NASA Summer Short Course for Earth System Modeling and Supercomputing

# Hands-on R for Climate Data Analysis

*James McCreight*
*NASA Postdoctoral Fellow*
*email: mccreigh (at) gmail*
*7/20/2012*

# Why use R ?

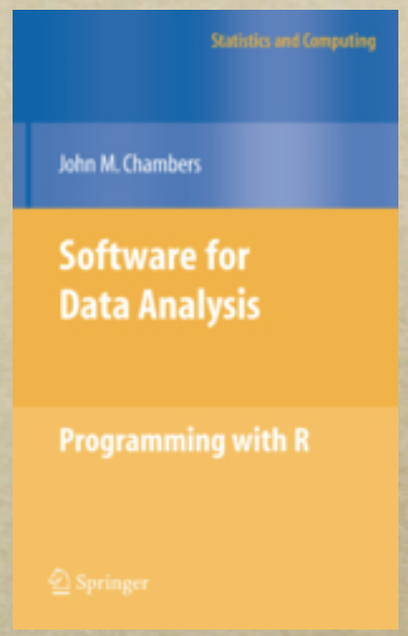
- *Free*
- *Platform independent (mostly)*
- *Open source*
- *A full-fledged programming language*
- *High-performance: parallelism & BLAS*
- *Sweave and knitr integration*

# Why use R? (2)

- *Packages: CRAN, bioconductor, omegahat*
  - *many tools developed by experts with accompanying books or papers(e.g. 1(1), 2(2), 3(3))*
- *WinBUGS/OpenBUGS/JAGS*
- *GRASS GIS*
- *Vibrant community/communities*
  - *used across a wide variety of fields*
  - *tremendous resources: books, papers, forums, conferences*

# Why use R.

- *SoDA: software for data analysis*



- *ggplot2*

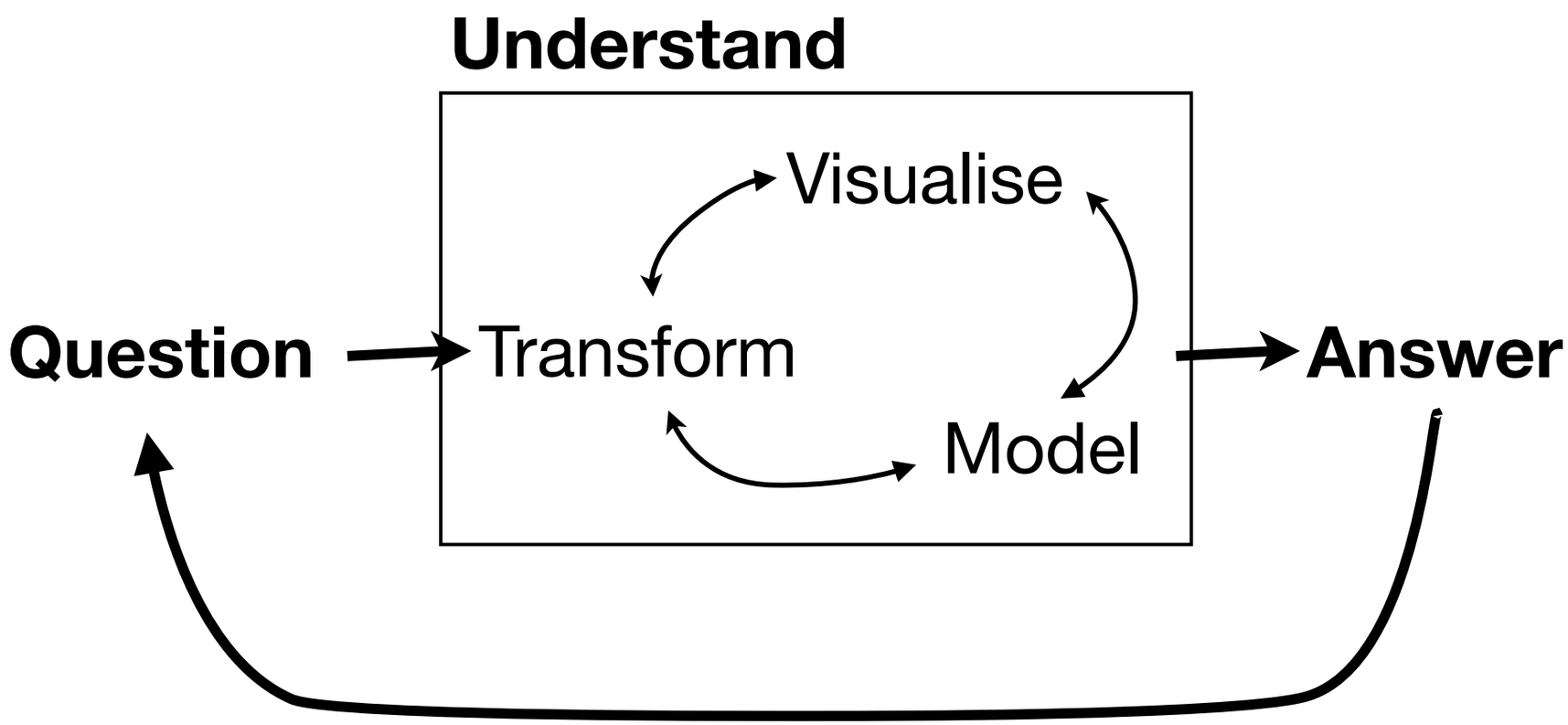
- *plyr*

# SoDA

"This is a book about *Software for Data Analysis*: using computer software to extract information from some source of data by organizing, visualizing, modeling, or performing any other relevant computation on the data."

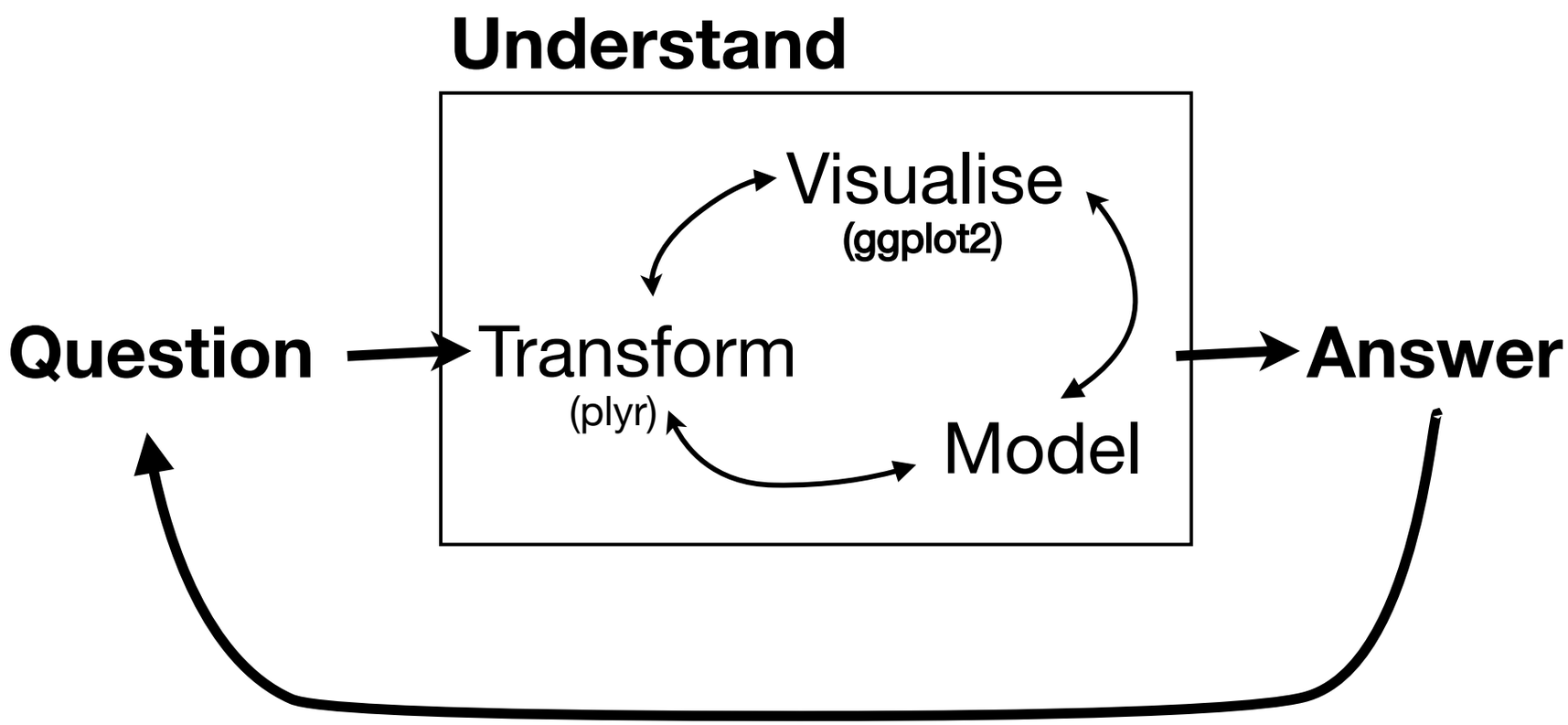
- John Chambers, award winning author of the S language from which R evolved

# SoDA




Adapted from Hadley Wickham

# SoDA




Adapted from Hadley Wickham

# Goal

- *Illustrate fundamentals for efficient data exploration in R.*
  - *(R is different from most scientific programming languages.)*

# How to use R.

*Hadley Wickham*

- *http://had.co.nz/*

*Advanced R Development (book in progress)*

- *https://github.com/hadley/devtools/wiki*

- *Some material in this presentation is adapted from Hadley's resources and about his packages (ggplot2 and plyr among these).*

# Outline

- *Fundamentals:*
  - *Cheat sheet, configuration tips (0)*
  - *Functional programming (10)*
  - *Lists and data frames (2)*
  - *Object oriented programming (1)*
  - *plyr and parallelization (2)*
- *Extended example using TRMM 3B42*

# Functional programming

- *first-class functions: "<u>functions are first class citizens</u>"*
  - ➡ *functions can be stored in data structures*
  - ➡ *functions can take functions as arguments (higher-order functions)*
  - ➡ *functions don't need names (anonymous)*
  - ➡ *functions can be written by functions (closures)*

If you're not using functions in R, you're doing it inefficiently.

# First-class functions examples

```
data <- rnorm(1000) ## gaussian data

## non-example! more an excuse to show some R basics.
## not really functions in the list, rather their evaluations
mySummary <- function(x) {
  list(mean=mean(x), median=median(x), sd=sd(x), mad=mad(x), IQR=IQR(x))
}
dataSummary <- mySummary(data)
dataSummary
str(dataSummary)
dataSummary$median
dataSummary['median']
dataSummary[['median']]
dataSummary[2]
dataSummary[[2]]
```

# Functions in a list, anonymous function in an HOF

```
## let's put functions in the list
summary2 <- function(x, statList) {
  for(iStat in 1:length(statList))
    statList[iStat]<-statList[[iStat]](x)
  statList
}
funcList <- list(mean=mean, min=min, max=max)
str(funcList)
dataSummary2 <- summary2(data, funcList)
str(dataSummary2)

## let's not use for on a list.
also, note an anonymous function. and lapply accepts functions (HOF).
summary2.1 <- function(x, statList) lapply(statList, function(iStat) iStat(x))
summary2.1(data, list(mean=mean, min=min, max=max))
```

# Closures

```
## closure: a function which returns a function (with its environment).
## creates a new function which has fixed arguments to the orig func.
makeSummary <- function(allFun)
  function(x) lapply(allFun, function(iFun) iFun(x))
summary3 <- makeSummary(list(mean=mean, min=min, max=max))
str(summary3)
summary3
dataSummary3 <- summary3(data)
dataSummary3
makeSummary(c(mean, min, max))(data)
```

# Make your own closure

*Create a function to create logarithm function of arbitrary base.*

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$$

# Make your own closure

*Create a function to create logarithm function of arbitrary base.*

$$\log_b(x) = \frac{\log_k(x)}{\log_k(b)}$$

```
logBaseB <- function(b) function(x) log(x)/log(b)
log17 <- logBaseB(17)
str(log17)
17^log17(3)
17^logBaseB(17)(3)
pi^logBaseB(pi)(3)
```

# Lexical scoping

```r
## a not defined in the function env,
## looks for a in its enclosing env WHERE DEFINED.
f <- function() a/2
a=10
f()

## where is f enclosed? where is it called?
g <- function() { a=100; f() }
g()

## f is now defined in h but not modified in the global env.
h <- function() { a=100; f <- function() a/5; f() }
h()
## note neither a nor f is modified in the enclosing env.
a; f

## "rescope" f to g2. some frown on this but it can be useful.
g2 <- function() { a=100; environment(f) <- environment(); f() }
g2()
## copy on modify! f was not harmed as orig defined.
f()
```

# Copy on modify

```
## copy on modify: there are no pointers.
## if not modified, not copied.
k <- function(x) x <- x/2
## note the return does not involve a print, just assignment.
a=10
k(a)
print(k(a))
a

## likewise for variables not passed.
k2 <- function() a <- a/2
print(k2())
a
```

# Lexical scoping & copy on modify

- *Careful! Don't use lexical scoping unintentionally: you could accidentally call a variable you forgot to set if the same name is used in an enclosing env!*
- *Lexical scoping + copy on modify provide plug-and-play modularity for functions*

# Lexical assignment

```
## <<- is lexical assignment. often frowned upon by gurus.
if (exists('qq')) rm(qq)  ## erase qq variable.
## qq is set in the environment of h.
h <- function(x) qq <- x
h(1)
exists('qq')

## lexical assignment searches for qq in enclosing environs.
## if not found before or global, it's created in the global env.
h2 <- function(x) qq <<- x
h2(1)
exists('qq'); rm(qq)

## remember, h2 is enclosed in the global env.
hLex <- function(x) {qq=3; h2(x); print(qq)}
hLex(1)
qq; rm(qq)

## nothing gets set in the global env
## because qq was found within hLex2
hLex2 <- function(x) {qq=3; environment(h2) <- environment(); h2(x); print(qq)}
hLex2(1)
qq
```

# A few details about functions

- *Function arguments: rules and defaults*
- *Lazy evaluation of function arguments*
- ...

# Lists and data frames

- *Functions can only return one object*
- *Lists can contain anything and recursively*
- *Data frames are 2D+non-ragged lists*

```
theList <- list(a=factor(1:4,levels=1:5),b=letters[1:4],runif(4,0,5))
str(theList)
str(theDf <- data.frame(a=factor(1:4,levels=1:5),b=letters[1:4],runif(4,0,5)))
str(as.data.frame(theList)) ## annoyingly converts the 3rd element to factor.

theList2 <- list(a=1:4, b=pi, c=c('a','j','k'), f=lm)
as.data.frame(theList2)
```

# Lists and data frames: call it by name

```
theDf ## dataframes have names=colnames and rownames
str(theDf) ## nrow obs of ncol vars
names(theDf)
colnames(theDf)
names(theDf)[3] <- 'unif'
rownames(theDf)
rownames(theDf) <- c("jan",'feb','mar','apr')
theDf
theDf$month <- as.factor(rownames(theDf)) ## indexing shown in str()
theDf
## blank selects the entire dimension (row)
theDf[,c('month','b')] ##  single [ returns a sublist
theDf[[,'month']] ## [[ returns the object in the sublist, can only select 1 column
theDf[-3,] ## can exclude indices with neg index numbers (cant negate names)
theDf[-1*(which(rownames(theDf)=='mar')),] ## which returns index on logical
theDf[!(rownames(theDf)=='mar'),] ## can also use logical indices
```

# OO in R: S3 & S4

- *S3: lite objects = lists with attributes (more common).*
- *S4 is a more formal OO.*
- *both have methods. it's important to be aware that R uses methods a lot.*

```
df <- data.frame(x1=1:100, x2=abs(-50:49))
df$y <- 4*df$x1 + -5*df$x2 + rnorm(100)
str(theFit <- lm(y~., df)) ## S3 object
class(theFit)  ## this is how it's different from a list.
predict        ## a general method
predict.lm     ## the above calls this after checking the class
predict(theFit, data.frame(x1=75,x2=-4))
predict.lm(theFit, data.frame(x1=75,x2=-4))
```

# plyr

- *DRY (Dont Repeat Yourself) principle*
- *names (lists, dfs, arrays) + functions*
- *Streamlined "for" syntax, streamlined "apply" syntax: [i][o]ply(), e.g. llply, laply, daply, etc...*
- *similar to mapReduce: <u>The Split-Apply-Combine Strategy for Data Analysis</u>*
- *parallel backends for plyr: convenient/embarrass ||*
- *parallelization vs paralyzation: memory is more important than CPU*

# plyr examples

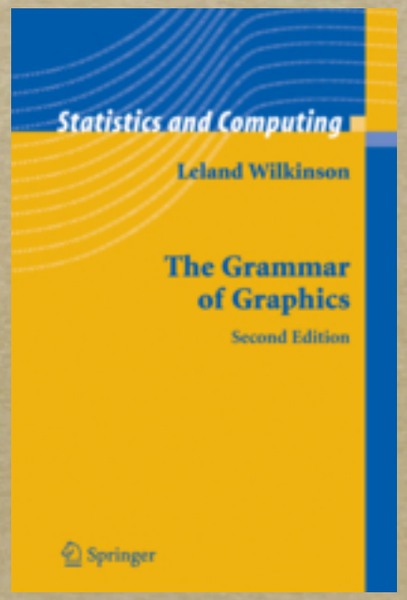
```
?llply
theList <- list(a=1:4, c=c('a','j','k'),
                f=rnorm(100), l=list(x=1:4,y=rnorm(4)))
str(ele2 <- llply(theList, '[[', 2)) ## subsetting functions '[','[['
str(list2 <- llply(theList, '[', 2)) ## the list data is different
```

# Rieman sum example in R

```
theFunc <- function(x) 4/(1+x^2) ## x is a vector
riemannMid <- function(n,f) sum( f(seq(0+(1/n)/2,1-(1/n)/2,1/n)) *1/n )
riemannMidClosure <- function(f)
  function(n) sum( f(seq(0+(1/n)/2,1-(1/n)/2,1/n)) *1/n)
system.time( print(riemannMid(1e7,theFunc)) )['elapsed']
system.time( print(riemannMidClosure(theFunc)(1e7)) )['elapsed']

## let's look at the accuracy as a function of n.
laply( 10^(1:6), riemannMid, f=theFunc)-pi
## look at time as a function of n.
laply( 10^(1:7), function(n) system.time(riemannMid(n,theFunc)-pi)[3] )
## do it in a nicer way, actually use a named list for input.
inN <- as.list(10^(1:7)); names(inN) <- unlist(inN)
laply( inN, function(n) system.time(riemannMid(n,theFunc)-pi)[3] )
llply( inN, function(n) system.time(riemannMid(n,theFunc)-pi)[3] )
## clearly, this is a much nicer format for plotting than any of the above.
print(timeDf <-
     ldply( inN, function(n) system.time(riemannMid(n,theFunc)-pi)[3] ))
plot(timeDf$.id, timeDf$elapsed,log='x')
ggplot( timeDf, aes(x=.id,y=elapsed)) + geom_point()
ggplot( timeDf, aes(x=as.numeric(.id),y=elapsed)) + geom_point() +
  geom_line() + scale_x_log10()
```

# TRMM example and ggplot

- *TRMM 3B42 v7 daily*
  - *daily avg precip rate (mm/hr)*
  - *gridded, quater degree*
  - *50S-50N: 400x1440=576,000 tiles*
  - *1/1/1998 - 1/2/2012=5,115 days*
  - *2,946,240,000 precip values*

- *grammar of graphics*