Introduction: Course Aims

- Under the hood of R
  - R essentials, and programming skills
  - Examples implement common genetic analyses, efficiently

- Using R for more sophisticated analyses
  - Using tools, packages produced by others
  - Bioconductor-based bioinformatics

This is not a first course in R – unless you are an experienced programmer. Basic use of R (reading in data, simple manipulation of data, supplying arguments to functions) is assumed. We will review some topics; further background is available here; http://faculty.washington.edu/kenrice/bigr
Introduction: About Prof Lumley

- Professor, Univ of Auckland
- R Core developer
- Genetic/Genomic research in Cardiovascular Epidemiology
- Sings bass
Introduction: About Prof Rice

• Associate Prof, UW Biostat

• Not an author, but a user (and a teacher)

• Genetic/Genomic research in Cardiovascular Epidemiology

• Sang bass (last week)

... and you?

(who are you, what area of genetics, what are you looking for from the course)
Introduction: Course structure

10 sessions over 2.5 days

- Day 1; Review of R, graphics, data-wrangling
- Day 2; Programming (loops), big data, bioconductor

Web page: http://faculty.washington.edu/kenrice/bigr/
Introduction: Session structure

- 45 mins teaching (questions welcome – please interrupt!)

- 30 mins hands-on

- 15 mins summary, discussion

Keys are posted after each session – though there is more than one correct answer
1. Review of R

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Auckland, November 2013
What is R?

R is a ‘programming environment for statistics and graphics’

The base R has fewer prepackaged statistics procedure than SPSS or SAS, but it is much easier to extend with new procedures.

There are about 3500 published extension packages for R, many aimed at genetics and genomics research.
Using R

R is a free implementation of S, for which John Chambers won the ACM Software Systems award.

For the S system, which has forever altered how people analyze, visualize, and manipulate data.

The downside is that using R effectively may require changing how you analyze, visualize, and manipulate data.

R is a command-line system, not a point-and-click system.
A calculator

> 2+2
[1] 4
> 1536/317000
[1] 0.004845426
> exp(pi)-pi
[1] 19.9991
> x <- 3
> y <- 2
> x+y
[1] 5
> ls()
[1] "x" "y"
> round(pi, 6)
[1] 3.141593
> round(pi,
+ 6)
[1] 3.141593
Scripts

For longer analyses (and for this course), it's better to type code into a script and then run it. In base R;

- Windows: File | New script, CTRL-R to run lines or regions
- Mac: File | New Document, command-return to run
- Some other text editors offer this: Emacs, Tinn, WinEDT, JGR, Eclipse. RStudio has a script editor and data viewer

Please note we strongly recommend doing the exercises in pairs
Scripts

Default interface (and how to use it);
Reading data

Ability to read in data is assumed. But to illustrate commands, and show some less-standard approaches, we’ll review reading in:

- Text files
- Other statistics packages datasets
- Web pages

Much more information is in the Data Import/Export manual.
Reading text data

The easiest format has variable names in the first row

case  id  gender  deg  yrdeg  field  startyr  year   rank   admin
 1    1    F     Other  92    Other  95    95    Assist   0
 2    2    M     Other  91    Other  94    94    Assist   0
 3    2    M     Other  91    Other  94    95    Assist   0
 4    4    M     PhD   96    Other  95    95    Assist   0

and fields separated by spaces. In R, use

```r
code
salary <- read.table("salary.txt", header=TRUE)
```

to read the data from the file `salary.txt` in the current working directory into the data frame `salary`. 
Syntax notes

• Spaces in commands don’t matter (except for readability), but Capitalisation Does Matter.

• TRUE (and FALSE) are logical constants

• Unlike many systems, R does not distinguish between commands that do something and commands that compute a value. Everything is a function: i.e. it returns a value.

• Arguments to functions can be named (header=TRUE) or unnamed ("salary.txt")

• A whole data set (called a data frame) is stored in a variable (salary), so more than one dataset can be available at the same time.
Did it work?

The `head()` function shows the first few lines of the data frame

```r
> head(salary)
case id gender deg yrdeg field startyr year rank admin salary
1 1 1 F Other 92 Other 95 95 Assist 0 6684
2 2 2 M Other 91 Other 94 94 Assist 0 4743
3 3 2 M Other 91 Other 94 95 Assist 0 4881
4 4 4 M PhD 96 Other 95 95 Assist 0 4231
5 5 6 M PhD 66 Other 91 91 Full 1 11182
6 6 6 M PhD 66 Other 91 92 Full 1 11507
```

It *should* look like this!

The `View()` command similarly shows your data frame, in a manageable way.
Where’s my file?

- Find out your current directory with `getwd()`

- Change directory with the menus

- Or with `setwd("I:/like/this/directory/better")`

- `file.choose()` pops up a dialog for choosing a file and returns the name, so

  ```r
  salary <- read.table(file.choose(), header=TRUE)
  ```
Reading text data

Sometimes columns are separated by commas (or tabs)

Ozone,Solar.R,Wind,Temp,Month,Day
41,190,7.4,67,5,1
36,118,8,72,5,2
12,149,12.6,74,5,3
18,313,11.5,62,5,4
NA,NA,14.3,56,5,5

Use

ozone <- read.table("ozone.csv", header=TRUE, sep="",""

or

ozone <- read.csv("ozone.csv")
Syntax notes

- Forgetting `header=TRUE` in `read.table()` is bad (try it!)

- Functions can have optional arguments (*sep* wasn’t used the first time). Use `help("read.table")` or `?read.table` for a complete description of the function and all its arguments.

- There’s more than one way to do it.

- **NA** is the code for missing data. Think of it as “Don’t Know”. R handles it sensibly in computations: eg `1+NA`, `NA & FALSE`, `NA & TRUE`. You cannot test `temp==NA` (Is temperature equal to some number I don’t know?), so there is a function `is.na()`.
### Reading text data

Sometime the variable names aren’t included

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.2</td>
<td>115</td>
<td>90</td>
<td>1</td>
<td>3</td>
<td>68</td>
<td>42</td>
<td>yes</td>
</tr>
<tr>
<td>2</td>
<td>0.7</td>
<td>193</td>
<td>90</td>
<td>3</td>
<td>1</td>
<td>61</td>
<td>48</td>
<td>yes</td>
</tr>
<tr>
<td>3</td>
<td>0.2</td>
<td>58</td>
<td>90</td>
<td>1</td>
<td>3</td>
<td>63</td>
<td>40</td>
<td>yes</td>
</tr>
<tr>
<td>4</td>
<td>0.2</td>
<td>5</td>
<td>80</td>
<td>2</td>
<td>3</td>
<td>65</td>
<td>75</td>
<td>yes</td>
</tr>
<tr>
<td>5</td>
<td>0.2</td>
<td>8.5</td>
<td>90</td>
<td>1</td>
<td>2</td>
<td>64</td>
<td>30</td>
<td>yes</td>
</tr>
</tbody>
</table>

and you have to supply them

```r
psa <- read.table("psa.txt", col.names=c("ptid","nadirpsa", "pretxpsa", "ps","bss","grade","age", "obstime","inrem"))
```

or

```r
psa <- read.table("psa.txt")
names(psa) <- c("ptid","nadirpsa","pretxpsa", "ps", "bss","grade","age","obstime","inrem")
```
Syntax notes

- Assigning a single vector (or anything else) to a variable uses the same syntax as assigning a whole data frame.

- `c()` is a function that makes a single vector from its arguments.

- `names()` is a function that accesses the variable names of a data frame.

- Some functions, such as `names()`, can be used on the LHS of an assignment.
library("foreign")
stata <- read.dta("salary.dta")
spss <- read.spss("salary.sav", to.data.frame=TRUE)
sasxport <- read.xport("salary.xpt")
epiinfo <- read.epiinfo("salary.rec")

Notes:

- Many functions in R live in optional packages. The library() function lists packages, shows help, or loads packages from the package library.

- The foreign package is in the standard distribution. It handles import and export of data. Thousands of extra packages are available at http://cran.r-project.org.
The web

Files for `read.table()` can live on the web

```r
fl2000<-read.table("http://faculty.washington.edu/tlumley/data/FLvote.dat", header=TRUE)
```

It’s also possible to read from more complex web services (such as the genome databases)
Operating on data

We assume you know how to use the $ sign, to indicate columns of interest in a dataset, e.g. *antibiotics$duration* means the variable *duration* in the data frame *antibiotics*.

```r
## This is a comment
## Convert temperature to real degrees
antibiotics$tempC <- (antibiotics$temp-32)*5/9
## display mean, quartiles of all variables
summary(antibiotics)
```
Subsets

Everything in R is a vector (but some have only one element). There are several ways to use `[]` to extract subsets

```r
## First element
antibiotics$temp[1]
## All but first element
antibiotics$temp[-1]
## Elements 5 through 10
antibiotics$temp[5:10]
## Elements 5 and 7
antibiotics$temp[c(5,7)]
## People who received antibiotics (note ==)
antibiotics$temp[ antibiotics$antib==1 ]
## or
with(antibiotics, temp[antib==1])
```
Notes

• Positive indices select elements, negative indices drop elements

• 5:10 is the sequence from 5 to 10

• You need == to test equality, not just =

• with() temporarily sets up a data frame as the default place to look up variables.
More subsets

For data frames you need two indices – naming rows and columns can also be useful;

```r
## First row
antibiotics[1,]
## Second column
antibiotics[,2]
## Some rows and columns
antibiotics[3:7, 2:4]
## Columns by name
antibiotics[, c("id","temp","wbc")]
## People who received antibiotics
antibiotics[antibiotics$antib==1, ]
## Put this subset into a new data frame
yes <- antibiotics[antibiotics$antib==1,]
```
Computations

mean(antibiotics$temp)
median(antibiotics$temp)
var(antibiotics$temp)
sd(antibiotics$temp)
mean(yes$temp)
mean(antibiotics$temp[, antib==1])
with(antibiotics, mean(temp[sex==2]))
toohot <- with(antibiotics, temp>99)
mean(toohot)
Factors

Factors represent categorical variables. You can't do mathematical operations on them (except for `==`)

```r
> table(salary$rank, salary$field)

         Arts Other Prof
    Assist  668   2626   754
    Assoc  1229   4229  1071
    Full   942   6285  1984

> antibiotics$antib <- factor(antibiotics$antib, labels=c("Yes","No"))
> antibiotics$agegp <- cut(antibiotics$age, c(0,18,65,100))
> table(antibiotics$agegp)

 (0,18] (18,65] (65,100]
      2     19       4
```
Help!

- `?fn` or `help(fn)` for help on `fn`

- `help.search("topic")` for help pages related to "topic"

- `apropos("tab")` for functions whose names contain "tab"

- `RSiteSearch("FDR")` to search the R Project website (requires internet access)
2. Graphics in R

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Auckland, November 2013
Graphics

R can produce graphics in many formats, including:

- on screen
- PDF files for \texttt{\LaTeX} or emailing to people
- PNG or JPEG bitmap formats for web pages (or on non-Windows platforms to produce graphics for MS Office). PNG is also useful for graphs of large data sets.
- On Windows, metafiles for Word, Powerpoint, and similar programs
Setup

Graphs should usually be designed on the screen and then may be replotted on eg a PDF file (for Word/Powerpoint you can just copy and paste)

For printed graphs, you will get better results if you design the graph at the size it will end up, eg:

```plaintext
## on Windows
windows(height=4,width=6)
## on Unix
x11(height=4,width=6)
```

Word or \LaTeX can rescale the graph, but when the graph gets smaller, so do the axis labels...
The Topography of Maunga Whau
BETTER SCALING

CREATED AT 6 × 5 INCHES

THE TOPOGRAPHY OF MAUNGA WHAU

HEIGHT (METERS)

METERS WEST

METERS NORTH

FILLED CONTOUR( ) FROM R VERSION 2.5.1 (2007−06−27)
The real thing

After you have the right commands to draw the graph you can produce it in another format: eg

```r
## start a PDF file
pdf("picture.pdf",height=4,width=6)
## your drawing commands here
...
### close the PDF file
dev.off()
```

You may find `dir()`, `getwd()` and `setwd()` helpful
Drawing

Usually, use `plot()` to create a graph and then `lines()`, `points()`, `legend()`, `text()`, `rect()`, `segments()`, `symbols()`, `arrows()` and other commands to annotate it. There is no ‘erase’ function, so keep your commands in a script.

`plot()` is a **generic function**: it does appropriate things for different types of input.

```r
## scatterplot
plot(salary$year, salary$salary)

## boxplot
plot(salary$rank, salary$salary)

## stacked barplot
plot(salary$field, salary$rank)
```

and others for other types of input.
Formula interface

The \texttt{plot()} command can also be used this way;

\begin{verbatim}
plot(salary~rank, data=salary)
\end{verbatim}

where we introduce the \texttt{formula} system, that is also used for regression models. Here, think of $Y \sim X$.

The variables in the formula are automatically looked up in the \texttt{data=} argument.
Designing graphs

Two important aspects of designing a graph

- It should have something to say

- It should be legible

Having something to say is your problem; software can help with legibility.
Designing graphs

Important points

- Axes need labels (with units, large enough to read)

- Color can be very helpful (but not if the graph is going to be printed in black and white).

- Different line or point styles usually should be labelled.

- Points plotted on top of each other won’t be seen

After these are satisfied, it can’t hurt to have the graph look nice.
First we set up the data – in this case, a built-in dataset, containing daily ozone concentrations in New York, summer 1973

data(airquality)
names(airquality)
airquality$date <- with(airquality, ISOdate(1973, Month, Day))

All these graphs were designed at 4in×6in and stored as PDF files;
Options

```r
plot(Ozone ~ date, data = airquality)
```
Options

plot(Ozone ~ date, data = airquality, type = "l")
plot(Ozone~date, data=airquality,type="h")
plot(Ozone~date, data=airquality,type="n")
Options

Commands to do a more complex plot;
bad <- ifelse(airquality$Ozone>=90, "orange", "forestgreen")
plot(Ozone~date, data=airquality, type="h", col=bad)
abline(h=90, lty=2, col="red")
Notes

- `type=` controls how data are plotted. `type="n"` is not as useless as it looks: it can set up a plot for latter additions.
- Colors can be specified by name (the `colors()` function gives all the names), by red/green/blue values ("#rrggbb" with six base-sixteen digits) or by position (1:8) in the standard palette of 8 colors.
- `abline` draws a single straight line on a plot
- `ifelse()` selects between two vectors based on a logical variable.
- `lty` specifies the line type: 1 is solid, 2 is dashed, 3 is dotted, then it gets more complicated. See `?par`, then search for `lty`
Adding to a plot

More example commands

data(cars)
plot(dist~speed,data=cars)
with(cars, lines(lowess(speed, dist), col="tomato", lwd=2))

plot(dist~speed,data=cars, log="xy")
with(cars, lines(lowess(speed, dist), col="tomato", lwd=2))
with(cars, lines(supsmu(speed, dist), col="purple", lwd=2))

legend("bottomright", legend=c("lowess","supersmoother"), bty="n", lwd=2, col=c("tomato","purple"))
Adding to a plot
Adding to a plot
Adding to a plot
Notes

- `lines()` adds lines to an existing plot (and `points()` adds points)

- `lowess()` and `supsmu()` are scatterplot smoothers. They calculate smooth curves that fit the relationship between $y$ and $x$ locally. Their output has attributes $x$ and $y$, that generic function `lines()` can cope with

- `log="xy"` asks for both axes to be logarithm (`log="x"` would just be the x-axis)

- `legend()` adds a legend
Conditioning plots

Ozone is a secondary pollutant, it is produced from organic compounds and atmospheric oxygen in reactions catalyzed by nitrogen oxides and powered by sunlight.

However, looking at ozone concentrations in NY in summer we see a non-monotone relationship with sunlight.
Conditioning plots
Conditioning plots

Here we draw a scatterplot of Ozone vs Solar.R for various subranges of Temp and Wind. For more examples like this, see the commands in the lattice package.

data(airquality)
coplot(Ozone ~ Solar.R | Temp * Wind, number = c(4, 4),
       data = airquality,
       pch = 21, col = "goldenrod", bg = "goldenrod")
Conditioning plots

Given : Temp

Given : Wind

Solar.R

Ozone

60 70 80 90

5 10 15 20

2.26
Conditioning plots

- A 4-D relationship is illustrated; the Ozone/sunlight relationship changes in strength depending on both the Temperature and Wind.

- The vertical bar | is statistician-speak for ‘conditioning on’ (nb this is different to use of |’s meaning as Boolean ‘OR’).

- The horizontal/vertical ‘shingles’ tell you which data appear in which plot. The overlap can be set to zero, if preferred.

- coplot()’s default layout is a bit odd; try setting rows, columns to different values.

- For more plotting commands that support conditioning, see library(help="lattice")
Conditioning plots: general

Functions in the lattice package allow various forms of conditioning plot – here for histograms;

```r
library("lattice")
airquality$date <- with(airquality, ISOdate(1973,Month,Day))
airquality$month <- strftime(airquality$date, "%m %y")
histogram(~Ozone|month, data=airquality)
```

![Histograms for different months](image)
Conditioning plots: general

Some other plots that can be conditioned;

- `xyplot()` for scatterplots – more flexible version of `coplot()`
- `barchart()` for boxplots
- `dotplot()` and `stripchart()`
- `qq()` and `qqmath()`, e.g. comparing to $N(0, 1)$
- `levelplot()` – heatmaps, see also `image()`
- `contourplot()` – see also `contour()`
- `cloud()` and `wireframe()`, for fake 3D

For still others see `?Lattice after loading the lattice package ...` or `demo(lattice)`.

For fine control, these functions all permit `panel` arguments, i.e. you can supply a function implemented in each subplot. `?panel.abline` describes some pre-written versions.
persp() and friends

Should you ever need them; (and you may not!)

with(trees,

   per <- persp(x=1:2, y=1:2, z=matrix(NA,2,2), theta=35,
                  xlim=range(Girth), ylim=range(Height), zlim=range(Volume),
                  xlab="Girth", ylab="Height", zlab="Volume", axes=TRUE, ticktype="detailed")
   points(trans3d(x=Girth, y=Height, z=Volume, per), col="red", pch=19)
)

persp() is okay for wireframes, otherwise try cloud() first.
Toys: Mathematical annotation

An expression can be specified in R for any text in a graph (help(plotmath) for details). Here we annotate a figure drawn with `polygon()`.

```r
x<-seq(-10,10,length=400)
y1<-dnorm(x)
y2<-dnorm(x,m=3)
par(mar=c(5,4,2,1))
plot(x,y2,xlim=c(-3,8),type="n",
     xlab=quote(Z==frac(mu[1]-mu[2],sigma/sqrt(n))),
     ylab="Density")
polygon(c(1.96,1.96,x[240:400],10),
     c(0,dnorm(1.96,m=3),y2[240:400],0),
     col="grey80",lty=0)
lines(x,y2)
lines(x,y1)
polygon(c(-1.96,-1.96,x[161:1],-10),
     c(0,dnorm(-1.96,m=0),y1[161:1],0),
     col="grey30",lty=0)
polygon(c(1.96,1.96,x[240:400],10),
     c(0,dnorm(1.96,m=0),y1[240:400],0),
     col="grey30")
```

2.31
Toys: Mathematical annotation

legend(4.2,.4,fill=c("grey80","grey30"),
       legend=expression(P(abs(Z)>1.96,H[1])==0.85,
                        P(abs(Z)>1.96,H[0])==0.05),bty="n")
text(0,.2,quote(H[0]:~~mu[1]==mu[2]))
Toys: Mathematical annotation

\[ Z = \frac{\mu_1 - \mu_2}{\sigma/\sqrt{n}} \]

\[ P(Z > 1.96, H_1) = 0.85 \]

\[ P(Z > 1.96, H_0) = 0.05 \]

\[ H_0 : \mu_1 = \mu_2 \]

\[ H_1 : \mu_1 = \mu_2 + \delta \]
Toys: Maps

> library("maps")
> map('county', 'washington', fill = TRUE,

   col = grey(sqrt(wa[,10]/(wa[,1])))

> title(main="Proportion Hispanic")
Toys: Maps

Proportion Hispanic
Even conditioned maps!

Health insurance coverage by age and state

<table>
<thead>
<tr>
<th></th>
<th>50–65</th>
<th>65+</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;35</td>
<td></td>
<td></td>
</tr>
<tr>
<td>35−50</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50−65</td>
<td></td>
<td></td>
</tr>
<tr>
<td>65+</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

![Maps showing health insurance coverage by age and state](image)
3. More Advanced Graphics

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Auckland, November 2013
Outline

- Colour and pre-attentive perception: facts about graphics
- Too many variables: parallel coordinates, transparency
- Too many dimensions: hexagonal binning, transparency
Colour coding

‘Simple’ plots involve two-dimensional data, which we measure on the $x$ and $y$ axes.

For higher-dimensions, some traditional approaches are;

- Different colors for e.g. men, women (col)
- Different-shaped symbols (pch), or different sizes (cex)

For $\leq 100$’s of data points, modest use of these is fine. But your eye is not good at concentrating e.g. just on the purple points, in a fully Technicolor plot;
Which point is different?
Which point is different?
Which point is different?
Which point is different?
Which point is different?
Pre-attentive perception

Some differences are processed by the brain before you get to see the image: pre-attentive perception.

Important because

- It’s easier to see things
- You can look at just one subset of the points and see patterns
- Like colour-blindness, illustrates that there are facts about graphics, not just artistic taste
Preattentive perception
Color schemes

Color choice is best left to experts, or people with taste.

http://www.colorbrewer.org has color schemes designed for the National Cancer Atlas, also in package RColorBrewer

colorspace package has color schemes based on straight lines in a perceptually-based color space (rather than RGB).

dichromat package attempts to show the impact of red:green color blindness on your R color schemes.

[Code for examples is in file colorpalettes.R on course website]
Color choice

(nb B&W printed copies of this slide may not be helpful!)
Color blindness

(nb B&W printed copies of this slide may not be helpful!)
Larger data

For large(ish) data, ‘overlap’ is a fundamental problem...

(California Academic Performance Index on 6194 schools)
Larger data

... which remains, when we color-code.

Colors denote Elementary, Middle & High Schools
Larger data

With three dimensions + color-codes, this can happen;

(R does have `persp()`, for occasional use)
Parallel Coordinate Plots

For even higher-dimensional data, scatterplots can not provide adequate summaries. For data where the dimensions can be ordered, the **parallel co-ordinates plot** is useful;

![Leading Principal Components, n=279, 10000 SNPs](image)
Parallel Coordinate Plots

- Each multi-dimensional data point (i.e. each person) is represented by a line – not a point
- `parcoord()` in the MASS package is one simple implementation – writing your own version is not a big job
- Coloring the lines also helps (example later)
- Scaling of axes, and their vertical positions are arbitrary
- Doing ‘Principal Components Analysis’ is just choosing axes for your data so that their variance is maximized on axis 1, then axis 2, ...

3.17
Parallel Coordinate Plots

A `pairs()` plot of the same thing; (nasty!)
Parallel Coordinate Plots

The pin cushion data++: colors indicate self-report ancestry
The colors in the last examples were transparent. As well as specifying e.g. `col=2` or `col="red"`, you can also specify

`col="#FF000033"`

– coded as RRGGBB in hexadecimal, with transparency 33 (also hexadecimal). This is a ‘pale’ red – $33/FF \approx 20\%.$

Get from color names to RGB with `col2rgb()`, and from base 10 to base 16 using `format(as.hexmode(11), width=2)`
Transparency

An example; (also shows other graphics commands)

curve(0.8*dnorm(x), 0, 6, col="blue", ylab="density", xlab="z")
curve(0.2*dnorm(x,3,2), 0, 6, col="red", add=T)

xvals <- seq(1, 6, l=101)
polygon(
c(xvals,6,1), c(0.8*dnorm(xvals), 0,0),
density=NA, col="#0000FF80" ) # transparent blue
polygon(
c(xvals,6,1), c(0.2*dnorm(xvals,3,2), 0,0),
density=NA, col="#FF000080" ) # transparent red

legend("topright", bty="n", lty=1, col=c("blue","red"),
c("80% null: N(0,1)", "20% signal: N(3,2)"))
axis(3, at=qnorm(c(0.25, 0.5*10^(-1:-7)), lower=F), c(0.5, 10^(-1:-7)) )
mtext(side=3, line=2, "unadjusted p")

text(2.2, 0.07, adj=c(0,1), paste("FDR beyond 1 = ",
round(0.8*pnorm(1,lower=F)/(0.8*pnorm(1,lower=F) + 0.2*pnorm(1,3,2,lower=F)),3))))
Transparency

Here's the output;

![Graph showing the z-density of two distributions: 80% null: N(0,1) and 20% signal: N(3,2). The FDR beyond 1 is 0.43.](image-url)
Hexagonal binning

Using transparent plotting symbols is a quick-and-dirty way to adapt scatterplots for use with large datasets.

A better method is ‘hexagonal binning’; this is a 2D analog of a histogram – where you would count the number of data in one area, and then draw a bar with height proportional to that count.
Hexagonal binning

Binning in two dimensions;
Hexagonal binning

Binning in two dimensions;
Hexagonal binning

Binning in two dimensions;
Hexagonal binning

Binning in two dimensions;
Hexagonal binning

The `hexbin()` package does all the bin construction, and counting. It has a `plot` method for its `hexbin` objects;

```r
install.packages(c("hexbin","survey"))
library("hexbin")
library("survey") # for apipop data frame

with(apipop, plot(hexbin(api99, api00), style="centroids"))
```
Hexagonal binning
Hexagonal binning

Hexbin is used when you don't *really* care about the exact location of every single point

- Singleton points are plotted ‘as usual’; you do (perhaps) care about them

- `hexbin` centers the ‘ink’ at the cell data’s ‘center of gravity’

- `style="centroids"` gives the center-of-gravity version; the default style is `colorscale` – usually grayscale. See `?gplot.hexagons` for more options
Hexagonal binning

For keen people: the `hexbin` package doesn’t use the standard R graphics plotting devices; instead, it operates through the `Grid` system (in the `grid` package) which defines rectangular regions on a graphics device; these `viewport` regions can have a number of coordinate systems. To add lines to a hexbin plot, the options are:

- Use `hexVP.abline()` to add these directly
- Move everything into ‘standard’ graphics – not `Grid` graphics (see `?Grid`). The `Grid` system lets you alter graphics after plotting them
- Write your own plot method for hexbin objects, with standard R graphics commands
- Make do with `hexBinning()` in the `fMultivar` package
Hexagonal binning

An example; color-coded lines of best fit, by school type;

```r
lm.e <- coef(lm(api00~api99, data=apipop, subset=stype=="E"))
lm.m <- coef(lm(api00~api99, data=apipop, subset=stype=="M"))
lm.h <- coef(lm(api00~api99, data=apipop, subset=stype=="H"))

hexVP.abline(vp1$plot.vp, lm.e[1], lm.e[2], col="coral")
```
Large data: multiple groups

For showing multiple groups, a scatterplot smoother or perhaps boxplots or conditioning may be better.

```r
boxplot(api00~cut(api99,(3:10)*100), data=apipop)
par(las=1)
par(mar=c(5.1,10.1,2.1,2.1))
boxplot(api00~interaction(stype,
        cut(api99,(3:10)*100)),
        data=apipop, horizontal=TRUE,col=1:3)
```

- **cut()** turns a variable into a factor by cutting it at the specified points.

- **par(mar=)** sets the margins around the plot. We need a large left margin for the labels.
Large data: multiple groups
Large data: multiple groups
Smoothers

We don't plot the data at all, just means (could also plot quantiles with quantreg package)

```r
plot(api00~api99,data=apipop,type="n")
with(subset(apipop, stype=="E"),
    lines(lowess(api99, api00), col="tomato"))
with(subset(apipop, stype=="H"),
    lines(lowess(api99, api00), col="forestgreen"))
with(subset(apipop, stype=="M"),
    lines(lowess(api99, api00), col="purple"))
```

Note the use of `type="n"`

`subset()` returns a subset of a data frame.
Smoothers
Conditioning

`hexbinplot(api00~api99|stype, data=apipop, style="centroid")`
4. Data manipulation

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Auckland, November 2013
Merging and matching

The data for an analysis often do not come in a single file. Combining multiple files is necessary.

If two data sets have the same individuals in the same order, they can simply be pasted together side by side.

```r
## CHS baseline data
baseline <- read.spss("I:/DISTRIB/BASEBOTH.SAV", to.data.frame=TRUE)
## Events data (eg death, heart attack, ...)
events <- read.spss("I:/SAVEFILES/EVSUM04.SAV", to.data.frame=TRUE)

if (!all(baseline$IDNO==events$IDNO)) {
  stop("PANIC: They don’t match!"")
} else {
  alldata <- cbind(baseline, events[,c("TTODTH","DEATH", "TTOMI","INCMI")])
}
```
Merging: order

The data might need to be sorted first

```r
index1 <- order(baseline$IDNO)
baseline <- baseline[index1,]
index2 <- order(events$IDNO)
events <- events[index2,]
if (!all(baseline$IDNO==events$IDNO)) {
  stop("PANIC: They still don’t match!"")
} else {
  alldata <- cbind(baseline, events[,c("TTODTH","DEATH",
                                     "TTOMI","INCMI")])
}
```

Note that `order(baseline$IDNO)` gives a subset of row numbers containing all the rows but in a different (increasing) order.
Merging: merge

Or there might be different rows in the two data sets

- Some people are missing from one or other data set (eg baseline and year 5 visits)

- Some people have multiple records in one data set (eg baseline data and all hospitalisations)

The `merge()` function can do an database outer join, giving a data set that has all the possible matches between a row in one and a row in the other
Merging: merge

combined <- merge(baseline, hospvisits, by="IDNO", all=TRUE)

- by=IDNO says that the IDNO variable indicates individuals who should be matched.

- all=TRUE says that even people with no records in the hospvisits data set should be kept in the merged version.
How does it work: match

You could imagine a dumb algorithm for merging

```
for(row in firstdataset){
    for(otherrow in seconddataset){
        if (row$IDNO==otherrow$IDNO)
            ##add the row to the result
    }
}
```

More efficiently, the `match` function gives indices to match one variable to another

```
> match(c("B","I","O","S","T","A","T"),LETTERS)
[1]  2  9 15 19 20  1 20
> letters[match(c("B","I","O","S","T","A","T"),LETTERS)]
[1] "b" "i" "o" "s" "t" "a" "t"
```
Reshaping

Sometimes data sets are the wrong shape. Data with multiple observations of similar quantities can be in long form (multiple records per person) or wide form (multiple variables per person).

Example: The SeattleSNPs genetic variation discovery resource supplies data in a format

```
SNP    sample  al1  al2
000095 D001  C T
000095 D002  T T
000095 D003  T T
```

so that data for a single person is broken across many lines. To convert this to one line per person
col.names=c("SNP","sample","allele1","allele2"))
> dim(data)
[1] 2303  4
> wideData<-reshape(data, direction="wide", idvar="sample",
timevar="SNP")
> dim(wideData)
[1] 47 99
> names(wideData)
[1] "sample"   "allele1.95" "allele2.95" "allele1.205"
[5] "allele2.205" "allele1.276" "allele2.276" "allele1.321"
[9] "allele2.321" "allele1.657" "allele2.657" "allele1.1086"
...
- **direction="wide"** says we are going from long to wide format.

- **idvar="sample"** says that **sample** identifies the rows in wide format.

- **timevar="SNP"** says that **SNP** identifies which rows go into the same column in wide form (for repeated measurements over time it would be the time variable).
Broken down by() age and sex

A common request for Table 1 or Table 2 in a medical paper is to compute means and standard deviations, percentages, or frequency tables of many variables broken down by groups (e.g., case/control status, age and sex, exposure,...).

That is, we need to apply a simple computation to subsets of the data, and apply it to many variables. One useful function is by(), another is tapply(), which is very similar (but harder to remember).
> by(airquality$Ozone, list(month=airquality$Month),
    mean, na.rm=TRUE)

month: 5
[1]  23.61538

month: 6
[1]  29.44444

month: 7
[1]  59.11538

month: 8
[1]  59.96154

month: 9
[1]  31.44828
The first argument is the variable to be analyzed.

The second argument is a list of variable defining subsets. In this case, a single variable, but we could do list(month=airquality$Month, toohot=airquality$Temp>85) to get a breakdown by month and temperature.

The third argument is the analysis function to use on each subset.

Any other arguments (na.rm=TRUE) are also given to the analysis function.

The result is really a vector (with a single grouping variable) or array (with multiple grouping variables). It prints differently.
**Confusing digression: str()**

How do we know it is an array? Because \texttt{str()} summarises the internal structure of a variable.

\begin{verbatim}
> a<- by(airquality$Ozone, list(month=airquality$Month,
   toohot=airquality$Temp>85),
   mean, na.rm=TRUE)

> str(a)
by [1:5, 1:2] 23.6 22.1 49.3 40.9 22.0 ...
- attr(*, "dimnames")=List of 2
  ..$ month : chr [1:5] "5" "6" "7" "8" ...
  ..$ toohot: chr [1:2] "FALSE" "TRUE"
- attr(*, "call")= language by.data.frame(data =
  as.data.frame(data), INDICES = INDICES,
  FUN = FUN, na.rm = TRUE)
- attr(*, "class")= chr "by"
\end{verbatim}
One function, many variables

There is a general function, `apply()` for doing something to rows or columns of a matrix (or slices of a higher-dimensional array).

```
> apply(psa[,1:8],2,mean,na.rm=TRUE)

       id    nadir     pretx      ps      bss
25.500000 16.360000  670.751163  80.833333  2.520833

       grade   age  obstime
2.146341 67.440000 28.460000
```

In this case there is a special, faster, function `colMeans`, but `apply()` can be used with other functions such as `sd()`, `IQR()`, `min()`,...
apply

- the first argument is an array or matrix or dataframe
- the second argument says which margins to keep (1=row, 2=column, ...), so 2 means that the result should keep the columns: apply the function to each column.
- the third argument is the analysis function
- any other arguments are given to the analysis function

There is a widespread belief that `apply()` is faster than a `for()` loop over the columns. This is a useful belief, since it encourages people to use `apply()`, but it is not true. (We’ll see `for()` loops later)
# New functions

Suppose you want the mean and standard deviation for each variable. One solution is to apply a new function;

```r
> apply(psa[,1:8], 2,
    function(x){c(mean=mean(x,na.rm=TRUE), stddev=sd(x,na.rm=TRUE))}
)

   id  nadir  pretx  ps  bss  grade
mean 25.50000 16.3600 670.7512 80.83333 2.520833 2.1463415
stddev 14.57738 39.2462 1287.6384 11.07678 0.6838434 0.7924953

   age  obstime
mean 67.440000 28.46000
stddev 5.771711 18.39056
```
New functions

function(x){ c(mean=mean(x, na.rm=TRUE), stddev=sd(x, na.rm=TRUE)) }

translates as: “If you give me a vector, which I will call \(x\), I will mean it and sd it and give you the results”

We could give this function a name and then refer to it by name

mean.and.sd <- function(x){ c(mean=mean(x, na.rm=TRUE),
stddev=sd(x, na.rm=TRUE))
}

apply(psa[,1:8], 2, mean.and.sd)

which would save typing if we used the function many times. The \{curly brackets\} are optional for a function with just one expression, but necessary for longer functions.
With our own functions, we can use `by()` more generally

```r
> by(psa[,1:8], list(remission=psa$inrem),
    function(subset){round( apply(subset, 2, mean.and.sd), 2)} )

remission: no

<table>
<thead>
<tr>
<th>id</th>
<th>nadir</th>
<th>pretx</th>
<th>ps</th>
<th>bss</th>
<th>grade</th>
<th>age</th>
<th>obstime</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>31.03</td>
<td>22.52</td>
<td>725.99</td>
<td>79.71</td>
<td>2.71</td>
<td>2.11</td>
<td>67.17</td>
</tr>
<tr>
<td>stddev</td>
<td>11.34</td>
<td>44.91</td>
<td>1362.34</td>
<td>10.29</td>
<td>0.52</td>
<td>0.83</td>
<td>5.62</td>
</tr>
</tbody>
</table>

remission: yes

<table>
<thead>
<tr>
<th>id</th>
<th>nadir</th>
<th>pretx</th>
<th>ps</th>
<th>bss</th>
<th>grade</th>
<th>age</th>
<th>obstime</th>
</tr>
</thead>
<tbody>
<tr>
<td>mean</td>
<td>11.29</td>
<td>0.53</td>
<td>488.45</td>
<td>83.57</td>
<td>2.07</td>
<td>2.23</td>
<td>68.14</td>
</tr>
<tr>
<td>stddev</td>
<td>12.36</td>
<td>0.74</td>
<td>1044.14</td>
<td>12.77</td>
<td>0.83</td>
<td>0.73</td>
<td>6.30</td>
</tr>
</tbody>
</table>
```
function(subset){  round(apply(subset, 2, mean.and.sd), 2) }

translates as “If you give me a data frame, which I will call subset, I will apply the mean.and.sd function to each variable, round to 2 decimal places, and give you the results”
5. Permutation Tests (i.e. replication)

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Auckland, November 2013
Overview

- Simulation
- Permutation tests
- Power (briefly)
- Smallest p-value across multiple models
- Cautionary notes, which we won’t get to
What is the expected value of the median of a sample, size \( n = 51 \), of independent data from \( \text{Exp}(1) \)?

What is its variance?
Simulation: easier than doing maths

The picture didn’t make it obvious? Here are the exact answers;

\[ E[\text{Median}_{51}] = \frac{2178178936539108674153}{3099044504245996706400} \]

\[ E[\text{Median}_{51}^2] = \frac{2467282316063667967459233232139257976801959}{4802038419648657749001278815379823900480000} \]

These are 0.70286 and 0.51380 to 5 d.p. – so the variance is

\[ 0.51380 - 0.70286^2 = 0.01978. \]

• Yes, there are ‘pretty’ answers here

• In general there aren’t – but the ‘expectation’ (\(E[...]\)) terms just mean averaging over lots of datasets – which is easy, with a computer

• We can get a good-enough answer very quickly
Simulation: easier than doing maths

We’ll write code that;

1. Generates samples of size $n = 51$ from $\text{Exp}(1)$

2. Calculates their median and returns this number

3. Replicate steps 1 and 2 many times, then work out the mean and variance of the stored numbers

Steps 1 and 2 are inside the curly brackets;

```r
set.seed(4)
many.medians <- replicate(10000, {
    mysample <- rexp(n=51, rate=1) # take a sample, size 51
    median(mysample) # calculate & output its median
})
```

The function `set.seed()` tells R where to start its random-number generator – this is important, as it means we can repeat the code and get the same answers. Choose any ‘seed’ you like.
Simulation: easier than doing maths

To finish, take mean & variance of our 10000 sample medians;

> mean(many.medians)
[1] 0.702171 # exact answer is 0.70286
> var(many.medians)
[1] 0.01955728 # exact answer is 0.01978

NB: for large-enough values of 10000, we could work basically anything about the sample median, with little extra work;
Simulation: easier than doing maths

Notes on the coding;

- Yes, you could write a `for()` loop. (See Session 6, or `?Control`) But this approach helps break down the job into manageable pieces – then *finally* deal with the looping

- This approach is also easier to setup – just create `many.medians` and easier to edit afterwards, e.g. changing the value of 10000. Using `for()` loops, it’s surprisingly easy to mess this up (more in Session 6)

- By default, the last object evaluated in a function is what it returns. Can also use `return()`

- We used `rexp()`, see also `rnorm()`, `rgamma()`, `rbinom`, `rpois()` etc etc
Example: permutation test

A classical statistical question: are the data we've observed *unexpected*, if there's nothing going on?

An example where we can answer this is the sleep data, from a small clinical trial;

- 10 subjects per group
- Groups receive different treatments, we record how many hours sleep they get, compared to baseline
- Mean extra hours sleep is higher in group 2 (2.33 hrs vs 0.75 hrs, so difference is 1.58 hrs)
**Example: permutation test**

What if there were nothing going on*, i.e. what if any differences in mean were just chance? If so, the data we saw would be just as likely as that obtained assigning the group labels *at random*;

* Formally, what if the *null hypothesis* of equal means held, in the population from which this data has been sampled?
Example: permutation test

To measure how unexpected our data is, we compute the red/green difference in means for many of these permutations, and see how the observed data compares.

```
orig.mean.diff <-
    mean(sleep$extra[group==2]) - mean(sleep$extra[group==1])
orig.mean.diff
```

```
set.seed(4)
many.mean.diff <- replicate(10000,
    group.shuffle <- sample(sleep$group)
    mean(sleep$extra[group.shuffle==2]) - mean(sleep$extra[group.shuffle==1])
)
```

- `sample()` returns a random shuffle of a vector

- The same calculation is made, for the original data and the shuffled versions; the difference in means is called the test statistic
Example: permutation test

How does original data (w/ mean diff=1.58) compare to these?

```r
> table(many.mean.diff>orig.mean.diff)
FALSE  TRUE
  9601   399
> mean(many.mean.diff>orig.mean.diff)
[1] 0.0399
> mean(abs(many.mean.diff)>abs(orig.mean.diff))
[1] 0.0789
```
Example: permutation test

- The proportion of sample in the RH tail is a (valid) $p$-value for a one-tailed test, where the alternative is that green $> \text{red}$. $p = 0.04$, here

- The proportion in both tails is the $p$-value for a two-tail test; $p = 0.079$

- There is some ‘Monte Carlo’ error in these $p$-values; roughly $\pm 0.004$ here, i.e. 2 decimal places in $p$. If that’s not good enough, use more permutations. (Here, could use all 184,756 – but in larger samples it’s not possible)

For a quicker but *somewhat* approximate version of this test:

```r
> t.test(extra~group, data=sleep)
Welch Two Sample t-test
data:  extra by group
t = -1.8608, df = 17.776, p-value = 0.07939
alternative hypothesis: true difference in means is not equal to 0
```

The $t$-test makes fewer assumptions than many people think!
How many permutations?

With 10000 permutations the smallest possible \( p \)-value is 0.0001, and the uncertainty near \( p = 0.05 \) is about \( \pm 0.4\% \).

If we have multiple testing we may need much more precision.

Using 100,000 permutations reduces the uncertainty near \( p = 0.05 \) to \( \pm 0.1\% \) and allows accurate \( p \)-values as small as 0.00001.

A useful strategy is to start with 1000 permutations and continue to larger numbers only if \( p \) is small enough to be interesting, eg \( p < 0.1 \).

Parallel computing of permutations is easy: just run \( R \) on multiple computers.
Example: power calculation

A reminder: statistical tests reject $H_0$ whenever $p(Y) < \alpha$; $\alpha$ is known as the size or level of the test; it is the probability of declaring a signal when none is present. (By convention, we almost always use $\alpha = 0.05$.)

The power of the test is the probability you reject $H_0$, (i.e. get $p(Y) < \alpha$) when a signal is truly present.

Note: probabilities are means of binary (0/1) variables.
Example: power calculation

To evaluate power, for a comparison of two groups – like the sleep study;

- Assume outcomes are Normal, means $\mu_1, \mu_2$, SDs $\sigma_1, \sigma_2$
- Sample sizes $n_1, n_2$ in each group
- Use unequal-variance two-sided $t$-test for analysis
Example: power calculation

Using `replicate()` to do the work;

```r
do.one <- function(n1, n2, mu1, mu2, s1, s2, alpha){
  y1 <- rnorm(n1, mu1, s1)
  y2 <- rnorm(n2, mu2, s2)
  t.test(y1, y2)$p.value < alpha # default is unequal variance
}
```

```r
set.seed(4)
bigB <- 10000
mean(replicate(bigB, do.one(20, 20, 0, 0.7, 1, 1.5, 0.05)))
```

- Mean here is 3895/10000 = 0.3895, i.e. about 40% power. Precision to multiple decimal places matters much less than earlier example, calculating $p$.
- For equal variances, can use `power.t.test()`, i.e. built-in maths-only version.
- This version is slower to compute – but much more flexible, e.g. regression-based analyses, multi-step analyses, any distribution/design you like.
Debugging

R error messages are sometimes hard to follow, when the error occurs in a low-level function. To see what happened after an error, `traceback()` reports the entire call stack, which is useful for seeing where things went wrong.

For example, in the permutation test example, suppose our outcome variable were actually a data frame with one column rather than a vector:

```r
> wrong.extra <- as.data.frame(sleep$extra) # easy mistake!
> many.mean.diff <- replicate(10000,{
+   group.shuffle <- sample(sleep$group)
+   mean(wrong.extra[group.shuffle==2]) - mean(wrong.extra[group.shuffle==1])
+ })
Error in ‘[.data.frame‘(wrong.extra, group.shuffle == 2) : undefined columns selected
```

We didn’t know we were calling ‘‘[.data.frame‘, so the message appears opaque and unhelpful.
Debugging

> traceback()
8: stop("undefined columns selected")
7: `\[.data.frame\`(wrong.extra, group.shuffle == 2)
6: wrong.extra[group.shuffle == 2]
5: mean(wrong.extra[group.shuffle == 2]) at #3
4: FUN(c(0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L,
3: lapply(X = X, FUN = FUN, ...)
2: sapply(integer(n), eval.parent(substitute(function(...)) expr),
       simplify = simplify)
1: replicate(10000, {
    group.shuffle <- sample(sleep$group)
    mean(wrong.extra[group.shuffle == 2]) - mean(wrong.extra[group.shuffle == 1])
  })

This mean the problem happens in our code
mean(wrong.extra[group.shuffle == 2]) and is a problem with computing wrong.extra[group.shuffle == 2].

We want to have a look at wrong.extra and group.shuffle...
Debugging

The post-mortem debugger lets you look inside the code where the error occurred;

```r
> options(error=recover)
> many.mean.diff <- replicate(10000,
+   group.shuffle <- sample(sleep$group)
+   mean(wrong.extra[group.shuffle==2]) - mean(wrong.extra[group.shuffle==1])
+ }

Error in ‘[.data.frame‘(wrong.extra, group.shuffle == 2) : undefined columns selected

Enter a frame number, or 0 to exit

1: replicate(10000, {
   group.shuffle <- sample(sleep$group)
   mean(wrong.ex
2: sapply(integer(n), eval.parent(substitute(function(...)) expr)), simplify =
3: lapply(X = X, FUN = FUN, ...)
4: FUN(c(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
5:  #3: mean(wrong.extra[group.shuffle == 2])
6: wrong.extra[group.shuffle == 2]
7: ‘[.data.frame‘(wrong.extra, group.shuffle == 2)

We want 6, the last step of ‘our’ code
Debugging

Called from: eval(substitute(browser(skipCalls = skip), list(skip = 7 - which)),
         envir = sys.frame(which))
Browse[1]> ls()
character(0)
Browse[1]> str(wrong.extra)
'data.frame': 20 obs. of 1 variable:
$ sleep$extra: num 0.7 -1.6 -0.2 -1.2 -0.1 3.4 3.7 0.8 0 2 ...
Browse[1]> is.vector(wrong.extra)
[1] FALSE
Browse[1]> c

As well as data in unexpected formats, watch out for ‘weird’ data
that might lead to e.g. missing values in regression output.

Finally; turn the post-mortem debugger off with

options(error=NULL) # turn it off! turn it off!!!
Minimum \( p \)-value

Little point in permutation test for the mean: same result as \( t \)-test

Permutation test is useful when we do not know how to compute the distribution of a test statistic.

Suppose we test additive effects of 8 SNPs, one at a time, and we want to know if the most significant association is real.

For any one SNP the \( z \)-statistic from a logistic regression model has a Normal distribution.

We need to know the distribution of the most extreme of eight \( z \)-statistics. This is not a standard distribution, but a permutation test is still straightforward.
Minimum \( p \)-value

dat <- data.frame(y=rep(0:1,each=100),
SNP1=rbinom(200,size=2,prob=.1), SNP2=rbinom(200,size=2,prob=.2),
SNP3=rbinom(200,size=2,prob=.2), SNP4=rbinom(200,size=2,prob=.4),
SNP5=rbinom(200,size=2,prob=.1), SNP6=rbinom(200,size=2,prob=.2),
SNP7=rbinom(200,size=2, prob=.2), SNP8=rbinom(200,size=2,prob=.4))

> head(dat)
   y SNP1 SNP2 SNP3 SNP4 SNP5 SNP6 SNP7 SNP8
1  1  0  0  0  0  0  0  1  0  0
2  0  0  1  0  1  0  1  0  2
3  0  0  1  0  1  1  0  0  0
4  0  0  1  1  0  0  0  0  0
5  0  0  1  0  1  1  0  0  0
6  0  0  0  0  1  0  1  0  1
Minimum $p$-value

```r
oneZ <- function(outcome, snp) {
  model <- glm(outcome ~ snp, family = binomial)
  coef(summary(model))[["snp", "z value"]]
}

maxZ <- function(outcome, snps) {
  allZs <- sapply(snps,
      function(this.snp) { oneZ(outcome, snp = this.snp) })
  max(abs(allZs))
}

true.maxZ <- maxZ(outcome = dat$y, snps = dat[,-1])

manypermZ <- replicate(10000, 
    maxZ(outcome = sample(dat$y), snps = dat[,-1]))
```

5.22
Minimum $p$-value
**Minimum $p$-value**

The histogram shows the permutation distribution for the maximum $Z$-statistic.

The blue curve is the theoretical distribution for one $Z$-statistic.

The yellow curve is the theoretical distribution for the maximum of eight independent $Z$-statistics.

Clearly the multiple testing is important: a $Z$ of 2.5 gives $p = 0.012$ for a single test but $p = 0.075$ for the permutation test.

The theoretical distribution for the maximum has the right range but the permutation distribution is quite discrete. The discreteness is more serious with small sample size and rare SNPs.

[The theoretical distribution is not easy to compute except when the tests are independent.]
More debugging

Permutation tests on other people’s code might reveal a lack of robustness.

For example, a permutation might result in all controls being homozygous for one of the SNPs and this might give an error message

We can work around this with `tryCatch()`

```r
oneZ<-function(outcome, snp){
    tryCatch({model <- glm(outcome~snp, family=binomial())
              coef(summary(model))['snp','z value'],
              error=function(e) NA
        },
    )
}
```

Now `oneZ()` will return `NA` if there is an error in the model fitting.
Caution: wrong null

Permutation tests cannot solve all problems: they are valid only when the null hypothesis is ‘no association’

Suppose we are studying a set of SNPs that each have some effect on outcome and we want to test for interactions (epistasis).

Permuting the genotype data would break the links between genotype and outcome and created shuffled data with no main effects of SNPs.

Even if there are no interactions the shuffled data will look different from the real data.
Caution: weak null hypothesis

A polymorphism could increase the variability of an outcome but not change the mean.

In this case the strong null hypothesis is false, but the hypothesis of equal means is still true.

- If we want to detect this difference the permutation test is unsuitable because it has low power
- If we do not want to detect this difference the permutation test is invalid, because it does not have the correct Type I error rate.

When groups are the same size the Type I error rate is typically close to the nominal level, otherwise it can be too high or too low.

To illustrate this we need many replications of a permutation test. We will do 1000 permutation tests for a mean, each with 1000 permutations.
meandiff <- function(x, trt) {
  mean(x[trt == 1]) - mean(x[trt == 2])
}

meanpermtest <- function(x, trt, n = 1000) {
  observed <- meandiff(x, trt)
  perms <- replicate(n, meandiff(x, sample(trt)))
  mean(abs(observed) > abs(perms))
}

trt1 <- rep(c(1, 2), c(10, 90))

perm.p <- replicate(1000, {
  x1 <- rnorm(100, 0, s = trt1)
  meanpermtest(x1, trt1)
})

table(cut(perm.p, c(0, 0.05, 0.1, 0.5, 0.9, 0.95, 1))/1000
The $p$-values are too small, relative to a uniform distribution. If we reverse the standard errors we get

\[
\begin{array}{cccccc}
(0,0.05] & (0.05,0.1] & (0.1,0.5] & (0.5,0.9] & (0.9,0.95] & (0.95,1] \\
86 & 99 & 564 & 244 & 6 & 0 \\
\end{array}
\]

If the two groups each have 50 observations we get

\[
\begin{array}{cccccc}
(0,0.05] & (0.05,0.1] & (0.1,0.5] & (0.5,0.9] & (0.9,0.95] & (0.95,1] \\
27 & 28 & 275 & 354 & 67 & 249 \\
\end{array}
\]

which is much better.
6. Writing Big Loops

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Auckland, November 2013
Writing loops in R

We saw that `replicate()`, and `apply()`, `sapply()` are R's preferred way of **looping** (doing the same thing many times)

Even for expert users, their use requires some careful thought; debugging code may be complex.

In this session we’ll talk about some alternatives, and their application to **genome-wide** studies.
for() loops

Your first computer program?

```r
for(i in 1:100) {
  print("Hello world!")
  print(i*i)
}
```

- Everything inside the curly brackets {...} is done 100 times
- Looped commands can depend on i (or whatever you called the counter)
- R creates a vector i with 1:100 in it. You could use any vector that’s convenient
for() loops

for() loops are very intuitive, but have some drawbacks;

- They can be slow;
  - ‘growing’ a large dataset is a bad idea;
    mydata <- cbind(mydata, rnorm(1000, mean=i))
  - so set up blank output first, then ‘fill it in’

- apply() is interpreted slightly faster than for() – but typically this will not matter, contrary to some urban myths

- for() loops require more typing than apply()! For tasks which will be repeated, writing a function really is the Right Thing to do, in the long run.

Using for(i in 1:N) sets up a vector (i) of length N. Do you really need this?
for() loops

Two alternatives; (see ?Control for details)

\[
i <- 1; \text{my.mat} <- \text{matrix}(\text{NA}, N, 3)\\
\text{while}(i <= N)\{\\
    z <- \text{work.on.gene}(i)\\
    \text{my.mat}[i,] <- \text{summary}(z)\\
    i <- i+1\\
\}\n\]

– note that we avoided ‘growing’ the output

\[
i <- 1; \text{my.mat} <- \text{matrix}(\text{NA}, N, 3)\\
\text{repeat}\{\\
    z <- \text{work.on.gene}(i)\\
    \text{my.mat}[i,] <- \text{summary}(z)\\
    i <- i+1\\
    \text{if}(i>=N) \text{break}\\
\}\n\]

Use apply(), sapply() to avoid the ‘setup’ stage
Whole genome studies look very intimidating ...

Genome-wide association study identifies new susceptibility loci for Crohn disease and implicates autophagy in disease pathogenesis

John D Rioux1,2, Rannik J Xavier3, Kent D Taylor4, Mark S Silverberg5, Philippe Goyette1, Alan Huett5, Todd Green5, Petric Kuksalla3, Michael Barmada6, Lisa Wu Datta7, Yin Yao Shugart8, Anne M Griffiths9, Stephan R Targan4, Andrew F Ippoliti4, Edmond-Jean Bernard10, Ling Mei5, Dan I Nicolau11, Miguel Regueiro5, I Philip Schumm13, A Hillary Steinbart7, Jerome I Rotter4, Richard H Duerr8,9,12, Judy H Cho14,16, Mark J Daly3,13,16 & Steven R Brant7,8,16
Application to whole-genome study

... however, each \( p \)-value on that picture comes from a single logistic regression.

There may be 304,413 tests in total; if each one takes 1/10 sec, the analysis is done in under an hour;

<table>
<thead>
<tr>
<th>Time per test</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01 sec</td>
<td>51 mins</td>
</tr>
<tr>
<td>0.1 sec</td>
<td>8 hours 27 mins</td>
</tr>
<tr>
<td>1 sec</td>
<td>3 days 12.5 hrs</td>
</tr>
<tr>
<td>5 sec</td>
<td>17 days 15 hrs (!)</td>
</tr>
<tr>
<td>5 mins</td>
<td>2 yrs 11 months (!!!)</td>
</tr>
</tbody>
</table>

Cutting time per test from 1 sec \( \rightarrow \) 0.1 sec is clearly worthwhile – but proposing analyses where each test takes \( \geq 5 \) secs is silly.
Some easy ‘streamlining’ ideas;

- Write a function to do just the analysis you want
  > my.output <- apply(my.data, 1, my.function)

- Pre-process/‘clean’ your data before analysis; e.g. \( \frac{\text{sum}(x)}{\text{length}(x)} \)
  doesn’t error-check like \( \text{mean}(x) \)

- Similarly, you can streamline \texttt{glm()} to just \texttt{glm.fit()} [as we’ll see, in some examples]

- Use vectorized operations, where possible

- Store data as matrices, not data.frames
Making code run faster, part 2

Streamlining, for experts;

- Write small but important pieces of code in C, and call these from R

- **Batch mode** processing lets you break down e.g. the whole genome into 23 chromosomes – great if you have 23 processors to use.
  - Save your analysis in 23 output files
  - read in the answers
  - finally produce e.g. multi-color pictures

Various packages help implement this, but use is platform-specific
Timing

“Premature optimization is the root of all evil”

Donald Knuth

Do you need to optimize your code? Running 2 or 3 times faster may not be worth the time spent coding/debugging!

But going an order of magnitude faster is A Good Thing.

After you have code that works, you may need to speed it up. Experienced users may be able to ‘eyeball’ the problem; measurement is an easier and more reliable approach.
Timing

• `proc.time()` returns the current time. Save it before a task and subtract from the value after a task.

• `system.time()` times the evaluation of an expression

• R has a **profiler**; this records which functions are being run, many times per second. `Rprof(filename)` turns on the profiler, `Rprof(NULL)` turns it off. `summaryRprof(filename)` reports how much time was spent in each function.

Remember that a 1000-fold speedup in a function used 10% of the time is **less helpful** than a 30% speedup in a function used 50% of the time.
High-throughput code – caveats

We saw yesterday that ‘weird’ datasets can crash your code. These will appear in genome-wide studies, and a crash at SNP 299,999 will be very frustrating.

• Some ‘weirdness’ is easy to spot;
  – Everyone is homozygous
  – All cases missing
  – No variation in outcome ...

• In more complex models, it’s easier to ‘try it and see’. Use tryCatch()

• When ‘weirdness’ is found, high-throughput code should;
  – Produce sensible output (NA, -999 etc)
  – Handle these appropriately in summary output
7. Working with Big Data

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Auckland, November 2013
**Large data**

“R is well known to be unable to handle large data sets.”

Solutions:

- Get a bigger computer: 8-core Linux computer with 32Gb memory for < $3000

- Don’t load all the data at once (methods from the mainframe days).
Large data

Data won't fit in memory on current computers: R can comfortably handle data up to

- About 1/3 of physical RAM
- About 10% of address space (ie, no more than 400MB for 32-bit R, no real constraint for 64-bit R)

R can't (currently) handle a single matrix with more than $2^{31} - 1 \approx 2$ billion entries even if your computer has memory for it.

Storing data on disk means extra programming work, but has the benefit of making you aware of data reads/writes in algorithm design.
Storage formats

R has two convenient data formats for large data sets

- For ordinary large data sets, direct interfaces to relational databases allow the problem to be delegated to the experts.

- For very large ‘array-structured’ data sets the \texttt{ncdf} package provides storage using the netCDF data format.
SQL-based interfaces

Relational databases are the natural habitat of large datasets.

- Optimized for loading subsets of data from disk
- Fast at merging, selecting
- Standardized language (SQL) and protocols (ODBC, JDBC)
Elementary SQL

Basic statement: `SELECT var1, var2 FROM table`

- `WHERE condition` to choose rows where condition is true

- `table1 INNER JOIN table2 USING(id)` to merge two tables on a identifier

- Nesting: `table` can be a complete `SELECT` statement
R database interfaces

- RODBC package for ODBC connections (mostly Windows)

- DBI: standardized wrapper classes for other interfaces
  - RSQLite: small, zero-configuration database for embedded storage
  - RJDBC: Java interface
    - also for Oracle, MySQL, PostgreSQL.
Setup: DBI

Needs DBI package + specific interface package

```r
library("RSQLite") ## also loads DBI package

sqlite <- dbDriver("SQLite")
conn <- dbConnect(sqlite, "example.db")

dbListTables(conn) ## what tables are available?
    ## see also dbListFields()

dbDisconnect(conn) ## when you are done

Now use `conn` to identify this database connection in future requests
```
Queries: DBI

- `dbGetQuery(conn, "select var1, var2, var22 from smalltable")`: runs the SQL query and returns the results (if any)

- `dbSendQuery(conn, "select var1, var2, var22 from hugetable")`: runs the SQL and returns a result set object

- `fetch(resultset, n=1000)`: asks the database for the next 1000 records from the result set

- `dbClearResult(resultset)`: releases the result set
Whole tables: DBI

- `dbWriteTable(conn, "tablename", dataframe)` writes the whole data frame to a new database table (use `append=TRUE` to append to existing table)

- `dbReadTable(conn, "tablename")` reads a whole table

- `dbDropTable(conn, "tablename")` deletes a table.
Setup: ODBC

Just needs RODBC package. Database must be given a "Data Source Name" (DSN) using the ODBC adminstrator on the Control Panel

library("RODBC")

conn <- odbcConnect(dsn)

close(conn) ## when you are done

Now use conn to identify this database connection in future requests
Queries: ODBC

- `sqlQuery(conn, "select var1, var2, var22 from smalltable")`: runs the SQL query and returns the results (if any)

- `odbcQuery(conn, "select var1, var2, var22 from hugetable")`: runs the SQL and returns a result set object

- `sqlGetResults(con, max=1000)`: asks the database for the next 1000 records from the result set
Whole tables: ODBC

- `sqlSave(conn, dataframe, "tablename")` writes the whole data frame to a new database table (use `append=TRUE` to append to existing table)

- `sqlFetch(conn, "tablename")` reads a whole table
Example: SQLite and Bioconductor

Bioconductor’s AnnotationDBI system maps from one system of identifiers (eg probe ID) to another (eg GO categories).

Each annotation package contains a set of two-column SQLite tables describing one mapping.

‘Chains’ of tables allow mappings to be composed so, eg, only gene ids need to be mapped directly to GO categories.

Original annotation system kept all tables in memory; they are now getting too large.
AnnotationDBI

> library("hgu95av")  # a Bioconductor package: see Session 8
> hgu95av2CHR[["1001_at"]]
 [1] "1"
> hgu95av2OMIM[["1001_at"]]
 [1] "600222"
> hgu95av2SYMBOL[["1001_at"]]
 [1] "TIE1"
> length(hgu95av2GO[["1001_at"]])
 [1] 16
Under the hood

> ls("package:hgu95av2.db")
[1] "hgu95av2"        "hgu95av2_dbconn"        "hgu95av2_dbfile"
[4] "hgu95av2_dbInfo" "hgu95av2_dbschema" "hgu95av2ACCNUM"
[7] "hgu95av2ALIAS2PROBE" "hgu95av2CHR" "hgu95av2CHRLENGTHS"
[10] "hgu95av2CHRLOC"  "hgu95av2CHRLOCEND" "hgu95av2ENSEMBL"
> hgu95av2_dbconn()
<SQLiteConnection: DBI CON (7458, 1)>
> dbGetQuery(hgu95av2_dbconn(), "select * from probes limit 5")

<table>
<thead>
<tr>
<th>probe_id</th>
<th>gene_id</th>
<th>is_multiple</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000_at</td>
<td>5595</td>
<td>0</td>
</tr>
<tr>
<td>1001_at</td>
<td>7075</td>
<td>0</td>
</tr>
<tr>
<td>1002_f_at</td>
<td>1557</td>
<td>0</td>
</tr>
<tr>
<td>1003_s_at</td>
<td>643</td>
<td>0</td>
</tr>
<tr>
<td>1004_at</td>
<td>643</td>
<td>0</td>
</tr>
</tbody>
</table>
Under the hood

The `[[` method calls the `mget` method, which also handles multiple queries.

These (eventually) produce SQLite `SELECT` statements with `INNER JOIN` across the tables needed for the mapping.
**netCDF**

netCDF was designed by the NSF-funded UCAR consortium, who also manage the National Center for Atmospheric Research.

Atmospheric data are often array-oriented: eg temperature, humidity, wind speed on a regular grid of \((x, y, z, t)\).

Need to be able to select ‘rectangles’ of data – eg range of \((x, y, z)\) on a particular day \(t\).

Because the data are on a regular grid, the software can work out where to look on disk without reading the whole file: efficient data access.

Many processes can read the same netCDF file at once: efficient parallel computing.
Current uses in biology

- Whole-genome genetic data (us and people we talk to)
  - Two dimensions: genomic location $\times$ sample, for multiple variables
  - Data sizes in tens to thousands of gigabytes.

- Flow cytometry data (proposed new FCS standard)
  - 5–20 (to 100, soon) fluorescence channels $\times$ 10,000–10,000,000 cells $\times$ 5–5000 samples
  - Data sizes in gigabytes to thousands of gigabytes.
Using netCDF data

Interact with netCDF files via the `ncdf` package, in much the same way as databases;

`open.ncdf()` opens a netCDF file and returns a connection to the file (rather than loading the data)

`get.var.ncdf()` retrieves all or part of a variable.

`close.ncdf()` closes the connection to the file.

To see what variables a file contains, and their dimensions, e.g.

```r
conn <- open.ncdf("mynetCDFfile.nc")
conn
```
Variables can use one or more array dimensions of a file
Example

Finding long homozygous runs (possible deletions)

```r
library("ncdf")
nc <- open.ncdf("hapmap.nc")

## read all of chromosome variable
chromosome <- get.var.ncdf(nc, "chr", start=1, count=-1)
## set up list for results
runs<-vector("list", nsamples)

for(i in 1:nsamples){
  ## read all genotypes for one person
  genotypes <- get.var.ncdf(nc, "geno", start=c(1,i),count=c(-1,1))
  ## zero for htzygous, chrm number for hmzygous
  hmzygous <- genotypes != 1
  hmzygous <- as.vector(hmzygous*chromosome)
}
## Example

```r
## consecutive runs of same value
r <- rle(hmzygous)
begin <- cumsum(c(1, r$lengths))
end <- cumsum(r$lengths)
long <- which (r$lengths > 250 & r$values !=0)
runs[[i]] <- cbind(begin[long], end[long], r$lengths[long])
```

```r
close.ncdf(nc)
```

### Notes

- **chr** uses only the 'SNP' dimension, so start and count are single numbers.
- **geno** uses both SNP and sample dimensions, so start and count have two entries.
- **rle** compresses runs of the same value to a single entry.
Advanced: creating netCDF files

Creating files is more complicated

- Define dimensions
- Define variables and specify which dimensions they use
- Create an empty file
- Write data to the file.
Advanced: defining dimensions

Specify the name of the dimension, the units, and the allowed values in the `dim.def.ncdf` function.

One dimension can be ’unlimited’, allowing expansion of the file in the future. An unlimited dimension is important, otherwise the maximum variable size is 2Gb.

```r
snpdim <- dim.def.ncdf("position","bases", positions)
sampledim <- dim.def.ncdf("seqnum","count",1:10, unlim=TRUE)
```
Advanced: defining variables

Variables are defined by name, units, and dimensions

varChrm <- var.def.ncdf("chr","count",dim=snpdim,
    missval=-1, prec="byte")
varSNP <- var.def.ncdf("SNP","rs",dim=snpdim,
    missval=-1, prec="integer")
vargeno <- var.def.ncdf("geno","base",dim=list(snpdim, sampledim),
    missval=-1, prec="byte")
vartheta <- var.def.ncdf("theta","deg",dim=list(snpdim, sampledim),
    missval=-1, prec="double")
varr <- var.def.ncdf("r","copies",dim=list(snpdim, sampledim),
    missval=-1, prec="double")
The file is created by specifying the file name and a list of variables.

```r
genofile<-create.ncdf("hapmap.nc", list(varChrm, varSNP, vargeno, vartheta, varr))
```

The file is empty when it is created. Data can be written using `put.var.ncdf()`. Because the whole data set is too large to read, we might read raw data and save to netCDF for one person at a time;

```r
for(i in 1:4000){
    temp.geno.data <-readRawData(i) ## somehow
    put.var.ncdf(genofile, "geno", temp.geno.data,
                  start=c(1,i), count=c(-1,1))
}
```
Efficient use of netCDF

Read all SNPs, one sample

<table>
<thead>
<tr>
<th>SNP</th>
<th>Sample</th>
<th>Genotypes</th>
<th>Chromosome</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Efficient use of netCDF

Read all samples, one SNP
Efficient use of netCDF

Read some samples, some SNPs.
Efficient use of netCDF

Random access is not efficient: eg read probe intensities for all missing genotype calls.
Efficient use of netCDF

- Association testing: read all data for one SNP at a time

- Computing linkage disequilibrium near a SNP: read all data for a contiguous range of SNPs

- QC for aneuploidy: read all data for one individual at a time (and parents or offspring if relevant)

- Population structure and relatedness: read all SNPs for two individuals at a time.
8. Bioconductor Intro and Annotation

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Auckland, November 2013
What is Bioconductor?

Bioconductor provides tools for the analysis and comprehension of high-throughput genomic data. Bioconductor uses the R statistical programming language, and is open source and open development. It has two releases each year, more than 450 packages, and an active user community.

Use Bioconductor for...

- **Microarrays**

- **High Throughput Assays**
  Import, transform, edit, analyze and visualize flow cytometric, mass spec, HTqPCR, cell-based, and other assays.

- **Sequence Data**
  Import fasta, fastq, ELAND, MAQ, BWA, Bowtie, SAM, gff, bed, wig, and other sequence formats. Trim, transform, align, and manipulate sequences. Perform quality assessment, ChIP-seq, differential expression, RNA-seq, and other workflows. Access the Sequence Read Archive.

- **Annotation**
  Use microarray probe, gene, pathway, gene ontology, homology and other annotations. Access GO, KEGG, NCBI, Biomat, UCSC, vendor, and other sources.

Mailing Lists

- [Re: views on Bio using GRanges object](#) about an hour ago
- [How to output Normalised count data from](#) about 2 hours ago
- [Re: RFS volumes with the Bioconductor...](#) about 7 hours ago

Events

- useR! 2011
  16 - 18 August 2011 — University of Warwick, Coventry, UK
- Statistical Analyses for Next Generation Sequencing
  26 - 27 September 2011 — Birmingham, AL, USA
- [See all events »](#)

News

- [BioC 2011 conference material](#) BioC 2011 conference material is now available.

Bioconductor 2.8 released

Following the usual 6-month cycle, the Bioconductor community released Bioconductor 2.8 on April 14th, 2011. This release comprises 466 software packages and more than 500 up-to-date annotation packages. It has been expressly designed to work with R 2.13.
What is Bioconductor?

• www.bioconductor.org

• Software project for analysis of genomic data – and related tools, resources/datasets

• Open source and Open development

• Free

You could use commercial software; but experts typically write R code first. Also, the help manuals are not a sales pitch and encourage appropriate use.
Bioconductor basics

- Begun in 2001, based at Harvard and now FHCRC (Seattle)
- A large collection of R packages (they also convert good software to R)
- Far too much for our little course!

We’ll give examples of what Bioconductor can do, and how to learn more. Hahne et al (above) is a helpful reference text
Bioconductor basics

Getting started...

Home » Install

- Install Packages • Find Packages • Update Packages • Install R

Install Bioconductor Packages

Use the biocLite.R script to install Bioconductor packages. To install a particular package, e.g., limma, type the following in an R command window:

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

After downloading and installing this package, the script prints "Installation complete" and "TRUE". Install several packages, e.g., "GenomicFeatures" and "AnnotationDbi", with

```r
biocLite(c("GenomicFeatures", "AnnotationDbi"))
```

To install a selection of core Bioconductor packages, use

```r
biocLite()
```

Packages and their dependencies installed by this usage are: affy, affydata, affyPLM, affyQCReport, annaffy, annotate, Biobase, biomart, Biostrings, DynDoc, genefilter, genepattern, GenomicChanges, hub95av2.db, limma, array, multtest, vsn, and stable. After downloading and installing these packages, the script prints "Installation complete" and "TRUE".

The biocLite.R script has arguments that change its default behavior:

- pkgs
  - Character vector of Bioconductor packages to install.
- destdir
  - File system directory for downloaded packages.
- lib
  - R library where packages are installed.
> source("http://bioconductor.org/biocLite.R")
> biocLite()

installs the following general-purpose libraries;

Biobase, IRanges, AnnotationDbi

... then you use e.g. library("Biobase") as before. (NB older versions used to download much more than this)

vignette(package="Biobase") tells you to look at vignette("esApply") for a worked example – a very helpful introduction. (Or use e.g. openVignette(), which is in the Biobase package itself)
Bioconductor basics

To get other packages, use the `source()` command as before, then use e.g.

```r
biocLite("SNPchip")
biocLite(c("limma", "siggenes"))
```

You do not need to type `biocLite()` again (even in a new R session). This would install the general-purpose packages again – which is harmless, but a waste of time.

Note; if, due to access privileges, you need to write to non-default directories, follow the onscreen commands and then start again. On Windows, ‘Run as Administrator’ may cut out this step.
What to install?

Back to the front page – click ‘Help’
What to install?

• **Software** – probably what you want

• **Metadata** – e.g. annotation data, probe sequence data for microarrays of different types

• **Experiment data** – e.g. datasets from hapmap.org, some expression datasets
Simple QC graphics

The "splots" package plots values from 96 or 384-well plates, for QC purposes.

First, install it

biocLite("splots")

Then load into R

library("splots")

There is a single function: plotScreen() for displaying the results.
The file "drosophila.rda" contains 12 of 114 plates from a RNAi gene-knockout study in fruit flies. Each spot represents a gene, and the intensity is low if knockout of that gene is lethal (data from the "RNAeither" package)

```r
load("drosophila.rda")
plotScreen(rnai)
```

The positive controls in the same position each plate are clear, and there are obvious plate effects that you might need to correct by normalization.
Example
One goal of Bioconductor is to provide efficient access inside R to the genome databases that are vital to interpreting associations.

We will look at a few of these

- annotate
- biomaRt
- genomeGraphs

The reason to have an R interface to these databases is to be able to analyze annotation data for many SNPs or RNA transcripts.
Online or stored data

Annotation data can be downloaded in a single file or retrieved for each query from an online database.

Local storage is faster, but may require too much space (e.g., Ensembl) or become obsolete too quickly.

Local storage is ideal for fixed annotation data such as gene names for a microarray or SNP chip.
Types of database

Translations of names: Affy probe 32972_at is the gene **NADPH oxidase 1** with symbol **NOX1** and Ensembl gene id **ENSG00000007952**

Location: NOX1 is on Xq22.1, from 99984969 to 100015990, coded on the negative strand. There are 120 known polymorphisms (SNPs or indels) in this range.

Homology: The mouse version of NOX1 is also on the X chromosome, starting at 130621066 (and called Nox1)

Structure and function: **NOX1** is a membrane protein (location), involved in voltage-gated ion channel activity (molecular function), and involved in signal transduction (biological process).
Annotate

Bioconductor distributes annotation packages for a wide range of gene expression microarrays. The `annotate` package is one way to use this annotation information.

```r
> library("annotate")
> library("hgu95av2.db")
> library("GO.db")
```

loads the `annotate` package and the databases for the Gene Ontology and one of the Affymetrix human microarray chips.
Lookups

The databases are queried with \texttt{get()} or \texttt{mget()} for multiple queries

\begin{verbatim}
> mget(c("738_at", "40840_at", "32972_at"), envir=hgu95av2GENENAME)
$‘738_at’
[1] "5’nucleotidase, cytosolic II"

$‘40840_at’
[1] "peptidylprolyl isomerase F (cyclophilin F)"

$‘32972_at’
[1] "NADPH oxidase 1"

> go<-get("738_at", envir=hgu95av2GO)
> names(go)
[1] "GO:0009117" "GO:0005829" "GO:0005737" "GO:0000166" "GO:0000287"
[6] "GO:0008253" "GO:0008253" "GO:0016787"
\end{verbatim}
> get("GO:0009117", envir=GOTERM)
GOID: GO:0009117
Term: nucleotide metabolic process
Ontology: BP
Definition: The chemical reactions and pathways involving a nucleotide, a nucleoside that is esterified with (ortho)phosphate or an oligophosphate at any hydroxyl group on the glyco moiety; may be mono-, di- or triphosphate; this definition includes cyclic nucleotides (nucleoside cyclic phosphates).
Synonym: nucleotide metabolism
What lookups are there?

> library(help="hgu95av2.db")

hgu95av2ALIAS2PROBE Map between Common Gene Symbol Identifiers and Manufacturer Identifiers

> get("NOX1", envir=hgu95av2ALIAS2PROBE)
[1] "32972_at"  "32973_s_at"

You can also reverse a lookup table with revmap()

> get("NOX1", envir=revmap(hgu95av2SYMBOL))
[1] "32972_at"  "32973_s_at"
> get("X",revmap(hgu95av2CHR))
[1] "1016_s_at"  "107_at"  "1100_at"  "112_g_at"  "1155_at"
.... and lots more
BioMart

BioMart (www.biomart.org) is a query-oriented data management system developed jointly by the European Bioinformatics Institute (EBI) and Cold Spring Harbor Laboratory (CSHL).

biomaRt is an R interface to BioMart systems, in particular to Ensembl (www.ensembl.org). Ensembl is a joint project between EMBL - European Bioinformatics Institute (EBI) and the Wellcome Trust Sanger Institute (WTSI) to develop a software system which produces and maintains automatic annotation on selected eukaryotic genomes.
**BioMart**

We begin by choosing which BioMart to use

```r
> library(biomaRt)
Loading required package: RCurl
> listMarts()

    biomart                                      ensembl
    1    ensembl                             ENSEMBL GENES 63 (SANGER UK)
    2      snp                              ENSEMBL VARIATION 63 (SANGER UK)
    3 functional_genomics                   ENSEMBL REGULATION 63 (SANGER UK)
    4      vega                            VEGA 43 (SANGER UK)
    5 bacteria_mart_10                       ENSEMBL BACTERIA 10 (EBI UK)
    6 fungi_mart_10                          ENSEMBL FUNGI 10 (EBI UK)
    7 fungi_variations_10                        
    8 metazoa_mart_10                       ENSEMBL METAZOA 10 (EBI UK)
    9 metazoa_variations_10                  ENSEMBL METAZOA VARIATION 10 (EBI UK)
    ...
    60 ENSEMBL_MART_PLANT                   GRAMENE 30 ENSEMBL GENES (CSHL/CORNELL US)
    61 ENSEMBL_MART_PLANT_SNP               GRAMENE 30 VARIATION (CSHL/CORNELL US)
    62 GRAMENE_MARKER_30                    GRAMENE 30 MARKERS (CSHL/CORNELL US)
    63 GRAMENE_MAP_30                       GRAMENE 30 MAPPINGS (CSHL/CORNELL US)
    64 QTL_MART                           GRAMENE 32 QTL DB (CSHL/CORNELL US)
    65 salmosalar2_mart                  UNIGENE SALMO SALAR DATABASE (CMM CHILE)
    66 trucha_mart                        UNIGENE ONCORHYNCHUS MYKISS DATABASE (CMM CHILE)

> ens <- useMart("ensembl")
```
BioMart

We then choose a database to use

> listDatasets(ens)

<table>
<thead>
<tr>
<th>dataset</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>oanatinus_gene_ensembl</td>
<td>Ornithorhynchus anatinus genes (OANA5)</td>
</tr>
<tr>
<td>tguttata_gene_ensembl</td>
<td>Taeniopygia guttata genes (taeGut3.2.4)</td>
</tr>
<tr>
<td>cporcellus_gene_ensembl</td>
<td>Cavia porcellus genes (cavPor3)</td>
</tr>
<tr>
<td>gaculeatus_gene_ensembl</td>
<td>Gasterosteus aculeatus genes (BROADS1)</td>
</tr>
<tr>
<td>lafricana_gene_ensembl</td>
<td>Loxodonta africana genes (loxAfr3)</td>
</tr>
<tr>
<td>pvampyrus_gene_ensembl</td>
<td>Pteropus vampyrus genes (pteVam1)</td>
</tr>
<tr>
<td>btaurus_gene_ensembl</td>
<td>Bos taurus genes (UMD3.1)</td>
</tr>
<tr>
<td>meugenii_gene_ensembl</td>
<td>Macropus eugenii genes (Meug_1.0)</td>
</tr>
<tr>
<td>sharrisii_gene_ensembl</td>
<td>Sarcophilus harrisii genes (DEVIL7.0)</td>
</tr>
<tr>
<td>cfamiliaris_gene_ensembl</td>
<td>Canis familiaris genes (CanFam3.1)</td>
</tr>
</tbody>
</table>

> hsap <- useDataset("hsapiens_gene_ensembl",mart=ens)
BioMart

The `getGene` function queries the database for gene information. It accepts many forms of gene identifier, e.g., Entrez, HUGO, Affy transcript.

```r
> getGene(id=1440, type="entrezgene", mart=hsap)
  entrezgene hgnc_symbol
  1  1440    CSF3
description
  1 colony stimulating factor 3 (granulocyte) [Source:HGNC Symbol;Acc:2438]
  chromosome_name  band  strand  start_position  end_position  ensembl_gene_id
  1  17  q21.1  1  38171614  38174066  ENSG00000108342

> getGene(id=c("AGT","AGTR1"), type="hgnc_symbol", mart=hsap)
  hgnc_symbol  hgnc_symbol
  1    AGT    AGT
  2    AGTR1   AGTR1

description
  1 angiotensinogen (serpin peptidase inhibitor, clade A, member 8) [Source:HGNC Symbol;Acc:333]
  2 angiotensin II receptor, type 1 [Source:HGNC Symbol;Acc:336]
  chromosome_name  band  strand  start_position  end_position  ensembl_gene_id
  1  1  q42.2  -1  230838269  230850043  ENSG00000135744
  2  3  q24   1  148415571  148460795  ENSG00000144891
```
BioMart

getBM is more general than getGene. It specifies a list of filters for selecting genes or SNPs and attributes to return from the database.

```r
> affyids <- c("202763_at", "209310_s_at", "207500_at")
> getBM(attributes = c("affy_hg_u133_plus_2", "hgnic_symbol", "chromosome_name", "start_position", "end_position", "band"), filters = "affy_hg_u133_plus_2", values = affyids, mart = hsap)

<table>
<thead>
<tr>
<th>affy_hg_u133</th>
<th>hgnic</th>
<th>chromosome_name</th>
<th>start_position</th>
<th>end_position</th>
<th>band</th>
</tr>
</thead>
<tbody>
<tr>
<td>202763_at</td>
<td>CASP3</td>
<td>4</td>
<td>185785844</td>
<td>185807623</td>
<td>q35.1</td>
</tr>
<tr>
<td>207500_at</td>
<td>CASP5</td>
<td>11</td>
<td>104370180</td>
<td>104384957</td>
<td>q22.3</td>
</tr>
<tr>
<td>209310_s_at</td>
<td>CASP4</td>
<td>11</td>
<td>104318804</td>
<td>104344535</td>
<td>q22.3</td>
</tr>
</tbody>
</table>
```

listAttributes(hsap) and listFilters(hsap) list the available attributes and filters (hundreds)
```r
> getBM(mart=hsap, attributes=c("band","hgnc_symbol"),
    filters=c("band_start","band_end","chromosome_name"),
    values=list("p21.33","p21.33",6))

<table>
<thead>
<tr>
<th>band</th>
<th>hgnc_symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p21.33</td>
</tr>
<tr>
<td>2</td>
<td>p21.33</td>
</tr>
<tr>
<td>3</td>
<td>p21.33</td>
</tr>
<tr>
<td>4</td>
<td>p21.33</td>
</tr>
<tr>
<td>5</td>
<td>p21.33</td>
</tr>
<tr>
<td>6</td>
<td>p21.33</td>
</tr>
<tr>
<td>7</td>
<td>p21.33</td>
</tr>
<tr>
<td>8</td>
<td>p21.33</td>
</tr>
<tr>
<td>9</td>
<td>p21.33</td>
</tr>
<tr>
<td>10</td>
<td>p21.33</td>
</tr>
<tr>
<td>11</td>
<td>p21.33</td>
</tr>
<tr>
<td>12</td>
<td>p21.33</td>
</tr>
<tr>
<td>121</td>
<td>p21.33</td>
</tr>
<tr>
<td>122</td>
<td>p21.33</td>
</tr>
<tr>
<td>123</td>
<td>p21.33</td>
</tr>
<tr>
<td>124</td>
<td>p21.33</td>
</tr>
<tr>
<td>125</td>
<td>p21.33</td>
</tr>
</tbody>
</table>
```

getLDS() combines two data marts, for example to homologous genes in other species. We can look up the mouse equivalents of a particular Affy transcript, or of the NOX1 gene.

```r
> human = useMart("ensembl", dataset = "hsapiens_gene_ensembl")
> mouse = useMart("ensembl", dataset = "mmusculus_gene_ensembl")
> getLDS(attributes = c("hgnc_symbol","chromosome_name", "start_position"),
+ filters = "hgnc_symbol", values = "NOX1", mart = human,
+ attributesL = c("chromosome_name","start_position","external_gene_id"),
+ martL = mouse)

          V1     V2     V3     V4     V5     V6
1       NOX1   X 100098313   X 134086421  Nox1
```

The mouse gene name is the same as the human one apart from capitalisation.
Homology

The `getSequence` function looks up DNA or protein sequences by chromosome position or gene identifiers

```r
> agt<-getSequence(id="AGT", type="hgnc_symbol", seqType="peptide", mart=hsap)
> agt

1  MRKRAPQSEMAGVSLRATILCLAWAGLAAGDVRYIHHPHVLVIHNESTCEQLAKANAGKPKDPTFIFAPIQAKTS
   PVDEKALQDQLVLVAAKLIDTEDKLRRAAMVGMLANFLGFRYGMHSELWGVVHATVLSPTAVFGLASLYLGALDHTAD
   RLQAILGVWPKDKNCTSRLDAHKVLSSALQAVQGLLVAQGRADSQALLLLSTTVGVFTAPGLHLKQPFPVQGLALYTPVVL
   PRSDLFTELDVAAEKIDRFRMQAVTGKWTCGSMGASVDSLAFNITYVHFQGKMKGFSLLAEPQEFWVDNSTSVSVPMLS
   GMGTQHWSDIQDNSVTQVPFTESACLLLIQPHYASDLKVEGLTFQQNLSLNWKKLSRPTIHLTMPQLQGSYDLQ
   DLLAQAEIPAILHTELNLQKLSNDRIRVGEVLNSIFFELEADEREPDESTQQLNKPEVLEVTNRNPFLFAVDQSATAL
   HFLGRVANPLSTA*
```
Example: finding chromosomes

We had a set 1524 SNPs, of which 409 did not have their chromosome listed.

I needed to know which SNPs were on the X chromosome, to estimate sex from DNA intensity and heterozygous X-chromosome loci, for QC.

> head(unknown)
[1] "UGT1A3-001449-0_B_R_1538822"  "LIPC-002761-0_B_R_1538453"
[3] "CETP-001265-0_B_R_1538254"  "F8-165293-0_T_F_1538626"
[5] "CPB2-051208-0_B_F_1539402"  "VDRDIL-1355-0_T_F_1539404"

A hand-search would be easy but tedious, so we want an automated approach
Example: finding chromosomes

First extract the gene names

genes <- sapply(unknown, function(snp) strsplit(snp,"-"))[1][1])
ugenesis <- unique(genes)

Now call out to Ensembl

getBM(attributes="chromosome_name", filters="hgnc_symbol",values=ugenesis,
mart=hsap)

works for all except VRDIL, which isn't recognized.
Finding SNPs

Human SNPs (and short indels) are in a separate database from gene information. We can look up known SNPs and other polymorphisms for the NOX1 gene.

```r
> snpmart = useMart("snp", dataset = "hsapiens.snp")
Checking attributes ... ok
Checking filters ... ok
> getBM(c("refsnip_id", "allele", "chrom_start", "chrom_strand"),
filters = c("chr_name", "chrom_start","chrom_end"),
values = list("X",99984969,100015990), mart = snpmart)
```

<table>
<thead>
<tr>
<th>refsnip_id</th>
<th>allele</th>
<th>chrom_start</th>
<th>chrom_strand</th>
</tr>
</thead>
<tbody>
<tr>
<td>rs7054049</td>
<td>T/A</td>
<td>99985184</td>
<td>1</td>
</tr>
<tr>
<td>rs60975472</td>
<td>G/T</td>
<td>99985304</td>
<td>1</td>
</tr>
<tr>
<td>rs58902780</td>
<td>A/G</td>
<td>99985571</td>
<td>1</td>
</tr>
<tr>
<td>rs182188185</td>
<td>G/A</td>
<td>99985618</td>
<td>1</td>
</tr>
<tr>
<td>rs186748080</td>
<td>A/G</td>
<td>99985798</td>
<td>1</td>
</tr>
</tbody>
</table>

8.29
More metadata

The `citation()` function prints out information about how to cite a package

```r
> citation("biomaRt")
```

To cite the biomaRt package in publications use:


Citations are one way academic software authors can prove to funders and promotion committees that software is worthwhile.
GenomeGraphs

This package makes pretty pictures from the annotation data.

For example, a pictures showing the standard and alternative splices for the NOX1 gene and the location of the gene on the X chromosome

> library(GenomeGraphs)
> gene <- makeGene(id = "NOX1", type = "hgnc_symbol",
                   biomart = hsap)
> transcript <- makeTranscript(id="NOX1",type="hgnc_symbol",
                               biomart=hsap)
> ideogram <- makeIdeogram(chromosome = "X")
> gdPlot(list(ideogram,gene,transcript))
GenomeGraphs