R / Bioconductor: Curso Intensivo

Leonardo Collado Torres
Licenciatura en Ciencias Genómicas, UNAM
www.lcg.unam.mx/~lcollado/index.php

Cuernavaca, México
Oct-Nov, 2008

Bioconductor

Motivación

- `Bioconductor` surgió por la gran cantidad de datos experimentales que estaban siendo generados. Son tal cantidad, que requerimos de la estadística y de la bioinformática para analizarlos.
- Por otro lado, muchos laboratorios pierden tiempo re-escribiendo programas que ya otros habían hecho.
- El otro problema es tener acceso a los programas que otros hacen. Por eso, `Bioconductor` es de distribución libre.

De su funcionamiento

- `Bioconductor` empezó en el otoño del 2001.
- Las personas principales que mantienen a `Bioconductor` son del `Fred Hutchinson Cancer Research Center` en Seattle, USA.
- Salen dos versiones nuevas por año.
- En sí, muchos de los paquetes de `Bioconductor` están basados en la programación orientada a objetos y usan la clase S4.
- Acuérdense de que `R` puede ser útil para programar aunque muchos de los que mantienen a `Bioconductor` tienen una formación fuerte en estadística.

Visitemos la página

- Entren a la página de Bioconductor y navegen un poco.
- Notarán que hay:
  - Información sobre cursos.
  - Un FAQ.
  - Una sección donde te puedes registrar a las "mailing lists" por si quieres pedir ayuda y/o ayudar a otros.
  - Información sobre como instalar los paquetes.
  - Tal vez la sección más importante para nosotros es la parte de software. Generalmente cada paquete tiene PDFs donde lo explican y/o ponen ejercicios demo.

CRAN

- Mientras que `Bioconductor` es el repositorio de información y paquetes más relacionado a la genómica, acuérdense de CRAN.
- Allí pueden encontrar paquetes útiles para ustedes, como es el RMySQL que veremos adelante.
- En fin, en `Bioconductor` pueden bajar datos experimentales, software, etc.

limma: intro

- Vamos a empezar con el paquete limma cuya información se encuentra aquí.
- `limma` fue principalmente creado para analizar microarreglos, relaciones lineales y encontrar genes diferencialmente expresados.
- Sus derivados gráficos son `limmaGUI` y `affylmGUI` mientras que de cierta forma su competencia es `marray`.
- En sí, solo vamos a ver una parte del paquete ya que es bastante extenso. Incluso su guía oficial no cubre todo.
- Para ejemplificar este paquete y sus funciones, vamos a hacer una de las prácticas básicas.

Problema

- En el caso básico de `limma` trabajamos con 4 mediciones por gene en un microarreglo. Se usan dos colores: Cy3 y Cy5. Primero se hace una medición: WT Cy3, Experimento Cy5. Luego cambian los colores.
- Tenemos datos de *zebrafish*, ya que lo usan para estudiar el desarrollo temprano en vertebrados. Swirl es una mutante puntal del gene BMP2 que afecta al eje "dorsal/ventral" del cuerpo. Nuestro objetivo es usar los datos de este experimento para encontrar los genes con un nivel de expresión alterado en esta mutante comparado con el WT.
- Las hibridaciones del experimento se hicieron en dos sets con tintes intercambiados, por lo que tenemos cuatro mediciones por gene. En cada microarreglo, compararon el RNA del pez mutante swirl con el del pez normal.

Datos

- Para empezar, bajen los siguientes archivos a un directorio:
  - fish.gal
  - swirl.1.spot
  - swirl.2.spot
  - swirl.3.spot
  - swirl.4.spot
  - SpotTypes.txt
  - SwirlSample.txt
- Abran `R` y cambien su directorio[1] a donde bajaron los archivos. Podríamos hacer todo vía Internet, pero esto nos va a ahorrar líneas de código.

[1] Usen setwd.

readTargets

- Ahora, cargen el paquete `limma` que ya debería estar instalado en los servidores[2].
- Usen la función readTargets para leer una tabla con información de nuestro experimento.

```
> library(limma)
> targets <- readTargets("SwirlSample.txt")
> targets
```

```
  SlideNumber     FileName        Cy3
1          81 swirl.1.spot      swirl
2          82 swirl.2.spot wild type
3          93 swirl.3.spot      swirl
4          94 swirl.4.spot wild type
        Cy5      Date
```

readTargets


```
1 wild type 2001/9/20
2     swirl 2001/9/20
3 wild type 2001/11/8
4     swirl 2001/11/8
```

- Claro, siempre podrían haber llenado esto a mano :P Pero bueno, ahora ya tenemos información sobre como fue el experimento.
- En sí, nuestros archivos no son los más basicos, ya que los microarreglos fueron leídos por un escáner Axon para producir una imagen `TIFF` que después fue analizada con el software `SPOT`.

[2] Si no lo tienen en su lap, siempren pueden instalarlo con las funciones que vimos la 1era clase

read.maimages

- Vamos a usar la función read.maimages para leer los archivos generados por SPOT.
- Nos lee la intensidad *foreground* y *background* para los colores verde y rojo.

```
> RG <- read.maimages(targets$FileName,
+      source = "spot")
```

```
Read swirl.1.spot
Read swirl.2.spot
Read swirl.3.spot
Read swirl.4.spot
```

- Usamos nuestro objeto `targets` para obtener los nombres de los archivos. Ahora chequen su objeto `RG`.

```
> RG
```

readGAL

- De una vez nos podemos dar cuenta que los microarreglos de este experimento tienen 8448 puntos.
- Como info aparte, los experimentalisatas usaron 768 puntos de control. Además, la impresora del arreglo funcionó con una cabeza de impresión de arreglo 4x4. Cada cuadrícula consiste de 22x24 puntos que se imprimieron con una punta de impresión.
- En el archivo `GAL` tenemos el nombre del gene asociado con dada punto. Usamos la función readGAL para leer esta info:

```
> RG$genes <- readGAL("fish.gal")
```

getLayout

- Ahora, como se dieron cuenta, tenemos mucha información previa de como hicieron el microarreglo.
- Podríamos pasar la información (las dimensiones) de la impresora del arreglo manualmente. Claro, es posible que nos equivoquemos :P
- La función getLayout nos recupera esta información.

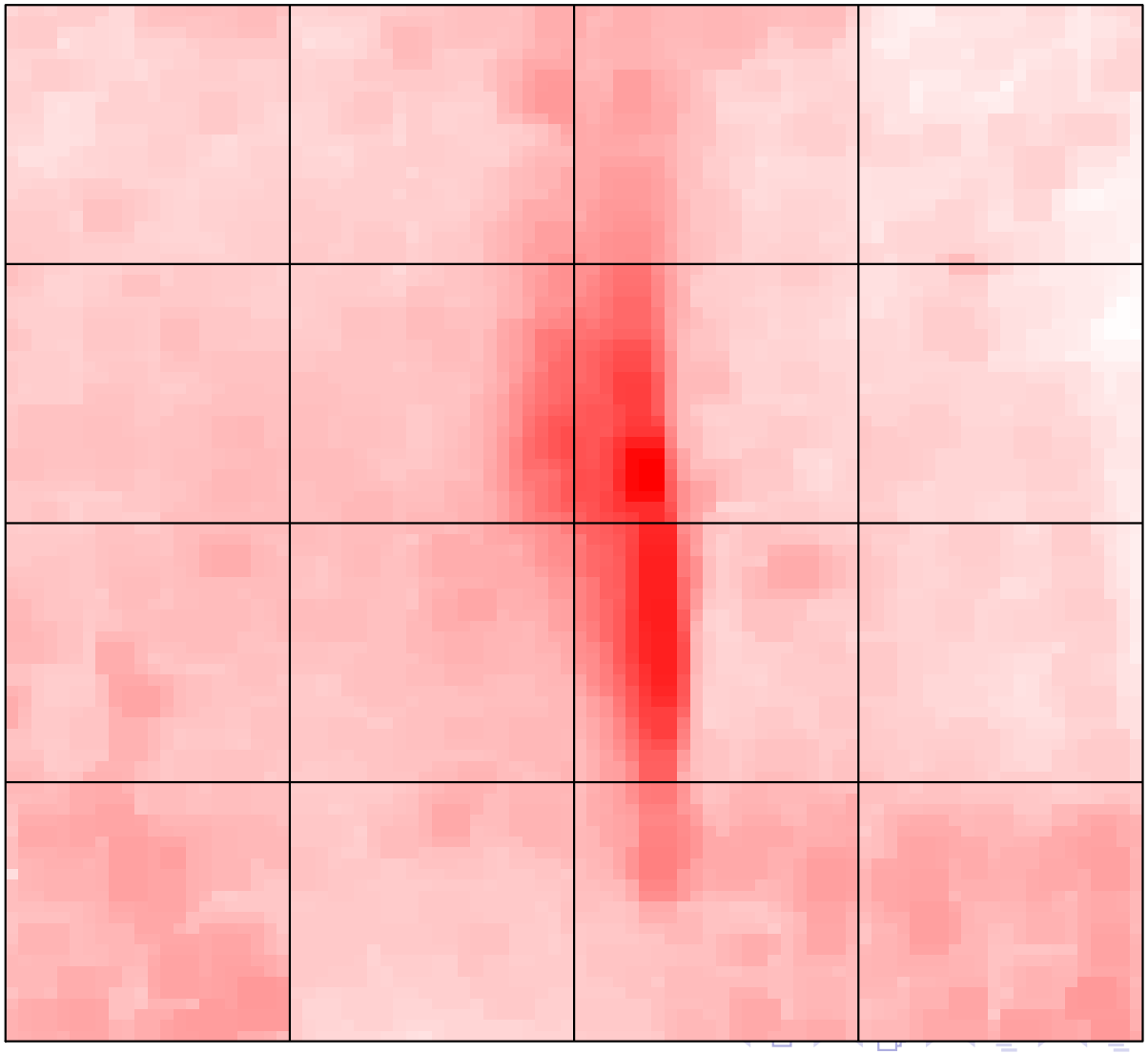
```
> RG$printer <- getLayout(RG$genes)
```

imageplot

- Una función del R básico que nos ayuda a visualizar una matriz es image.
- Para esto de los microarreglos hay una función similar (pero especializada) llamada imageplot.
- Vamos a usar esa función para ver si hay variación en el *background* de nuestros microarreglos.

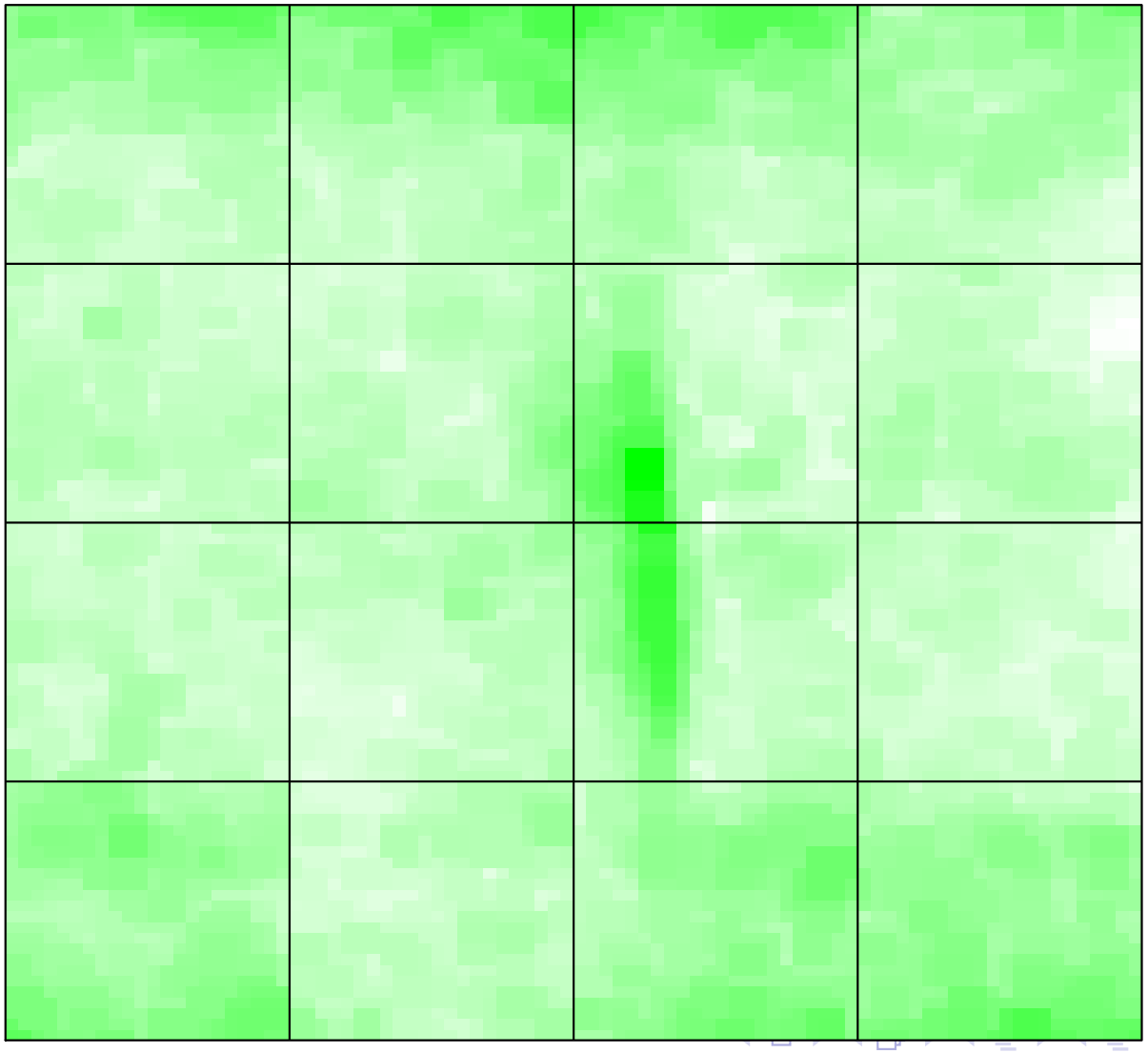
```
> imageplot(log2(RG$Rb[, 1]), RG$printer,
+     low = "white", high = "red")
> imageplot(log2(RG$Gb[, 1]), RG$printer,
+     low = "white", high = "green")
```

imageplot: rojo



z−range 5.9 to 11.1 (saturation 5.9, 11.1)

imageplot: verde



z−range 6.2 to 8.2 (saturation 6.2, 8.2)

normalizeWithinArrays

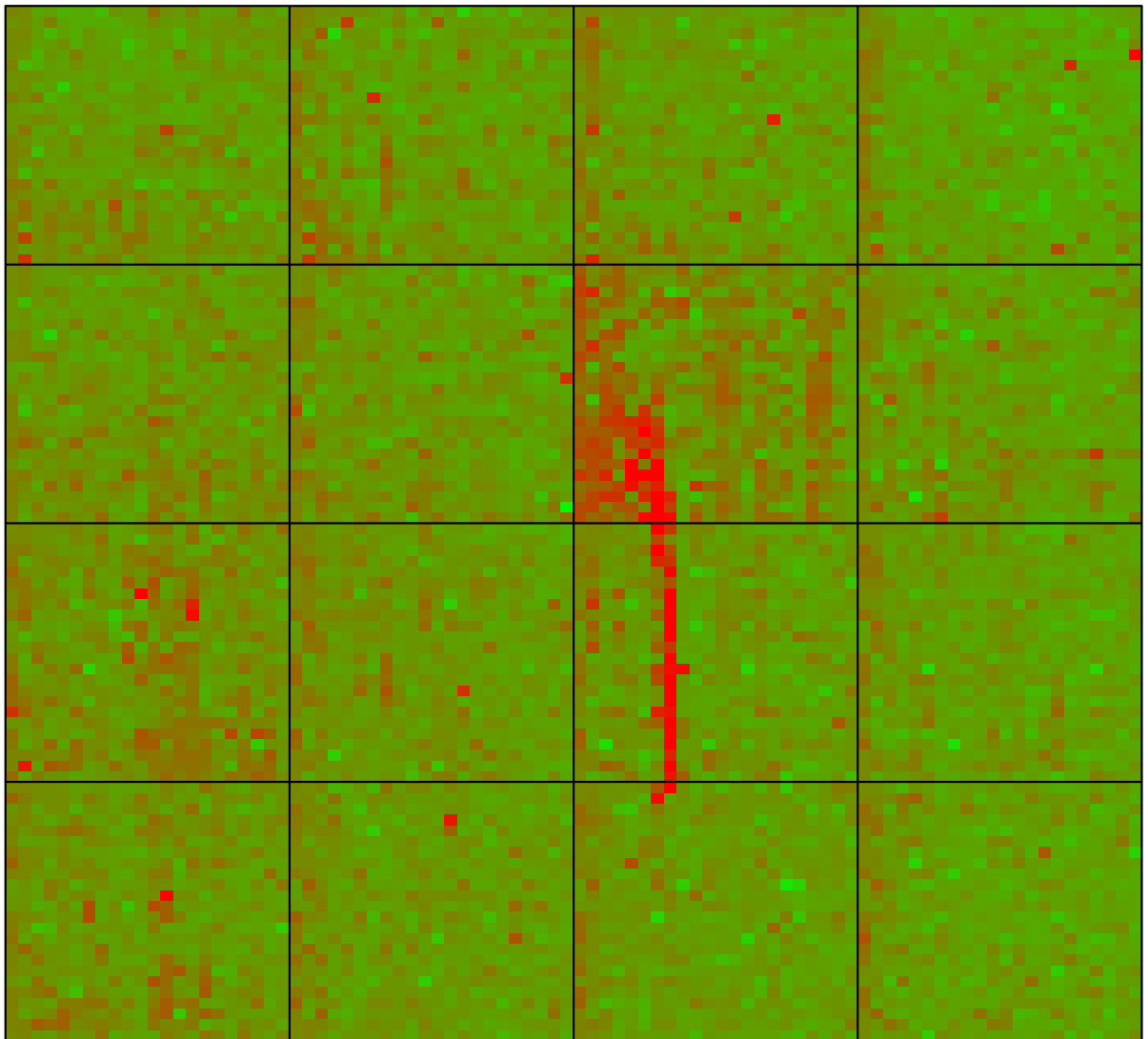
- Solo vimos la info del primer arreglo, pero nos dimos cuenta de que hay que normalizar los datos.
- Usemos la función normalizeWithinArrays, la cual nos normaliza los datos con sus *log-ratio* con tal de que la media de estos sea 0 en cada arreglo.

```
> MA <- normalizeWithinArrays(RG,
+      method = "none")
```

- Ahora veamos una imagen del primer arreglo ya normalizado:

```
> imageplot(MA$M[, 1], RG$printer,
+      zlim = c(-3, 3))
```

imageplot: normalizado



z−range −2.7 to 4.4 (saturation −3, 3)

o_O

- La función `imageplot` nos rota al arreglo, por lo que el grupo de hasta abajo a la izquierda es el primero.
- En la última gráfica podemos ver que hay una raya roja. Esto nos dice que había polvo o que el microarreglo estaba rayado.
- Los puntos que están en esa zona del artefacto, pues tendrán valores sospechosos.

plotMA

- Para esto de los microarreglos, sirve hacer una gráfica "MA".
- En estas, vamos a graficar la razón R vs G contra la intensidad del punto.
- El valor M está determinado por:

$$M = log_2(R) - log_2(G)$$

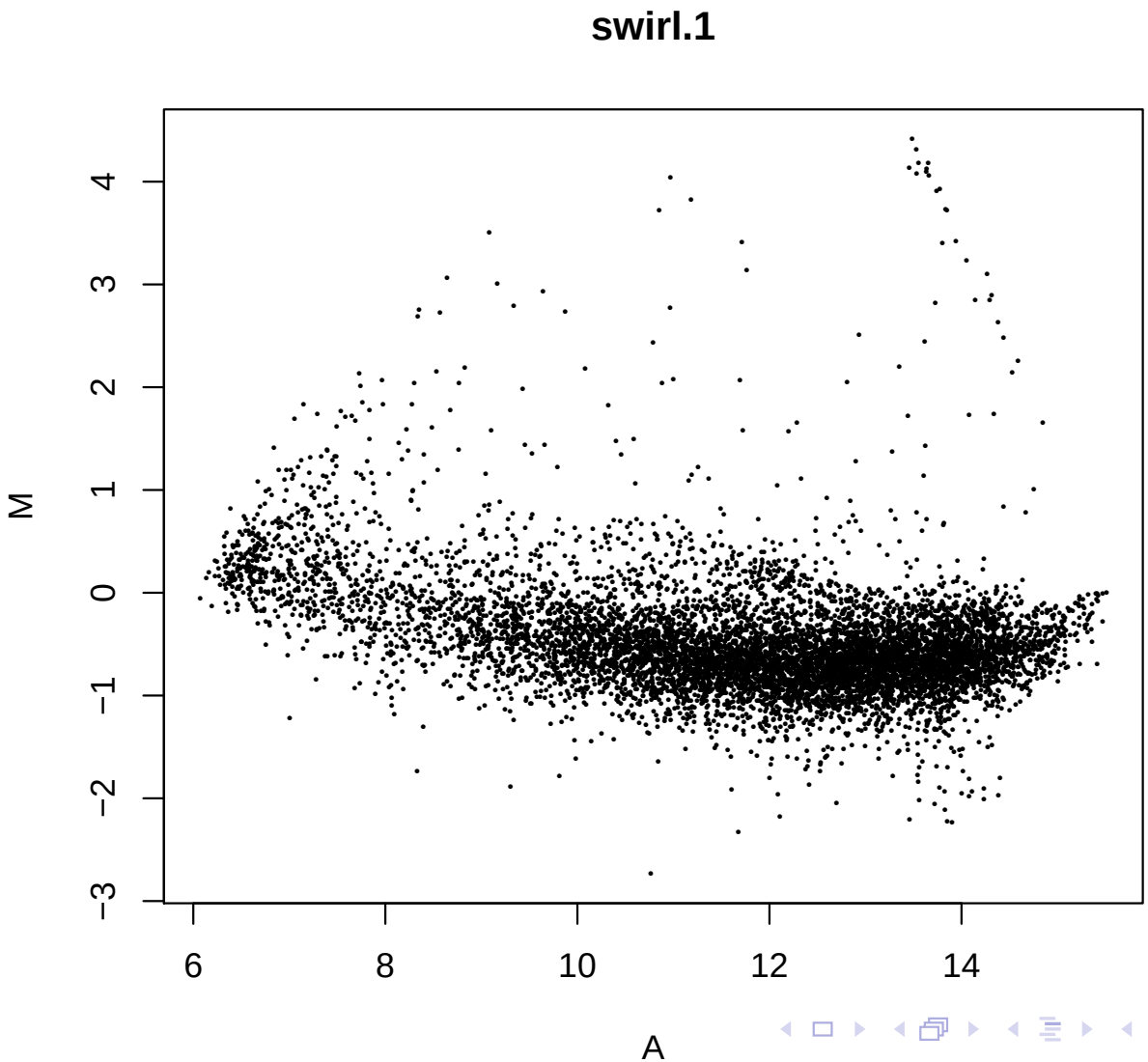
- El valor A representa la intensidad y está dado por:

$$A = (log_2(R) + log_2(G))/2$$

- Para hacer este tipo de gráficas, usamos plotMA.

```
> plotMA(MA)
```

plotMA: del 1er arreglo

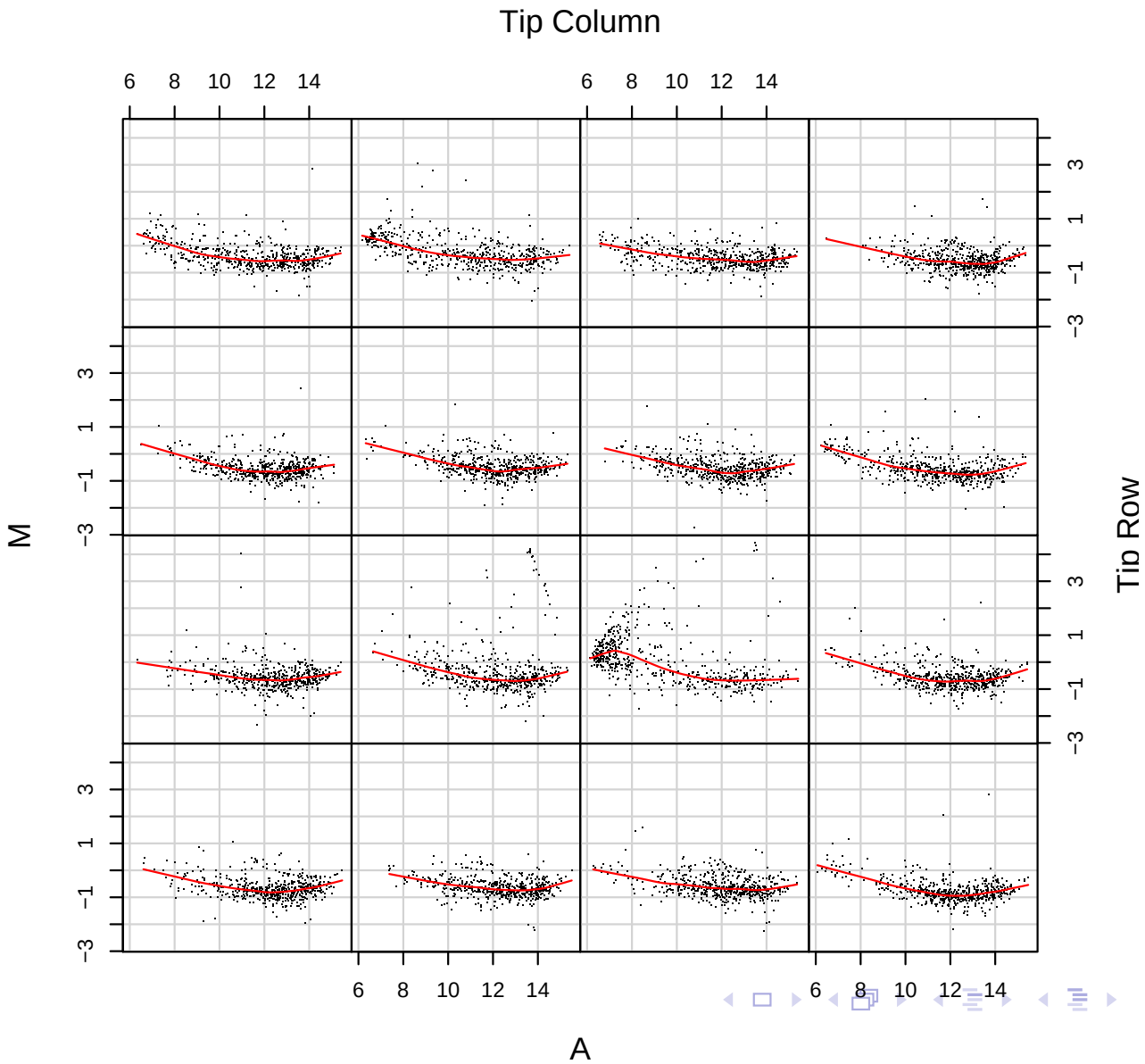


plotPrintTipLoess

- En la anterior gráfica podemos ver como esos valores derivados del artefacto salen en la esquina superior derecha.
- Cuando tenemos muchos datos, luego nos distraemos por los *outliers*. Por eso podemos usar las funciones de R lowess y loess. `lowess` hace una "locally weighted polynomial regression". Es más vieja por lo que no utiliza la notación de fórmula.
- Ahora vamos a usar plotPrintTipLoess para visualisar los datos completos de nuestro primer arreglo y la curva loess con la cual vamos a normalizarlos.

```
> plotPrintTipLoess(MA)
```

plotPrintTipLoess: arreglo 1



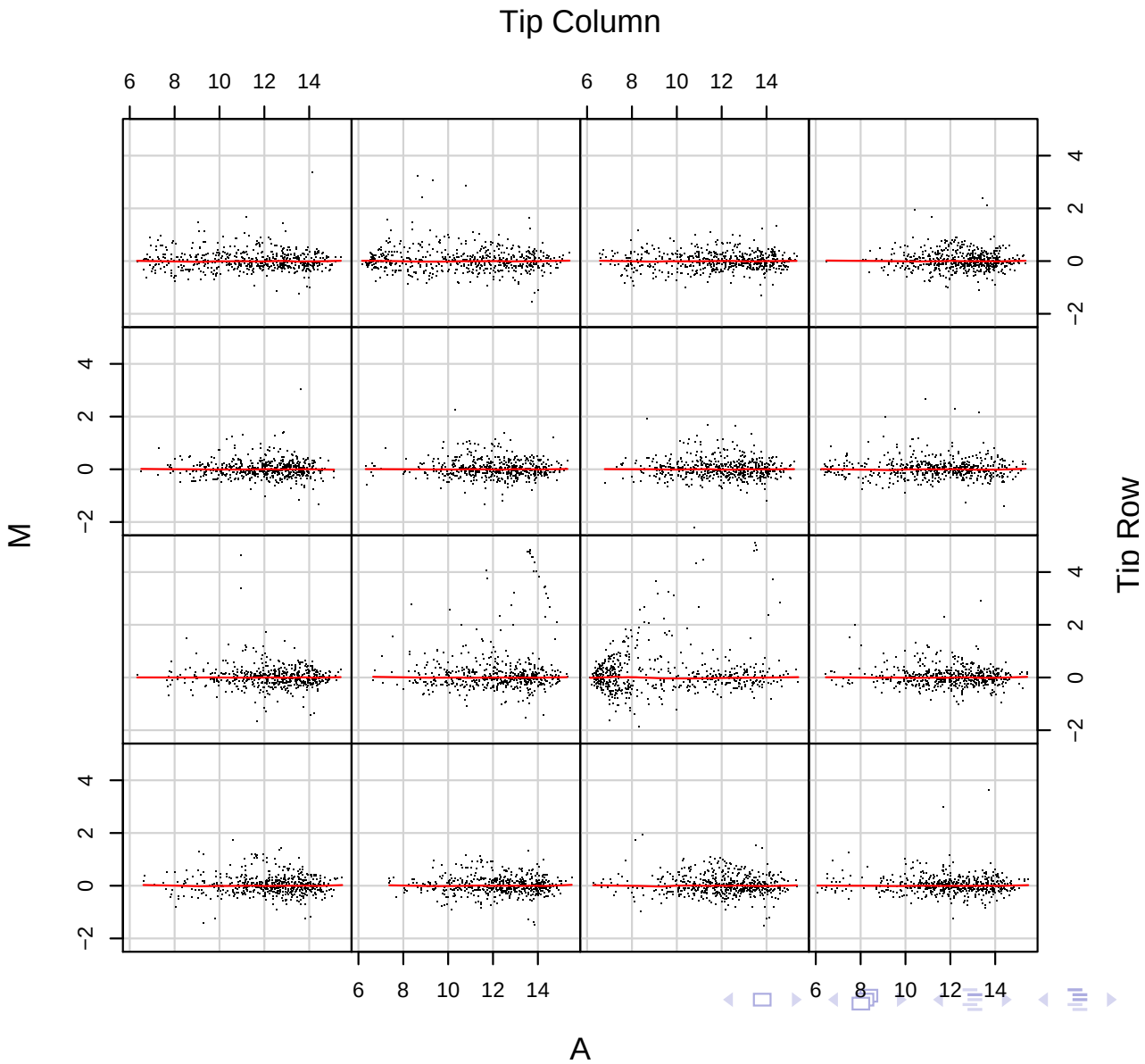
Normalizando

- Ahora normalizamos los datos con los parámetros default y volvemos a hacer el mismo tipo de gráfica.
- En realidad, vamos a normalizar los valores M para cada arreglo.

```
> MA <- normalizeWithinArrays(RG)
```

```
> plotPrintTipLoess(MA)
```

Grafica normalizado

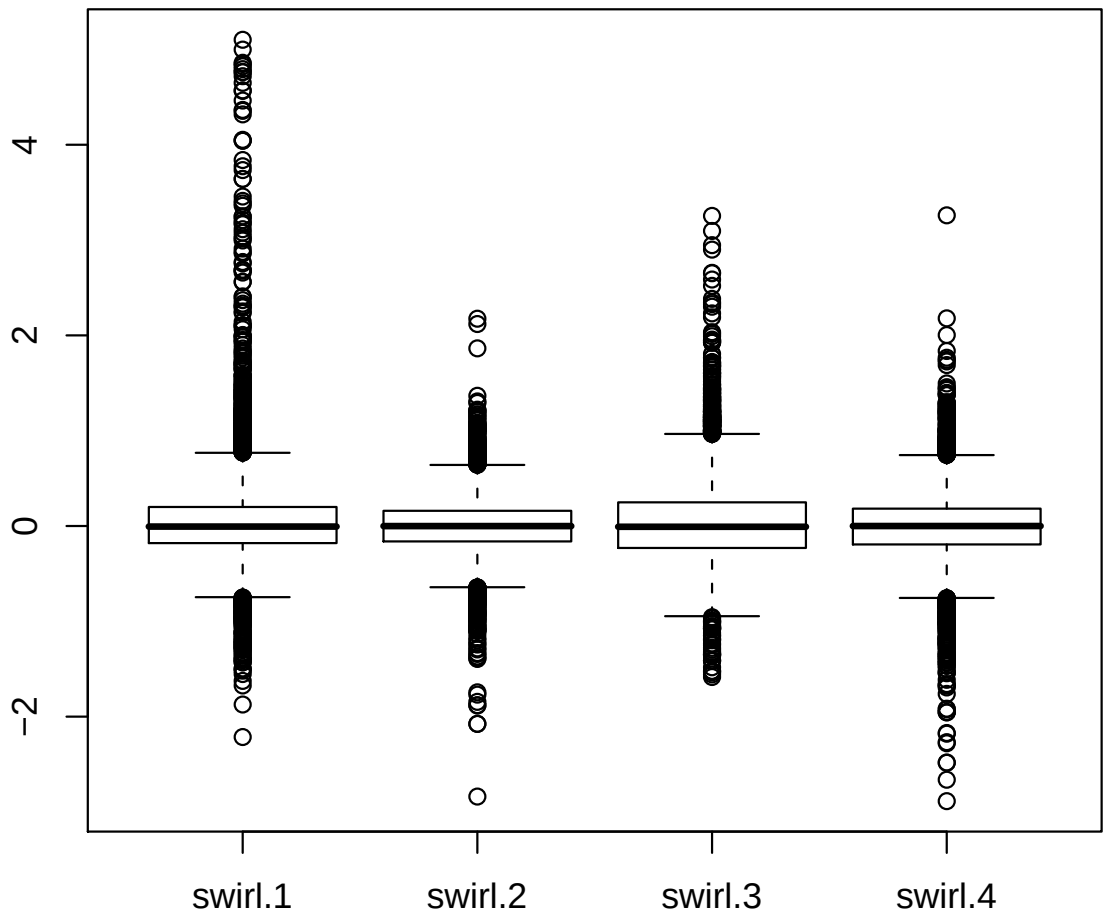


Entre arreglos?

- Una vez hecho lo anterior, nos podemos preguntar si es que hay que normalizar entre nuestros 4 microarreglos.
- Para eso usamos la ya tan conocida función `boxplot`:

```
> boxplot(MA$M ~ col(MA$M), names = colnames(MA$M)
```

boxplot: nuestra evidencia

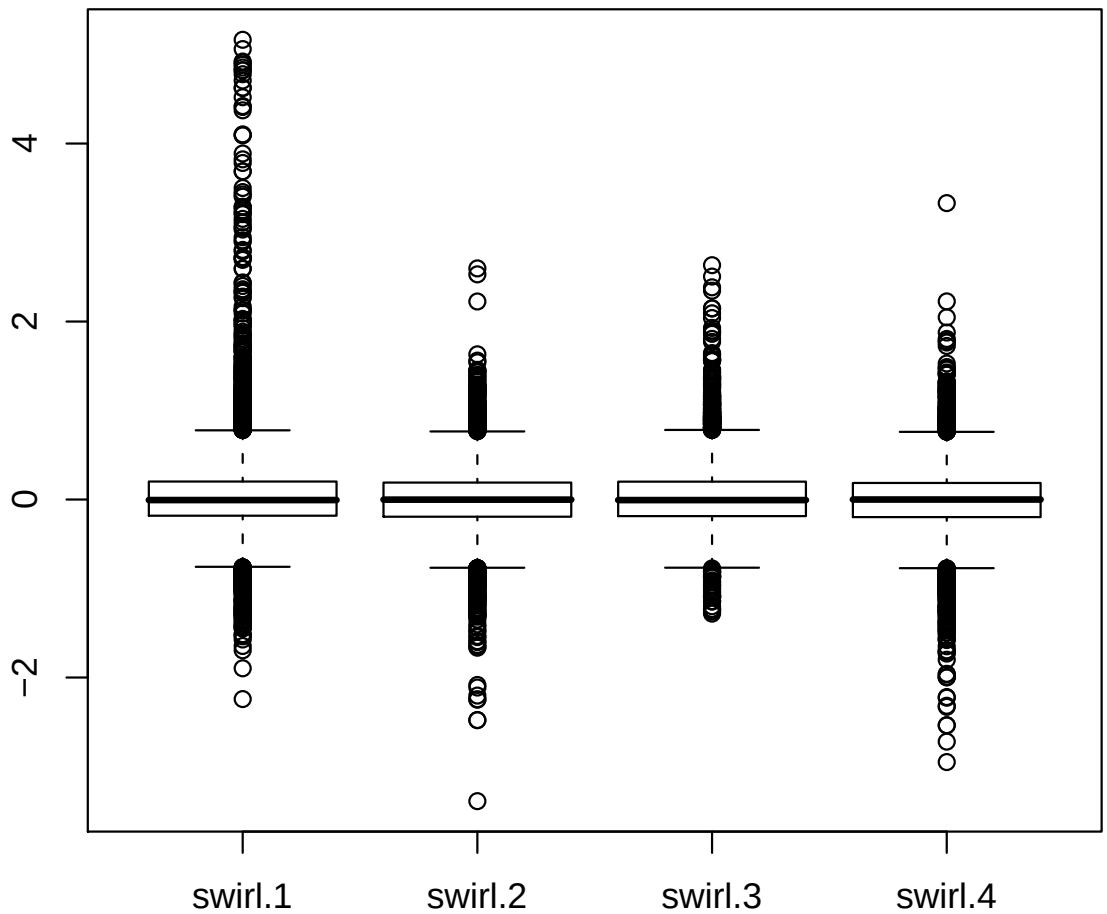


normalizeBetweenArrays

- Ahora, como si hay evidencia de variación entre los arreglos, pues vamos a normalizarlos. Esto no se hace por rutina ya que muchas veces no es necesario.
- Para eso usamos la función normalizeBetweenArrays y luego comprobamos el resultado con un `boxplot`. Hay varios métodos los cuales pueden checar en la ayuda, pero solo usaremos el default.
- Dicho método es el `Aquantile` que se asegura de que los valores A tengan la misma distribución empírica en los arreglos sin cambiar los valores M.

```
> MA <- normalizeBetweenArrays(MA,
+     method = "scale")

> boxplot(MA$M ~ col(MA$M), names = colnames(MA$M)
```

normalizeBetweenArrays: res



lmFit

- Ahora vamos a hacer un modelo lineal para poder estimar el valor M de cada gene.
- Primero tenemos que especificar el diseño del experimento. Osea, cuando usaron que tinte.

```
> design <- c(-1, 1, -1, 1)
```

- Luego, hacemos nuestra regresión lineal con la función lmFit que está específicamente diseñada para microarreglos.
- Dicha función nos va a regresar mucha información :)

```
> fit <- lmFit(MA, design)
```

- Corran el siguiente comando para obtener un despliegue con mayor información:

```
> fit
```

Pruebas *t*

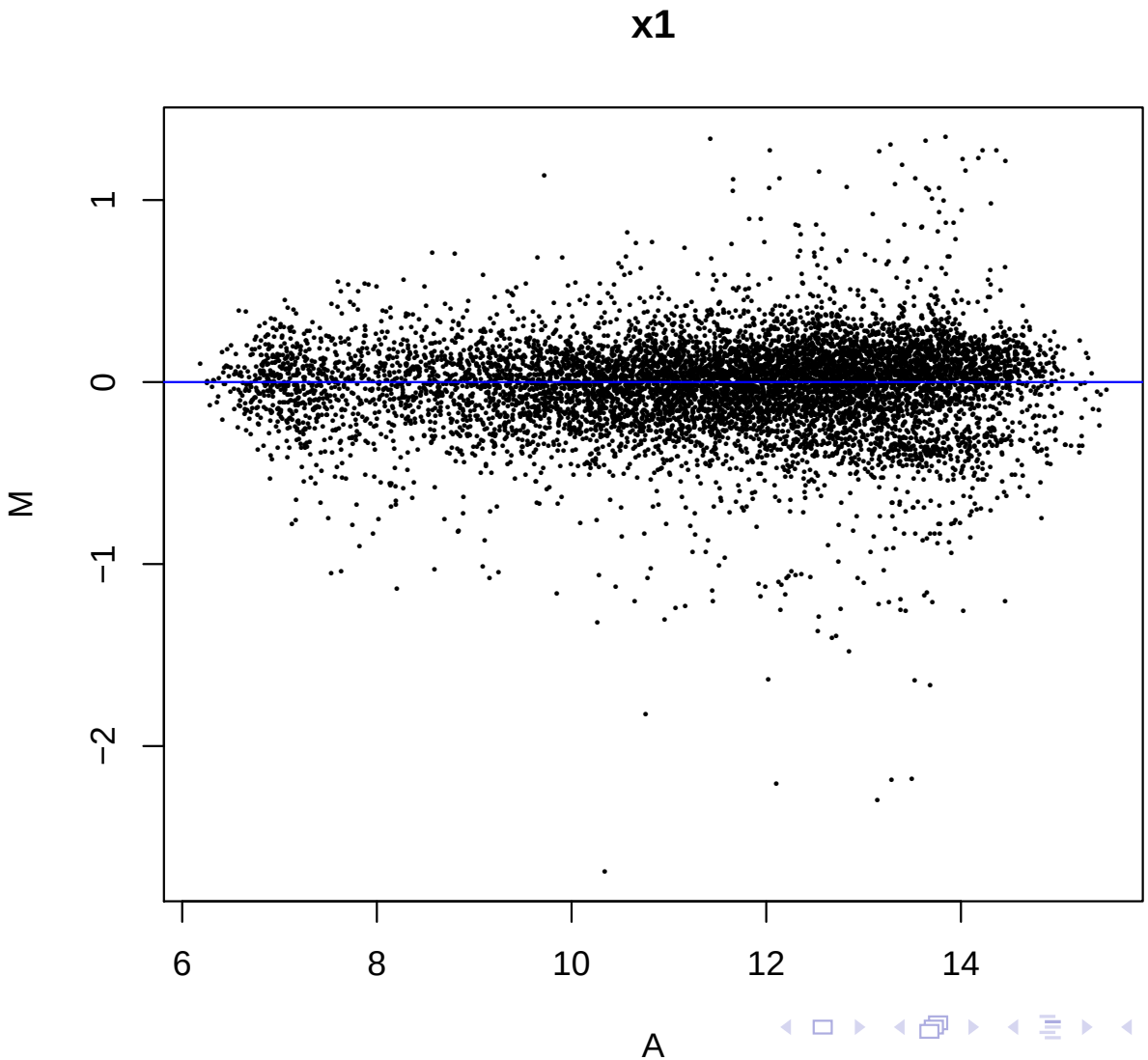
- En el caso de nuestro objeto `fit`, `coefficients` es el valor M promedio mientras que `sigma` es la desviación estándar muestral para cada gene.
- Podemos hacer pruebas t comparar la mutante con el WT gene por gene así:

```
> ordinary.t <- fit$coef/fit$stdev.unscaled/fit$s
```

- Ahora hacemos una gráfica de los valores promedio M y A para cada gene.

```
> plotMA(fit)
> abline(0, 0, col = "blue")
```

Promedio M vs A



eBayes

- De acuerdo a los creadores de `limma` es mejor usar pruebas t moderadas con Bayes empírico con la función eBayes.
- Estos nos van a servir para encontrar los genes diferencialmente expresados.
- En sí, `eBayes` utiliza la estimación de Bayes empírica para minimizar los errores estándares hacia un valor común.

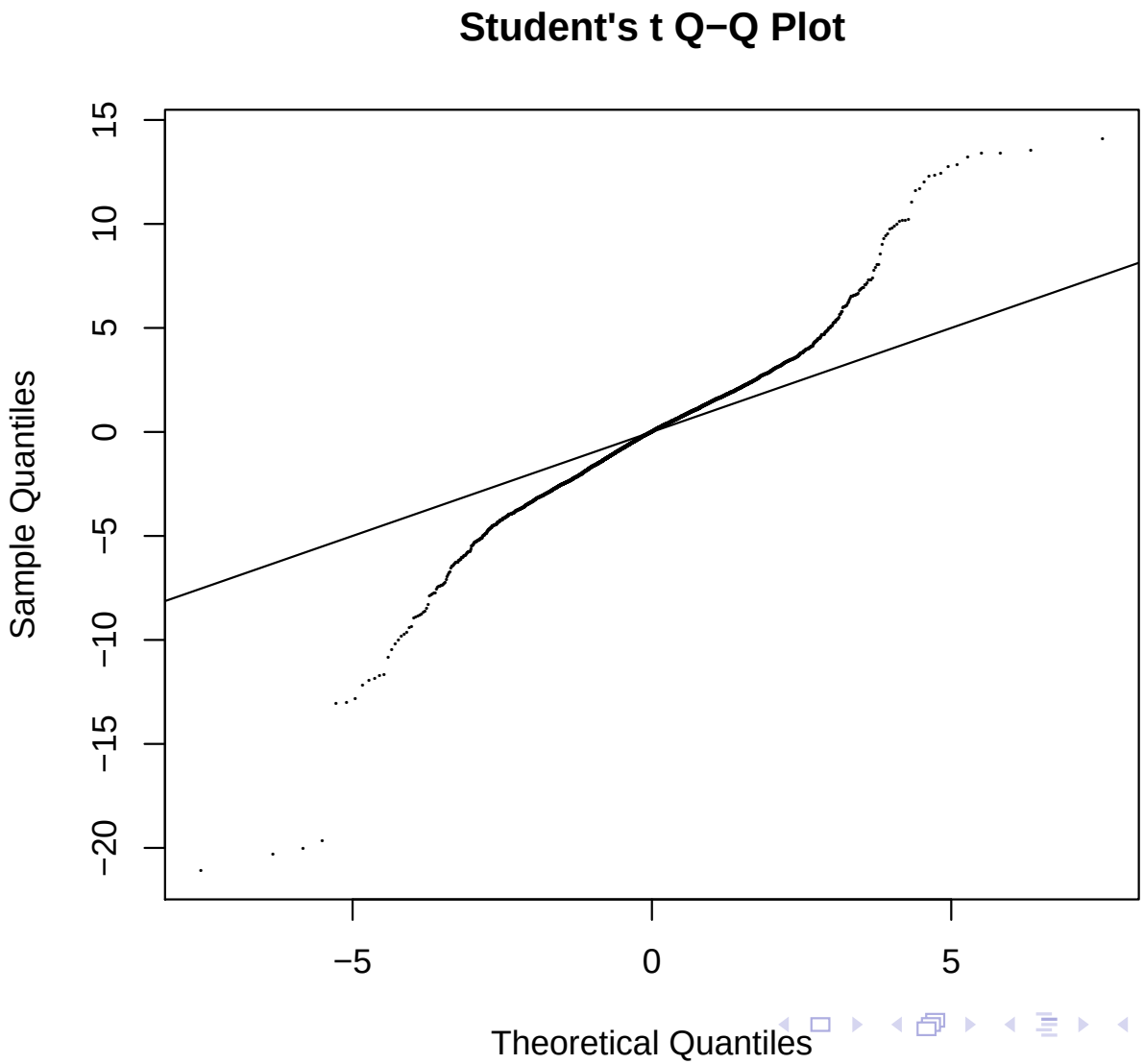
  ```
  > fit <- eBayes(fit)
  ```

- Luego vamos a hacer una gráfica QQ para ver si tenemos genes diferencialmente expresados. Usamos qqt en vez de qq, ya que queremos comparar nuestros cuantiles contra los de una distribución t y no una normal.

eBayes


```
> qqt(fit$t, df = fit$df.prior +
+     fit$df.residual, pch = 16,
+     cex = 0.2)
> abline(0, 1)
```

qqt


Student's t Q−Q Plot
Sample Quantiles
Theoretical Quantiles
15
10
5
0
−5
−10
−15
−20
−5
0
5

topTable

- En la anterior gráfica nos podemos dar cuenta de que tenemos muchos genes diferencialmente expresados en nuestro experimento.
- Queremos saber cuales son, así que usamos la función topTable. Un argumento importante es `adjust.method`, ya que aquí especificamos como vamos a corregir nuestros valores p.
- Por ejemplo, con el siguiente código podemos ver los 30 (o el 1er) genes con una expresión diferencial más marcada ajustando los valores p por FDR.

```
> options(digits = 3)
> topTable(fit, number = 30, adjust = "BH")

> topTable(fit, number = 1, adjust = "BH")
```

topTable


```
      Block Row Column      ID Name
3721      8   2      1 control BMP2
          logFC  AveExpr         t
3721 -2.205288 12.10451 -21.06952
           P.Value    adj.P.Val       B
3721 1.028468e-07 0.0003572816 7.96075
```

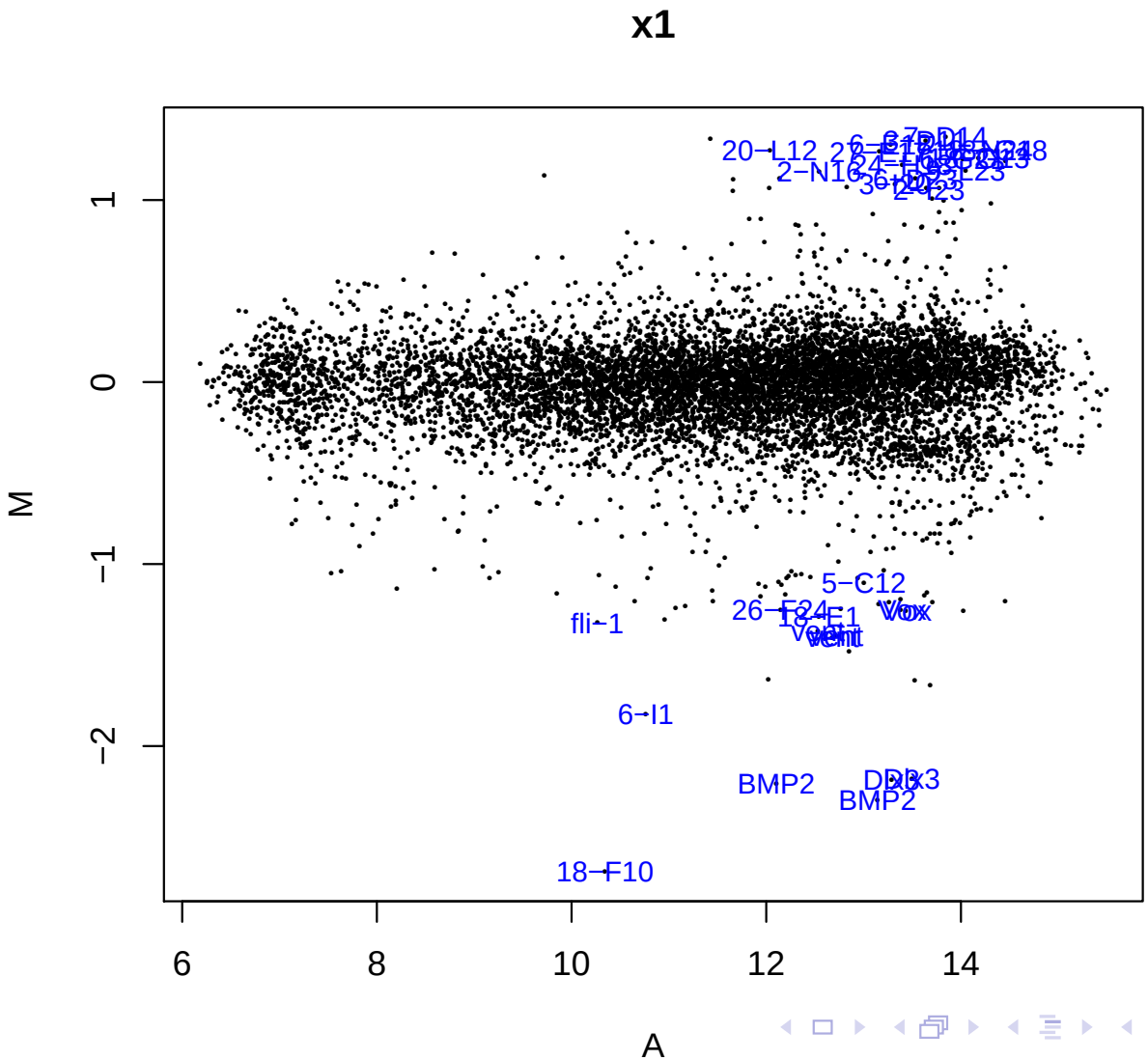
- Como era de esperarse, nuestro gene con mayor diferencia es BMP2 ya que este lo tenemos mutado en la línea Swirl. A parte, salen otros que son controles.
- También podemos ver los valores p originales, los ajustados y los *log odds* de la estadística empírica de Bayes.

Terminando con limma

- Ya para terminar con nuestra práctica de limma vamos a marcar nuestros 30 genes en nuestra gráfica anterior.
- Usaremos order y text del R base para hacer esto.

```
> plotMA(fit)
> ord <- order(fit$lods, decreasing = TRUE)
> top30 <- ord[1:30]
> text(fit$Amean[top30], fit$coef[top30],
+     labels = fit$genes[top30, "Name"],
+     cex = 0.8, col = "blue")
```

limma: FIN



Una práctica

- Hay un grupo muy fuerte que trabaja con `R` para analizar sus microarreglos de `Affimetrix`.
- En sí, si checan el software de `Bioconductor` verán muchos paquetes al respecto.
- Vamos a hacer un ejercicio con datos de `Affy`, aunque si quieren checar con más detalle la introducción pueden hacerlo aquí.
- Para empezar, entramos a `GEO`[3] y bajamos unos archivos en formato `CEL` usando `Arabidopsis[Organism] AND Atgenexpress[Title] AND light[Title]`. Para esta práctica, pueden bajarlos de arabidopsis_CEL.zip.
- Por ahora quiero que chequen la descripción de los datos en GEO.

[3] Gene Expression Omnibus

Targets

- Hay que generar nuestro archivo "targets.txt" con la información de nuestros 9 archivos.

Name	FileName	Target
DS REP1	GSM131177.CEL	dark45m
DS REP2	GSM131192.CEL	dark45m
DS REP3	GSM131207.CEL	dark45m
PS REP1	GSM131179.CEL	red1m dark44m
PS REP2	GSM131193.CEL	red1m dark44m
PS REP3	GSM131209.CEL	red1m dark44m
BS REP1	GSM131181.CEL	blue45m
BS REP2	GSM131195.CEL	blue45m
BS REP3	GSM131211.CEL	blue45m

Normalizaciones

- Vamos a hacer 3 normalizaciones diferentes que no son sencillas. Pueden checar la liga anterior y encontrar los artículos relacionados.
  - RMA: corrige el BG por el fondo del arreglo, normaliza por cuantiles y utiliza un modelo lineal robusto para resumir los datos.
  - GCRMA: corrige el BG por contenido de GC, normaliza por cuantiles y utiliza un modelo lineal robusto para resumir los datos.
  - MAS5: corrige el BG por regiones, normaliza al escalar por una constante, utiliza una correción MM y resume los datos utilizando un promedio bi-peso de Tukey.

RMA

- Primero cargamos los paquetes y leemos los datos. Una función nueva es ReadAffy.

```
> library(affy)
> library(limma)
> library(gcrma)
> targets <- readTargets("targets.txt")
> data <- ReadAffy(filenames = targets$FileName)
```

- Ahora normalizamos los datos con RMA y se los asignamos a un objeto exprSet.

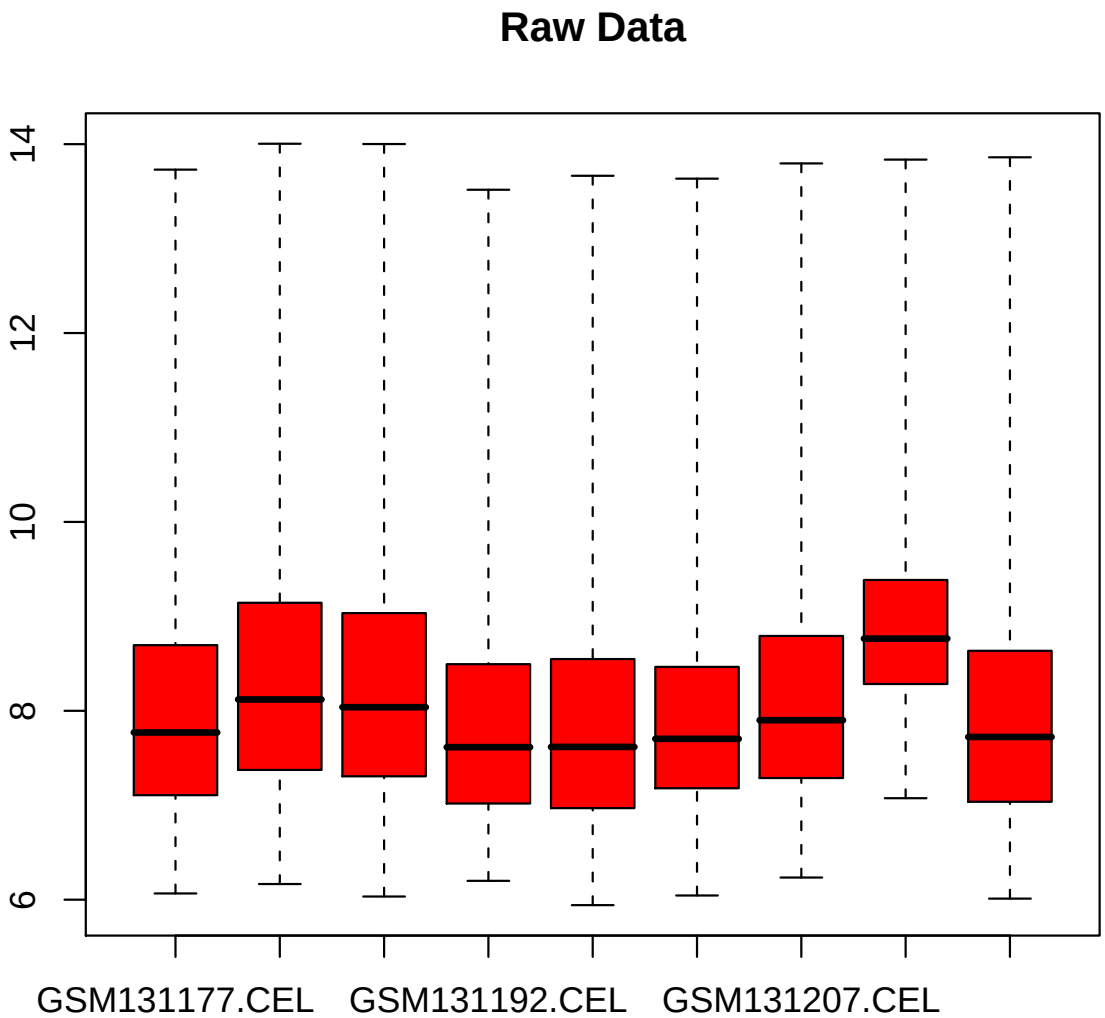
```
> eset_rma <- rma(data)

Background correcting
Normalizing
Calculating Expression
```

Visualizar

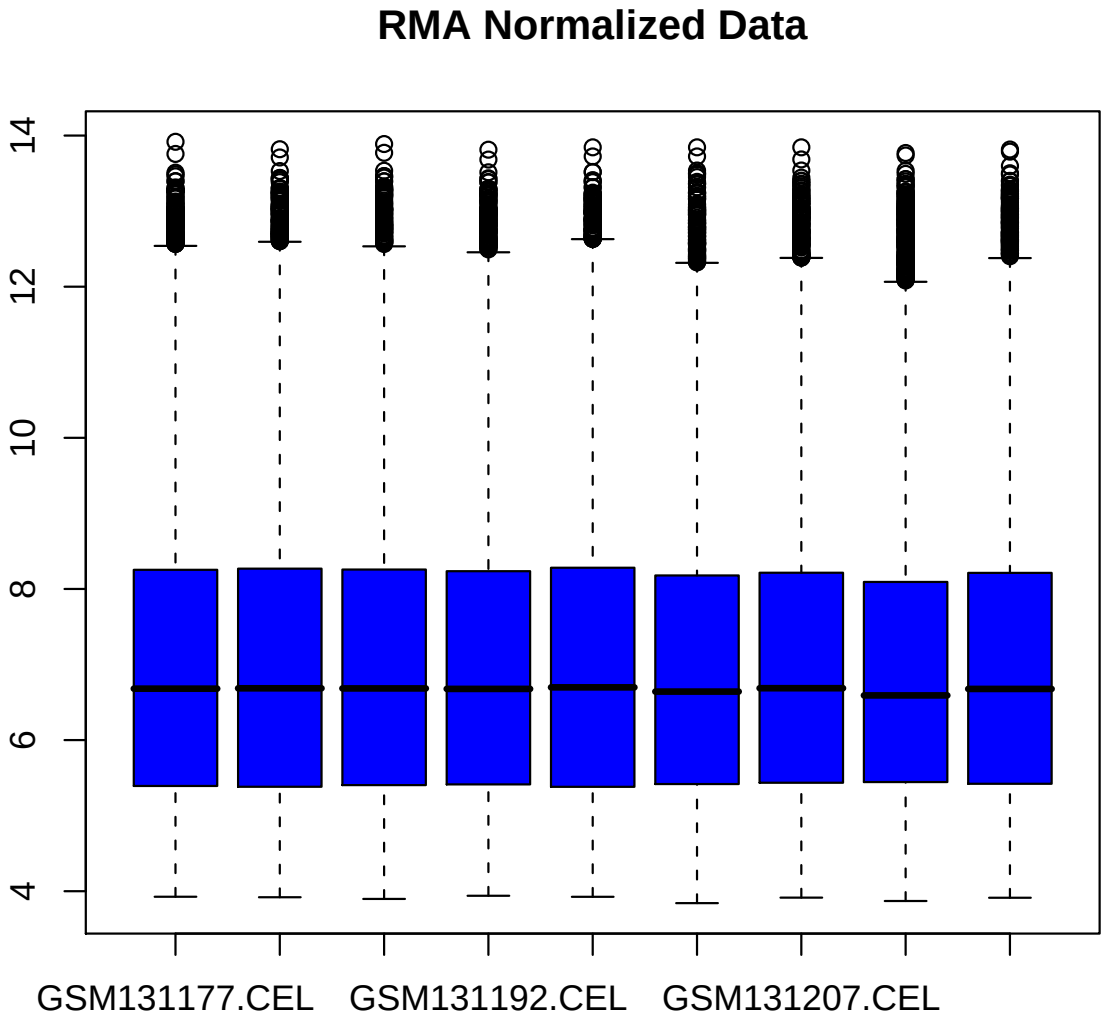
- Ahora visualizamos nuestros datos crudos y los normalizados

```
> boxplot(data, col = "red", main = "Raw Data")
> boxplot(data.frame(exprs(eset_rma)),
+         col = "blue", main = "RMA Normalized Data")
```

boxplot: datos crudos



boxplot: datos normalizados con RMA



Análisis GDE: RMA

- Procedemos a encontrar los genes diferencialmente expresados.
- Primero creamos nuestra matrix del diseño del experimento usando model.matrix y le agregamos mejores nombres.

```
> design <- model.matrix(~-1 + factor(c(1,
+     1, 1, 2, 2, 2, 3, 3, 3)))
> colnames(design) <- c("S1", "S2",
+     "S3")
```

- Luego definimos las comparaciones que queremos hacer con makeContrasts y ajustamos cada gene a un modelo lineal.

```
> contrast.matrix <- makeContrasts(S2 -
+     S1, S3 - S2, S3 - S1, levels = design)
> fit <- lmFit(eset_rma, design)
```

Análisis GDE: RMA

- Además, calculamos coeficientes estimados y errores estándares para nuesto conjunto de contrastes. Luego volvemos a usar eBayes.

```
> fit2 <- contrasts.fit(fit, contrast.matrix)
> fit2 <- eBayes(fit2)
```

- Finalmente, filtramos nuestros resultados y los exportamos a un archivo Excel usando la función nueva rma_deg_result.

```
> rma_deg_result <- topTable(fit2,
+     coef = 1, adjust = "fdr", sort.by = "B",
+     number = 50000)
> rma_deg_result <- rma_deg_result[rma_deg_result$
+     0.05, ]
> write.table(rma_deg_result, "rma_deg_result.xls
```

Análisis GDE: RMA


```
+       quote = FALSE, row.names = FALSE,
+       sep = "\t")
```

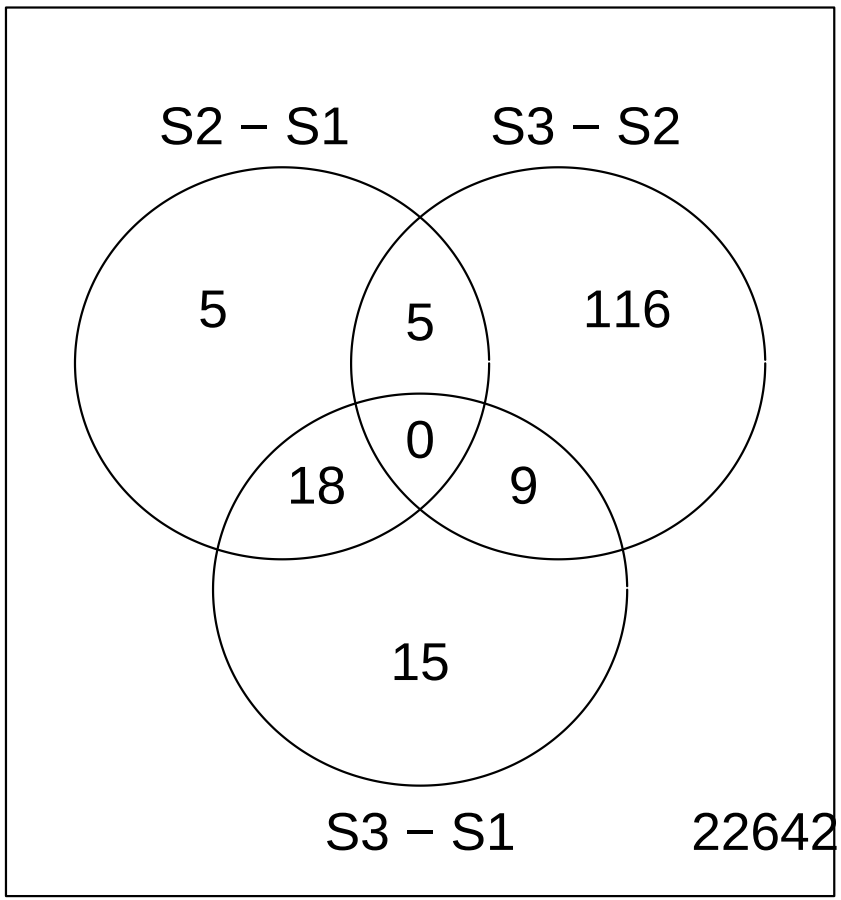
Venn: RMA

- Finalmente creamos nuestro diagrama de Venn para todos los genes con un valor p menor o igual a 0.05

```
> rma_venn <- decideTests(fit2, p.value = 0.05)
```

```
> vennDiagram(rma_venn)
```

Venn: RMA



GCRMA y MAS5

- Ahora, hacemos lo mismo para GCRMA y MAS5.
- El código en sí es muy parecido aunque algunas funciones cambian de nombre. Por ejemplo, usamos gcrma_deg_result en vez de `rma_deg_result`.

```
Adjusting for optical effect.........Done.
Computing affinities.Done.
Adjusting for non-specific binding.........Done.
Normalizing
Calculating Expression
```

- La otra diferencia, es que para MAS5 hay que usar valores absolutos de las intensidades y no los logarítmicos.

```
> eset_mas5 <- mas5(data)
```

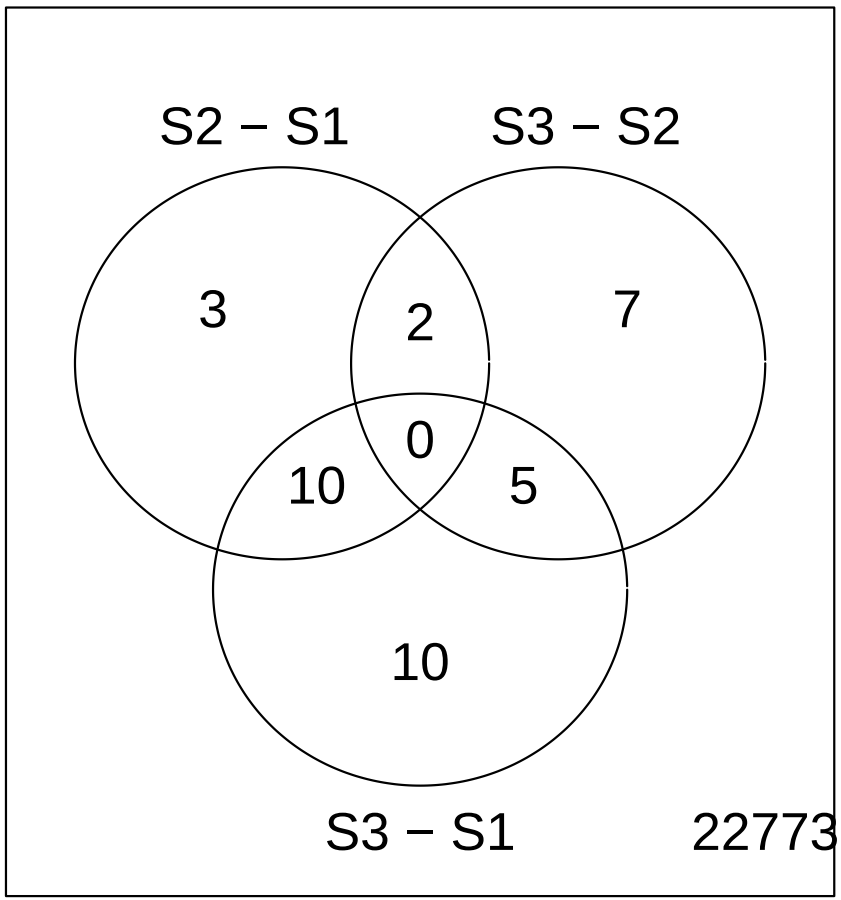
GCRMA y MAS5


```
background correction: mas
PM/MM correction : mas
expression values: mas
background correcting...done.
22810 ids to be processed
|                    |
|####################|
```

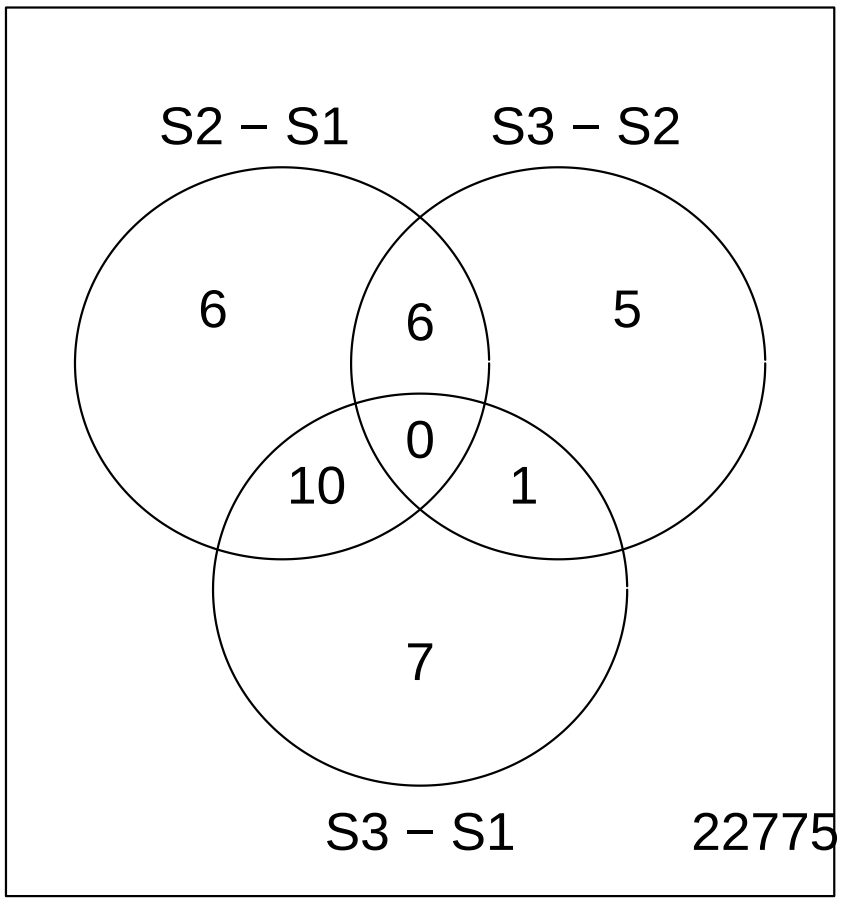
```
> exprs(eset_mas5) <- log2(exprs(eset_mas5))
```

- Obtenemos los siguientes diagramas de Venn:

Venn: GCRMA



Venn: MAS5



Sobrelape

- Queremos ahora ver que tanto se sobrelapan los resultados de los diferentes métodos. Para eso usamos dos veces la función merge.
- Finalmente creamos un archivo `Excel` con la información resultante

```
> overlap2 <- (merge(rma_deg_result,
+     gcrma_deg_result, by.x = "ID",
+     by.y = "ID", all = FALSE))
> overlap3 <- (merge(overlap2, mas5_deg_result,
+     by.x = "ID", by.y = "ID", all = FALSE))
> write.table(overlap3, "overlap.xls",
+     quote = FALSE, row.names = FALSE,
+     sep = "\t")
```

Sobrelape

- Completamos nuestra comparación de métodos para encontrar los GDE :). Si se fijaron, usamos mucho a `limma` para analizar los datos de `Affy`.

Paquete mcmc

- Como les dije al principio, hay otros paquetes que les pueden ser interesantes en `CRAN`.
- Por ejemplo, está el paquete mcmc con el cual podemos hacer una cadena de Markov Monte Carlo para un vector de números al azar contínuos usando el algoritmo de Metropolis.

Un ejemplo básico

- Primero llamamos al paquete y definimos una función para ver cuando cambiamos de punto:

```
> library(mcmc)
> h <- function(x) if (all(x >= 0) &&
+     sum(x) <= 1) return(1) else return(-Inf)
```

- Además, lo corremos para generar 1000 datos usando un vector de cinco 0s.

```
> out <- metrop(h, rep(0, 5), 1000)
```

- Podemos var el porcentaje de nuestros puntos que son aceptados de acuerdo a nuestra función `h`.

```
> out$accept
```

```
[1] 0
```

Mejorando

- Como nuestra taza de aceptación es muy baja, lo corremos de nuevo. Pero ahora cambiamos el tamaño de la ventana sobre el cual estamos trabajando usando el argumento `scale`.

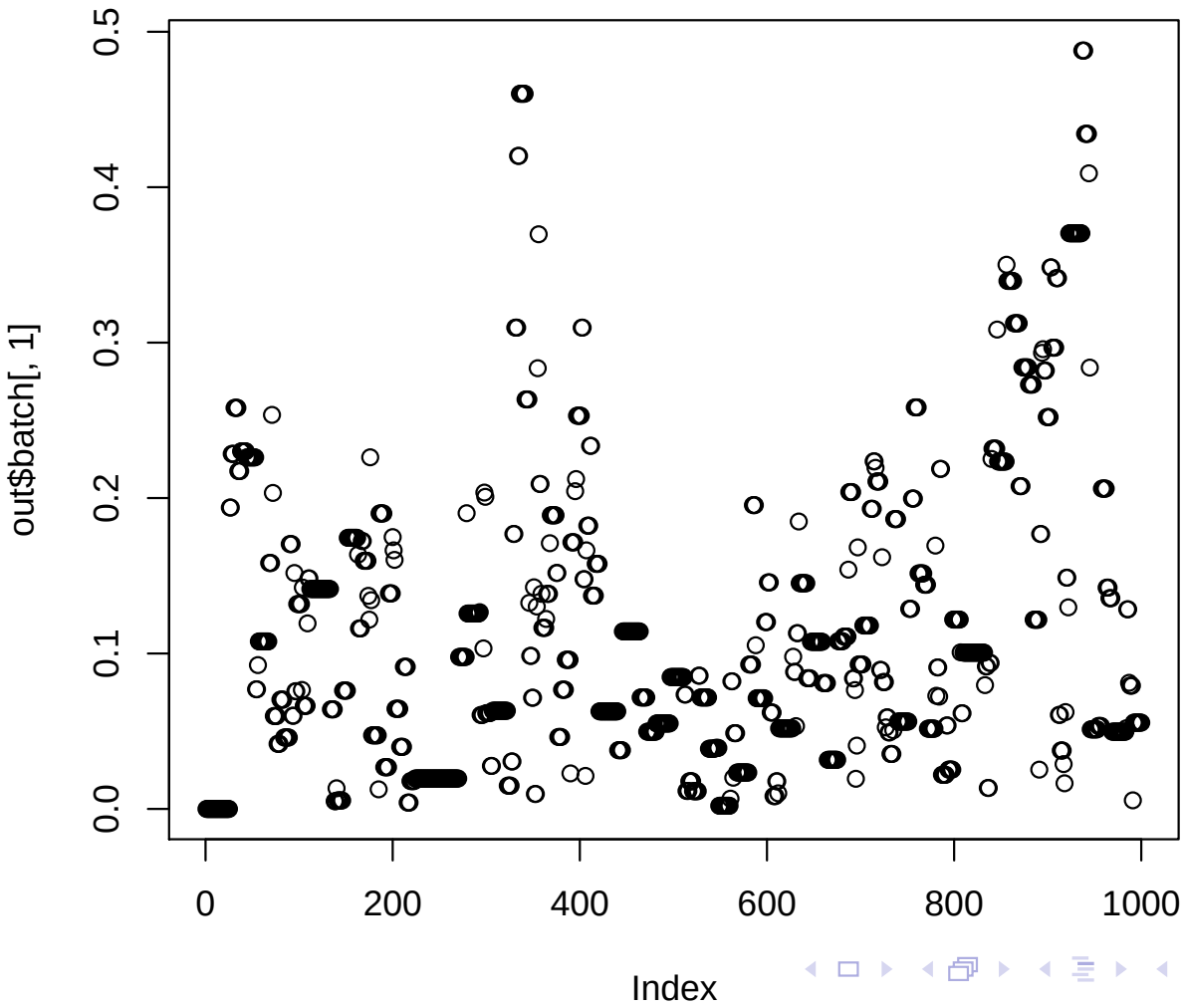
```
> out <- metrop(out, scale = 0.1)
> out$accept
```

```
[1] 0.239
```

- Como estamos cerca del 25 por ciento, pues ahora queremos hacer una gráfica de nuestros datos que generamos:

```
> plot(out$batch[, 1])
```

Con 25 %



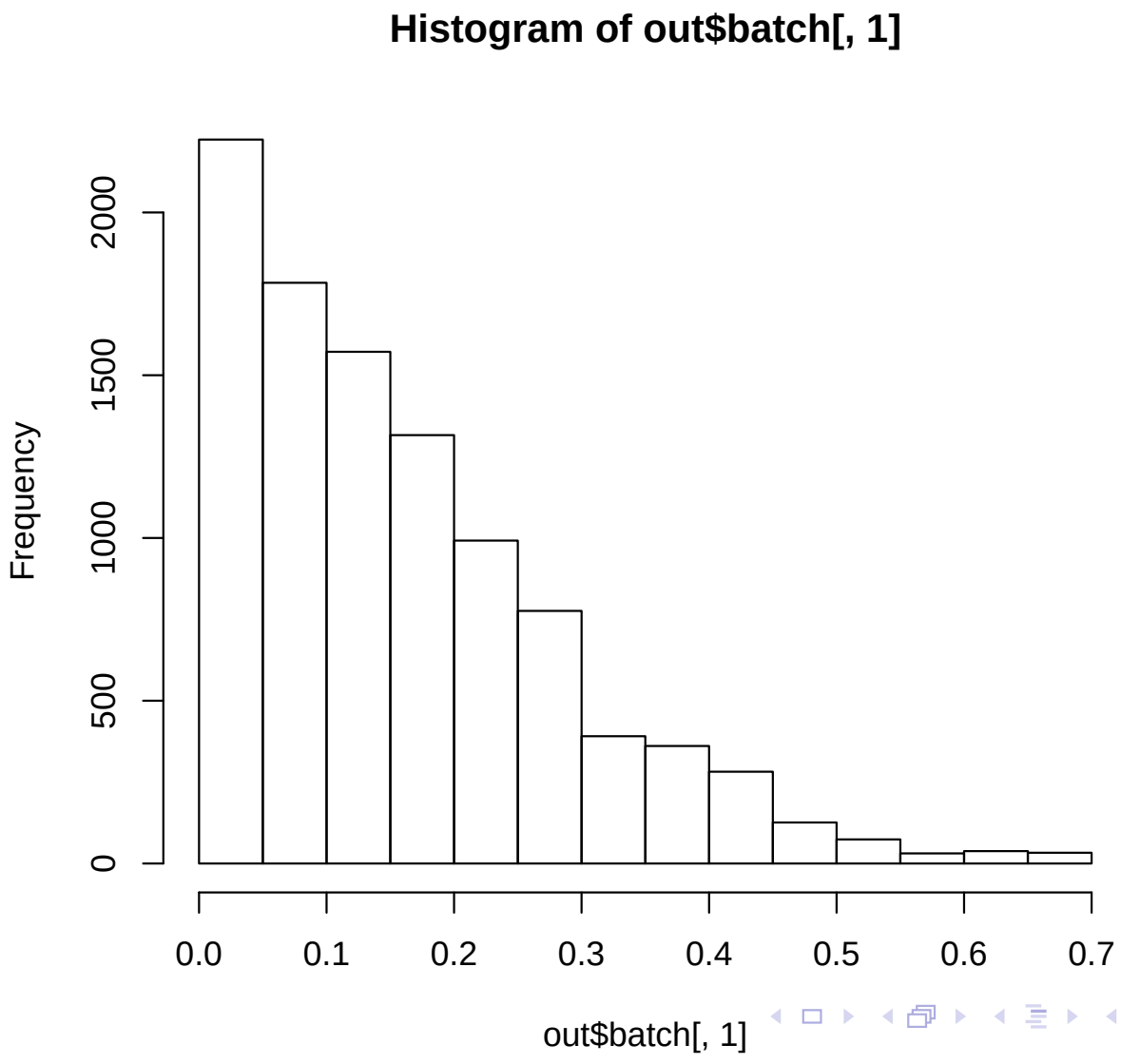
Finalizando

- Sin embargo, no tenemos suficientes datos por lo que volvemos a correr para generar 10 000 datos.
- Ya con esta cantidad, podemos obtener nuestro histograma final, que es para lo que usamos este paquete.

```
> out <- metrop(out, nbatch = 10000)
> out$accept

[1] 0.2339

> plot(out$batch[, 1])
> hist(out$batch[, 1])
```

- En sí, MCMC es complicado pero se los mencioné ya que Ziheng Yang les dio una plática donde usa el MCMC.

Histograma final



Ligando a R con MySQL

- Como nos lo pidieron los de la 4ta y 5ta, Osam y yo averiguamos sobre como enlazar R con MySQL.
- La solución es usar el paquete RMySQL que ya está instalado en los servidores. Sin embargo, este no es sencillo de instalar en una laptop con Windows[4].
- Con este paquete es fácil hacer consultas o simplemente bajar toda una tabla a R.
- La velocidad con la que leen una tabla dependerá si están en red local con el servidor o si están desde su casa.

[4] Pondré los pasos a seguir en el foro

Uso básico

- En fin, podemos explorar ciertas funcionalidades del paquete con el siguiente comando o en su guía disponible en CRAN.

  ```
  > help(package = "RMySQL")
  ```

- Es importante que recuerden cerrar toda sesión de MySQL que abran con R.

dbConnect

- Lo más básico es aprender a conectarse a su base de datos. Esto lo hacen con la función dbConnect y con sus argumentos `user`, `password`, `dbname` y `host`. Por ejemplo:

```
> con <- dbConnect(MySQL(), user = "lcollado",
+      password = "LOL", dbname = "BPdB",
+      host = "kabah.lcg.unam.mx")
```

- Para desconectarse, simplemente usen dbDisconnect:

```
> dbDisconnect(con)
```

Pasamos datos a R

- Una vez conectados, pueden usar funciones como dbListTables y dbReadTable para ver sus tablas o descargarlas a R.

```
> dbListTables(con)
 [1] "coinfection"
 [2] "coinfection_host_link"
 [3] "coinfection_phage_link"
 [4] "fields"
 [5] "gene_ref_link"
 [6] "gene_ref_link2"
 [7] "genome_phage_ref"
 [8] "host"
 [9] "host_phage_link"
[10] "host_taxonomy"
```

Pasamos datos a R


```
[11] "image"
[12] "p_family"
[13] "p_genus"
[14] "p_order"
[15] "phage"
[16] "phage_gene2"
[17] "promoter"
[18] "ref2"
[19] "reference_type2"
[20] "transcription_unit"
[21] "transcription_unit_gene_link"
```

```
> d <- dbReadTable(con, "phage")
```

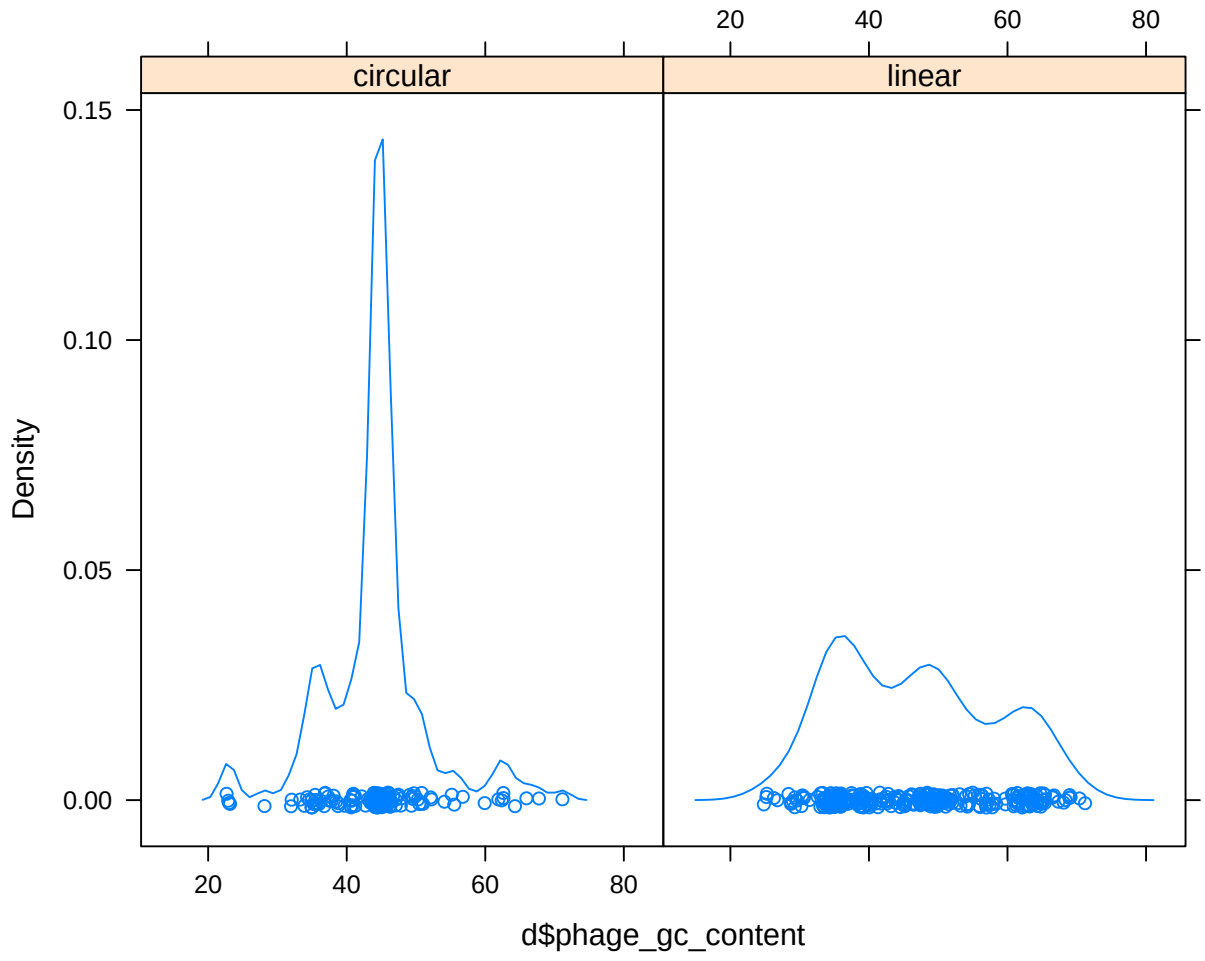
- Cual es la clase del objeto d?

Usen sus datos!

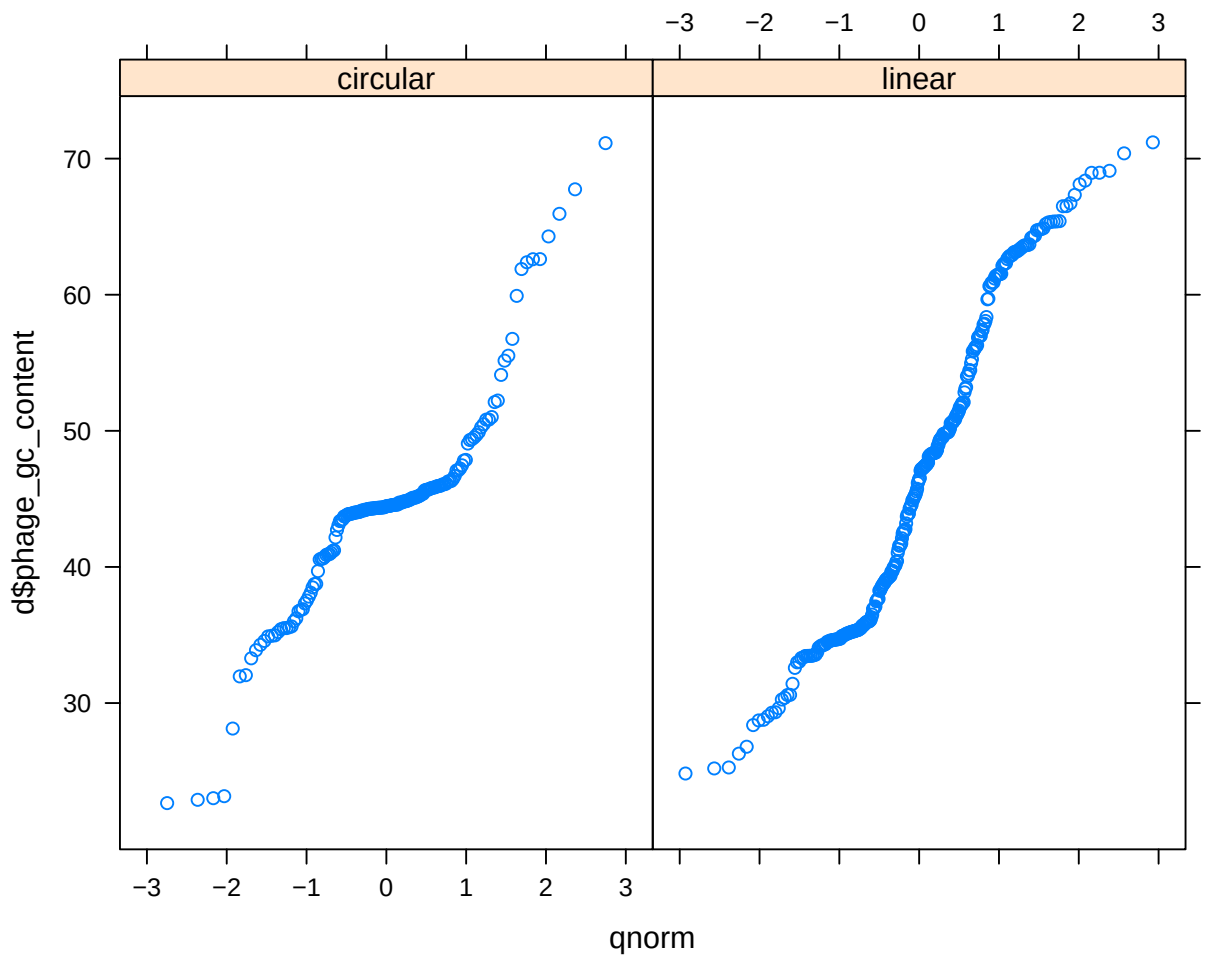
- `d` es clase `data.frame`. Estos son fáciles de manipular en `R` y ya los hemos manejado muchas veces :)
- Ya con sus datos en `R` pueden utilizar herramientas que vimos en las anteriores clases para explorarlos.
- Por ejemplo, usemos a `lattice`

```
> library(lattice)
> table(factor(d$phage_topology))

circular   linear
     166      293

> densityplot(~d$phage_gc_content |
+     factor(d$phage_topology))
> qqmath(~d$phage_gc_content | factor(d$phage_topo
```

Explorando nuestros datos:



QQplot: tenemos datos normales?



Nos desconectamos

- Otra gráfica que pudimos hacer es:

```
> bwplot(d$phage_gc_content ~ factor(d$phage_topo
```

- En fin, hay que desconectarse!

```
> dbDisconnect(con)
[1] TRUE
```

Fin de la clase

- Como se dieron cuenta hoy, el universo de `Bioconductor` es amplio y complicado. Hay paquetes sencillos como el `GeneR` y otros mucho más avanzados com `Biostrings` y `Shortreads` que son más para manejar datos de `Solexa`.
- De tarea ya no les voy a dejar ejercicios puntuales, pues la idea es que empiezen a trabajar en la parte de `R` de su proyecto.
- Esta vez tienen que entregar vía la página de Cursos un solo código por equipo. En dicho código tienen que hacer 3 gráficas, o pruebas de estadística, etc. y poner en comentarios las conclusiones que saquen[5]. Evidentemente tienen que usar datos de su proyecto.
- Suerte!

[5] No tienen que ser conclusiones complicadas