SISG
2014

SISG Module 13: Advanced R Programming in Bioinformatics

19th Summer Institute in Statistical Genetics

UNIVERSITY of WASHINGTON

(This page left intentionally blank.)

SISG Module 13
Advanced R for Bioinformatics

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Seattle, July 2014

Introduction: Course Aims

- Programming with R
  - Efficient coding
  - Code that other people can use
- Using R for sophisticated analyses
  - Some useful tools for large-scale problems
  - Making R play nicely with others
  - Knowing where to look when you need more

Introduction: About Prof Lumley

- Prof, University of Auckland
- R Core developer
- Genetic/Genomic research in Cardiovascular Epidemiology
- Sings bass (sometimes)

Introduction: About Prof Rice

- Associate Prof, UW Biostat
- Not an authoR, but a useR (and a teacheR)
- Genetic/Genomic research in Cardiovascular Epidemiology
- Sings bass (in Seattle!)

... and you?

(who are you, what area of genomics, what are you looking for from the course)

Introduction: Course structure

10 sessions over 2.5 days

- Day 0; Programming in R, Graphics
- Day 1; Objects, Packages, XML
- Day 2; C code, large datasets

Download everything from here;

`http://faculty.washington.edu/kenrice/sisg-adv`

Introduction: Session structure

We will alternate teaching (questions welcome) and hands-on exercises (questions and discussions welcome!)

For some topics, within a single 90 minute session;

- 45 mins teaching (Questions welcome! Please interrupt!)
- 30 mins hands-on
- 15 mins summary, discussion

For other topics, we'll separate sessions (90 mins) and hands-on exercises (90 mins)

1. Introduction to R: First steps

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Seattle, July 2014

Important pre-takeoff announcement:

We are assuming you know;

- How to use `R` from the command line, and how to write and use script files (and spot e.g. missing commas and }'s)
- How to manipulate basic data structures in `R`; in particular vectors and data frames
- How to write functions
- What `NA` means, and that `42+NA==NA`
- Enough programming (in `R` or elsewhere) to recognize loops, and manage files external to your `R` session
- How to look up help files

Of course, familiarity with (non-advanced) statistical & genetic concepts will also help

Programmers: what is R?

- R is a free implementation of a dialect of the S language, the interactive statistics and graphics environment developed at Bell Labs.

- R/S are probably the most widely used software for research in statistical methodology and in genomics, and is popular in financial modelling and medical statistics.

- John Chambers won the 1999 ACM Software Systems award for S, which *will forever alter the way people analyze, visualize, and manipulate data*.

- Ross Ihaka won the Royal Society of New Zealand's 2008 Pickering Medal, recognizing *excellence and innovation in the practical application of technology* for the creation of R.

Programmers: a little prehistory

The design of R is largely based on S version 3, which predates Java, Python, JavaScript, Linux, MacOS X, and usable versions of Windows.

Much of the design was fixed in S version 2, which predates C++, Perl, the ANSI C standard, the IBM PC, the GNU project, and Miami Vice.

The basic graphics system is older than Space Invaders.

Yes, some things would be done differently today.

Simulation

This really is how calculations and simulation studies were done! Simulations have **always** been part of statistical research.

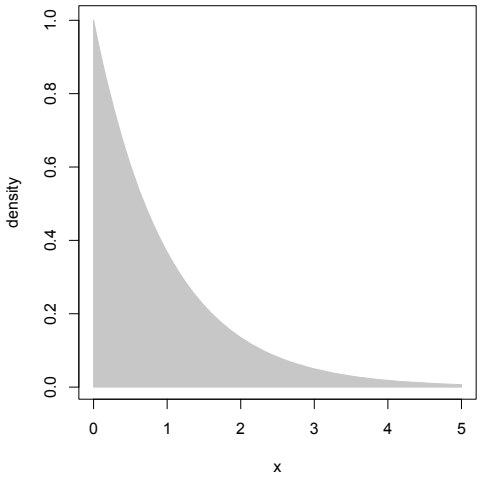
Simulations: a simple example?

Here's a simple problem, for which we *can* work out the exact answer;

For samples of i.i.d $Exp(1)$ *data with* n*=51...*
What is the mean value of the sample median?
What is the mean value of the median-squared?

If you had, say, 51 survival times to analyze, from a distribution of times not unlike $Exp(1)$, these are sane questions.

$Exp(1)$ looks like this (right) any guesses?



Simulations: a simple example?

...guessing these would require a lot of luck;

$$\mathbb{E}Y = \frac{2178178936539108674153}{3099044504245996706400}$$

$$\mathbb{E}Y^2 = \frac{24672823160636679674592332321392579768019 59}{4802038419648657749001278815379823900480000}$$

- They are 0.70286, 0.51380, to 5 d.p.
- They are *about* 2/3 and 1/2
- 3–4 significant figures is probably enough for most practical purposes. Being *able* compute more accurately is re-assuring
- In the 'post-genome' era, being able to compute quickly *is* important (again)

Simulations: a simple example?

Brute force provides perfectly acceptable answers; the `replicate()` function replicates evaluation of an expression

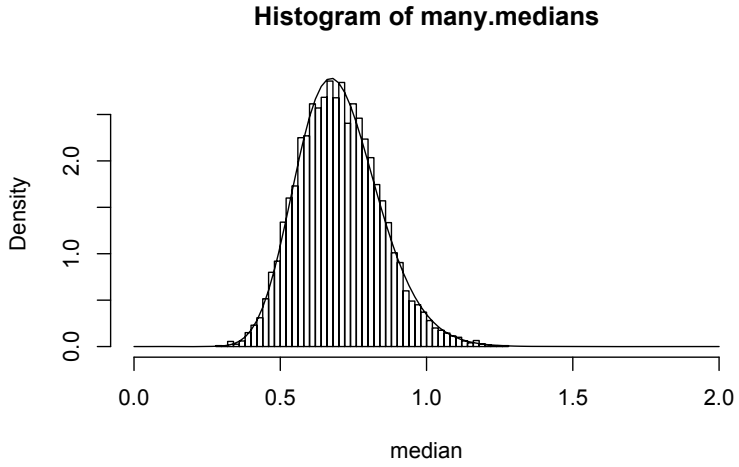
```
> bigB <- bazillion <- 10000
> set.seed(4) # a specific "start" value
> many.medians <- replicate(bigB, { median(rexp(51)) } )
> round( mean(many.medians), 3)
[1] 0.702
> round( mean(many.medians^2), 3)
[1] 0.513
```

The 'right' answers averages over an infinite number of replicatsions. `bigB=10,000` here, which $\approx \infty$.

This calculation takes < 2 seconds, on my desktop

Simulations: a simple example?

Our simulations get us very close to the true distribution of the median;



Having done the 'hard work' of simulation, we can also compute skewness, kurtosis, quantiles, etc – all for 'free'. This technique is very powerful – and often under-rated by statisticians.

Simulations: a simple example?

Here are some other statistical concepts, interpreted in the same way;

- *[If] we simulated data, a bazillion times* ($B \approx \infty$)..."
- *...and applied our procedure to each dataset – and recorded the output*

- Does our estimate usually get close? [consistency]
- How close does our estimate typically get? [bias]
- How variable is our estimate? [standard error, efficiency]
- How often does our interval cover the truth? [coverage]
- How often does our test make a Type I/Type II error? [size/power]

Effective coding

We need to be able to program simulations effectively. A good default for any simulation study follows this 'pseudo-code';

```
do.one <- function(n, beta, f){
   ... commands to do one analysis
   ... last command spits out what you want
}

many.sim <- replicate(bigB, do.one(my.n, my.beta, my.f))
   ... commands to work out observed coverage, bias, etc
```

Once this works, wrap it inside further loops, e.g.

```
n.vals        <- c(10, 20, 30, 40, 50, 1000)
coverage.vals <- sapply(n.vals, function(n){
   ... commands to do the replication, with my.n=n
   })
```

At each stage, you must first write a function, *then check it*. This requires a bit of sanity-checking (i.e. trying it where you know *at least roughly* what should happen) and debugging.

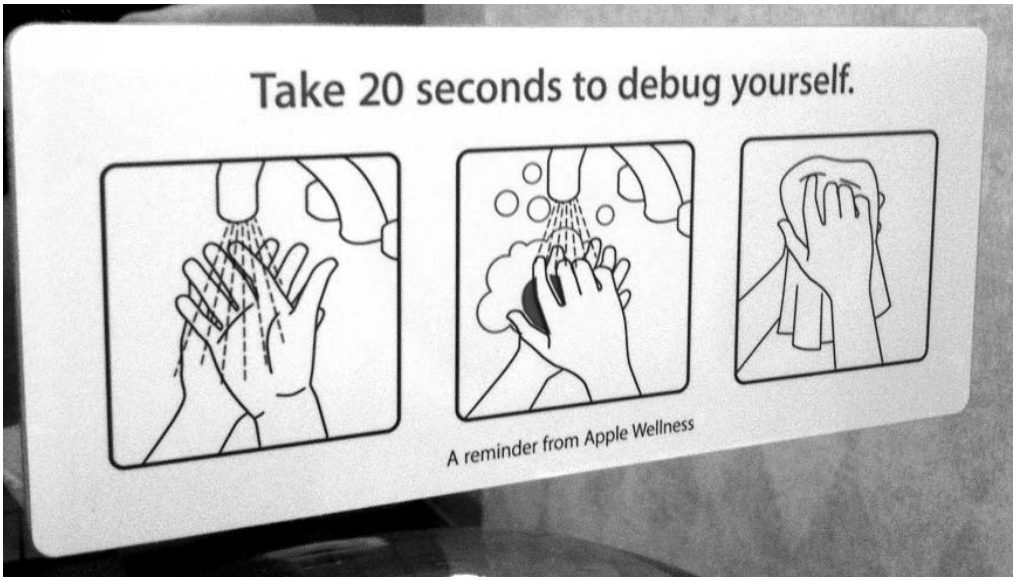
Effective coding

The use of `sapply()` (and `apply()`, `lapply()`) can be unfamiliar – many programmers have used `for()` loops elsewhere. `R` does have `for()` loops (see `?Control`) but;

- 'Growing' the dataset is a terrible idea;
  ```
  for(i in 1:n){
  mydata <- cbind(mydata, rnorm(1000, mean=i)) # noooooo!
  }
  ```
  Always set up blank output first, then 'fill it in'
- Use of `replicate()`, `apply()` etc means slightly faster interpretation of code than `for()` – but not by much. `for()` loops are not *intrinsically* evil
- `for()` requires more typing than `replicate()` etc, and is often more work to edit
- Using functions makes your ultimate `R` package easier to produce... right?

Debugging

A ‘handy’ hint from the Apple Corporation;



Debugging

Beyond the level of spotting missed commas and mis-matching parentheses, debugging is difficult.

We’ll discuss use of `traceback()` and `recover()`, which can help;

```
> # a trite example of traceback()
> f1 <- function(x){ print(x); f2(x) }
> f2 <- function(x){ x + i.dont.exist }
> f1(10) # gives this strange error;
[1] 10
Error in f2(x) : object 'i.dont.exist' not found
> traceback()
2: f2(x)
1: f1(10)
```

The error occurred inside the execution of `f2()`

Debugging

If the error's not obvious, try using `recover()`;

```
options(error=recover) # enter c to close
set.seed(4)
replicate(1000, {
y <- rnorm(10)
x <- rbinom(10, 1, 0.5)
lm1 <- lm(y~x) # regress Y on X
c(coef(lm1)[2], vcov(lm1)[2,2]) # terms of interest
})

# Hint: look at the highest number frame first

#turn it off! turn it off!
options(error=NULL)
```

Use `ls()` to list local objects; the highest frame number is a good place to start

Debugging

`trace()` adds instrumentation to a function

- `trace(rnorm)` prints a message when `rnorm` is started/ended
- `trace(rnorm, recover)` calls the debugger when `rnorm()` is entered.
- `trace(lm, quote(if(all(mf$x==1)) recover()), at=12)` calls the debugger if `mf$x` is all 1s at line 12 of `lm()`

Use `untrace(rnorm)` to remove tracing from `rnorm()`

Exceptions

While you might never see them in practice (due to data cleaning) in simulation studies your replications may produce 'pathological' data, e.g. all X are identical, or all minor allele-carriers smoke. If your regressions estimate differences per allele-copy, adjusting for smoking, it *should* complain.

If this is just too tedious (and rare) to bother fixing, you can use `tryCatch()`;

```
one.glm <- function(outcome, x){
   tryCatch(
      {model <- glm(outcome~x, family=binomial())
       coef(summary(model))[2,]
      },
   error=function(e){rep(NA, 4)} # puts 4 NAs in output
   )
}
```

... but check your simulation output's rates of `NA`-ness. It's better to pre-empt these problems – but this is not easy

Timing

Premature optimization is the root of all evil

Donald Knuth

If you **already have** the capacity to generate reasonably accurate results within a sane time limit, optimizing code is a *waste of effort*

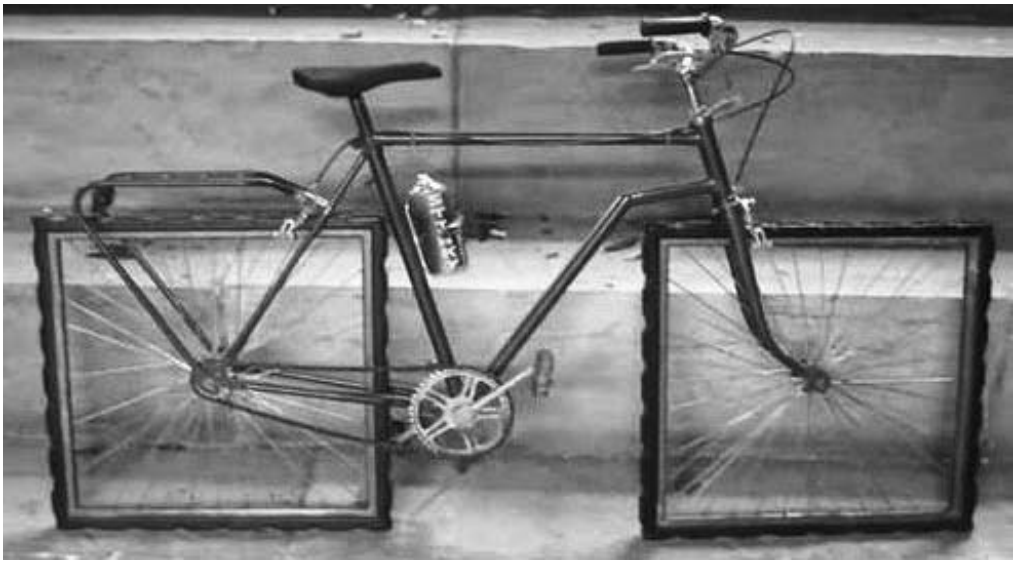
If you need to do things an **order of magnitude** faster, or use your code again (repeatedly) then optimizing your code **may** be worthwhile

To optimize, you need to know;

- What's the bottleneck?
- How much faster can I make that step?

Timing

Obvious bottleneck/easy solution;



Timing

...What's the bottleneck?

Experienced useRs may be able to 'eyeball' this from code; measurement is an **easier and more reliable** approach (!)

To find out how long operations are taking;

- `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 a given 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: A 1000-fold speedup in a function used 10% of the time is **less helpful** than a 2-fold speedup in a function used 50% of the time.

Timing

A small example of this in action;

```
# what is taking all the time?
Rprof("deleteme.txt")
   many.sim <- replicate(1000, {
      y <- rnorm(10)
      x <- rbinom(10, 1, 0.5)
      if( all(x==0)|all(x==1))  return(c(NA,NA))
      lm1 <- lm(y~x)
      c(coef(lm1)[2], vcov(lm1)[2,2])
   })
Rprof(NULL) # turn it off! turn it off!
summaryRprof("deleteme.txt")
```


Timing

...How much faster can I make that step?

Some simple tips;

- Pre-process/clean your data before analysis; e.g. `sum(x)/length(x)` doesn't error-check like `mean(x)`
- Similarly, use `glm.fit` not `glm` – use matrix calculations in place of `lm()`
- Use vectorized operations, where possible
- Store data as matrices, not data frames
- Delete objects you are finished with

Timing

More advanced methods;

- Write **small but important** pieces of code in `C`, and call these from `R`
- Run multiple batches. Store your commands in one script file (which you should do anyway) and call it with e.g.

  ```
  R CMD BATCH myscript.R myconsoleoutput.txt &
  ```

  ... and finally assemble all the (saved) results

The second option applies when there is no available speedup; if your `R` session is mostly waiting for `C` to do matrix work, writing the whole thing in `C` offers no important benefit

More advanced: short cuts to C

For a limited number of jobs, it may be worth getting `R` to send a (large) number of generated datasets to `C` simultaneously.

- For example, instead of looping over datasets with $n = 20$ outcomes Y and $n = 20$ covariates X , generate $B \times 20$ matrices **Y** and **X**; using `rowSums(X)`, `rowSums(X*Y)` etc to construct $\widehat{\beta}$ avoids `replicate()` or similar
- For large n or large B one can quickly run out of memory
- This is a massive pain! I have only used it productively for one real job – doing 2.5 million cookie-cutter meta-analyses
- Less of a pain is `cor(large.matrix)` – for all pairwise correlations of columns of `large.matrix`, where all the looping is done in `C`

For complex methods, this approach will not help

Bonus tracks: how big?

Q. What's the 'Monte Carlo' error in my estimates?

One quick-and-dirty measure of uncertainty is given by these intervals;

```
many.thetahat <- replicate(bigB, {...calculate an estimate...} )
lm1 <- lm(many.thetahat~1)
confint.default(lm1)
```

For binary outcomes, (i.e. when you want coverage, size, power)

```
z <- replicate(bigB, {... calculate theta.hat/est.std.err ...})
mean( z^2 < 1.96^2 ) # how many give p>0.05?
lm2 <- lm( I(z^2 < 1.96^2) ~ 1 )
confint.default(lm2)
```

For GWAS-style levels of e.g. 5×10^{-8}, simulations with e.g. $B = 10^{10}$ may be needed; efficient coding of them can save many days of processor time.

2. Graphics

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Seattle, July 2014

Important pre-takeoff announcement:

We are assuming you know;

- ... that graphics are useful! (and may be worth $\leq$ 1000 words)
- How to make some simple plots e.g. making a scatterplot with `plot()`, adding to existing plots using `points()`, `lines()`, `text()`, and `legend()`
- That these functions can take many three letter arguments; `lwd`, `lty`, `pch` and many others, which can be looked up via `?par`
- That, ultimately, we want PDFs, JPEGs and other output formats – not just a window in an R session

Plotting large & high-dimensional data

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

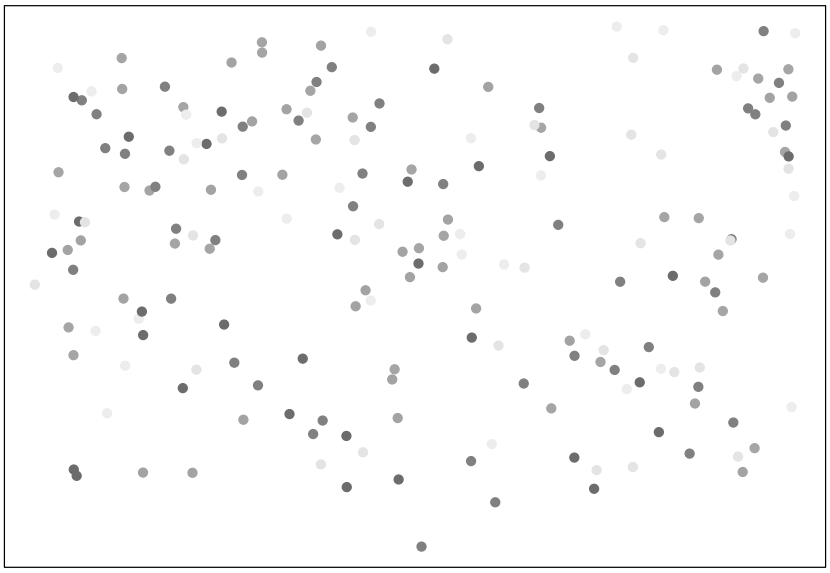
For higher-dimensions, some traditional approaches are;

- 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;

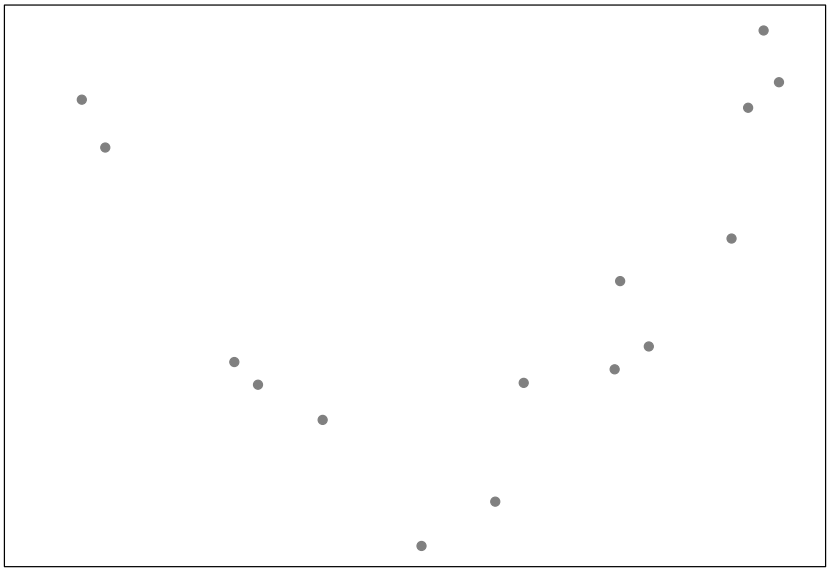
Plotting large & high-dimensional data

Some of these points are not like the others...



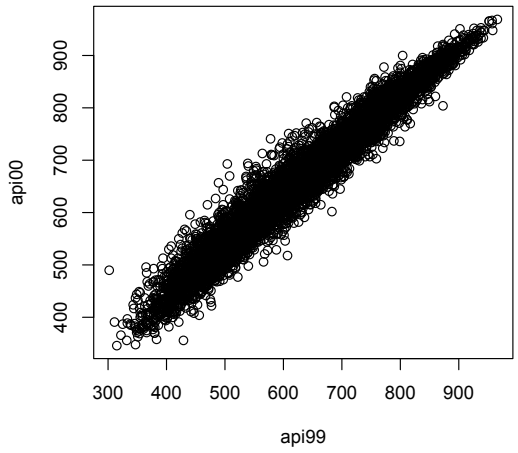
Plotting large & high-dimensional data

Some of these points are not like the others...



Plotting large & high-dimensional data

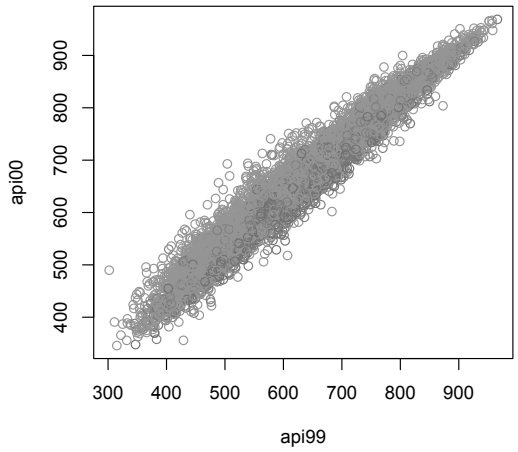
For large(ish) data, 'overlap' is a fundamental problem...



(California Academic Performance Index on 6194 schools)

Plotting large & high-dimensional data

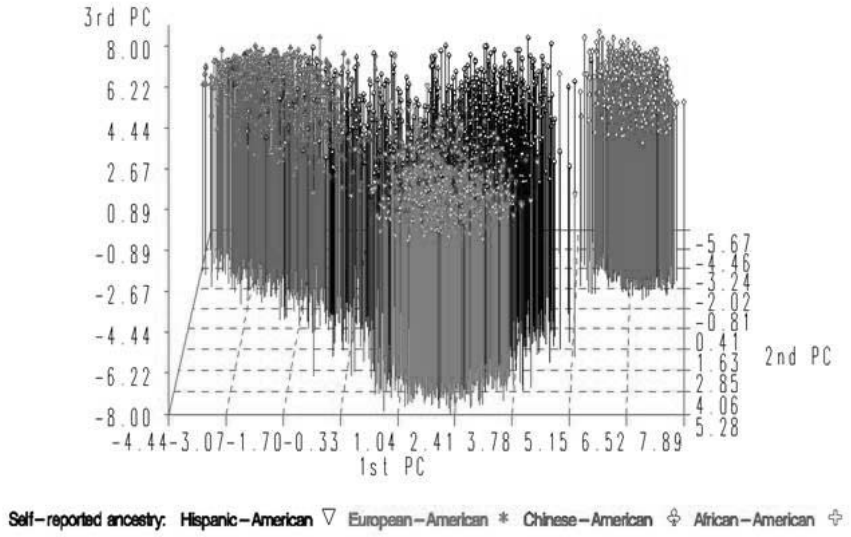
... which remains, when we color-code.



Colors denote Elementary, Middle & High Schools

Plotting large & high-dimensional data

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



(R does have `persp()`, for occasional use)

Conditioning plots

A typical goal for measuring Z is to see whether the $Y - X$ relationship changes at different values of Z. For example, we might want to see if a Blood Pressure/genotype association varies by Body Mass Index (weight/height^2)

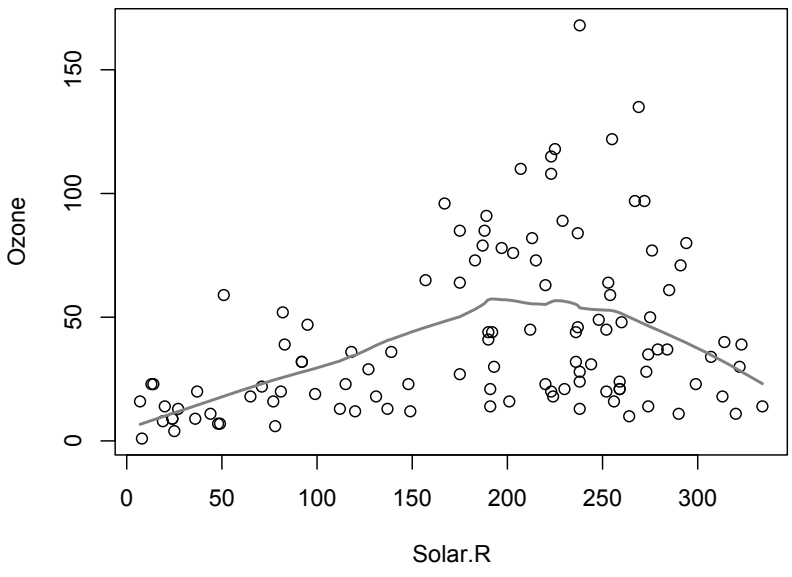
In this case, it's useful to show plots of Y against X conditioned on the value of Z, i.e. Y versus X for all data with Z in a small range. This is known as a conditioning plot, and can be produced with `coplot()`.

Conditioning plots

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

However, looking at ozone concentrations in NY in summer (Y) we see a non-monotone relationship with sunlight (X)

Conditioning plots

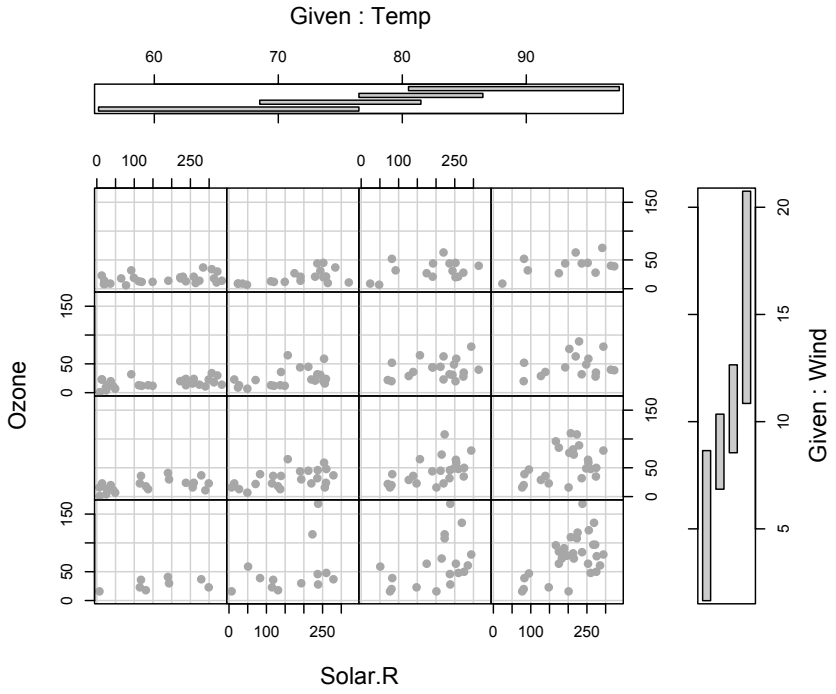


Conditioning plots

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



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')

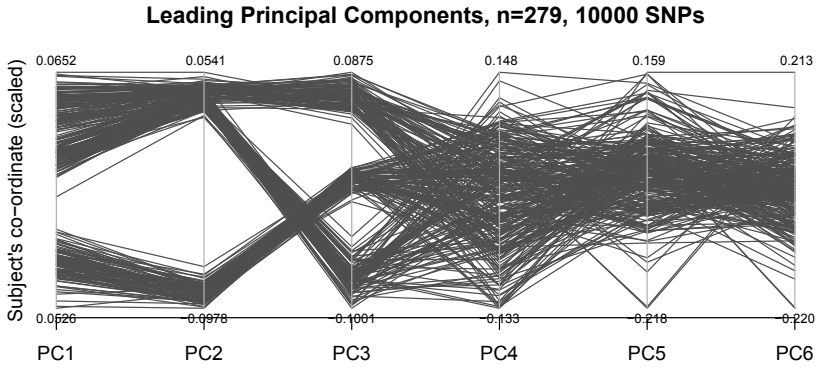
- The horizontal/vertical 'shingles' tell you which data appear in which plot. The overlap can be set to zero, if preferred

- `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")`

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;

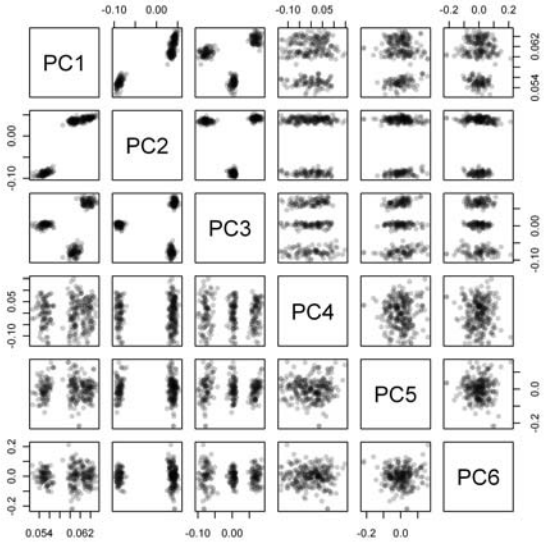


Parallel Coordinate Plots

- 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, ...

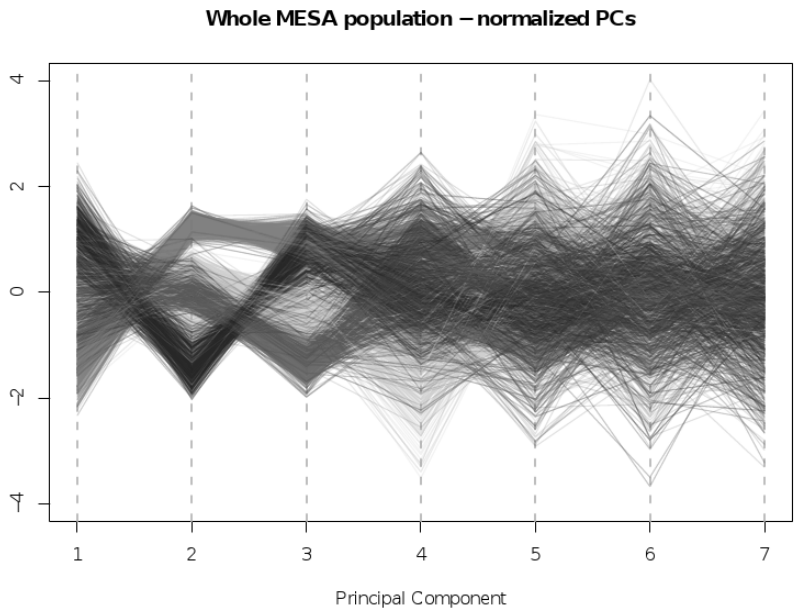
Parallel Coordinate Plots

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



Parallel Coordinate Plots

The pin cushion data++ : colors indicate self-report ancestry



Transparency

The colors in the last examples were transparent. As well as specifiying 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)`

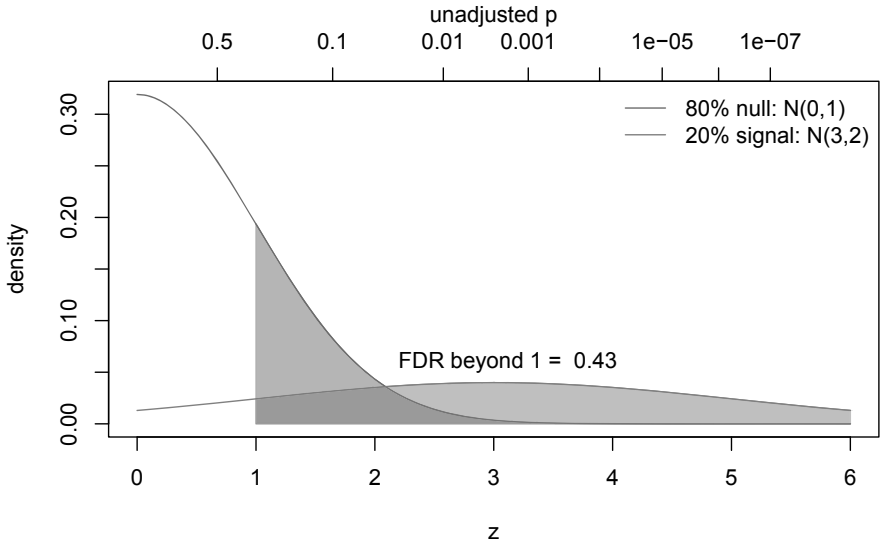
Transparency

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" ) # tranparent blue
polygon(
c(xvals,6,1), c(0.2*dnorm(xvals,3,2), 0,0),
density=NA, col="#FF000080" ) # tranparent 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;



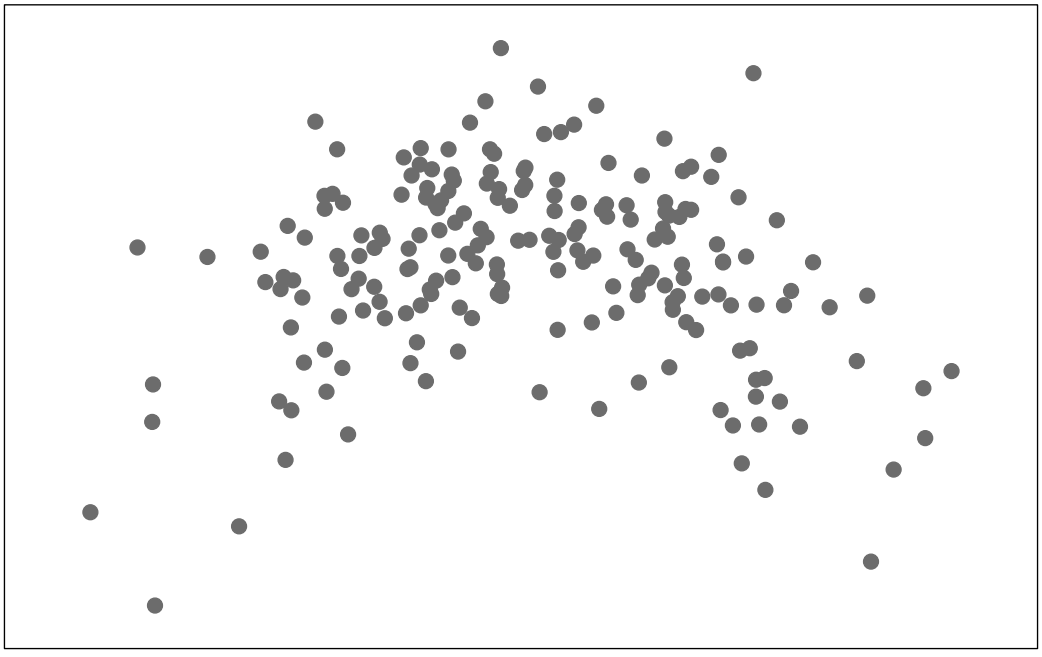
Hexagonal binning

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.

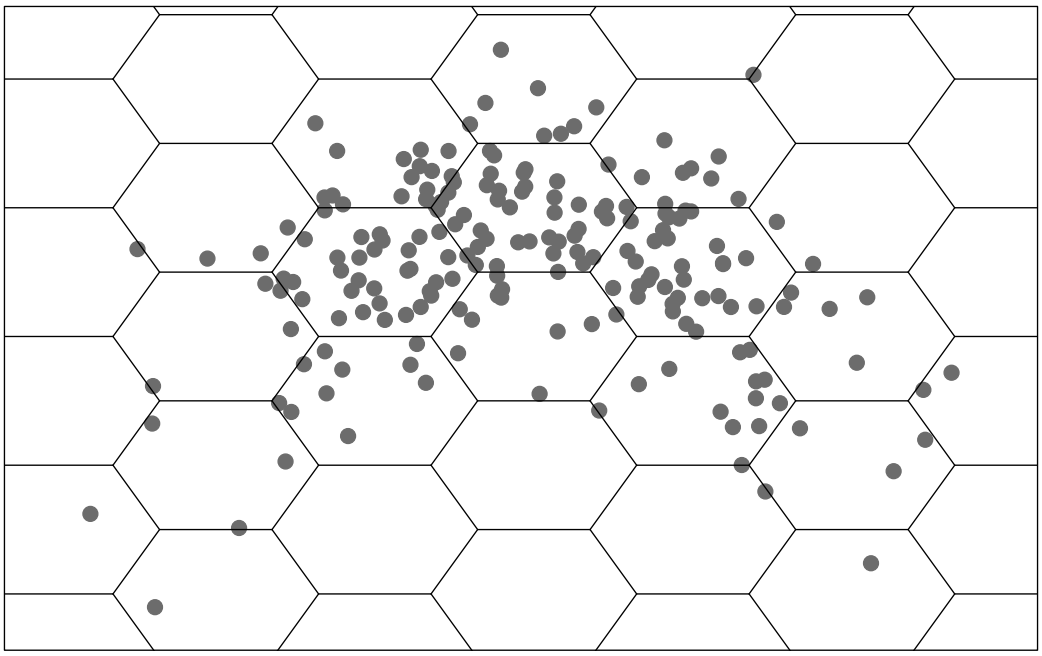
Hexagonal binning

Binning in two dimensions;



Hexagonal binning

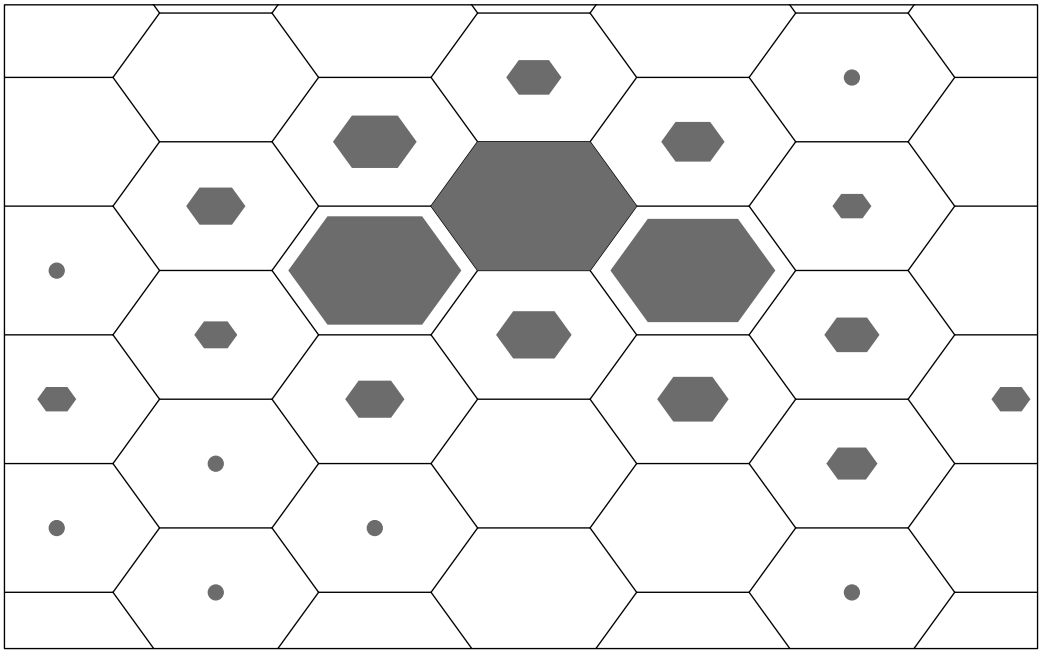
Binning in two dimensions;



2.23

Hexagonal binning

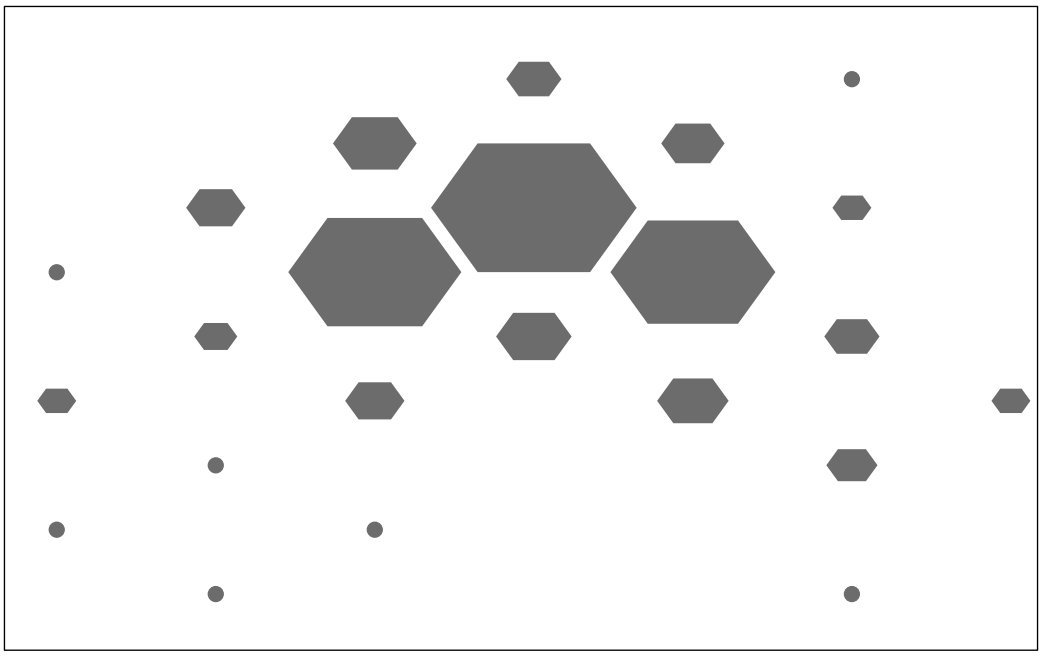
Binning in two dimensions;



2.24

Hexagonal binning

Binning in two dimensions;

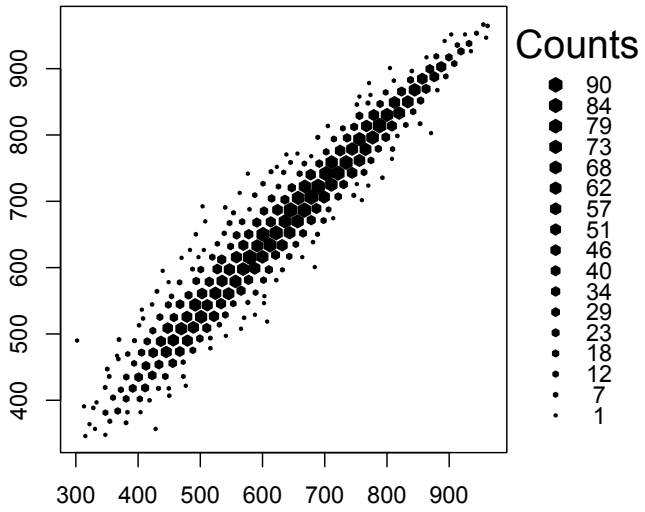


Hexagonal binning

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

```
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

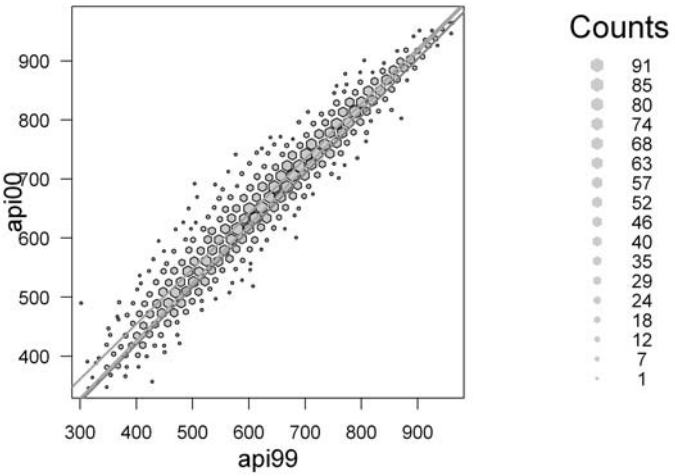
Hexagonal binning

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;




```
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")
```

File formats

Ultimately, we want to output the graph in an appropriate file format. (Cut-and-paste is possible, but not recommended)

R knows more about font sizes and spacing than most users – so first design the graph at the size it will end up, eg:

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

... and, when that's done, write a version to a file

File formats

For example, for a 6×4 PDF file;

```
pdf("myprettypic.pdf", height=4, width=6) # inches
... plotting commands here ...
dev.off()             # close the file
```

Some other formats: (see `?Devices` for a full list)

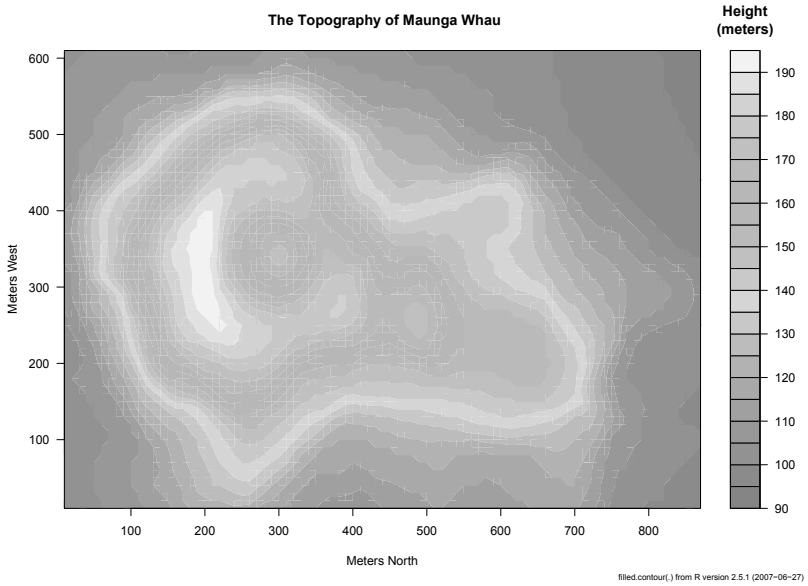
- `jpeg("mypic.jpg", w=6*288, h=4*288, res=288)` – lossy
- `png("mypic.png", w=6*288, h=4*288, res=288)` – lossless

– point size of text can also be manipulated, which can be useful when making posters

PowerPoint, or Word, or LaTeX can all rescale graphs. But when the graph gets smaller, so do the axis labels...

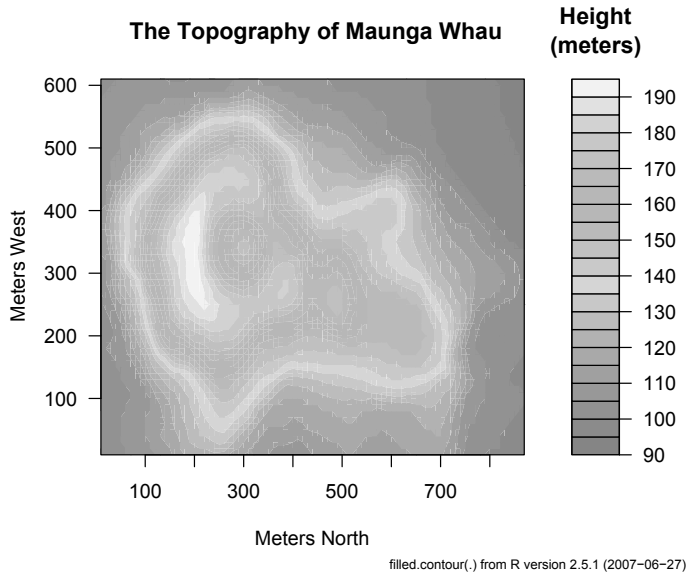
File formats

Created at full-page size (11×8.5 inches)



File formats

Created at 6×5 inches



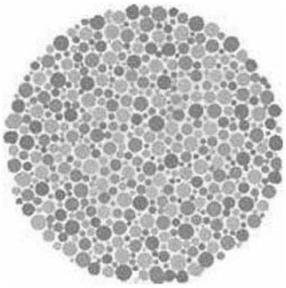
Color schemes

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`

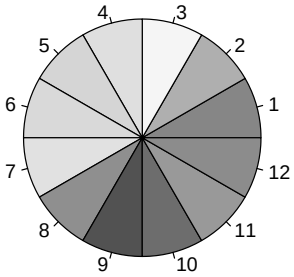
`colorspace` package has color schemes based on straight lines in a perceptually-based color space (rather than RGB).

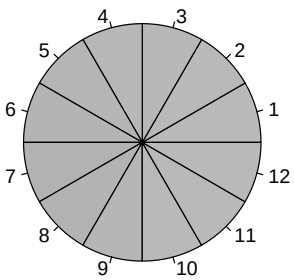
`dichromat` package attempts to show the impact of red:green color blindness on your R color schemes.

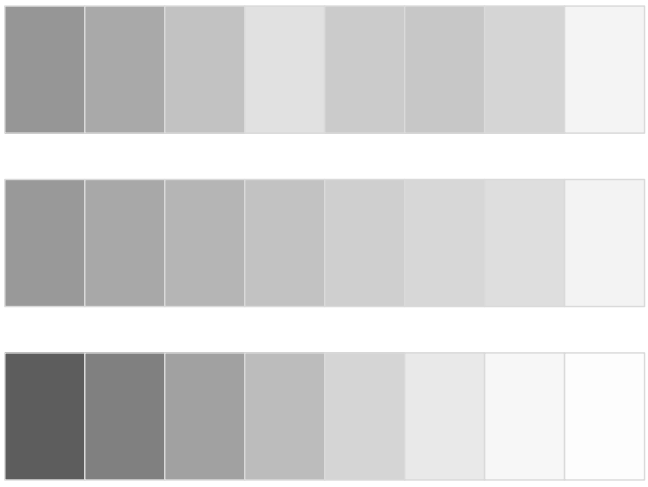


[Code for examples is in file `colorpalettes.R` on course website]

Color choice

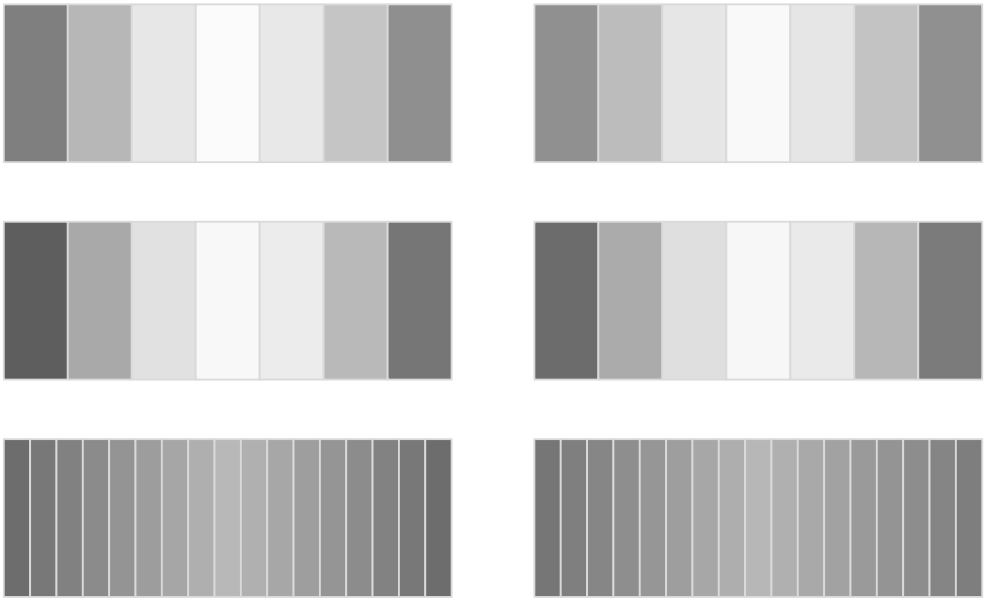






(nb B&W printed copies of this slide may not be helpful!)

Color blindness



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

3. The object system(s)

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Seattle, July 2014

Generics and methods

Many functions in R are generic. This means that the function itself (eg `plot`, `summary`, `mean`) doesn't do anything. The work is done by methods that know how to plot, summarize or average particular types of information.

If you call `summary` on a `data.frame`, R works out that the correct function to do the work is `summary.data.frame` and calls that instead. If there is no specialized method to summarize the information, R will call `summary.default`

You can find out all the types of data that R knows how to summarize with two functions...

Generics and methods

```
> methods("summary")
 [1] summary.Date            summary.POSIXct      summary.POSIXlt
 [4] summary.aov             summary.aovlist      summary.connection
 [7] summary.data.frame      summary.default      summary.ecdf*
[10] summary.factor          summary.glm          summary.infl
[13] summary.lm              summary.loess*       summary.manova
[16] summary.matrix          summary.mlm          summary.nls*
[19] summary.packageStatus*  summary.ppr*         summary.prcomp*
[22] summary.princomp*       summary.stepfun      summary.stl*
[25] summary.table           summary.tukeysmooth*

   Non-visible functions are asterisked
> getMethods("summary")
NULL
```

There are two functions because S has two object systems, for historical reasons.

Generics and methods

Use the `class` argument to see which generics are available

```
> methods(class="lm")
 [1] add1.lm*            alias.lm*
 [3] anova.lm            case.names.lm*
 [5] confint.lm*         cooks.distance.lm*
 [7] deviance.lm*        dfbeta.lm*
 [9] dfbetas.lm*         drop1.lm*
[11] dummy.coef.lm*      effects.lm*
[13] extractAIC.lm*      family.lm*
[15] formula.lm*         hatvalues.lm
[17] influence.lm*       kappa.lm
```

... and many more; packages you load may have their own generics

Methods

The class and method system makes it easy to add new types of information (e.g. survey designs) and have them work just like the built-in ones.

Some standard methods are

- `print`, `summary`: everything should have these
- `plot` or `image`: if you can work out an obvious way to plot the thing, one of these functions should do it.
- `coef`, `vcov`: Anything that estimates parameters and corresponding covariance matrices should have these.
- `anova`, `logLik`, `AIC`: models fitted by maximum likelihood should have these.
- `residuals`: anything that has residuals should have this.

[Informal analogue of Java interfaces]

New classes: S3

Creating a new class is easy

```
class(x) <- "duck"
```

R will now automatically look for the `print.duck` method, the `summary.duck` method, and so on.

There is no formal registration or documentation of the structure of the object. *You* need to make sure that anything of class `duck` can `look.duck`, `walk.duck`, `quack.duck`.

Yes, this is different from Java and C++.

Generic functions: S3

A generic function has a call to `UseMethod()`, which does the method dispatching.

```
> print
function (x, ...)
{
UseMethod("print")
}
```

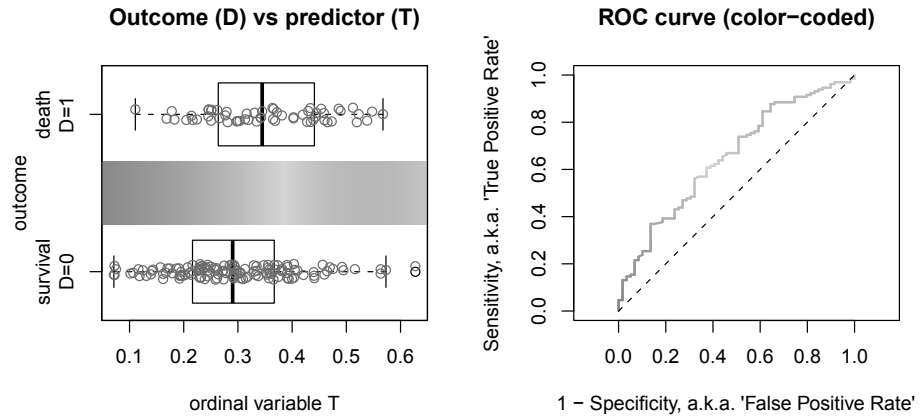
By default, method dispatch is on the first argument. It can be on any (single) argument.

```
> svymean
function (x, design, na.rm = FALSE, ...)
{
    UseMethod("svymean", design)
}
```

Example: ROC curves

The Receiver Operating Characteristic (ROC) curve describes the ability of an ordinal variable T to predict a binary variable D.

The ROC curve graphs $P(T > c|D = 1)$ against $P(T > c|D = 0)$ for every cutpoint c;



Example: ROC curves

Here's a simple way to code it:

```
ROC <- function(test, disease){ #test = e.g. levels of a biomarker
  # where is the curve going to change?
  cutpoints <- c(-Inf, sort(unique(test)), Inf)

  # what values will it take when it does change?
  sensitivity <- sapply(cutpoints,
      function(result){ mean(test>result & disease)/mean(disease)}
      )

  specificity <- sapply(cutpoints,
      function(result){mean(test<=result & !disease)/mean(!disease)}
      )

  # plot the curve, return the coordinates
  plot(1-specificity, sensitivity, type="l")
  abline(0,1,lty=2)

  return(list(sens=sensitivity, spec=specificity))
}
```

Example: ROC curve

Here's a more efficient version of the calculation;

```
drawROC<-function(T,D){
  DD <- table(-T,D)
  tpr <- cumsum(DD[,2])/sum(DD[,2])
  fpr <- cumsum(DD[,1])/sum(DD[,1])
  plot(fpr, tpr, type="l")
}
```

Note that we use the vectorized `cumsum()` rather than the implied loop of `sapply()`.

We want to make this return an ROC object that can be plotted and operated on in other ways

ROC curve object

```
ROC<-function(T,D){
  DD <- table(-T,D)
  tpr <- cumsum(DD[,2])/sum(DD[,2])
  fpr <- cumsum(DD[,1])/sum(DD[,1])
  rval <- list(tpr=tpr, fpr=fpr,
               cutpoints=rev(sort(unique(T))),
               call=sys.call())
  class(rval)<-"ROC"
  rval
}
```

Instead of plotting the curve we return the data needed for the plot – plus some things that might be useful later; `sys.call()` is a copy of the call.

Methods

We need a `print` method to stop the whole contents of the object being printed

```
print.ROC<-function(x,...){
   cat("ROC curve: ")
   print(x$call)
}
```


Methods

A plot method

```
plot.ROC <- function(x, xlab="1-Specificity",
                         ylab="Sensitivity", type="l",...){
        plot(x$fpr, x$tpr, xlab=xlab, ylab=ylab, type=type, ...)
}
```

We specify some graphical parameters in order to set defaults for them. Others are automatically included in `...`.

Methods

We want to be able to add lines to an existing plot

```
lines.ROC <- function(x, ...){
lines(x$fpr, x$tpr, ...)
}
```

and also be able to identify cutpoints by clicking on a graph

```
identify.ROC<-function(x, labels=NULL, ...,digits=1)
{
  if (is.null(labels))
    labels<-round(x$cutpoints,digits)
  identify(x$fpr, x$tpr, labels=labels,...)
}
```


Syntax notes

Methods should have at least the same arguments as the generic, in the same order, with the same defaults (so the first argument to a `print` method is `x`, but to a `summary` method is `object`).

For inheritance to work, methods must have a `...` argument to allow unknown arguments to be ignored.

The language does not enforce these requirements, but the package checking system does.

Inheritance

The `class` attribute can be a vector, e.g. `c("glm", "lm")`

R will look for a method for each element in turn until it finds one.

Inside a method, use `NextMethod()` to call the next method in the inheritance.

Inheritance is not used much: statisticians extend by generalization, not by specialization. The relationship of `glm` to `lm` should really be delegation, not inheritance.

An exception is data infrastructure (e.g. Bioconductor), which tends to use S4 methods.

S4 classes

Introduced in version 4 of Bell Labs' S, since extended and refined in R.

Still uses generic functions, with methods belonging to functions rather than to classes.

- Formal declaration of class structure: `setClass()`
- Formal declaration of methods: `setMethod()`
- Multiple dispatch
- Multiple inheritance

Example: ROC curve

Define `ROC` class

```
setClass("ROC",
       representation(tpr="numeric",fpr="numeric",
          cutpoints="numeric",call="call")
)
```

Or we could factor out the 'curve' structure and declare

```
setClass("xycurve", representation(x="numeric", y="numeric"))
setClass("ROC",  contains="xycurve",
       representation(cutpoints="numeric",call="call")
)
```

taking advantage of inheritance

Example: ROC curve

Other options include validity checks at object creation

```
setClass("ROC",
     representation(tpr="numeric",fpr="numeric",
          cutpoints="numeric",call="call"),
     validity=function(object){
           if(length(object@tpr)!=length(object@fpr) ||
              length(object@tpr)!=length(object@cutpoints))
                  return("length mismatch")
           if(any(object@tpr>1) || any(object@fpr>1) ||
              any(object@tpr<0) || any(object@fpr<0))
                 return("outside [0,1]")
           return(TRUE)
        })
```

Example: ROC constructor

Objects are created with `new()`; code is otherwise the same.

```
ROC <- function(T,D){
        DD <- table(-T,D)
        tpr <- cumsum(DD[,2])/sum(DD[,2])
        fpr <- cumsum(DD[,1])/sum(DD[,1])
        new("ROC",tpr=tpr, fpr=fpr,
           cutpoints=rev(sort(unique(T))),call=sys.call())
        }
```


Example: ROC methods

`setMethod` specifies a method for a generic function and an argument signature giving the classes of all the arguments used for dispatch.

Use `@` to refer to slots (not `$`), otherwise similar to S3

```
setMethod("show",signature="ROC",
    function(object){
      cat("S4 ROC curve:")
      print(object@call)
    }
)
```

(Note that S4 uses `show` rather than `print`)

This generic has only one argument, so the signature is a single string.

Example: ROC methods

```
setMethod("plot",signature("ROC","ANY"),
   function(x,y,type="l", xlab="1 - Specificity",
            ylab="Sensitivity",...){
       plot(1-x@spec, x@sens, type=type, xlab=xlab, ylab=ylab ,...)
    }
)
```

This generic, for `plot()`, has two arguments (x, y).

The signature specifies this method when x is ROC and y is ANYthing.

Example: ROC methods

`lines()` is *not* an S4 generic, but we can re-use the S3 version;

```
setGeneric("lines")
setMethod("lines",signature("ROC"),
    function(x,...){
        lines(1-x@spec, x@sens,...)
    }
)
```

`setGeneric()` creates an S4 generic that defaults to calling the original `lines()` function.

Multiple dispatch

Generic functions with method choice based on all arguments are strictly more expressive than the Java/C++ model of methods belonging to classes.

Java/C++ style can be translated mechanically:
`object.method(arg1, arg2)` maps to `generic(object, arg1, arg2)`

The price is slower method lookup, but most of the cost is at installation time, and slower method lookup is inevitable for a system that allows one package to declare methods for another package's objects.

Multiple dispatch

Generic function style ;

- allows symmetric treatment of argument, e.g. matrix multiplication: `multiply(A, B)` not `A.rightmultiply(B)` or `B.leftmultiply(A)`
- allows the programmer to describe whether methods for two objects are actually doing the same thing.
- allows first-class functions, which mathematicians and statisticians like.

Multiple dispatch

`filter()` in the `flowCore` package for flow cytometry has two arguments: a data set, and an object specifying a subsetting operation. Methods are dispatched based on both arguments.

```
> showMethods("filter")
Function: filter (package flowCore)
x="flowFrame", filter="filter"
x="flowFrame", filter="filterSet"
x="flowSet", filter="filter"
x="flowSet", filter="filterList"
x="flowSet", filter="filterSet"
x="flowSet", filter="list"
```

The `Matrix` package has 70 multiplication methods for different combinations of matrix types (`showMethods("%*%")`)

More complex example

Class `AnnDbBimap` is used in the `AnnotationDbi` package in Bioconductor, to provide conversions from one system of identifiers to another (eg probe ids, gene ids, gene symbols, GO categories). More details in Session 9.

Examine the structure and inheritance relationships of the class with `getClass()`

More complex example

```
> getClass("AnnDbBimap")
Class "AnnDbBimap" [package "AnnotationDbi"]
Slots:
Name:     L2Rchain   direction       Lkeys       Rkeys  ifnotfound   datacache
Class:        list     integer   character   character        list environment
Name:   objName    objTarget
Class:  character  character

Extends:
Class "Bimap", directly
Class "AnnDbObj", directly
Class "AnnObj", by class "AnnDbObj", distance 2

Known Subclasses:
Class "InpAnnDbBimap", directly
Class "GoAnnDbBimap", directly
Class "GOTermsAnnDbBimap", directly
Class "AnnDbMap", directly
Class "ProbeAnnDbBimap", directly
Class "Go3AnnDbBimap", by class "GoAnnDbBimap", distance 2
Class "IpiAnnDbMap", by class "AnnDbMap", distance 2
Class "AgiAnnDbMap", by class "AnnDbMap", distance 2
Class "ProbeAnnDbMap", by class "AnnDbMap", distance 2    [..etc..]
```


More complex example

```
setClass("AnnDbBimap",
    contains=c("Bimap", "AnnDbObj"),
    representation(
        L2Rchain="list",            # list of L2Rlink objects
        direction="integer",        # 1L for left-to-right,
        Lkeys="character",
        Rkeys="character",
        ifnotfound="list"
    ),
    prototype(
        direction=1L,               # left-to-right by default
        Lkeys=as.character(NA),
        Rkeys=as.character(NA),
        ifnotfound=list()           # empty list => raise an error
    )
)
```

More complex example

Multiple inheritance used for 'mix-in' behavior:

- `"Bimap"` is a virtual class that is used only to define a set of methods for its subclasses
- Some implementation is inherited from `"AnnDBObj"`

is(), as()

- `is(object, "class")` tests whether `object` inherits from `"class"`
- `as(object, "class")` attempts to convert `object` to `"class"`. This will only work if `object` inherits from `"class"` or a conversion function has been provided with `setAs()`

```
setAs("ROC", "numeric",
   function(from){ cbind(from@fpr, from@tpr, from@cutpoints) }
)
```

New generics

When creating a completely new function with methods, you need to specify the arguments to the generic function:

```
setGeneric("increment",
  function(object, step, ...)
    standardGeneric("increment")
)
```

Recall in S3 we'd have just defined `increment.ROC`, `increment.lm`, etc, which a generic `increment()` function would pick from with `UseMethod("increment")`

In S4, methods for `increment` will have a signature specifying classes for `object` and `step`

Some Bioconductor infrastructure

- `eSet`: basic data structure including genomic data, phenotype, metadata; specializes to `ExpressionSet`, `SnpSet`, others
- `IRanges`: for manipulating numeric sequences.
- `Xstring`: stores long strings (specializes to `DNAstring`, `RNAstring`, `AAstring`)
- `AnnDbObj`, `Bimap`: Storage and lookup of annotation data

eSet

assayData Contains matrices with equal dimensions, and with column number equal to `nrow(phenoData)`. Class:AssayData-class

phenoData Contains experimenter-supplied variables describing sample (i.e., columns in assayData) phenotypes. Class: AnnotatedDataFrame-class

featureData Contains variables describing features (i.e., rows in assayData) unique to this experiment. Use the annotation slot to efficiently reference feature data common to the annotation package used in the experiment. Class: AnnotatedDataFrame-class

experimentData Contains details of experimental methods. Class: MIAME-class

annotation Label associated with the annotation package used in the experiment. Class: character

protocolData Contains microarray equipment-generated variables describing sample (i.e., columns in assayData) phenotypes. Class: AnnotatedDataFrame-class

eSet

`eSet` has accessor functions to extract or modify the data; the slots should not be used directly.

`eSet` is a virtual class that abstracts a set of data properties. Actual objects must be defined using a subclass of `eSet`, and `new("eSet")` is an error.

`ExpressionSet` is a subclass where the `assayData` slot contains one or more matrices (all the same size) for gene expression data

`SnpSet` is a subclass where the `assayData` slot contains two matrices of the same size, for SNP calls and call probabilities

Sequences

The `IRanges` package provides an alternative infrastructure to vectors, mostly as virtual classes.

- `Sequence`: virtual class for (potentially large) vectors
- `View`: virtual class for subsequences of a `Sequence`
- `Ranges`: sets of intervals of consecutive integers.
- `IntervalTree`: find overlaps between two `Ranges`

Biostrings package

`DNAString` and `RNAString` represent genomic sequences, `AAString` represents an amino-acid sequence

```
>  d <- DNAString("TTGAAAA-CTC-N")
>   length(d)
[1] 13
>   alphabet(d)                 # DNA_ALPHABET
 [1] "A" "C" "G" "T" "M" "R" "W" "S" "Y" "K" "V" "H" "D" "B" "N" "-" "+"
>   alphabet(d, baseOnly=TRUE)  # DNA_BASES
[1] "A" "C" "G" "T"
>
> d
  13-letter "DNAString" instance
seq: TTGAAAA-CTC-N
> reverseComplement(d)
  13-letter "DNAString" instance
seq: N-GAG-TTTTCAA
> RNAString(d)
  13-letter "RNAString" instance
seq: UUGAAAA-CUC-N
```

Efficiency

The underlying sequence is not copied on assignment.

The `subseq()` function (from `IRanges`) makes a view of a subset of the string without copying

...allows manipulation of whole-chromosome sequences.

```
>   data(yeastSEQCHR1)
>   yeast1 <- DNAString(yeastSEQCHR1)
> str(yeast1)
Formal class 'DNAString' [package "Biostrings"] with 6 slots
  ..@ shared         :Formal class 'SharedRaw' [package "IRanges"] with 2 slots
  .. .. ..@ xp                   :<externalptr>
  .. .. ..@ .link_to_cached_object:<environment: 0x1cf45fdc>
  ..@ offset         : int 0
  ..@ length         : int 230208
  ..@ elementMetadata: NULL
  ..@ elementType    : chr "ANY"
  ..@ metadata       : list()
```


Efficiency

```
>   dinucleotideFrequency(yeast1)
   AA    AC    AG    AT    CA    CC    CG    CT    GA
23947 12493 13621 19769 15224  9218  7089 13112 14478
   GC    GG    GT    TA    TC    TG    TT
 8910  9438 12938 16181 14021 15617 24151
>   trinucleotideFrequency(yeast1)
 AAA  AAC  AAG  AAT  ACA  ACC  ACG  ACT  AGA  AGC  AGG
8576 4105 4960 6306 3924 2849 2186 3534 4537 2680 2707
 AGT  ATA  ATC  ATG  ATT  CAA  CAC  CAG  CAT  CCA  CCC
3697 5242 3849 4294 6384 5147 2722 3091 4264 3696 1622
 CCG  CCT  CGA  CGC  CGG  CGT  CTA  CTC  CTG  CTT  GAA
1444 2456 2158 1380 1446 2105 2755 2556 3074 4727 5437
 GAC  GAG  GAT  GCA  GCC  GCG  GCT  GGA  GGC  GGG  GGT
2384 2645 4012 2993 1960 1259 2698 2983 1905 1594 2955
 GTA  GTC  GTG  GTT  TAA  TAC  TAG  TAT  TCA  TCC  TCG
3490 2455 2798 4195 4787 3282 2925 5187 4611 2786 2200
 TCT  TGA  TGC  TGG  TGT  TTA  TTC  TTG  TTT
4424 4800 2945 3691 4181 4694 5161 5451 8845
```

Efficiency

```
>   ## Get the least and most represented 6-mers:
>   f6 <- oligonucleotideFrequency(yeast1, 6)
>   f6[f6 == min(f6)]
CCCGGG
     3
>   f6[f6 == max(f6)]
TTTTTT
   705
```


Comparisons

The S3 system has less overhead, is more widely understood, and is very slightly faster. It is still useful for single-programmer work.

The S4 system is better for multi-person efforts or code that is likely to be reused by others.

- Formal definition of class structure, so the contents of an object can be relied on
- Registration of methods means that reflection (looking up what methods are available) is reliable
- Multiple inheritance is useful for mix-in behavior
- Multiple dispatch is only rarely important, but when you need it you really need it

4. Extended Exercise

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Seattle, July 2014

This session

... does not have a lecture component.

Instead, we'll do a more extensive programming project

5. Making packages

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Seattle, July 2014

Packages

The most important innovation in R over S-PLUS was the package system

- Standard format for distributing code contributions, allows anyone to contribute.
- Cross-platform production of binaries
- Documentation format integrated into the R help system
- Increasing number of QA checks to improve code quality.
- Dependency tracking

Main documentation reference: *Writing R Extensions*

Package format

- `DESCRIPTION`, `NAMESPACE`: overall structure
- `R/` R code
- `src/` Code to be compiled
- `man/` help pages
- `vignettes/` Vignettes
- `data/` example data sets.
- `tests/` extra tests

R function `package.skeleton()` makes the basic structure for you.

DESCRIPTION

```
Package: survey
Title: analysis of complex survey samples
Description: Summary statistics, generalised linear models, cumulative link mode
ls, Cox models, loglinear models, and general maximum pseudolikelihood estimatio
n for multistage stratified, cluster-sampled, unequally weighted survey samples.
 Variances by Taylor series linearisation or replicate weights. Post-stratificat
ion, calibration, and raking. Two-phase subsampling designs. Graphics. Predictiv
e margins by direct standardization. PPS sampling without replacement. Principal
 components, factor analysis.
Version: 3.22-2
Author: Thomas Lumley
Maintainer: Thomas Lumley <tlumley@u.washington.edu>
License: GPL-2 | GPL-3
Depends: R (>= 2.2.0)
Suggests: survival, MASS, KernSmooth, hexbin, mitools, lattice, RSQLite, RODBC,
quantreg, splines, Matrix, multicore
URL: http://faculty.washington.edu/tlumley/survey/
```

DESCRIPTION

- `Package`: name of package
- `Title`: one-line description
- `Description`: one-paragraph description
- `Version`: X.YY-zz version number
- `Author`, `Maintainer`: the maintainer is the person to contact, needs email address
- `License`: preferably a standard abbreviation for a standard license if you are going to distribute the package.
- `Depends`: other packages needed to install this one, version of R needed
- `Suggests`: other packages this can make use of but doesn't need
- `URL`: web page for package

Also `Imports` for other packages imported in the `NAMESPACE`, `LazyLoad: yes` to allow loading-on-demand, `Encoding:` if the character set is not ASCII (eg `UTF-8`).

NAMESPACE

Controls which functions in the package are exported and visible to the user.

- Fewer name collisions and accidental redefinitions of a function
- Internal functions not intended for the user (and, eg, with less error checking) can be hidden
- S3 methods can be exported as methods rather than by name, so they can only be called via `UseMethod`
- Compiled code loaded in the NAMESPACE file will be found by `.C` and `.Call` only for functions in the package [session 7]

Example: `leaps` package

```
useDynLib(leaps)
export(regsubsets, leaps)
S3method(regsubsets, biglm)
S3method(regsubsets, formula)
S3method(regsubsets, default)
S3method(summary, regsubsets)
S3method(print, summary.regsubsets)
S3method(print, regsubsets)
S3method(plot, regsubsets)
S3method(coef, regsubsets)
S3method(vcov, regsubsets)
```


Example: `annotate` package [edited]

```
importClassesFrom(Biobase, eSet)

importMethodsFrom(AnnotationDbi,
                  Definition, GOID, Secondary,  Synonym, as.list, colnames, dbmeta
                  eapply,  exists, get,  ls, mappedRkeys, mget, ncol,  nrow,
                  Ontology, revmap, Term)
importMethodsFrom(Biobase,
                  annotation,  contents,  exprs, featureNames)

importFrom(Biobase,
           addVigs2WinMenu)
importFrom(graphics,
           abline, identify, plot)
importFrom(utils
           browseURL,  compareVersion, packageDescription)

exportClasses( chromLocation,  FramedHTMLPage,homoData,
              HTMLPage,  pubMedAbst )

exportMethods( abstText, articleTitle, authors, chromInfo, chromLengths,
              chromLocs, chromNames, dataSource, Definition,  fileName,
              geneSymbols, GOID,  homoACC,   homoHGID, homoLL,
         [snip]              )
```

Example: `annotate` package [edited]

```
export(.buildAnnotateOpts, .getIdTag, .getNcbiURL, .handleXML,
       .pmfetch,   .transformAccession, ACC2homology, accessionToUID, ACCNUMStats,
       annPkgName, aqListGOIDs,  buildChromLocation,  buildPubMedAbst,
       checkArgs,  chrCats,  compatibleVersions,  createLLChrCats,
       createMAPIncMat,  dropECode, filterGOByOntology,  findChr4LL,
       findNeighbors,    genbank,    genelocator,  getAnnMap,
  [snip]
       setRepository, getRepositories, clearRepository, isValidKey, allValidKeys,
       updateSymbolsToValidKeys
       )
```

`import` makes available all functions exported from another package, but does not load that package (the functions are not visible to the user), `importFrom` imports just the specified functions.

`importClassesFrom`, `importMethodsFrom`, `exportClasses`, `exportMethods` are for S4 classes and methods

R/ subdirectory

R code, in files with extension `.r`, `.R`, `.q`, `.Q`, `.s` or `.S`

[Restriction on file names is to prevent, eg, editor backup files being accidentally used.]

Files are sorted in ASCII alphabetical order and concatenated into one file.

src/ subdirectory

R will try to compile

- C, Fortran 77: R uses these, so they work portably
- C++: Most systems with a C compiler also provide a compatible C++ compiler.
- Fortran 90, 95: Not all systems have Fortran 90/95 compilers. The linker needed for Fortran 95 is sometimes incompatible with C++ code.

Other types of code can be compiled by including a `Makefile`, but this tends not to be portable.

man/ subdirectory

Documentation for every user-visible object

- functions
- data objects
- S4 classes

Rd format

Similar to LaTeX

Markup language with tags distinguished by backslash: `\usage`, `\title`, `\arguments`

`prompt()` (or `package.skeleton()`) makes a skeleton help file for you to fill in, so you don't need to know that much of the format.

[show examples]

Vignettes

Help pages describe what a single function does, or the structure of a single class.

Vignettes describe how to do a task. They are longer than help pages and don't need to describe all the possibilities.

Vignettes are written in LaTeX or Markdown with specially formatted chunks of R. The `Sweave()` function (in the `tools` package, or the `knit()` function in the `knitr` package) extracts the R code chunks from the document, runs them, and then puts the output back into the document.

`latex` or `pdflatex` can then be used to make a PDF document.

Since the output in the document is automatically generated from the input, it is reliably correct, without cut-and-paste or version errors... a.k.a reproducible research!

data/ directory

This contains saved data sets that can be loaded with the `data()` function and used in help page examples.

Typically these are R binary data files created with the `save()` function and having a `.rda` or `.Rdata` extension, but R source files (`.R` extension) or white-space separated tables (`.txt` extension) are also allowed.

If the `DESCRIPTION` file has the line `LazyData: yes` the data files are automatically loaded when their names are used, and the `data()` function is not necessary.

Steps in making a package

1. You probably have at least partial code. Use `package.skeleton()` to set up a package directory with this code. Edit the help files. Put any code to be compiled in `src`

2. `R CMD INSTALL thepackage` installs the package

3. Run the functions and check that it works

4. `R CMD check thepackage` runs the QC tools

5. Fix all the errors and warnings, then go to step 2

6. `R CMD build thepackage` makes a package file for distribution to other people

R CMD check

Package QC checks

- Correct package structure, including portability of character sets.
- Documentation consistent with code
- Language rules not enforced by the parser, such as agreement in argument names between S3 methods and generics
- Absence of common coding errors
- All the help page examples run without errors
- Any tests in the `tests/` directory run correctly

Packages submitted to CRAN or Bioconductor should complete `R CMD check` in at most a minute or so: it may not be possible to put all your package tests in the `tests/` directory.

R-forge and winbuilder

`http://win-builder.r-project.org/` will compile Windows binaries of your (working, checked) package: upload by ftp, it sends email when compilation is done.

`http://r-forge.r-project.org/` hosts R package development:

- version control via `subversion`, including browseable code on web pages.
- mailing lists
- bug tracker
- daily build/check on Windows, Mac, 32-bit and 64-bit Linux

6. XML

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Seattle, July 2014

Complex text data

XML is a format for constructing and describing data formats for plain text data.

- HTML (almost)
- SVG graphics format
- MS Word and Open Office files
- MAGE-ML for microarray data
- Web service communications
- KML for Google Earth

Components of XML

Tag A markup construct that begins with < and ends with >. Tags come in three flavors: start-tags, for example `<section>`, end-tags, for example `</section>`, and empty-element tags, for example `<line-break/>`.

Element A start-tag and matching end-tag and what is between them. The characters between the start- and end-tags,may contain markup, including other elements, which are called child elements. An example of an element is `<Greeting>Hello, world.</Greeting>` Empty-element tags also count as elements: `<line-break/>`.

Attribute A name and value stored within the start tag, eg in HTML `<a href="link.html">` the `href` is an attribute with value `link.html`.

Schemas

It is straightforward to parse any XML document into a tree of elements with no additional information. An XML schema or DTD defines a specific XML language by specifying which tags and attributes are valid, and what nesting is allowed.

```
<?xml version="1.0" encoding="UTF-8"?>
<kml xmlns="http://www.opengis.net/kml/2.2">
<Placemark>
  <name>SISG </name>
  <description>Registration, coffee and snacks</description>
  <Point>
    <coordinates>-122.3093834881461,47.6496137385853,0</coordinates>
  </Point>
</Placemark>
</kml>
```

Schemas

The KML schema is defined at http://schemas.opengis.net/kml/2.2.0/ogckml22.xsd.

- `kml` tag defines a KML document, `xmlns` defines a namespace, to avoid confusion between different definitions of tags such as `point`
- `placemark` tag defines a geographic location. There are many ways to do this, we have the simplest one
  - `name` tag for name of the place
  - `description` tag for longer description
  - `point` tag for location
    - `coordinates` tag relative to WGS84 reference spheroid.

Making XML

Simple XML such as a single KML point can be created just by manipulating strings.

```
kml<-function(conn,lat,lon,name){
     ss<-gsub(" ","",paste("<coordinates>",lon,",",lat,
         ",400</coordinates>"))
     cat("<?xml version=\"1.0\" encoding=\"UTF-8\"?>
        <kml xmlns=\"http://earth.google.com/kml/2.1\">
        <Placemark>
        <name>",name,"</name>
        <Point>",
        ss,
        " </Point>
       </Placemark>
        </kml>\n",
     file=conn)
}
```

Making XML

For more complex XML, prefer software that understands the tree structure.

`XML` package provides routines for reading and writing XML

To create XML, use `xmlOutputBuffer` to start a document.

The returned value is an XML object with a list of functions to add tags, close tags, add a whole child subtree, and print out the current state of the document.

[This also shows how to implement a familiar style of object-oriented programming in R]

Making XML

```
con <- xmlOutputBuffer(nsURI="http://www.opengis.net/kml/2.2",
      nameSpace="")

con$addTag("kml",close=FALSE)
  con$addTag("Placemark",close=FALSE)
    con$addTag("name",  "SISG")
    con$addTag("description", "Registration, Coffee, Snacks")
    con$addTag("Point", close=FALSE)
      con$addTag("coordinates", "-122.3093834881461,47.6496137385853,0")
    con$closeTag() # Point
  con$closeTag() # Placemark
con$closeTag() #kml

cat(con$value())
```

SVG

W3C Vector graphics format, allows animations, zooming, etc. Only partial support in most browsers.

R has `svg()` graphics device; `RSVGDevice`, `RSVGToolTips` packages.

```
for(i in 1:length(or)) {
   setSVGShapeToolTip(title=gene[i],
     desc1=snp[i],
     desc2=if(abs(lor[i]/se[i])>qnorm(0.5/n,lower.tail=FALSE))
         qvals[i]
     else
         NULL)
     setSVGShapeURL(paste("http://pga.gs.washington.edu/data",
         tolower(gene[i]),sep="/"))
     points(prec[i],lor[i], cex=1, pch=19, col='grey')
     invisible(NULL)
 }
```

Creates an SVG file showing SNP associations, with clickable links for annotation.

SVG file

File begins

```
<svg version="1.1" baseProfile="full" width="722.70" height="578.16"
viewBox="0,0,722.70,578.16" onload="Init(evt)" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink"
xmlns:ev="http://www.w3.org/2001/xml-events">
 <title>R SVG Plot</title>
 <desc>R SVG Plot with tooltips! (mode=2)</desc>
```

specifying the type of XML format and where its specification can be found.

Each point looks like

```
 <a href="http://pga.gs.washington.edu/data/f10">
  <circle cx="333.66" cy="365.76" r="2.34" stroke-width="1px" stroke="#BEBEBE"
             fill="#BEBEBE" stroke-opacity="1.000000" fill-opacity="1.000000">
   <title>F10</title>
   <desc1>rs3211744</desc1>
  </circle>
 </a>
```

Reading XML

XML package has two approaches to reading XML

- whole file into memory at once ('DOM')
- analyse and discard each node as it is read. ('event')

```
library(XML)
funnelplot <- xmlTreeParse("svgplot1.svg", useInternal=TRUE)
```

The `funnelplot` variable now holds the whole XML (SVG) document in a tree structure.

Manipulating XML

Read the x coordinate of each point (the `cx` attribute of the `<circle>` elements, nested in the `<a>` elements, nested in the `<svg>` element)

```
xpathApply(funnelplot, "/s:svg/s:a/s:circle",xmlGetAttr, "cx",
    namespaces=c(s="http://www.w3.org/2000/svg"))
```

`xpathApply` finds nodes in the tree satisfying the path `<svg> <a href> <circle>` and then applies `xmlgetAttr()` to each one, with the argument `cx`, to extract the x-axis coordinate. This returns a list of 171 numbers.

The `s:` prefix refers to the namespace, where the format is defined. The `namespaces` argument lets us abbreviate the full namespace `http://www.w3.org/2000/svg` by `s`.

Manipulating XML

Find the points corresponding to the gene TFPI:

```
xpathApply(funnelplot, "/s:svg/s:a/s:circle[s:title='TFPI']",
    namespaces=c(s="http://www.w3.org/2000/svg"))
```

`xpathApply` finds nodes in the tree satisfying the path `<svg> <a href> <circle>` and the condition `title='TFPI'`. Since we do not specify a function to apply to each node, a list of nodes is returned.

Example: CRANberries

CRANberries is an RSS feed that describes new and updated packages on CRAN.

These `XML` commands can be used to read the file, and count the number of new packages in the past week or so;

```
cr  <- readLines("http://dirk.eddelbuettel.com/cranberries/index.rss",
     encoding="ISO-8859-1")
cr <- iconv(cr, to="UTF-8")
crt <- xmlTreeParse(cr, useInternal=TRUE)
titles <-xpathApply(crt, "/rss/channel/item/title")
titlelist <- sapply(titles, xmlToList)
titlelist

length(grep("New package", titlelist)) # how many new ones?
```

biomaRt

Bioconductor's `biomaRt` package sends queries to the Ensembl database server (`www.ensembl.org`). Ensembl stores genome annotation data for about 50 species, including cross-species information.

A `biomaRt` operation like:

```
ens <- useMart("ensembl")
#listDatasets(ens)
human <- useDataset("hsapiens_gene_ensembl",mart=ens)
mouse <- useDataset("mmusculus_gene_ensembl",mart=ens)
getLDS(attributes = c("hgnc_symbol","chromosome_name",
           "start_position"),
        filters = "hgnc_symbol", values = "NOX1",
        mart = human,
      attributesL =c("chromosome_name","start_position"),
        martL = mouse)
```

translates to

biomaRt

```
<?xml version='1.0' encoding='UTF-8'?>
<!DOCTYPE Query>
<Query  virtualSchemaName = 'default' uniqueRows = '1'
         count = '0' datasetConfigVersion = '0.6'
         requestid= "biomaRt">
   <Dataset name = 'hsapiens_gene_ensembl'>
     <Attribute name = 'hgnc_symbol'/>
     <Attribute name = 'chromosome_name'/>
     <Attribute name = 'start_position'/>
     <Filter name = 'hgnc_symbol' value = 'NOX1' />
   </Dataset>
   <Dataset name = 'mmusculus_gene_ensembl' >
      <Attribute name = 'chromosome_name'/>
      <Attribute name = 'start_position'/>
   </Dataset>
</Query>
```

biomaRt

The `biomaRt` package constructs the XML by pasting character strings rather thafsn with the `XML` package.

It uses the `RCurl` package to post the XML request to the webserver and collect the response, which is a table of strings.

Modifying XML

More sophisticated XML interaction in the `SVGAnnotation` package, from `http://www.omegahat.org/SVGAnnotation/`, which reads in and modifies SVG graphs produced by the `svg()` graphics device.

Examples:

- add tooltips
- link points across panels of a graph
- use radio buttons to show/hide elements of a graph

Not very portable yet: the Javascript/SVG combination only works on a few browsers.

Uses `newXMLNode()` to add elements anywhere in an XML document.

7. Embedding C code

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Seattle, July 2014

Why C?

- Some tasks are slow and require explicit loops that can't be vectorized away.
- C is simple and allows for efficient implementation of exactly the kinds of algorithms that R is no good at.
- C is standardized, portable, and has good-quality free compilers on effectively all platforms.

You can also embed C++, Fortran, Java, Perl, Python, or even OCaml

A little example

The convolution of two sequences $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$ is the sum

$$z_i = \sum_{j+k=i} x_j y_k$$

If x and y are probability mass functions, z is the probability mass function of the sum, so this computation is useful in statistical calculations.

A simple R version is

```
m <- length(x)
n <- length(y)
z <- numeric(m+n)
for(j in 1:m){
    for (k in 1:n){
         z[j+k-1] = z[j+k-1] + x[j]*y[k]
    }
}
```


A little example

One loop can be removed easily, but removing both loops doesn't seem to be possible (without using $m \times n$ memory), so convolution is a good candidate for translating to C.

```
void convolve(double *x, int *n, double *y, int *m, double *z){
       int i, j, nz = *m + *n - 1;
       for(i = 0; i < nz; i++) z[i] = 0.0;

       for(i = 0; i < *n; i++) {
         for(j = 0; j < *m; j++){
           z[i + j] += x[i] * y[j];
         }
       }
     }
```

The C code is very similar to the R code, another indication that the R code will be slow.

Notes:

- All arguments are passed as pointers, since all R data types are vectors.
- Lengths of vectors can't be determined in C, so need to be passed in.
- Return type is `void`, so values are returned by modifying arguments
- The arguments are copies of the R objects, not the originals.

Compiling and linking

On Unix or Mac OS, or on Windows with the `Rtools` toolchain, from the OS command line

```
R CMD SHLIB convolve.c
```

to make a dynamic library (`convolve.dll` under Windows, `convolve.so` under most Unix).

In R

```
dyn.load("convolve.so")
```

or

```
dyn.load(paste("convolve", .Platform$dynlib.ext, sep=""))
```

for portability. [Or make a package]

Other compilers

You can make packages with almost any C compiler. On Unix-like systems this typically just works: the C binary interface is standardized.

On Windows there is less standardization and you need the right compiler options. Instructions for some popular ones under Windows. are at http://www.stats.uwo.ca/faculty/murdoch/software/compilingDLLs/

Not all compilers will correctly compile R itself, in particular it is difficult to compile R with the Microsoft C/C++ compiler. R relies heavily on details of the IEEE floating point standard.

Calling from R

```
conv <- function(x, y){
    .C("convolve", x=as.double(x), n=length(x),
        y=as.double(y), m=length(y),
        z=numeric(length(x)+length(y)-1))$z
}
```

Need to make sure the arguments are of the correct type (double or integer), and need to supply an empty vector for the result.

Argument names are ignored by R but help us keep track.

`.C()` returns a list with copies of all the arguments, but we only care about the last argument.

Calling from R

The C code gives the same answers as the R code above, but much faster

```
> system.time(for (i in 1:100) conv(rep(1,100),rep(1,100)))
   user  system elapsed
  0.006   0.000   0.006
> system.time(for (i in 1:100) Rconv(rep(1,100),rep(1,100)))
   user  system elapsed
  8.567   0.018   8.600
```


More realistic example

Given a gene expression value X and a phenotype Y, find the best (smallest p-value) way to divide X into two categories to predict Y.

```
cutpoints <- sort(unique(x))
n<-length(cutpoints)
pvalues <- sapply(cutpoints[3:(n-2)],
     function(c) {
       z <- x<c
       t.test(y~z)$p.value
     })
best <- which.min(pvalues)
cutpoints[3:(n-2)][best]
```

Computing the unique values of X is fast in R; the loop over cutpoints is slow.

More realistic example

Design for C

- Sort X in R first
- Keep sums `1:i`, `(i+1):n` of Y and Y^2, update by adding/subtracting ith term in loop
- Compute squared z statistic rather than p-value.

More realistic example

```
void bestz(double x[], double y[], int *n, int *best){
   double sum1 = 0, sum2 = 0,  sumsq1 = 0, sumsq2 = 0;
   double mean1, mean2, var1, var2;
   double best_zsq = -1, zsq;
   int N = *n;
   int i;

    for(i=2; i<N; i++){
      sum1 += y[i];
      sumsq1 += y[i]*y[i];
    }
    sum2=y[0]+y[1];
    sumsq2=y[0]*y[0]+y[1]*y[1];
    *best = -1;
```

More realistic example

```
    for(i=2; i<N-1;  i++){
      mean1 = sum1/(N-i);
      mean2 = sum2/i;
      var1 = (sumsq1/(N-i))- mean1*mean1;
      var2 = (sumsq2/i) - mean2*mean2;
      zsq= (mean1-mean2)*(mean1-mean2)/(var1/(N-i)+var2/i);
      if (zsq>best_zsq) {
        *best=i;
         best_zsq=zsq;
      }
      sum1 -= y[i];
      sum2 += y[i];
      sumsq1 -= y[i]*y[i];
      sumsq2 += y[i]*y[i];
    }  /* i */
}      /* function */
```


More realistic example

From R

```
dyn.load("bestz.so")

bestz <- function(x,y){
   i <- order(x)
   n <- length(x)
   if (length(y)!=n) stop("lengths don't agree")
   best <- .C("bestz", x=as.double(x[i]), y=as.double(y[i]),
          n=n, best=integer(1))$best
   ibest <- i[best]+1
   list(cutpoint= x[ibest], test= t.test(x<x[ibest], y))
}
```

More realistic example

C code is much faster, for two reasons

- In C
- C code is $O(n)$, R code is $O(n^2)$ because it recomputes the means and variances from scratch.

Note: If $Y|X$ has constant variance, an even faster pure-R approach, based on changepoint theory, is

```
i<-order(x)
which.max( cumsum(y[i]-mean(y)))
```


C in packages

Put C code in the `src/` subdirectory of your package. It will be compiled and linked automatically when the package is installed.

Put `useDynLib(pkgname)` in the NAMESPACE file to load `pkgname.dll` (or `pkgname.so` or whatever).

Calls to `.C` from code in your package will now automatically look only in `pkgname.dll` for compiled routines.

.Call()

The `.C` interface is useful only for arithmetic, logical, and string vectors.

Calling back to R is clumsy and handling more complicated R objects such as lists is not feasible.

Most error checking must be done in R as it cannot be done in C and type or length errors will corrupt memory.

`.Call` provides an alternative interface that passes pointers to R objects and returns an R object.

.Call and convolve

```
#include "Rinternals.h"

SEXP  convolve(SEXP x, SEXP y){
       int i, j, m,n, nz;
       SEXP z;

       m = LENGTH(x);
       n = LENGTH(y);

       PROTECT(z = allocVector(REALSXP, m+n-1));
       for(i =0; i< n+m-1; i++) REAL(z)[i]=0;

       for(i = 0; i < m; i++) {
         for(j = 0; j < n; j++){
           REAL(z)[i + j] += REAL(x)[i] * REAL(y)[j];
         }
       }
       UNPROTECT(1);  /*z*/
       return z;
     }
```

Notes

- `SEXP`, short for S-expression (from LISP) is the type of R objects.
- `LENGTH()` returns the length of vector
- `REAL()` is a pointer to the actual numbers in the R object (`INTEGER`, `LOGICAL` for other types). `REALSXP` indicates the numeric type.
- `PROTECT()` protects memory from the garbage collector, `UNPROTECT()` releases it.
- Pointer protection is a stack: need to match `PROTECT` and `UNPROTECT` calls. The returned value is Someone Else's Problem.

Improvements

Computing the vector lengths in C removes one source of errors.

`allocVector()` will give an R-level error if memory is not available.

The `REAL()` function will give an R-level error (rather than corrupting memory) if the arguments are not numeric.

Still better to check explicitly and to convert integer or logical arguments to numeric.

Also, there is some overhead to calling `REAL()` each time.

Improvements

```
SEXP xconv,yconv;
double *xdata, *ydata;
/*...*/

if (TYPEOF(x)==REALSXP){
  xdata = REAL(x);
  xconv = 0;
} else {
  xconv = coerceVector(x, REALSXP);
  xdata= REAL(xconv);
}
/* ... */

     for(i = 0; i < n; i++) {
         for(j = 0; j < m; j++){
           zdata[i+j] += xdata[i] * ydata[i];
         }
       }
/*...*/

if (yconv) UNPROTECT(1);
if (xconv) UNPROTECT(1);
UNPROTECT(1)  /* z */
```


Lists, functions, expressions

A stripped-down version of `lapply()` (used in deciding whether to move `lapply()` to C). Takes an expression in `x` rather than a function.

```
#include "Rinternals.h"
SEXP elapply(SEXP list, SEXP expr, SEXP rho)
     {
       R_len_t i, n = length(list);
       SEXP ans;

       if(!isNewList(list)) error("'list' must be a list");
       if(!isEnvironment(rho)) error("'rho' should be an environment");
       PROTECT(ans = allocVector(VECSXP, n));
       for(i = 0; i < n; i++) {
         defineVar(install("x"), VECTOR_ELT(list, i), rho);
         SET_VECTOR_ELT(ans, i, eval(expr, rho));
       }
       setAttrib(ans, R_NamesSymbol, getAttrib(list, R_NamesSymbol));
       UNPROTECT(1);
       return ans;
     }
```

Translation

For each element of the list in turn

- Set `x` to the `i`th element of the list
- evaluate `expr` with that value of x
- put the value in the `i`th element of the answer

```
l <- list(1, 2, 3, 75)
.Call("elapply", l, quote(x^2), new.env())
```


Notes

- `isNewList()` checks for a list, `isEnvironment()` checks for an environment.
- `defineVar()` assigns a value to a variable, `install("x")` puts `x` in R's symbol table and returns a code.
- `VECTOR_ELT` reads an element from a list, `SET_VECTOR_ELT` writes an element to a list
- `eval` evaluates an expression
- `getAttrib()` and `setAttrib()` read and set attributes.

`inline` package

Allows C (C++, Fortran, Objective-C) code to be written in-line in R code, as a character string or vector.

`cfunction()` writes the C function declaration, includes necessary header files, compiles the code, and writes an R function that uses `.C()` or `.Call()`.

Example: `.Call()` version of `convolve`

`inline` package

```
convinline <- cfunction(
    sig=signature(x="numeric",
                  y="numeric"),
    body="
       int i, j, m,n, nz;
       SEXP z;

       m = LENGTH(x);
       n = LENGTH(y);

       PROTECT(z = allocVector(REALSXP, m+n-1));
       for(i =0; i< n+m-1; i++) REAL(z)[i]=0;

       for(i = 0; i < m; i++) {
         for(j = 0; j < n; j++){
           REAL(z)[i + j] += REAL(x)[i] * REAL(y)[j];
         }
       }
       UNPROTECT(1);  /*z*/
       return z;
      ",
    convention=".Call", language="C")
```

inline package

produces an S4 object inheriting from `function`

```
> convinline@.Data
function (x, y)
{
    if (!file.exists(libLFile))
        libLFile <<- compileCode(f, code, language, verbose)
    if (!(f %in% names(getLoadedDLLs())))
        dyn.load(libLFile)
    .Call("file3c7812be", PACKAGE = f, x, y)
}
```


inline package

The C code has been wrapped up into a full program:

```
> cat(convinline@code)
#include <R.h>
#include <Rdefines.h>
#include <R_ext/Error.h>


SEXP file6f1696f5 ( SEXP x, SEXP y ) {

       int i, j, m,n, nz;
       SEXP z;

       m = LENGTH(x);
       n = LENGTH(y);

       PROTECT(z = allocVector(REALSXP, m+n-1));
       for(i =0; i< n+m-1; i++) REAL(z)[i]=0;
   [...snip...]
       UNPROTECT(1);  /*z*/
       return z;

  warning("your C program does not return anything!");
  return R_NilValue;
}
```

Debugging

You can run R under a debugger, such as `gdb`

```
R  --debugger="gdb"
```

Type `run` to run R, then load the compiled code, then `CTRL-C` to get back to the debugger.

Set break points with eg

```
break elapply
break elapply.c:22
```


Valgrind

Under Linux, `valgrind` is a memory access checker that runs code in a virtual machine. It catches many typical C errors such as reading or writing off the end of an array.

```
==12539== Invalid read of size 4
==12539==    at 0x1CDF6CBE: csc_compTr (Mutils.c:273)
==12539==    by 0x1CE07E1E: tsc_transpose (dtCMatrix.c:25)
==12539==    by 0x80A67A7: do_dotcall (dotcode.c:858)
==12539==    by 0x80CACE2: Rf_eval (eval.c:400)
==12539==    by 0x80CB5AF: R_execClosure (eval.c:658)
==12539==    by 0x80CB98E: R_execMethod (eval.c:760)
==12539==    by 0x1B93DEFA: R_standardGeneric (methods_list_dispatch.c:624)
==12539==    by 0x810262E: do_standardGeneric (objects.c:1012)
==12539==    by 0x80CAD23: Rf_eval (eval.c:403)
==12539==    by 0x80CB2F0: Rf_applyClosure (eval.c:573)
==12539==    by 0x80CADCC: Rf_eval (eval.c:414)
==12539==    by 0x80CAA03: Rf_eval (eval.c:362)
==12539==  Address 0x1C0D2EA8 is 280 bytes inside a block of size 1996 alloc'
==12539==    at 0x1B9008D1: malloc (vg_replace_malloc.c:149)
==12539==    by 0x80F1B34: GetNewPage (memory.c:610)
==12539==    by 0x80F7515: Rf_allocVector (memory.c:1915)
```

Other C resources

- *Writing R Extensions* manual
- *The C Programming Language* Kernighan & Ritchie (2nd edition)
- Steve Summit's notes at `http://www.eskimo.com/~scs/cclass/`

7.30

8. Extended Exercise

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Seattle, July 2014

This session

... does not have a lecture component.

Instead, we'll do a more extensive programming project

9. Handling large data

Thomas Lumley
Ken Rice

Universities of Washington and Auckland

Seattle, July 2014

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 `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

```
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

New databases: DBI

First part of this should look familiar;

```
library("RSQLite") ## also loads DBI package
sqlite <- dbDriver("SQLite")
conn <- dbConnect(sqlite, "mynewdatabase.db")
```

Then to write data to `mynewdatabase.db`, use

```
dbWriteTable(conn, "tablename", dataframe)
```

- Optional setting `row.names=FALSE` stops writing probably-unwanted junk to the file
- Close the connection after writing – or no-one else will be able to read it

Whole tables: DBI

Other whole-table commands;

- `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

SQLite and Bioconductor

Bioconductor 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 getting too large now.

AnnotationDBI

```
> library("hgu95av")
> 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")
    probe_id gene_id is_multiple
1    1000_at    5595           0
2    1001_at    7075           0
3  1002_f_at    1557           0
4  1003_s_at     643           0
5    1004_at     643           0
```

Under the hood

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

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

Databases for data

Large phenotype data sets are usually entered and stored in a relational database, and exported for statistical analysis.

Querying the database directly can save time and memory: only load variables as needed

To automate

- use `all.vars()` to give all the variable names in a formula or expression
- construct a `SELECT` statement with the variables

Databases for data

```
> all.vars( a~b+d+log(d)+ns(e))
[1] "a" "b" "d" "e"
> all.vars(quote( a+f(b+g(c+d)+elephant)))
[1] "a"        "b"        "c"        "d"        "elephant"

> v <- all.vars( a~b+d+log(d)+ns(e))
> vlist <- paste(v, collapse=", ")
> sub("@",vlist, "select @ from phenotypes")
[1] "select a, b, d, e from phenotypes"
```

Implemented in `survey` package to allow database-backed data objects with the same user interface as in-memory objects

Method for database-backed objects loads data and then calls method for in-memory object.

Databases for data

Translating an R expression to a SQL `WHERE` condition is a little more difficult because R and SQL syntax are not identical

Need to change `%in%` to `IN`, `&` to `AND`, `|` to `OR`, remove `c` from vectors, change from expression to string.

Work recursively on an R expression (inorder traversal of tree)

```
> sqlexpr(quote(a==b+c))
[1] "( (a==(b+c)) )"
> sqlexpr(quote(a==b+c & g>h))
[1] "( ((a==(b+c)) AND (g>h)) )"
> sqlexpr(quote(a==b+c & g>h | id %in% c(3,4,7)))
[1] "( (((a==(b+c)) AND (g>h)) OR (id IN (3,4,7))) )"
```

See also `translate_sql()` in Hadley Wickham's `dplyr` package. (It also provides R functions `filter()`, `arrange()`, `select()`, `mutate()` and `summarise()` that work on SQL datasets.)

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

With the `ncdf` package:

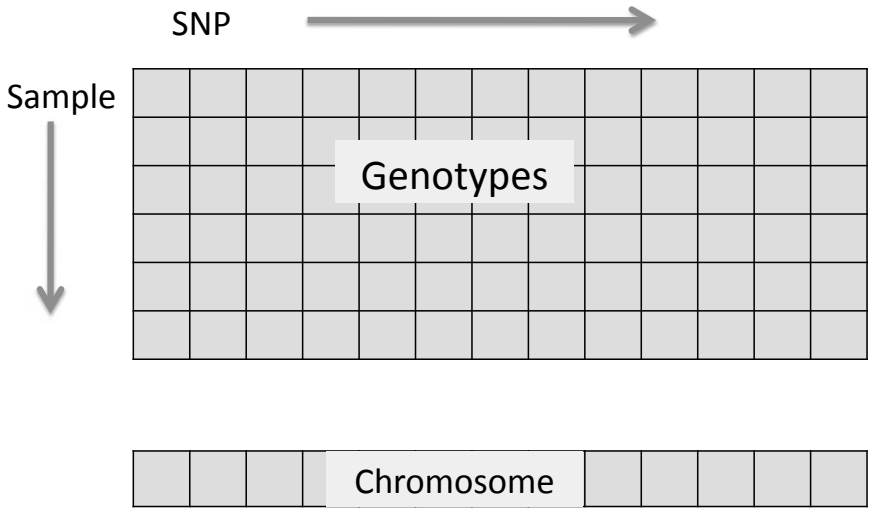
`open.ncdf()` opens a netCDF file and returns a connection to the file (rather than loading the data)

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

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

Dimensions

Variables can use one or more array dimensions of a file



Example

Finding long homozygous runs (possible deletions)

```
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

```
   ## 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])
}

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.

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.

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.

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

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")
```


Creating the file

The file is created by specifying the file name ad a list of variables.

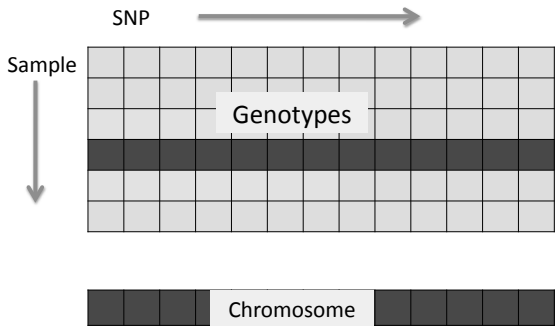
```
genofile<-create.ncdf("hapmap.nc", list(varChrm, varSNP, vargeno,
                 vartheta, varr))
```

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;

```
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))
}
```

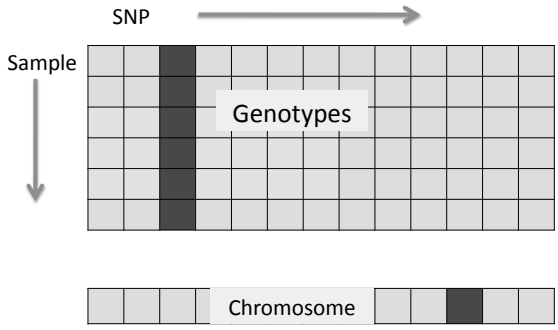
Efficient use of netCDF

Read all SNPs, one sample



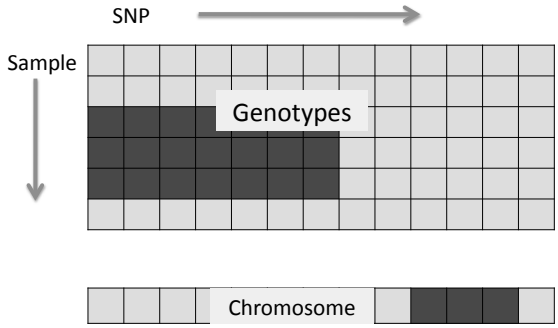
Efficient use of netCDF

Read all samples, one SNP



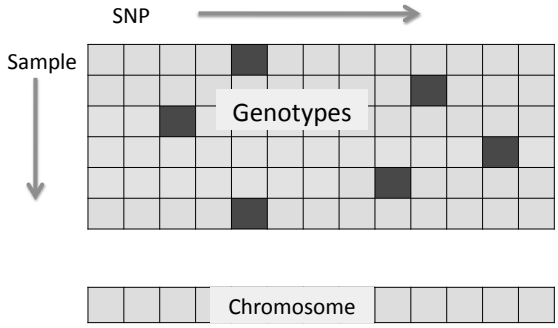
Efficient use of netCDF

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.

10. Extended Exercise

Ken Rice
Thomas Lumley

Universities of Washington and Auckland

Seattle, July 2014

This session

... does not have a lecture component.

Instead, we'll do a more extensive programming project