SIAM: Getting Started with R

Mark M. Fredrickson

fredric3@illinois.edu

March 6, 2014

Which implementation performs better?

- Imagine you have two implementations of an algorithm. You want to know which performs better on a data set.

Which implementation performs better?

- Imagine you have two implementations of an algorithm. You want to know which performs better on a data set.
- Strategy 1: Run each implementation once. Pick a winner based on that one run.

Which implementation performs better?

- Imagine you have two implementations of an algorithm. You want to know which performs better on a data set.
- Strategy 1: Run each implementation once. Pick a winner based on that one run.
- Strategy 2: Run the first implementation 100 times and then run the second implementation 100 times. Compare the average run time.

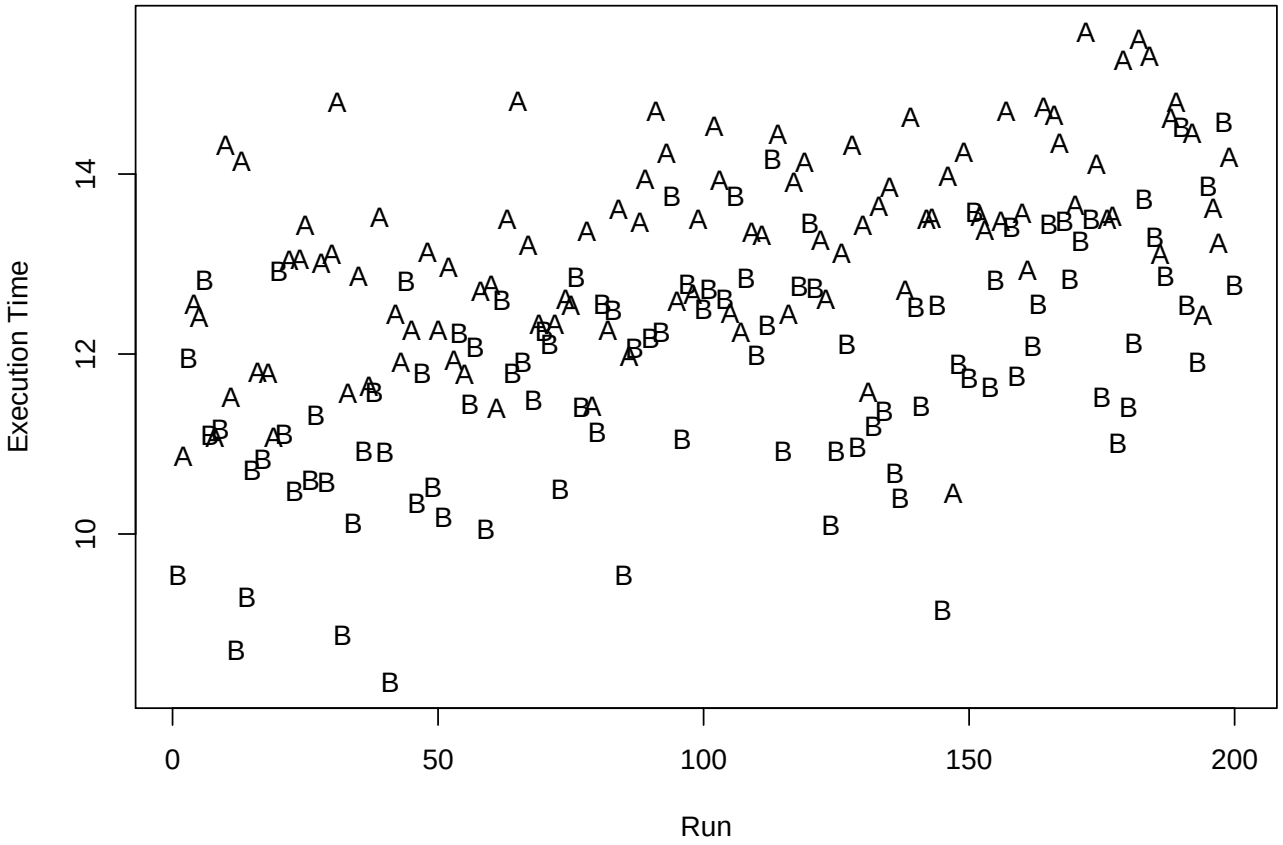
Which implementation performs better?

- Imagine you have two implementations of an algorithm. You want to know which performs better on a data set.
- Strategy 1: Run each implementation once. Pick a winner based on that one run.
- Strategy 2: Run the first implementation 100 times and then run the second implementation 100 times. Compare the average run time.
- Strategy 3: Randomly assign order of tests. Perform Wilcoxon-Mann-Whitney test of hypothesis that there is no difference in the two implementations.

Running experiment

```
> # number of times to run each implementation
> replicates <- 100
> # if z is 1, run A then B, otherwise run B then A
> z <- rbinom(replicates, size = 1, p = 0.5)
> times <- sapply(z, function(i) {
    if (i == 1) {
      return(c(A(), B()))
    } else {
      return(c(B(), A()))
    }
  })
```

Graphing the data



Hypothesis test: Two algorithms perform the same

```
> wilcox.test(times[1,], times[2,],
              paired = TRUE, exact = TRUE)

Wilcoxon signed rank test

data:  times[1, ] and times[2, ]
V = 2195, p-value = 0.2585
alternative hypothesis: true location shift is not equa
```

What is R?

- A language for statistical analysis (data manipulation, modeling, visualization).
- S started at Bell Labs in 1970s and 1980s.
- GNU R is an open source port of the language.
- Currently in version 3.0.x.
- Interpreted language with bindings to C/C++, Fortran, other languages
- Available for Windows, Mac, UNIXes; extensive package repository

Variables

- Variables can be assigned using either <- or =

```
> a <- 7
> b = c(1, 2, 3, 4)
```

- The basic data type is a vector (with optional names)

```
> is.vector(c(a = 3, foo = 1, 4, last123 = 1))
[1] TRUE
> is.vector(3)
[1] TRUE
```

- Assigning via special functions like `names<-`:

```
> names(b) <- c("A", "B", "C", "D")
> b
A B C D
1 2 3 4
```

Objects

- Objects have a *class* and a *mode*

```
> m <- matrix(c(1,2,3,4,5,6,7,8,9), nrow = 3)
> class(m)
[1] "matrix"
> mode(m)
[1] "numeric"
```

- Usual suspects for modes:
  - logical
  - numeric (integer, double, complex, factor)
  - character (strings)
  - raw
  - list

Exercise: Variables

- Create a vector with three numbers (use the c function).
- Create a vector with six numbers.
- Add them together. What happens?

Exercise: Variables

- Create a vector with three numbers (use the c function).
- Create a vector with six numbers.
- Add them together. What happens?
- R will "recycle" vectors to the length of the longer one.

Loops

The usual `for` and `while` constructs exist:

```
> a <- 0
> for (i in 1:5) {
    a <- a + i
 }
> print(a)
[1] 15
```

The `*apply` family of functions perform maps over vectors:

```
> sapply(c(1,2,3,4), function(x) { x * x })
[1]  1  4  9 16
```

Vectorization

Many explicit loops can be avoided with "vectorization":

```
> square <- function(x) { x * x }
> square(c(1,2,3))
[1] 1 4 9
```

There are some built in functions to "vectorize" other functions.

Functions: Creating and Passing

Functions are created like regular variables and can be treated like any other object:

```
> withfile <- function(fname, f) {
    sink(fname)
    f()
    sink()
  }
> withfile("myoutput.txt", function() {
    print("hi")
    print("bye")
  })
```

Functions: Arguments

Arguments can be named and given default values. Special `...` argument captures any other passed arguments.

```
> f <- function(a, b, c = 0.5, ...) {
    round(a:b * c, ...)
 }


> f(1, 3)
[1] 0 1 2
> f(b = 3, a = 1)
[1] 0 1 2
> f(c = 1/3, 1 , 3, digits = 3)
[1] 0.333 0.667 1.000
```

Booleans and Comparisons

- Standard boolean operations:

```
> c(TRUE && TRUE, TRUE && FALSE, FALSE || TRUE)
[1]  TRUE FALSE  TRUE
```

- Single boolean `if-else` statements; `switch` function
- Elementwise boolean operations `|` and `&`:

```
> c(TRUE, TRUE, FALSE) | c(FALSE, TRUE, FALSE)
[1]  TRUE  TRUE FALSE
```

- Elementwise comparisons:

```
> ifelse(c(1,2,3,4,5) %% 2 == 0, "even", "odd")
[1] "odd"  "even" "odd"  "even" "odd"
```

Exercise: Function to compute multiples

From Project Euler Problem 1:

> *If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.*
> *Find the sum of all the multiples of 3 or 5 below 1000.*

- Write a function `isThreeOrFive` that returns a boolean (hint: %% is modulo).
- Apply the function to all values from 1 to 1000 (hint: `1:1000`).
- Use the `sum` function to answer the question.

Example solutions

- Vectorization:

```
> isThreeOrFive <- function(x) {
    # by using elementwise operation, this is vector
    (x %% 3 == 0) | (x %% 5 == 0)
  }
> sum((1:999)[isThreeOrFive(1:999)])
[1] 233168
```

- Explicit looping:

```
> s <- 0
> for (x in 1:999) {
    if (isThreeOrFive(x)) s <- s + x
  }
> s
[1] 233168
```

- Functional one liner:

```
> sum(Filter(x = 1:1000, f = isThreeOrFive))
```

Missing Values

- Missing values for any data type is notated with the special `NA` value.
- Missing values (`NA`) are neither true or false, but short circuiting can still occur:

```
> c(FALSE && NA, TRUE || NA)
[1] FALSE  TRUE
> c(TRUE && NA, FALSE || NA)
[1] NA NA
```

- Many functions will have special `NA` handling arguments:

```
> nas <- c(1, 2, 3, NA, 5)
> mean(nas)
[1] NA
> mean(nas, na.rm = TRUE)
[1] 2.75
```

Matrices

The `matrix` class holds a single mode of data in a square format.

- Convenient substripting notation style:
- Standard linear algebra tools available.
- The `array` class generalizes to dimensions > 2.

```
> matrix <- matrix(1:12, nrow = 3)
> m[1:2, 2:3]
     [,1] [,2]
[1,]    4    7
[2,]    5    8
```

Data Frames

The `data.frame` class holds multiple modes, `$` operator to get specific columns as vectors.

```
> df <- data.frame(nums = 1:10, letters = letters[1:10])
> df[1:3, ]
  nums letters
1    1       a
2    2       b
3    3       c
> df$letters
 [1] a b c d e f g h i j
Levels: a b c d e f g h i j
```

Formulas

A common interface is a formula with a left and right hand side:

```
> y ~ x1 + x2 + x3 * x4 + log(x5)
```

- Used in many model fitting and plotting routines.
- Short hand notation for interactions (`:` and `*`).
- Many functions permitted.
- Can pull variables from the enviroment, but usally better to combine with a `data = mydata` argument.

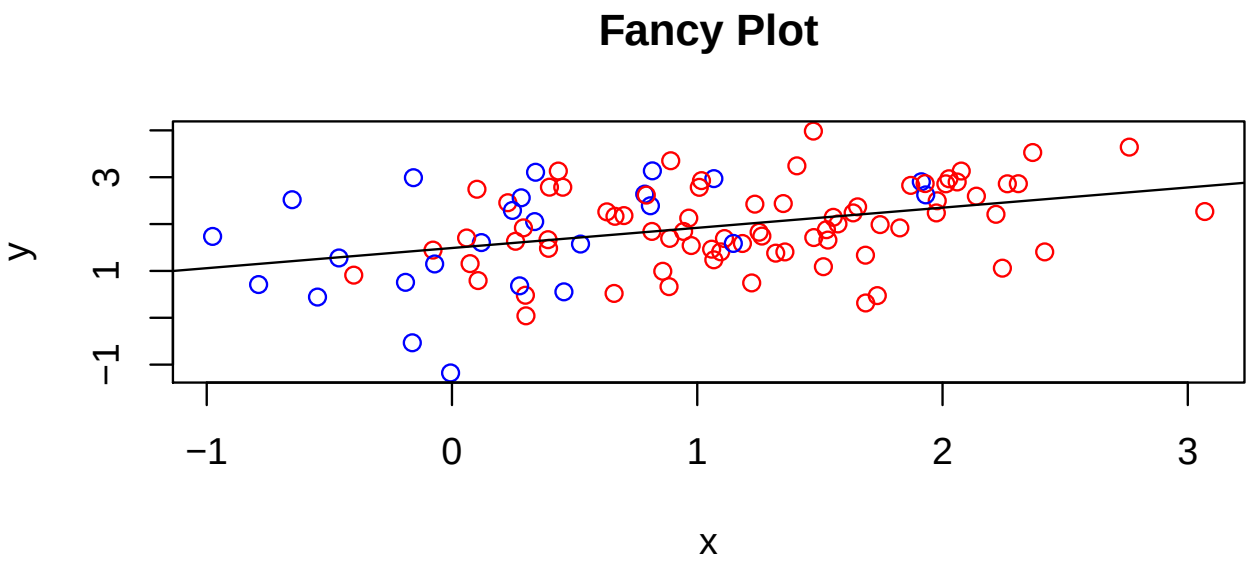
Example: Building a linear model

```
> library(MASS)
> randoms <- as.data.frame(
      mvrnorm(n = 100,
              mu = c(1,2),
              Sigma = matrix(c(1, 0.5, 0.5, 1),
                             nrow = 2)))
> colnames(randoms) <- c("x", "y")
> randoms$w <- (randoms$x + rnorm(100)) > 0
> model <- lm(y ~ x, data = randoms)
```

Plotting Basics

- Many objects have `plot` methods.
- These can be combined with other plotting primitives to create nice graphics.
- Output can be viewed interatively or saved as PDF, SVG, PNG, JPG, and others.

Example: Plotting regression model




```
> colors <- ifelse(randoms$w == 1, "red", "blue")
> plot(y ~ x, data = randoms, col = colors,
      main = "Fancy Plot")
> cs <- model$coefficients
> abline(a = cs[1], b = cs[2])
```

More examples

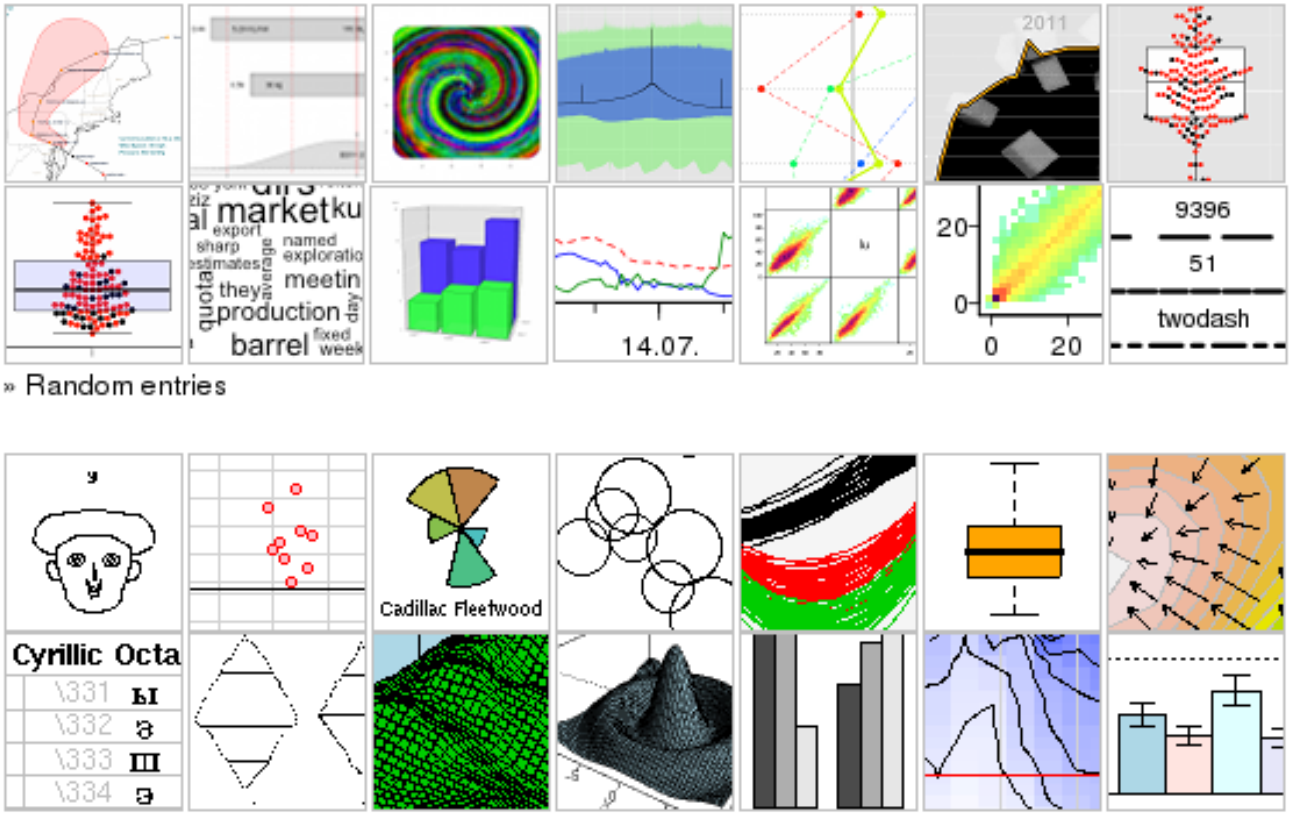
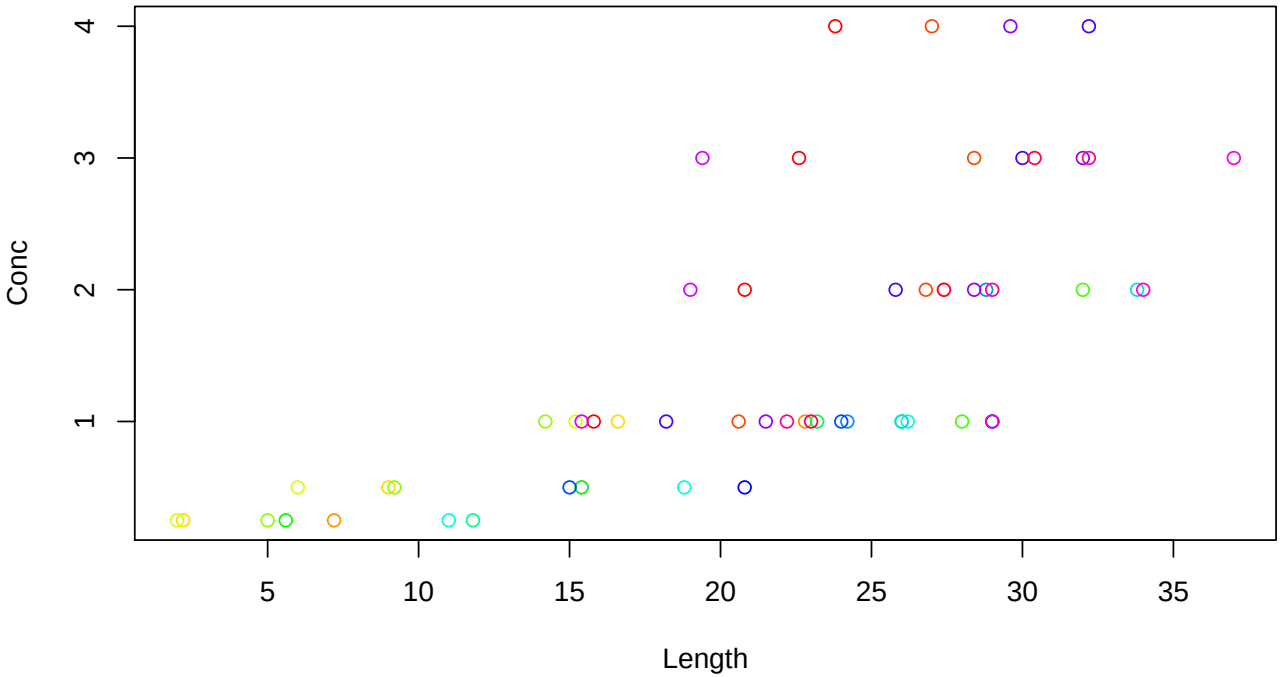


Figure : Mini examples from R Graphics Gallery

Exercise: Plotting Heart Muscle Experiment

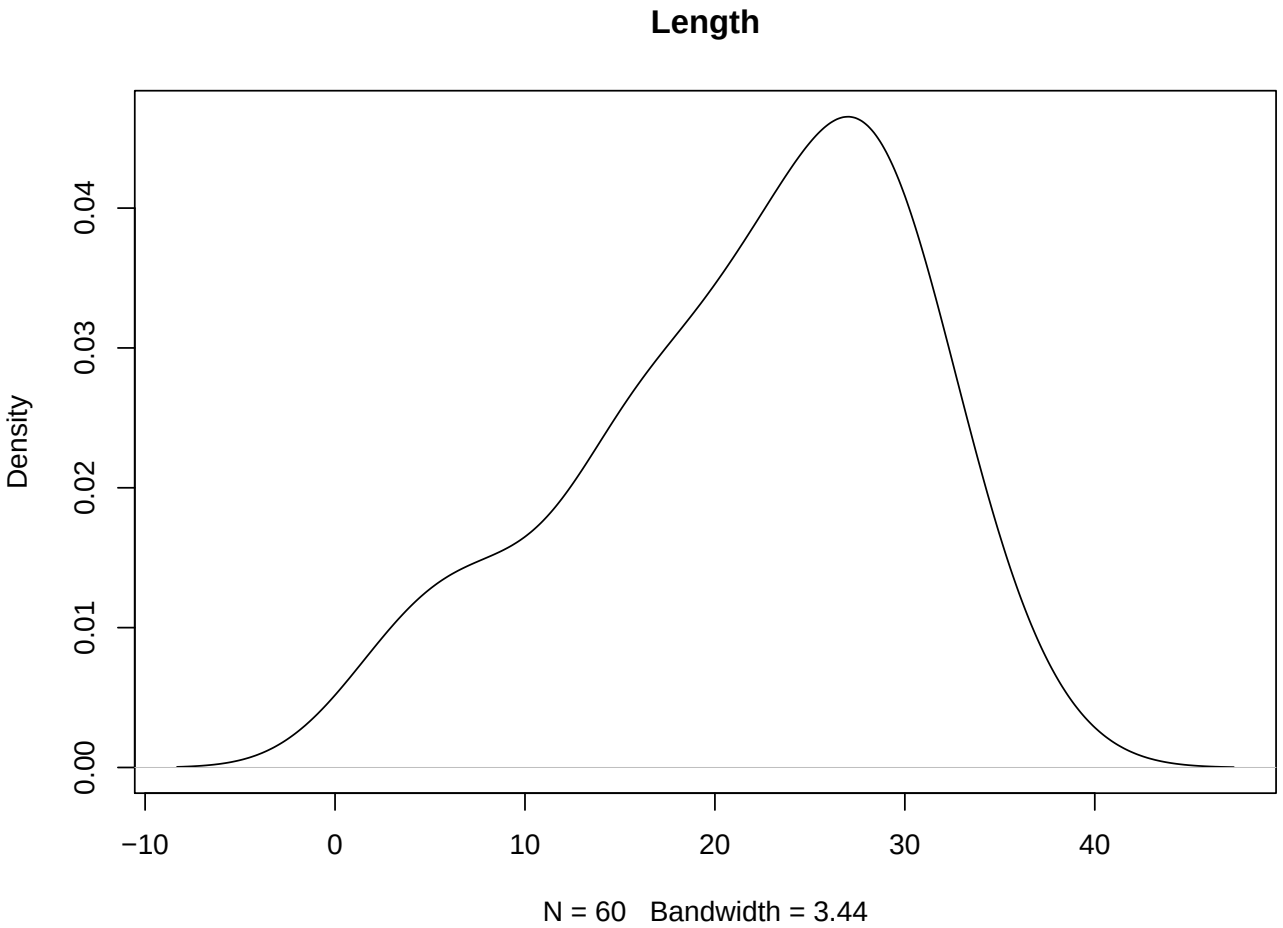
- Load the rat heart muscle experiment data: `data(muscle)`
- Plot `Conc` versus `Length`
- What is the distribution of the `Length` variable? (Hint: try plotting the results of `density` function)

Exercise: Plotting Heart Muscle Experiment




```
> cols <- rainbow(nlevels(muscle$Strip))
> plot(Conc ~ Length, data = muscle,
       col = cols[muscle$Strip])
```

Exercise: Length density plot




```
> plot(density(muscle$Length), main = "Length")
```

CRAN

The Comprehensive R Archive Network(CRAN) is a repository for community packages.

- 5066 packages as of 2013-12-30
- Easy to use with `install.packages("packageName")`
- Packages for different statistical techniques, plotting/graphics, parsing, development tools.
- Pre-built binaries for Windows and OS X. Source builds for other platforms.

Installing and Using Packages

```
> install.packages("e1071")
```

```
> library(e1071)
> data(Titanic)
> m <- naiveBayes(Survived ~ ., data = Titanic)
> me <- list(class = "3rd", Sex = "Male",
             Age = "Adult")
> predict(m, newdata = me)
[1] No
Levels: No Yes
```

Some recommend packages

- `caret`: Machine learning meta package
- `lattice` and `ggplot`: Advance plotting packages.
- `xtable`: Formats tables as LaTeXand HTML
- `Rcpp`: Simplifies interfacing with C/C++.
- `plyr`: Data manipulation routines.

Sweave

- Embed R in LaTeX(e.g. these slides)
- Chunks are evaluated and (optionally) output TeX.
- Figures can be generated as well

`file.Rnw`

```
<<chunkname, eval = TRUE>>=
c(1, 2, 3) + c(4, 5, 6)
@
```

`file.tex`

```
\begin{Schunk}
\begin{Sinput}
> c(1, 2, 3) + c(4, 5, 6)
\end{Sinput}
\begin{Soutput}
[1] 5 7 9
\end{Soutput}
\end{Schunk}
```

Using R to Create Artifacts

- Projects with multiple languages, phases
- We use `make` to map out dependencies and run `R`
- The `save` function serializes most data types.
- Typical workflow:
  - Load data (usually a `.csv` or similar)
  - Process data using `R` and save `.rda` file
  - Analysis phases consume `data.rda` and produce more `.rda` files
  - Figures, tables, etc. rely on these items and are built via `Makefile`
  - Output documents (`paper.pdf`, `presentation.pdf`) depend on entire collection.

Interfacing R with Other Languages

- `R` is largely built on C and Fortran and interfacing is pretty straight forward.
- `Rcpp` and `inline` packages make it even easier (including compiling writing C++ as a string)
- `RInside` is a package to allow calling `R` from C++
- `RPy2`, `RJava`, `RinRuby`, `statistics::R`, probably others
- `foreign` library can read many formats
- `RSQLite`: interact with `sqlite3` files, bindings for most languages

Books

- *The Art of R Programming* by Norman Matloff
- *Software for Data Analysis: Programming with R* by John Chambers
- *Reproducible Research in R and RStudio* by Christopher Gandrud
- O'Reilly `R` cookbooks
- Many books that teach specific statistical techniques with `R` demonstrations

Web

- `r-project.org`: downloads, introductory guides, packages
- Searching the web for "R" is an extreme frustration (get used to it)
- R Inferno: common pitfalls and workarounds
- Mailing lists: r-help, r-devel
- `stackoverflow.com` and `stats.stackexchange.com`
- `r-bloggers.com`

Editors

- `R` for Windows and OS X ship with REPL and editor
- `Rcmdr` and `tinR` attempt to add a full GUI
- emacs: *Emacs Speaks Statistics* is a set of modes, including a REPL and Sweave interaction
- vim: syntax highlighting, I was never able to get interactive stuff to work (I use emacs + EVIL + ESS)
- RStudio: commerical IDE freeware; available as Java app and in-browser interface

Thank You

Presentation can be found at:
https://github.com/SIAM-at-Illinois/Getting-Started-Series