Introduction to R

Le Yan

HPC User Services @ LSU

The History of R

- R is a dialect of the S language
  - S was initiated at the Bell Labs as an internal statistical analysis environment
  - Most well known implementation is S-plus (most recent stable release was in 2010)
- R was first announced in 1993
- The R core group was formed in 1997, who controls the source code of R (written in C)
- R 1.0.0 was released in 2000
- The current version is 3.1.3

Features of R

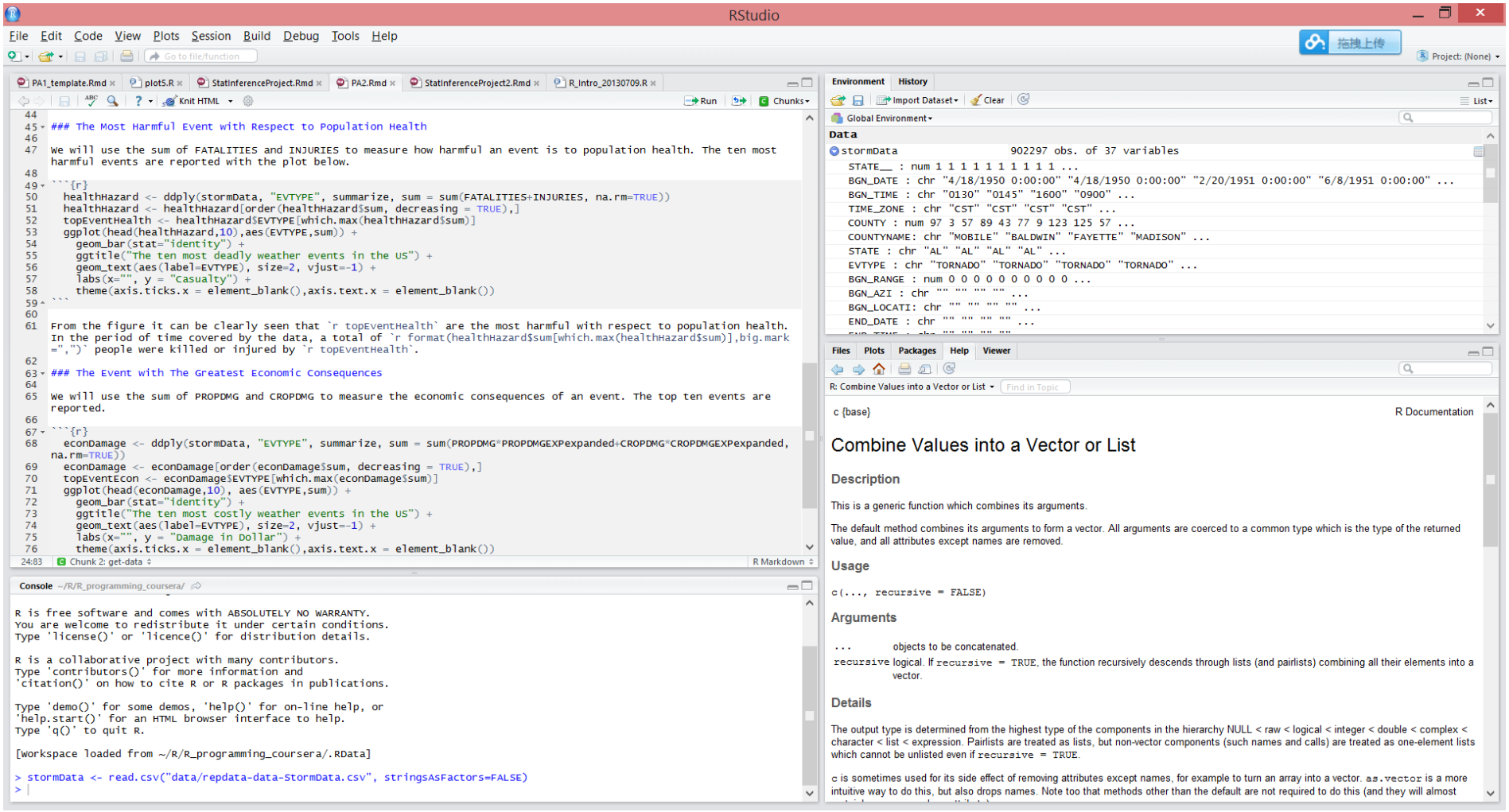
- R is a dialect of the S language
  - Language designed for statistical analysis
  - Similar syntax
- Available on most platform/OS
- Rich data analysis functionalities and sophisticated graphical capabilities
- Active development and very active community
  - CRAN: The **C**omprehensive **R** **A**rchive **N**etwork
    - Source code and binaries, user contributed packages and documentation
  - More than 6,000 packages available on CRAN as of last week
- Free to use

Two Ways of Running R

- With an IDE
  - Rstudio is the de facto environment for R on a desktop system
- On a cluster
  - R is installed on all LONI and LSU HPC clusters
    - QB2: `r/3.1.0/INTEL-14.0.2`
    - SuperMIC: `r/3.1.0/INTEL-14.0.2`
    - SuperMike2: `+R-2.15.1-gcc-4.4.6`

Rstudio

- Free to use
- Similar user interface to other IDEs or software such as Matlab; provides panes for
  - Source code
  - Console
  - Workspace
  - Others (help message, plot etc.)
- Rstudio in a desktop environment is better suited for development and/or a limited number of small jobs


RStudio
File Edit Code View Plots Session Build Debug Tools Help
PA1_template.Rmd
plot5.R
StatInferenceProject.Rmd
PA2.Rmd
StatInferenceProject2.Rmd
R_Intro_20130709.R
Knit HTML
Run
Chunks
The Most Harmful Event with Respect to Population Health
We will use the sum of FATALITIES and INJURIES to measure how harmful an event is to population health. The ten most harmful events are reported with the plot below.
```{r}
healthHazard <- ddply(stormData, "EVTYPE", summarize, sum = sum(FATALITIES+INJURIES, na.rm=TRUE))
healthHazard <- healthHazard[order(healthHazard$sum, decreasing = TRUE),]
topEventHealth <- healthHazard$EVTYPE[which.max(healthHazard$sum)]
ggplot(head(healthHazard,10),aes(EVTYPE,sum)) +
geom_bar(stat="identity") +
ggtitle("The ten most deadly weather events in the US") +
geom_text(aes(label=EVTYPE), size=2, vjust=-1) +
labs(x="", y = "Casualty") +
theme(axis.ticks.x = element_blank(),axis.text.x = element_blank())
```
From the figure it can be clearly seen that `r topEventHealth` are the most harmful with respect to population health. In the period of time covered by the data, a total of `r format(healthHazard$sum[which.max(healthHazard$sum)],big.mark=",")` people were killed or injured by `r topEventHealth`.
The Event with The Greatest Economic Consequences
We will use the sum of PROPDMG and CROPDMG to measure the economic consequences of an event. The top ten events are reported.
```{r}
econDamage <- ddply(stormData, "EVTYPE", summarize, sum = sum(PROPDMG*PROPDMGEXPexpanded+CROPDMG*CROPDMGEXPexpanded, na.rm=TRUE))
econDamage <- econDamage[order(econDamage$sum, decreasing = TRUE),]
topEventEcon <- econDamage$EVTYPE[which.max(econDamage$sum)]
ggplot(head(econDamage,10), aes(EVTYPE,sum)) +
geom_bar(stat="identity") +
ggtitle("The ten most costly weather events in the US") +
geom_text(aes(label=EVTYPE), size=2, vjust=-1) +
labs(x="", y = "Damage in Dollar") +
theme(axis.ticks.x = element_blank(),axis.text.x = element_blank())
24:83
Chunk 2: get-data
R Markdown
Console ~/R/R_programming_coursera/
R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.
[Workspace loaded from ~/R/R_programming_coursera/.RData]
> stormData <- read.csv("data/repdata-data-StormData.csv", stringsAsFactors=FALSE)
Environment History
Import Dataset
Clear
Global Environment
Data
stormData 902297 obs. of 37 variables
STATE__ : num 1 1 1 1 1 1 1 1 1 1 ...
BGN_DATE : chr "4/18/1950 0:00:00" "4/18/1950 0:00:00" "2/20/1951 0:00:00" "6/8/1951 0:00:00" ...
BGN_TIME : chr "0130" "0145" "1600" "0900" ...
TIME_ZONE : chr "CST" "CST" "CST" "CST" ...
COUNTY : num 97 3 57 89 43 77 9 123 125 57 ...
COUNTYNAME: chr "MOBILE" "BALDWIN" "FAYETTE" "MADISON" ...
STATE : chr "AL" "AL" "AL" "AL" ...
EVTYPE : chr "TORNADO" "TORNADO" "TORNADO" "TORNADO" ...
BGN_RANGE : num 0 0 0 0 0 0 0 0 0 0 ...
BGN_AZI : chr "" "" "" "" ...
BGN_LOCATI: chr "" "" "" "" ...
END_DATE : chr "" "" "" "" ...
Files Plots Packages Help Viewer
R: Combine Values into a Vector or List
Find in Topic
c {base}
R Documentation
Combine Values into a Vector or List
Description
This is a generic function which combines its arguments.
The default method combines its arguments to form a vector. All arguments are coerced to a common type which is the type of the returned value, and all attributes except names are removed.
Usage
c(..., recursive = FALSE)
Arguments
... objects to be concatenated.
recursive logical. If recursive = TRUE, the function recursively descends through lists (and pairlists) combining all their elements into a vector.
Details
The output type is determined from the highest type of the components in the hierarchy NULL < raw < logical < integer < double < complex < character < list < expression. Pairlists are treated as lists, but non-vector components (such names and calls) are treated as one-element lists which cannot be unlisted even if recursive = TRUE.
c is sometimes used for its side effect of removing attributes except names, for example to turn an array into a vector. as.vector is a more intuitive way to do this, but also drops names. Note too that methods other than the default are not required to do this (and they will almost
```

On LONI and LSU HPC Clusters

- Two modes to run R on clusters
  - Interactive mode
    - Type R command to enter the console, then run R commands there
  - Batch mode
    - Write the R script first, then submit a batch job to run it (use the `Rscript` command)
    - This is for production runs
- Clusters are better for resource-demanding jobs

```
[lyan1@qb1 ~]$ module add r
[lyan1@qb1 ~]$ R

R version 3.1.0 (2014-04-10) -- "Spring Dance"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-unknown-linux-gnu (64-bit)

…

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> getwd()
[1] "/home/lyan1"
> x <- 5
> x
[1] 5
>
Save workspace image? [y/n/c]: n

[lyan1@qb1 ~]$ cat hello.R
print("Hello World!")
[lyan1@qb1 ~]$ Rscript hello.R
[1] "Hello World!"
```

Getting Help

- Command line
  - `?<command name>`
  - `??<part of command name/topic>`
- Or search in the help page in Rstudio

Data Classes

- R has five atomic classes
  - Numeric
    - Double is equivalent to numeric.
    - Numbers in R are treated as numeric unless specified otherwise.
  - Integer
  - Complex
  - Character
  - Logical
    - TRUE or FALSE
- You can convert data from one type to the other using the `as.<Type>` functions

Data Objects - Vectors

- Vectors can only contain elements of the same class
- Vectors can be constructed by
  - Using the `c()` function (concatenate)
    - Coercion will occur when mixed objects are passed to the `c()` function, as if the `as.<Type>()` function is explicitly called
  - Using the `vector()` function
- One can use `[index]` to access individual element
  - Indices start from 1

```
# “#” indicates comment
# “<-” performs assignment operation (you can use “=“ as well, but
“<-” is preferred)

# numeric (double is the same as numeric)
> d <- c(1,2,3)
> d
[1] 1 2 3

# character
> d <- c("1","2","3")
> d
[1] "1" "2" "3"

# you can covert at object with as.TYPE
# as. numeric changes the character vector created above to numeric
> as.numeric(d)
[1] 1 2 3

# The conversion doesn't always work though
> as.numeric("a")
[1] NA
Warning message:
NAs introduced by coercion
```

```
> x <- c(0.5, 0.6) ## numeric
> x <- c(TRUE, FALSE) ## logical
> x <- c(T, F) ## logical
> x <- c("a", "b", "c") ## character
# The ":" operator can be used to generate integer sequences
> x <- 9:29 ## integer
> x <- c(1+0i, 2+4i) ## complex

> x <- vector("numeric", length = 10)
> x
[1] 0 0 0 0 0 0 0 0 0 0

# Coercion will occur when objects of different classes are mixed
> y <- c(1.7, "a") ## character
> y <- c(TRUE, 2) ## numeric
> y <- c("a", TRUE) ## character

# Can also coerce explicitly
> x <- 0:6
> class(x)
[1] "integer"
> as.logical(x)
[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE
```

Vectorized Operations

- Lots of R operations process objects in a vectorized way
  - more efficient, concise, and easier to read.

```
> x <- 1:4; y <- 6:9
> x + y
[1]  7  9 11 13
> x > 2
[1] FALSE FALSE  TRUE  TRUE
> x * y
[1]  6 14 24 36
> print( x[x >= 3] )
[1] 3 4
```

Data Objects - Matrices

- Matrices are vectors with a dimension attribute
- R matrices can be constructed
  - Using the `matrix()` function
  - Passing an `dim` attribute to a vector
  - Using the `cbind()` or `rbind()` functions
- R matrices are constructed column-wise
- One can use `[<index>,<index>]` to access individual element

```
# Create a matrix using the matrix() function
> m <- matrix(1:6, nrow = 2, ncol = 3)
> m
[,1] [,2] [,3]
[1,] 1 3 5
[2,] 2 4 6
> dim(m)
[1] 2 3
> attributes(m)
$dim
[1] 2 3

# Pass a dim attribute to a vector
> m <- 1:10
> m
[1] 1 2 3 4 5 6 7 8 9 10
> dim(m) <- c(2, 5)
> m
[,1] [,2] [,3] [,4] [,5]
[1,] 1 3 5 7 9
[2,] 2 4 6 8 10
```

```
# Row binding and column binding
> x <- 1:3
> y <- 10:12
> cbind(x, y)
x y
[1,] 1 10
[2,] 2 11
[3,] 3 12
> rbind(x, y)
[,1] [,2] [,3]
x 1 2 3
y 10 11 12

# Slicing
> m <- 1:10
> m[c(1,2),c(2,4)]
[,1] [,2]
[1,] 3 7
[2,] 4 8
```

Data Objects - Lists

- Lists are a special kind of vector that contains objects of different classes
- Lists can be constructed by using the `list()` function
- Lists can be indexed using `[[]]`

```
# Use the list() function to construct a list
> x <- list(1, "a", TRUE, 1 + 4i)
> x
[[1]]
[1] 1

[[2]]
[1] "a"

[[3]]
[1] TRUE
```

Data Objects - Data Frames

- Data frames are used to store tabular data
  - They are a special type of list where every element of the list has to have the same length
  - Each element of the list can be thought of as a column
  - Data frames can store different classes of objects in each column
  - Data frames also have a special attribute called `row.names`
  - Data frames are usually created by calling `read.table()` or `read.csv()`
    - More on this later
  - Can be converted to a matrix by calling `data.matrix()`

Names

- R objects can have names

```
# Each element in a vector can have a name
> x <- 1:3
> names(x)
NULL
> names(x) <- c("a","b","c")
> names(x)
[1] "a" "b" "c"
> x
a b c
1 2 3
```

```
# Lists
> x <- list(a = 1, b = 2, c = 3)
> x
$a
[1] 1

$b
[1] 2

$c
[1] 3

# Names can be used to refer to individual element
> x$a
[1] 1

# Columns and rows of matrices
> m <- matrix(1:4, nrow = 2, ncol = 2)
> dimnames(m) <- list(c("a", "b"), c("c", "d"))
> m
  c d
a 1 3
b 2 4
```

Querying Object Attributes

- The `class()` function
- The `str()` function
- The `attributes()` function reveals attributes of an object (does not work with vectors)
  - Class
  - Names
  - Dimensions
  - Length
  - User defined attributes
- They work on all objects (including functions)

```
> m <- matrix(1:10, nrow = 2, ncol = 5)
> str(matrix)
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE,
dimnames = NULL)
> str(m)
 int [1:2, 1:5] 1 2 3 4 5 6 7 8 9 10

> str(matrix)
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE,
dimnames = NULL)

> str(str)
function (object, ...)
```

Data Class - Factors

- Factors are used to represent categorical data.
- Factors can be unordered or ordered.
- Factors are treated specially by modelling functions like `lm()` and `glm()`

```
# Use the factor() function to construct a vector of factors
# The order of levels can be set by the levels keyword
> x <- factor(c("yes", "yes", "no", "yes", "no"),
levels = c("yes", "no"))
> x
[1] yes yes no yes no
Levels: yes no
```

Date and Time

- R has a Date class for date data while times are represented by POSIX formats
- One can convert a text string to date using the `as.Date()` function
- The `strptime()` function can deal with dates and times in different formats.
- The package "`lubridate`" provides many additional and convenient features

```
# Dates are stored internally as the number of days since 1970-01-01
> x <- as.Date("1970-01-01")
> x
[1] "1970-01-01"
> as.numeric(x)
[1] 0
> x+1
[1] "1970-01-02“

# Tmes are stored internally as the number of seconds since 1970-01-01
> x <- Sys.time()
> x
[1] "2015-03-17 09:40:43 CDT"
> as.numeric(x)
[1] 1426603244
> p <- as.POSIXlt(x)
> names(unclass(p))
 [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"
 [9] "isdst"  "zone"   "gmtoff"
> p$sec
[1] 43.88181
```

Missing Values

- Missing values are denoted by NA or NaN for undefined mathematical operations.
  - `is.na()` is used to test objects if they are NA
  - `is.nan()` is used to test for NaN
  - NA values have a class also, so there are integer NA, character NA, etc.
  - A NaN value is also NA but the converse is not true

```
> x <- c(1, 2, NA, 10, 3)
> is.na(x)
[1] FALSE FALSE TRUE FALSE FALSE
> is.nan(x)
[1] FALSE FALSE FALSE FALSE FALSE
> x <- c(1, 2, NaN, NA, 4)
> is.na(x)
[1] FALSE FALSE TRUE TRUE FALSE
> is.nan(x)
[1] FALSE FALSE TRUE FALSE FALSE
```

Arithmetic Functions

`exp()`	exponentiation
`log()`	log
`log10()`	log base10
`sqrt()`	square root
`abs()`	absolute value
`sin()`	sine
`cos()`	cosine
`floor()`	Rounding of numbers
`ceiling()`	
`round()`	

Simple Statistic Functions

`min()`	Minimum value
`max()`	Maximum value
`which.min()`	Location of minimum value
`which.max()`	Location of maximum value
`pmin()`	Element-wise minima of several vectors
`pmax()`	Element-wise maxima of several vectors
`sum()`	Sum of the elements of a vector
`mean()`	Mean of the elements of a vector
`prod()`	Product of the elements of a vector

```
> dim(x)
[1]  2  2 50
> min(x)
[1] -2.665878
> which.min(x)
[1] 123
```

Distributions and Random Variables

- For each distribution R provides four functions: density (d), cumulative density (p), quantile (q), and random generation (r)
  - The function name is of the form `[d|p|q|r]<name of distribution>`
  - e.g. `qbinom()` gives the quantile of a binomial distribution

Distribution	Distribution name in R
Uniform	`unif`
Binomial	`binom`
Poisson	`pois`
Geometric	`geom`
Gamma	`gamma`
Normal	`norm`
Log Normal	`lnorm`
Exponential	`exp`
Student's t	`t`

```
# Random generation from a uniform distribution.
> runif(10, 2, 4)
 [1] 2.871361 3.176906 3.157928 2.398450 2.171803 3.954051
3.084317 2.883278
 [9] 2.284473 3.482990
# You can name the arguments in the function call.
> runif(10, min = 2, max = 4)

# Given p value and degree of freedom, find the t-value.
> qt(p=0.975, df = 8)
[1] 2.306004
# The inverse of the above function call
> pt(2.306, df = 8)
[1] 0.9749998
```

User Defined Functions

- Similar to other languages, functions in Rare defined by using the `function()` directives
- The return value is the last expression in the function body to be evaluated.
- Functions can be nested
- Functions are R objects
  - For example, they can be passed as an argument to other functions

Control Structures

- Control structures allow one to control the flow of execution.

`if … else`	testing a condition
`for`	executing a loop (with fixed number of iterations)
`while`	executing a loop when a condition is true
`repeat`	executing an infinite loop
`break`	breaking the execution of a loop
`next`	skipping to next iteration
`return`	exit a function

Testing conditions

```
# Comparisons: <,<=,>,>=,==,!=
# Logical operations: !, &&, ||

if(x > 3 && x < 5) {
  print ("x is between 3 and 5")
} else if(x <= 3) {
  print ("x is less or equal to 3")
} else {
  print ("x is greater or equal to 5")
}
```

For Loops

```
x <- c("a", "b", "c", "d")

# These loops have the same effect

# Loop through the indices
for(i in 1:4) {
  print(x[i])
}

# Loop using the seq_along() function
for(i in seq_along(x)) {
  print(x[i])
}

# Loop through the name
for(letter in x) {
  print(letter)
}

for(i in 1:4) print(x[i])
```

The apply Function

- The `apply()` function evaluate a function over the margins of an array
  - More concise than the for loops (not necessarily faster)

```
# X: array objects
# MARGIN: a vector giving the subscripts which
the function will be applied over
# FUN: a function to be applied

> str(apply)
function (X, MARGIN, FUN, ...)
```

```
> x <- matrix(rnorm(200), 20, 10)

# Row means
> apply(x, 1, mean)
 [1] -0.23457304  0.36702942 -0.29057632 -0.24516988 -0.02845449  0.38583231
 [7]  0.16124103 -0.10164565  0.02261840 -0.52110832 -0.10415452  0.40272211
[13]  0.14556279 -0.58283197 -0.16267073  0.16245682 -0.28675615 -0.21147184
[19]  0.30415344  0.35131224

# Column sums
> apply(x, 2, sum)
 [1]  2.866834  2.110785 -2.123740 -1.222108 -5.461704 -5.447811 -4.299182
 [8] -7.696728  7.370928  9.237883

# 25th and 75th Quantiles for rows
> apply(x, 1, quantile, probs = c(0.25, 0.75))
           [,1]       [,2]       [,3]       [,4]      [,5]       [,6]
25% -0.52753974 -0.1084101 -1.1327258 -0.9473914 -1.176299 -0.4790660
75%  0.05962769  0.6818734  0.7354684  0.5547772  1.066931  0.6359116
          [,7]       [,8]       [,9]       [,10]      [,11]      [,12]
25% -0.1968380 -0.5063218 -0.8846155 -1.54558614 -0.8847892 -0.2001400
75%  0.7910642  0.3893138  0.8881821 -0.06074355  0.5042554  0.9384258
         [,13]       [,14]      [,15]      [,16]      [,17]      [,18]
25% -0.5378145 -1.08873676 -0.5566373 -0.3189407 -0.6280269 -0.6979439
75%  0.6438305 -0.02031298  0.3495564  0.3391990 -0.1151416  0.2936645
        [,19]      [,20]
25% -0.259203 -0.1798460
75%  1.081322  0.8306676
```

```
> dim(x)
[1] 20 10

# Change the dimensions of x
> dim(x) <- c(2,2,50)

# Take average over the first two dimensions
> apply(x, c(1, 2), mean)
            [,1]        [,2]
[1,] -0.0763205 -0.01840142
[2,] -0.1125101  0.11393513
> rowMeans(x, dims = 2)
            [,1]        [,2]
[1,] -0.0763205 -0.01840142
[2,] -0.1125101  0.11393513
```

Other Apply Functions

- `lapply` - Loop over a list and evaluate a function on each element
- `sapply` - Same as lapply but try to simplify the result
- `tapply` - Apply a function over subsets of a vector
- `mapply` - Multivariate version of lapply

Plyr Package

- In data analysis you often need to **split** up a big data structure into homogeneous pieces, **apply** a function to each piece and then **combine** all the results back together
- This split-apply-combine procedure is what the plyr package is for.

```
> library(ggplot2)
> library(plyr)
> str(mpg)
'data.frame':   234 obs. of  11 variables:
 $ manufacturer: Factor w/ 15 levels "audi","chevrolet",..: 1 1 1 1 1 1 1 1 1 1
...
 $ model       : Factor w/ 38 levels "4runner 4wd",..: 2 2 2 2 2 2 2 3 3 3 ...
 $ displ       : num  1.8 1.8 2 2 2.8 2.8 3.1 1.8 1.8 2 ...
 $ year        : int  1999 1999 2008 2008 1999 1999 2008 1999 1999 2008 ...
 $ cyl         : int  4 4 4 4 6 6 6 4 4 4 ...
 $ trans       : Factor w/ 10 levels "auto(av)","auto(l3)",..: 4 9 10 1 4 9 1 9 4
10 ...
 $ drv         : Factor w/ 3 levels "4","f","r": 2 2 2 2 2 2 2 1 1 1 ...
 $ cty         : int  18 21 20 21 16 18 18 18 16 20 ...
 $ hwy         : int  29 29 31 30 26 26 27 26 25 28 ...
 $ fl          : Factor w/ 5 levels "c","d","e","p",..: 4 4 4 4 4 4 4 4 4 4 ...
 $ class       : Factor w/ 7 levels "2seater","compact",..: 2 2 2 2 2 2 2 2 2 2
...
> str(ddply)
function (.data, .variables, .fun = NULL, ..., .progress = "none", .inform =
FALSE, .drop = TRUE, .parallel = FALSE, .paropts = NULL)
> ddply(mpg, "cyl", summarise, mean = mean(cty))
  cyl     mean
1   4 21.01235
2   5 20.50000
3   6 16.21519
4   8 12.57143
```

Reading and Writing Data

- R understands many different data formats and has lots of ways of reading/writing them

`read.table` `read.csv`	`write.table` `write.csv`	for reading/writing tabular data
`readLines`	`writeLines`	for reading/writing lines of a text file
`source`	`dump`	for reading/writing in R code files
`dget`	`dput`	for reading/writing in R code files
`load`	`save`	for reading in/saving workspaces
`unserialize`	`serialize`	for reading/writing single R objects in binary form

Reading Data with `read.table` (1)

```
> str(read.table)
function (file, header = FALSE, sep = "", quote = "\"'", dec = ".",
row.names, col.names, as.is = !stringsAsFactors, na.strings = "NA",
colClasses = NA, nrows = -1, skip = 0, check.names = TRUE, fill =
!blank.lines.skip, strip.white = FALSE, blank.lines.skip = TRUE,
comment.char = "#", allowEscapes = FALSE, flush = FALSE, stringsAsFactors =
default.stringsAsFactors(), fileEncoding = "", encoding = "unknown", text,
skipNul = FALSE)
```

Reading Data with `read.table` (2)

- `file` - the name of a file, or a connection
- `header` - logical indicating if the file has a header line
- `sep` - a string indicating how the columns are separated
- `colClasses` - a character vector indicating the class of each column in the dataset
- `nrows` - the number of rows in the dataset
- `comment.char` - a character string indicating the comment character
- `skip` - the number of lines to skip from the beginning
- `stringsAsFactors` - should character variables be coded as factors?

Reading Data with `read.table` (3)

- The function will
  - Skip lines that begin with a #
  - Figure out how many rows there are (and how much memory needs to be allocated)
  - Figure out what type of variable is in each column of the table
- Telling R all these things directly makes R run faster and more efficiently.
- `read.csv()` is identical to `read.table()` except that the default separator is a comma.

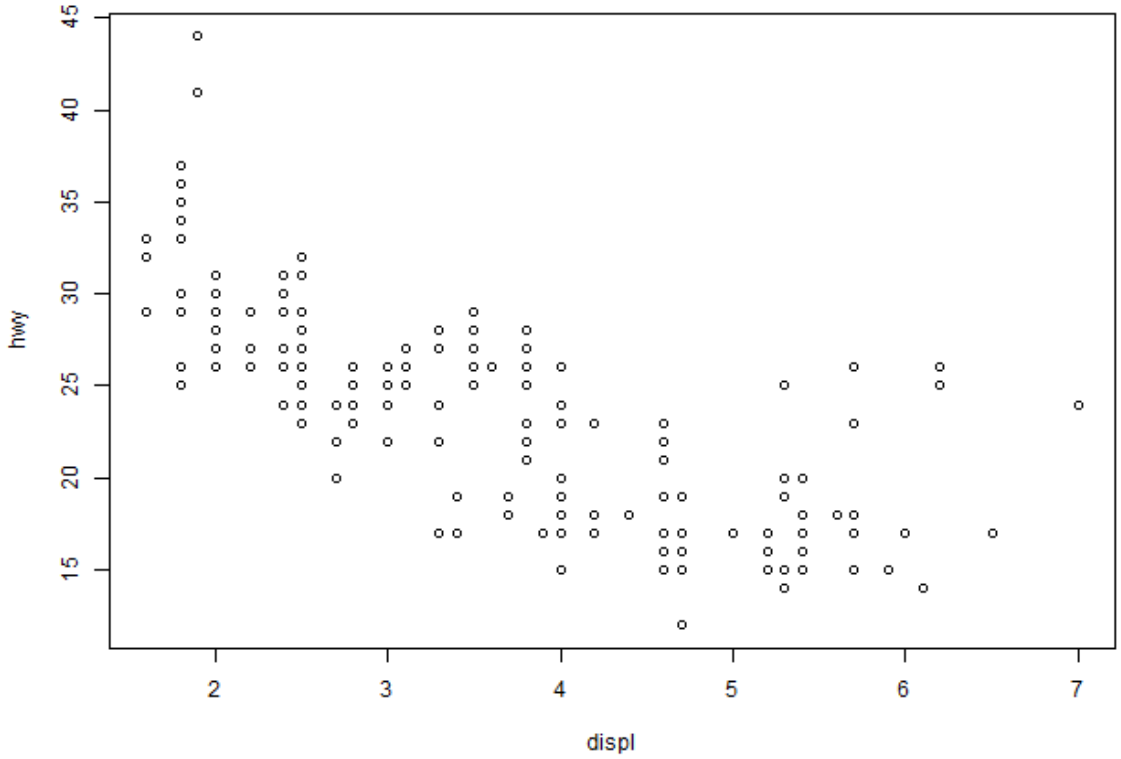
```
[lyan1@qb1 R]$ head household_power_consumption.txt
# This file contains the household power cosumption data.
Date;Time;Global_active_power;Global_reactive_power;Voltage;Global_intensity;Sub_m
etering_1;Sub_metering_2;Sub_metering_3
16/12/2006;17:24:00;4.216;0.418;234.840;18.400;0.000;1.000;17.000
16/12/2006;17:25:00;5.360;0.436;233.630;23.000;0.000;1.000;16.000
16/12/2006;17:26:00;5.374;0.498;233.290;23.000;0.000;2.000;17.000

> comsumpData <- read.table("household_power_consumption.txt",header=TRUE,sep=";")
> str(comsumpData)
'data.frame':   2075259 obs. of  9 variables:
 $ Date                  : Factor w/ 1442 levels "10/10/2007","10/10/2008",..: 326
326 326 326 326 326 326 326 326 326 ...
 $ Time                  : Factor w/ 1440 levels "00:00:00","00:01:00",..: 1045
1046 1047 1048 1049 1050 1051 1052 1053 1054 ...
 $ Global_active_power  : Factor w/ 4187 levels "?","0.076","0.078",..: 2082 2654
2661 2668 1807 1734 1825 1824 1808 1805 ...
 $ Global_reactive_power: Factor w/ 533 levels "?","0.000","0.046",..: 189 198 229
231 244 241 240 240 235 235 ...
 $ Voltage               : Factor w/ 2838 levels "?","223.200",..: 992 871 837 882
1076 1010 1017 1030 907 894 ...
```

Graphics in R

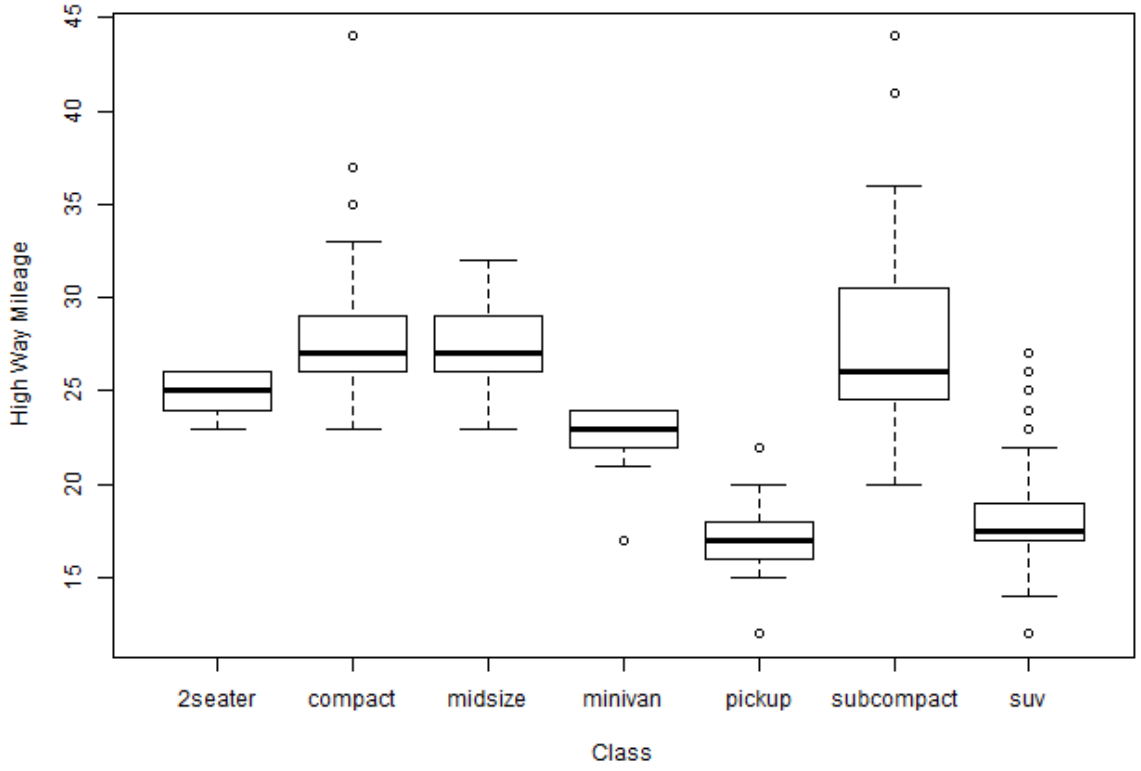
- There are three plotting systems in R
  - Base
    - Convenient, but hard to adjust after the plot is created
  - Lattice
    - Good for creating conditioning plot
  - Ggplot2
    - Powerful and flexible, many tunable feature, may require some time to master
- Each has its pros and cons, so it is up to the users which one to choose

Graphics - Base



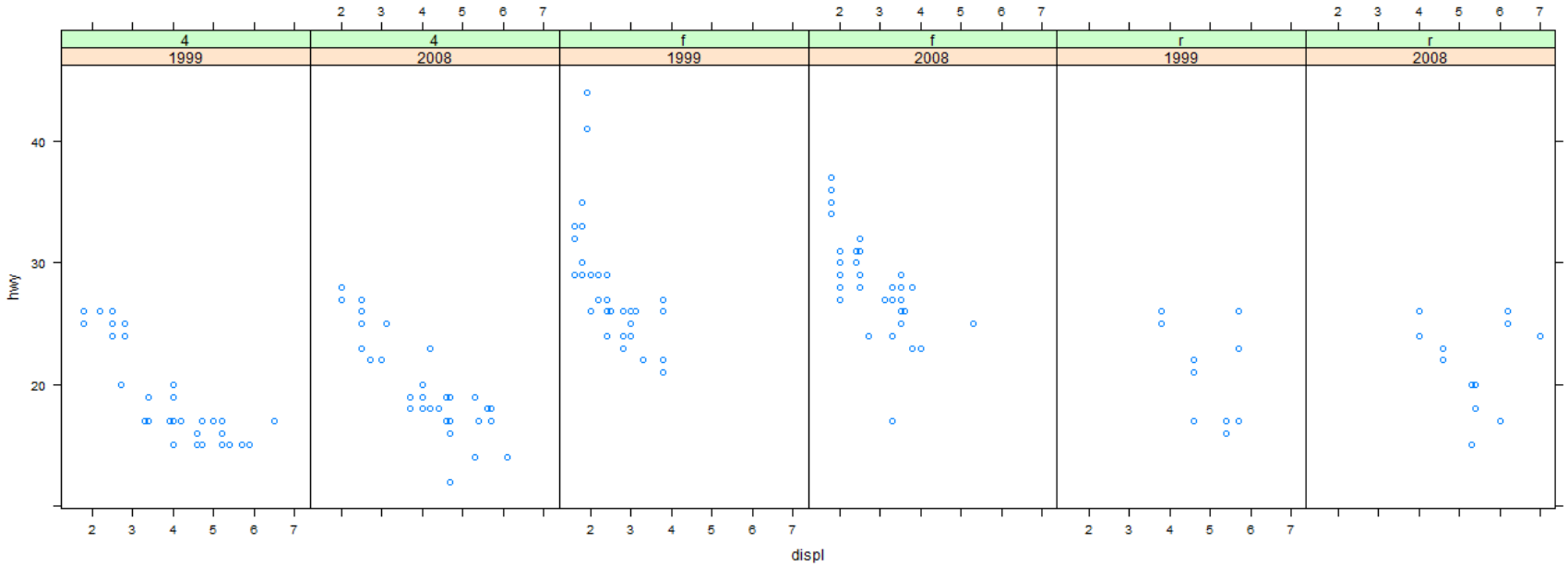
plot(hwy ~ displ, data=mpg)

Graphics - Base



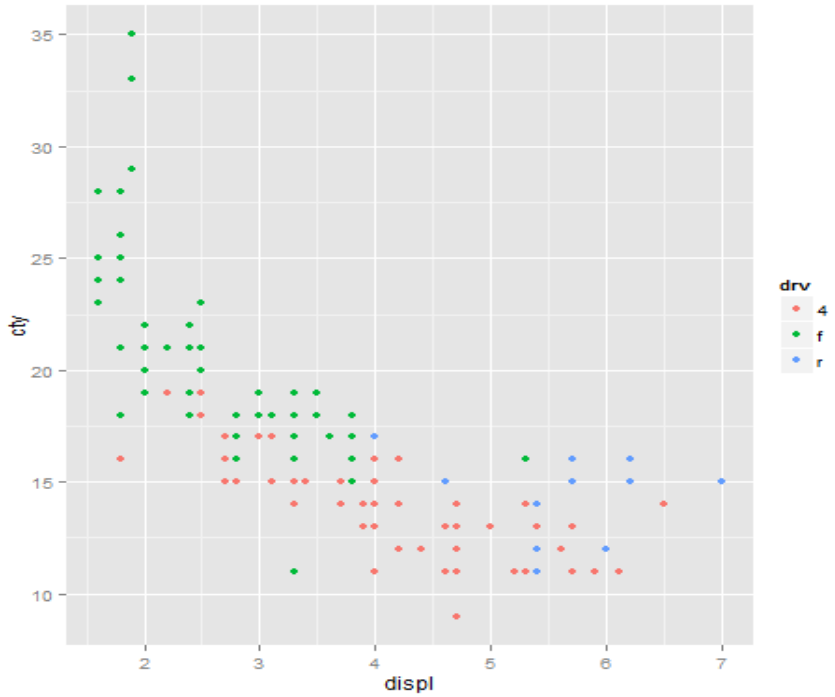

```
boxplot(hwy ~ class, data = mpg, xlab = "Class",
                     ylab = "High Way Mileage")
```

Graphics - Lattice



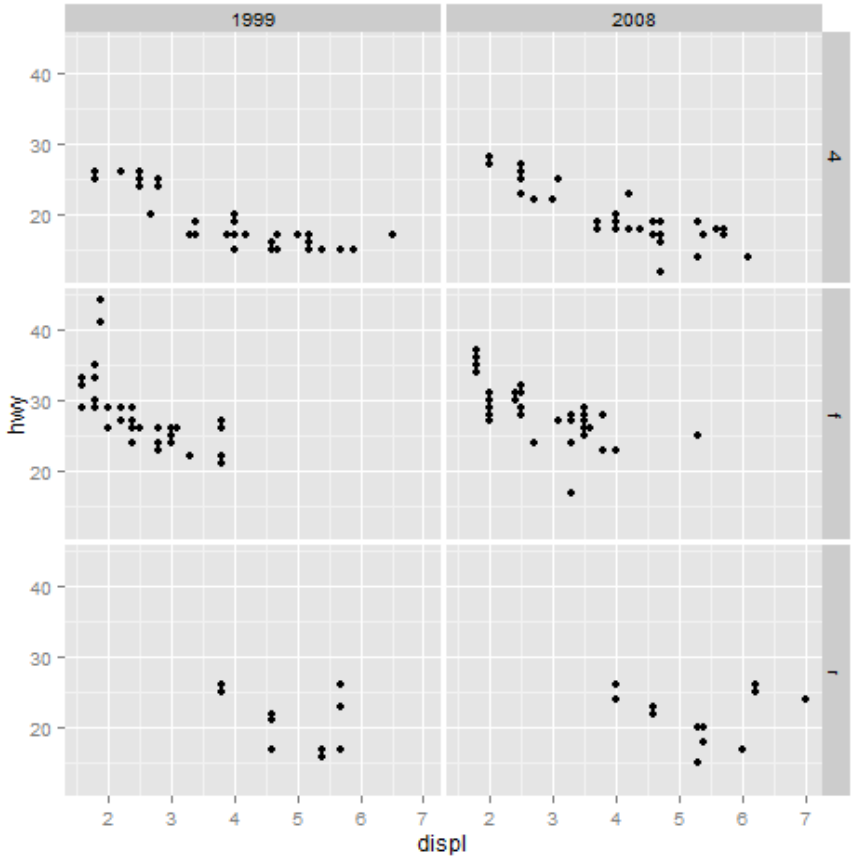

```
mpg <- transform(mpg, year = factor(year))
xyplot(hwy ~ displ | year*drv, mpg, layout = c(6,1))
```

Graphics – ggplot2



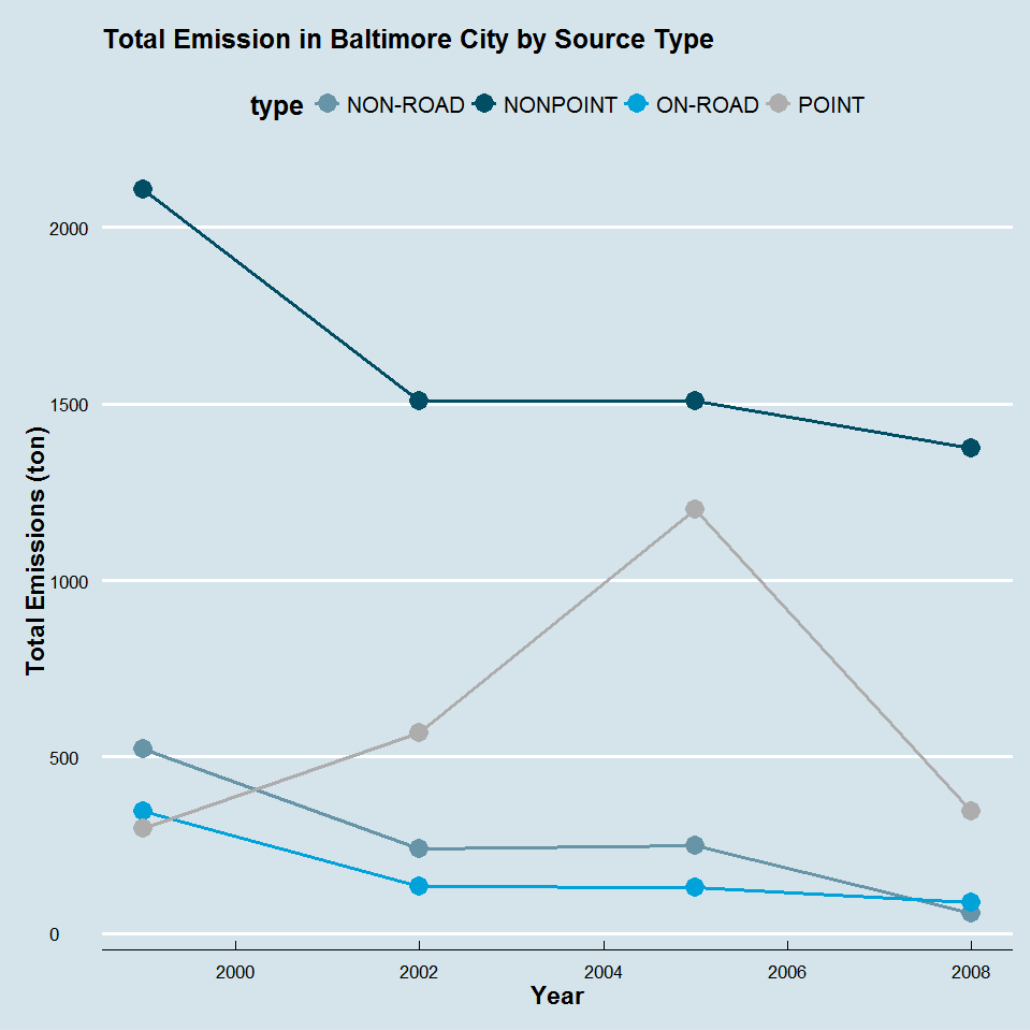

```
qplot(displ, cty, data = mpg, color = drv)
```

Graphics – ggplot2




```
qplot(displ, hwy, data = mpg, facets = drv ~ year)
```

Graphics – ggplot2



Graphics – ggplot2

```
ggplot(emiByType, aes(year, sum, colour = type)) +
  geom_point(size = 5) +
  geom_line(size = 1) +
  ggtitle("Total Emission in Baltimore City by Source Type") +
  labs(x = "Year", y = "Total Emissions (ton)", fontsize = 20) +
  theme_economist() + scale_colour_economist() +
  theme(axis.title=element_text(size=14,face="bold"), legend.title =
        element_text(size = rel(1.5), face = "bold"))
```

Installing and Loading R Packages

- Installation
  - With R Studio
    - You most likely have root privilege on your own computer
    - Use the `install.packages("<package name>")` function (double quotation is mandatory), or
    - Click on "install packages" in the menu
  - On a cluster
    - You most likely do NOT have root privilege
    - To install a R packages
      - Point the environment variable `R_LIBS_USER` to desired location, then
      - Use the `install.packages` function
- Loading: the `library()` function load previously installed packages

```
[lyan1@qb1 R]$ export R_LIBS_USER=/home/lyan1/packages/R/libraries
[lyan1@qb1 R]$ R

R version 3.1.0 (2014-04-10) -- "Spring Dance"
Copyright (C) 2014 The R Foundation for Statistical Computing
Platform: x86_64-unknown-linux-gnu (64-bit)
…

> install.packages("swirl")
```

R with HPC

- There are lots of efforts going on to make R run (more efficiently) on HPC platforms
  - http://cran.r-project.org/web/views/HighPerformanceComputing.html

Not Covered

- Data cleaning/preprocessing
- Profiling and debugging
- Regression Models
- Machine learning/Data Mining
- …
- Chances are that R has something in store for you whenever it comes to data analysis

Case Study 1: Central Limit Theorem

```
# The purpose of this segment of code is to verify the Central Limit Theorem
(CLT).
# CLT states that, when certain conditions are satisfied, sample means are
approximately normally distributed, no matter what the underlying
distribution is.

# First, show a histogram of the underlying distribution.
hist(rexp(1000,0.2), main = “Exponential distribution with a rate of 0.2")

# Draw 1000 samples (size = 100) and plot a histogram of the means.
mns=NULL
for (i in 1 : 1000) mns = c(mns, mean(rexp(100,0.2)))
hist(mns, main = "Distribtion of sample means")

# Test if the sample means are normally distributed.
shapiro.test(mns)
qqnorm(mns);qqline(mns,col = 2)
```

Case Study 2: Data Analysis with Reporting

- Typical data analysis workflow involves
  - Obtaining the data
  - Cleaning and preprocessing the data
  - Analyzing the data
  - Generating a report
- `knitr` is a R package that allows one to generate dynamic report by weaving R code and human readable texts together
  - It uses the markdown syntax
  - The output can be HTML, PDF or (even) Word

```
---
title: "R_tutorial_knitr"
author: "Le Yan"
date: "Tuesday, March 10, 2015"
output: pdf_document
---

This is a report generated by the "knitr" package for the LSU HPC "Introduction to R" tutorial.

The data source is from [Univerity of California at Irvine Machine Learning Repository]http://archive.ics.uci.edu/ml/):
[Electric Power Consumption.](https://archive.ics.uci.edu/ml/datasets/Individual+household+electric+power+consumption)

## Load the data

In this step, we load the dataset into R.

```{r, echo=FALSE, cache=TRUE}

  # Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.
  # The "cache=TRUE" parameter enables caching, which means that the objects created in this segment of code will not be recalculated next time.

  taball <- read.table("household_power_consumption.txt",header=TRUE,sep=";")
```

Preprocessing the data

In this step, we extract two days of data, add a column "datetime" to the data frame and convert the "Global_active_power"" column from factor to numeric.

```{r}

  tab2day <- subset(taball, Date == "1/2/2007" | Date == "2/2/2007")
  tab2day$datetime <- strptime(paste(tab2day$Date, tab2day$Time),
                                          format="%d/%m/%Y %H:%M:%S")
  tab2day$Global_active_power <- as.numeric(levels(tab2day$Global_active_power)
                                          [tab2day$Global_active_power])

```

Data analysis

The only data analysis performed in this step is to generate a histogram.

```{r}
  hist(tab2day$Global_active_power, col="red", main="Global Active Power",
       xlab = "Global Active Power (in kilowatts)")
```

26:155 (Top Level) R Markdown
```

Learning R

- User documentation on CRAN
  - An Introduction on R: http://cran.r-project.org/doc/manuals/r-release/R-intro.html
- Online tutorials
  - http://www.cyclismo.org/tutorial/R/
- Online courses (e.g. Coursera)
- Educational R packages
  - Swirl: Learn R in R

Next Tutorial – Xeon Phi Programming

- Xeon Phi coprocessors have the potential of accelerate the execution of your code.
- Most of the nodes on the LSU SuperMIC cluster have two Xeon Phi coprocessors installed.
- This tutorial will be of interest to those who would like to harness the power of this family of accelerators
- Date: March 25th, 2015

Getting Help

- User Guides
  - LSU HPC: http://www.hpc.lsu.edu/docs/guides.php#hpc
  - LONI:http://www.hpc.lsu.edu/docs/guides.php#loni
- Documentation: http://www.hpc.lsu.edu/docs
- Online courses: http://moodle.hpc.lsu.edu
- Contact us
  - Email ticket system: sys-help@loni.org
  - Telephone Help Desk: 225-578-0900
  - Instant Messenger (AIM, Yahoo Messenger, Google Talk)
    - Add “lsuhpchelp”

Questions?