

Getting Help

Computing for Data Analysis

Asking Questions

- Asking questions via email is different from asking questions in person
- People on the other side do not have the background information you have
 - they also don't know you personally (usually)
- Other people are busy; their time is limited
- The instructor (me) is here to help in all circumstances but may not be able to answer all questions!

Finding Answers

- Try to find an answer by searching the archives of the forum you plan to post to.
- Try to find an answer by searching the Web.
- Try to find an answer by reading the manual.
- Try to find an answer by reading a FAQ.
- Try to find an answer by inspection or experimentation.
- Try to find an answer by asking a skilled friend.
- If you're a programmer, try to find an answer by reading the source code.

Asking Questions

- It's important to let other people know that you've done all of the previous things already
- If the answer is in the documentation, the answer will be "Read the documentation"
 - one email round wasted

Example: Error Messages


```
> library(datasets)
```


```
> data(airquality)
```


```
> cor(airquality)
```

```
Error in cor(airquality) : missing observations in  
cov/cor
```

Google is Your Friend





About 508,000 results (0.28 seconds)

[\[R\] Error in cor.default\(x1, x2\) : missing observations in cov/cor](#)
<https://stat.ethz.ch/pipermail/r-help/2008.../155523.html> - Block stat.ethz.ch
Mar 1, 2008 – [R] Error in cor.default(x1, x2) : **missing observations in cov/cor**. Daniel Malter daniel at umd.edu. Thu Feb 28 01:40:58 CET 2008. Previous message: [R] Error in ...
[\[R\] Null values in R.](#) - Dec 3, 2008
[\[R\] cor\(data.frame\) infelicities](#) - Dec 3, 2007
[\[BioC\] maanova: missing observations in cov/cor](#) - Aug 25, 2006
[\[R-sig-Geo\] Some comments on calculations with 'asc' \(and 'kasc ...](#) - Mar 27, 2005
[More results from stat.ethz.ch »](#)

[EMBnet 2007 Course - Introduction to Statistics for Biologists](#)
www.ch.embnet.org/CoursEMBnet/Basel07/tp2b.html - Block ch.embnet.org
you will get an error: Error in cor(thuesen) : **missing observations in cov/cor** . [You will have seen a similar message above with sd .] This is because of the ...

[R help - Error in cor.default\(x1, x2\) : missing observations in cov/cor](#)
r.789695.n4.nabble.com/Error-in-cor-default-x1-x2-missing-... - Block
r.789695.n4.nabble.com
13 posts - 5 authors - Feb 27, 2008
Error in cor.default(x1, x2) : **missing observations in cov/cor**. Hello, I'm trying to do cor(x1,x2) and I get the following error: Error in cor.default(x1, ...

Asking Questions

- What steps will reproduce the problem?
- What is the expected output?
- What do you see instead?
- What version of the product (e.g. R, packages, etc.) are you using?
- What operating system?
- Additional information

Subject Headers

- Stupid:

HELP! Can't fit linear model!

- Smart:

R 2.15.0 lm() function produces seg fault with large data frame, Mac OS X 10.6.3

- Smarter:

R 2.15.0 lm() function on Mac OS X 10.6.3 -- seg fault on large data frame

Do

- Describe the goal, not the step
- Be explicit about your question
- Do provide the minimum amount of information necessary (volume is not precision)
- Be courteous (it never hurts)
- Follow up with the solution (if found)

Don't

- Claim that you've found a bug
- Grovel as a substitute for doing your homework
- Post homework questions on mailing lists (we've seen them all)
- Email multiple mailing lists at once
- Ask others to debug your broken code without giving a hint as to what sort of problem they should be searching for

Case Study: A Recent Post to the R-devel Mailing List

Subject: large dataset – confused

Message:

I'm trying to load a dataset into R, but I'm completely lost. This is probably due mostly to the fact that I'm a complete R newb, but it's got me stuck in a research project.

Response

Yes, you are lost. The R posting guide is at <http://www.r-project.org/posting-guide.html> and will point you to the right list and also the manuals (at e.g. <http://cran.r-project.org/manuals.html>, and one of them seems exactly what you need).

Analysis: What Went Wrong?

- Question was sent to the wrong mailing list (R-devel instead of R-help)
- Email subject was very vague
- Question was very vague
- Problem was not reproducible
- No evidence of any effort made to solve the problem
- RESULT: Recipe for disaster!

Places to Turn

- Class discussion board; your fellow students
- r-help@r-project.org
- Other project-specific mailing lists

(This talk inspired by Eric Raymond's "How to ask questions the smart way")

Overview and History of R

Computing for Data Analysis

What is R?

What is R?

What is R?

R is a dialect of the S language.

What is S?

- S is a language that was developed by John Chambers and others at Bell Labs.
- S was initiated in 1976 as an internal statistical analysis environment—originally implemented as Fortran libraries.
- Early versions of the language did not contain functions for statistical modeling.
- In 1988 the system was rewritten in C and began to resemble the system that we have today (this was Version 3 of the language). The book *Statistical Models in S* by Chambers and Hastie (the white book) documents the statistical analysis functionality.
- Version 4 of the S language was released in 1998 and is the version we use today. The book *Programming with Data* by John Chambers (the green book) documents this version of the language.

- In 1993 Bell Labs gave StatSci (now Insightful Corp.) an exclusive license to develop and sell the S language.
- In 2004 Insightful purchased the S language from Lucent for \$2 million and is the current owner.
- In 2006, Alcatel purchased Lucent Technologies and is now called Alcatel-Lucent.
- Insightful sells its implementation of the S language under the product name S-PLUS and has built a number of fancy features (GUIs, mostly) on top of it—hence the “PLUS”.
- In 2008 Insightful is acquired by TIBCO for \$25 million
- The fundamentals of the S language itself has not changed dramatically since 1998.
- In 1998, S won the Association for Computing Machinery's Software System Award.

In “Stages in the Evolution of S”, John Chambers writes:

“[W]e wanted users to be able to begin in an interactive environment, where they did not consciously think of themselves as programming. Then as their needs became clearer and their sophistication increased, they should be able to slide gradually into programming, when the language and system aspects would become more important.”

<http://www.stat.bell-labs.com/S/history.html>

- 1991: Created in New Zealand by Ross Ihaka and Robert Gentleman. Their experience developing R is documented in a 1996 *JCGS* paper.
- 1993: First announcement of R to the public.
- 1995: Martin Mächler convinces Ross and Robert to use the GNU General Public License to make R free software.
- 1996: A public mailing list is created (R-help and R-devel)
- 1997: The R Core Group is formed (containing some people associated with S-PLUS). The core group controls the source code for R.
- 2000: R version 1.0.0 is released.
- 2012: R version 2.15.1 is released on June 22, 2012.

- Syntax is very similar to S, making it easy for S-PLUS users to switch over.
- Semantics are superficially similar to S, but in reality are quite different (more on that later).
- Runs on almost any standard computing platform/OS (even on the PlayStation 3)
- Frequent releases (annual + bugfix releases); active development.

Features of R (cont'd)

- Quite lean, as far as software goes; functionality is divided into modular packages
- Graphics capabilities very sophisticated and better than most stat packages.
- Useful for interactive work, but contains a powerful programming language for developing new tools (user \longrightarrow programmer)
- Very active and vibrant user community; R-help and R-devel mailing lists and Stack Overflow

Features of R (cont'd)

It's free!

(Both in the sense of beer and in the sense of speech.)

With *free software*, you are granted

- The freedom to run the program, for any purpose (freedom 0).
- The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help your neighbor (freedom 2).
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this.

<http://www.fsf.org>

Drawbacks of R

- Essentially based on 40 year old technology.
- Little built in support for dynamic or 3-D graphics (but things have improved greatly since the “old days”).
- Functionality is based on consumer demand and user contributions. If no one feels like implementing your favorite method, then it's *your* job!
 - (Or you need to pay someone to do it)
- Objects must generally be stored in physical memory; but there have been advancements to deal with this too
- Not ideal for all possible situations (but this is a drawback of all software packages).

The R system is divided into 2 conceptual parts:

- 1 The “base” R system that you download from CRAN
- 2 Everything else.

R functionality is divided into a number of *packages*.

- The “base” R system contains, among other things, the **base** package which is required to run R and contains the most fundamental functions.
- The other packages contained in the “base” system include **utils**, **stats**, **datasets**, **graphics**, **grDevices**, **grid**, **methods**, **tools**, **parallel**, **compiler**, **splines**, **tcltk**, **stats4**.
- There are also “Recommend” packages: **boot**, **class**, **cluster**, **codetools**, **foreign**, **KernSmooth**, **lattice**, **mgcv**, **nlme**, **rpart**, **survival**, **MASS**, **spatial**, **nnet**, **Matrix**.

And there are many other packages available:

- There are about 4000 packages on CRAN that have been developed by users and programmers around the world.
- There are also many packages associated with the Bioconductor project (<http://bioconductor.org>).
- People often make packages available on their personal websites; there is no reliable way to keep track of how many packages are available in this fashion.

Available from CRAN (<http://cran.r-project.org>)

- An Introduction to R
- Writing R Extensions
- R Data Import/Export
- R Installation and Administration (mostly for building R from sources)
- R Internals (not for the faint of heart)

Some Useful Books on S/R

Standard texts

- Chambers (2008). *Software for Data Analysis*, Springer. (your textbook)
- Chambers (1998). *Programming with Data*, Springer.
- Venables & Ripley (2002). *Modern Applied Statistics with S*, Springer.
- Venables & Ripley (2000). *S Programming*, Springer.
- Pinheiro & Bates (2000). *Mixed-Effects Models in S and S-PLUS*, Springer.
- Murrell (2005). *R Graphics*, Chapman & Hall/CRC Press.

Other resources

- Springer has a series of books called *Use R!*.
- A longer list of books is at <http://www.r-project.org/doc/bib/R-books.html>

Introduction to the R Language

Data Types and Basic Operations

Computing for Data Analysis

R has five basic or “atomic” classes of objects:

- character
- numeric (real numbers)
- integer
- complex
- logical (True/False)

The most basic object is a vector

- A vector can only contain objects of the same class
- BUT: The one exception is a *list*, which is represented as a vector but can contain objects of different classes (indeed, that’s usually why we use them)

Empty vectors can be created with the `vector()` function.

- Numbers in R are generally treated as numeric objects (i.e. double precision real numbers)
- If you explicitly want an integer, you need to specify the L suffix
- Ex: Entering 1 gives you a numeric object; entering 1L explicitly gives you an integer.
- There is also a special number `Inf` which represents infinity; e.g. $1 / 0$; `Inf` can be used in ordinary calculations; e.g. $1 / \text{Inf}$ is 0
- The value `NaN` represents an undefined value (“not a number”); e.g. $0 / 0$; `NaN` can also be thought of as a missing value (more on that later)

R objects can have attributes

- names, dimnames
- dimensions (e.g. matrices, arrays)
- class
- length
- other user-defined attributes/metadata

Attributes of an object can be accessed using the `attributes()` function.

Entering Input

At the R prompt we type *expressions*. The `<-` symbol is the assignment operator.

```
> x <- 1
> print(x)
[1] 1
> x
[1] 1
> msg <- "hello"
```

The grammar of the language determines whether an expression is complete or not.

```
> x <- ## Incomplete expression
```

The `#` character indicates a *comment*. Anything to the right of the `#` (including the `#` itself) is ignored.

When a complete expression is entered at the prompt, it is *evaluated* and the result of the evaluated expression is returned. The result may be *auto-printed*.

```
> x <- 5  ## nothing printed
> x        ## auto-printing occurs
[1] 5
> print(x) ## explicit printing
[1] 5
```

The [1] indicates that x is a vector and 5 is the first element.

```
> x <- 1:20  
> x  
[1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15  
[16] 16 17 18 19 20
```

The `:` operator is used to create integer sequences.

Creating Vectors

The `c()` function can be used to create vectors of objects.

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

Using the `vector()` function

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

What about the following?

```
> y <- c(1.7, "a")    ## character  
> y <- c(TRUE, 2)     ## numeric  
> y <- c("a", TRUE)   ## character
```

When different objects are mixed in a vector, *coercion* occurs so that every element in the vector is of the same class.

Explicit Coercion

Objects can be explicitly coerced from one class to another using the `as.*` functions, if available.

```
> x <- 0:6
> class(x)
[1] "integer"
> as.numeric(x)
[1] 0 1 2 3 4 5 6
> as.logical(x)
[1] FALSE TRUE TRUE TRUE TRUE TRUE TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
> as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```


Explicit Coercion

Nonsensical coercion results in NAs.

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

```
> as.numeric(x)
```

```
[1] NA NA NA
```

Warning message:

NAs introduced by coercion

```
> as.logical(x)
```

```
[1] NA NA NA
```

Matrices are vectors with a *dimension* attribute. The dimension attribute is itself an integer vector of length 2 (nrow, ncol)

```
> m <- matrix(nrow = 2, ncol = 3)
```

```
> m
```

```
      [,1] [,2] [,3]  
[1,]   NA   NA   NA  
[2,]   NA   NA   NA
```

```
> dim(m)
```

```
[1] 2 3
```

```
> attributes(m)
```

```
$dim
```

```
[1] 2 3
```

Matrices (cont'd)

Matrices are constructed *column-wise*, so entries can be thought of starting in the “upper left” corner and running down the columns.

```
> m <- matrix(1:6, nrow = 2, ncol = 3)
```

```
> m
```

	[,1]	[,2]	[,3]
[1,]	1	3	5
[2,]	2	4	6

Matrices (cont'd)

Matrices can also be created directly from vectors by adding a dimension attribute.

```
> 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
```

cbind-ing and rbind-ing

Matrices can be created by *column-binding* or *row-binding* with `cbind()` and `rbind()`.

```
> 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
```

Lists are a special type of vector that can contain elements of different classes. Lists are a very important data type in R and you should get to know them well.

```
> x <- list(1, "a", TRUE, 1 + 4i)
```

```
> x
```

```
[[1]]
```

```
[1] 1
```

```
[[2]]
```

```
[1] "a"
```

```
[[3]]
```

```
[1] TRUE
```

```
[[4]]
```

```
[1] 1+4i
```

Factors are used to represent categorical data. Factors can be unordered or ordered. One can think of a factor as an integer vector where each integer has a *label*.

- Factors are treated specially by modelling functions like `lm()` and `glm()`
- Using factors with labels is *better* than using integers because factors are self-describing; having a variable that has values “Male” and “Female” is better than a variable that has values 1 and 2.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"))
> x
[1] yes yes no  yes no
Levels: no yes
> table(x)
x
no yes
 2  3
> unclass(x)
[1] 2 2 1 2 1
attr("levels")
[1] "no" "yes"
```


The order of the levels can be set using the `levels` argument to `factor()`. This can be important in linear modelling because the first level is used as the baseline level.

```
> x <- factor(c("yes", "yes", "no", "yes", "no"),  
              levels = c("yes", "no"))  
  
> x  
[1] yes yes no  yes no  
Levels: yes no
```

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

Missing Values

```
> 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
```

Data frames are used to store tabular data

- They are represented as 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 and the length of each element of the list is the number of rows
- Unlike matrices, data frames can store different classes of objects in each column (just like lists); matrices must have every element be the same class
- Data frames also have a special attribute called `row.names`
- Data frames are usually created by calling `read.table()` or `read.csv()`
- Can be converted to a matrix by calling `data.matrix()`

```
> x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
> x
  foo  bar
1   1 TRUE
2   2 TRUE
3   3 FALSE
4   4 FALSE
> nrow(x)
[1] 4
> ncol(x)
[1] 2
```

R objects can also have names, which is very useful for writing readable code and self-describing objects.

```
> x <- 1:3
> names(x)
NULL
> names(x) <- c("foo", "bar", "norf")
> x
  foo  bar norf
   1    2    3
> names(x)
[1] "foo"  "bar"  "norf"
```

Lists can also have names.

```
> x <- list(a = 1, b = 2, c = 3)
```

```
> x
```

```
$a
```

```
[1] 1
```

```
$b
```

```
[1] 2
```

```
$c
```

```
[1] 3
```

And 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
```


Data Types

- atomic classes: numeric, logical, character, integer, complex
- vectors, lists
- factors
- missing values
- data frames
- names

Introduction to the R Language

Data Types and Basic Operations

Computing for Data Analysis

There are a number of operators that can be used to extract subsets of R objects.

- `[]` always returns an object of the same class as the original; can be used to select more than one element (there is one exception)
- `[[` is used to extract elements of a list or a data frame; it can only be used to extract a single element and the class of the returned object will not necessarily be a list or data frame
- `$` is used to extract elements of a list or data frame by name; semantics are similar to hat of `[]`.

```
> x <- c("a", "b", "c", "c", "d", "a")
> x[1]
[1] "a"
> x[2]
[1] "b"
> x[1:4]
[1] "a" "b" "c" "c"
> x[x > "a"]
[1] "b" "c" "c" "d"
> u <- x > "a"
> u
[1] FALSE TRUE TRUE TRUE TRUE FALSE
> x[u]
[1] "b" "c" "c" "d"
```

Subsetting a Matrix

Matrices can be subsetting in the usual way with (i,j) type indices.

```
> x <- matrix(1:6, 2, 3)
> x[1, 2]
[1] 3
> x[2, 1]
[1] 2
```

Indices can also be missing.

```
> x[1, ]
[1] 1 3 5
> x[, 2]
[1] 3 4
```

Subsetting a Matrix

By default, when a single element of a matrix is retrieved, it is returned as a vector of length 1 rather than a 1×1 matrix. This behavior can be turned off by setting `drop = FALSE`.

```
> x <- matrix(1:6, 2, 3)
```

```
> x[1, 2]
```

```
[1] 3
```

```
> x[1, 2, drop = FALSE]
```

```
  [,1]
```

```
[1,] 3
```

Subsetting a Matrix

Similarly, subsetting a single column or a single row will give you a vector, not a matrix (by default).

```
> x <- matrix(1:6, 2, 3)
> x[1, ]
[1] 1 3 5
> x[1, , drop = FALSE]
      [,1] [,2] [,3]
[1,]    1    3    5
```

Subsetting Lists

```
> x <- list(foo = 1:4, bar = 0.6)
```

```
> x[1]
```

```
$foo
```

```
[1] 1 2 3 4
```

```
> x[[1]]
```

```
[1] 1 2 3 4
```

```
> x$bar
```

```
[1] 0.6
```

```
> x[["bar"]]
```

```
[1] 0.6
```

```
> x["bar"]
```

```
$bar
```

```
[1] 0.6
```


Subsetting Lists

Extracting multiple elements of a list.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
```

```
> x[c(1, 3)]
```

```
$foo
```

```
[1] 1 2 3 4
```

```
$baz
```

```
[1] "hello"
```

Subsetting Lists

The `[[` operator can be used with *computed* indices; `$` can only be used with literal names.

```
> x <- list(foo = 1:4, bar = 0.6, baz = "hello")
> name <- "foo"
> x[[name]] ## computed index for 'foo'
[1] 1 2 3 4
> x$name     ## element 'name' doesn't exist!
NULL
> x$foo
[1] 1 2 3 4 ## element 'foo' does exist
```

Subsetting Nested Elements of a List

The `[[` can take an integer sequence.

```
> x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
```

```
> x[[c(1, 3)]]
```

```
[1] 14
```

```
> x[[1]][[3]]
```

```
[1] 14
```

```
> x[[c(2, 1)]]
```

```
[1] 3.14
```

Partial Matching

Partial matching of names is allowed with `[[` and `$`.

```
> x <- list(aardvark = 1:5)
> x$a
[1] 1 2 3 4 5
> x[["a"]]
NULL
> x[["a", exact = FALSE]]
[1] 1 2 3 4 5
```

Removing NA Values

A common task is to remove missing values (NAs).

```
> x <- c(1, 2, NA, 4, NA, 5)
> bad <- is.na(x)
> x[!bad]
[1] 1 2 4 5
```

Removing NA Values

What if there are multiple things and you want to take the subset with no missing values?

```
> x <- c(1, 2, NA, 4, NA, 5)
> y <- c("a", "b", NA, "d", NA, "f")
> good <- complete.cases(x, y)
> good
[1] TRUE TRUE FALSE TRUE FALSE TRUE
> x[good]
[1] 1 2 4 5
> y[good]
[1] "a" "b" "d" "f"
```

Removing NA Values

```
> airquality[1:6, ]
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5     NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> good <- complete.cases(airquality)
> airquality[good, ][1:6, ]
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
7    23     299  8.6   65     5   7
```

Introduction to the R Language

Vectorized Operations

Computing for Data Analysis

Vectorized Operations

Many operations in R are *vectorized* making code 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 >= 2
[1] FALSE  TRUE  TRUE  TRUE
> y == 8
[1] FALSE FALSE  TRUE FALSE
> x * y
[1]  6 14 24 36
> x / y
[1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Vectorized Matrix Operations

```
> x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
> x * y          ## element-wise multiplication
      [,1] [,2]
[1,]    10    30
[2,]    20    40
> x / y
      [,1] [,2]
[1,]  0.1  0.3
[2,]  0.2  0.4
> x %*% y        ## true matrix multiplication
      [,1] [,2]
[1,]    40    40
[2,]    60    60
```

Introduction to the R Language

Reading and Writing Data

Computing for Data Analysis

There are a few principal functions reading data into R.

- `read.table`, `read.csv`, for reading tabular data
- `readLines`, for reading lines of a text file
- `source`, for reading in R code files (inverse of `dump`)
- `dget`, for reading in R code files (inverse of `dput`)
- `load`, for reading in saved workspaces
- `unserialize`, for reading single R objects in binary form

There are analogous functions for writing data to files

- `write.table`
- `writeln`
- `dump`
- `dput`
- `save`
- `serialize`

Reading Data Files with `read.table`

The `read.table` function is one of the most commonly used functions for reading data. It has a few important arguments:

- `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?

For small to moderately sized datasets, you can usually call `read.table` without specifying any other arguments

```
data <- read.table("foo.txt")
```

R will automatically

- skip lines that begin with a `#`
- figure out how many rows there are (and how much memory needs to be allocated)
- figure 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.

Reading in Larger Datasets with `read.table`

With much larger datasets, doing the following things will make your life easier and will prevent R from choking.

- Read the help page for `read.table`, which contains many hints
- Make a rough calculation of the memory required to store your dataset. If the dataset is larger than the amount of RAM on your computer, you can probably stop right here.
- Set `comment.char = ""` if there are no commented lines in your file.

Reading in Larger Datasets with read.table

- Use the `colClasses` argument. Specifying this option instead of using the default can make 'read.table' run MUCH faster, often twice as fast. In order to use this option, you have to know the class of each column in your data frame. If all of the columns are "numeric", for example, then you can just set `colClasses = "numeric"`. A quick and dirty way to figure out the classes of each column is the following:

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)
tabAll <- read.table("datatable.txt",
                    colClasses = classes)
```

- Set `nrows`. This doesn't make R run faster but it helps with memory usage. A mild overestimate is okay. You can use the Unix tool `wc` to calculate the number of lines in a file.

In general, when using R with larger datasets, it's useful to know a few things about your system.

- How much memory is available?
- What other applications are in use?
- Are there other users logged into the same system?
- What operating system?
- Is the OS 32 or 64 bit?

Calculating Memory Requirements

I have a data frame with 1,500,000 rows and 120 columns, all of which are numeric data. Roughly, how much memory is required to store this data frame?

$$\begin{aligned} 1,500,000 \times 120 \times 8 \text{ bytes/numeric} &= 1440000000 \text{ bytes} \\ &= 1440000000 / 2^{20} \text{ bytes/MB} \\ &= 1,373.29 \text{ MB} \\ &= 1.34 \text{ GB} \end{aligned}$$

- `dumping` and `dputing` are useful because the resulting textual format is edit-able, and in the case of corruption, potentially recoverable.
- Unlike writing out a table or csv file, `dump` and `dput` preserve the *metadata* (sacrificing some readability), so that another user doesn't have to specify it all over again.
- Textual formats can work much better with version control programs like subversion or git which can only track changes meaningfully in text files
- Textual formats can be longer-lived; if there is corruption somewhere in the file, it can be easier to fix the problem
- Textual formats adhere to the “Unix philosophy”
- Downside: The format is not very space-efficient

dput-ting R Objects

Another way to pass data around is by deparsing the R object with `dput` and reading it back in using `dget`.

```
> y <- data.frame(a = 1, b = "a")
> dput(y)
structure(list(a = 1,
               b = structure(1L, .Label = "a",
                             class = "factor")),
          .Names = c("a", "b"), row.names = c(NA, -1L),
          class = "data.frame")
> dput(y, file = "y.R")
> new.y <- dget("y.R")
> new.y
  a b
1 1 a
```

Dumping R Objects

Multiple objects can be deparsed using the `dump` function and read back in using `source`.

```
> x <- "foo"
> y <- data.frame(a = 1, b = "a")
> dump(c("x", "y"), file = "data.R")
> rm(x, y)
> source("data.R")
> y
  a b
1 1 a
> x
[1] "foo"
```

Interfaces to the Outside World

Data are read in using *connection* interfaces. Connections can be made to files (most common) or to other more exotic things.

- `file`, opens a connection to a file
- `gzipfile`, opens a connection to a file compressed with gzip
- `bzfile`, opens a connection to a file compressed with bzip2
- `url`, opens a connection to a webpage

```
> str(file)
function (description = "", open = "", blocking = TRUE,
         encoding = getOption("encoding"))
```

- description is the name of the file
- open is a code indicating
 - “r” read only
 - “w” writing (and initializing a new file)
 - “a” appending
 - “rb”, “wb”, “ab” reading, writing, or appending in binary mode (Windows)

In general, connections are powerful tools that let you navigate files or other external objects. In practice, we often don't need to deal with the connection interface directly.

```
con <- file("foo.txt", "r")  
data <- read.csv(con)  
close(con)
```

is the same as

```
data <- read.csv("foo.txt")
```

Reading Lines of a Text File

The `readLines` function can be used to simply read lines of a text file and store them in a character vector.

```
> con <- gzfile("words.gz")
> x <- readLines(con, 10)
> x
[1] "1080"      "10-point" "10th"      "11-point"
[5] "12-point"  "16-point" "18-point"  "1st"
[9] "2"        "20-point"
```

`writeLines` takes a character vector and writes each element one line at a time to a text file.

Reading Lines of a Text File

`readLines` can be useful for reading in lines of webpages

```
## This might take time
```

```
con <- url("http://www.jhsph.edu", "r")
```

```
x <- readLines(con)
```

```
> head(x)
```

```
[1] "<!DOCTYPE HTML PUBLIC \"-//W3C//DTD HTML 4.0 Transitional//EN\">"
```

```
[2] ""
```

```
[3] "<html>"
```

```
[4] "<head>"
```

```
[5] "\t<meta http-equiv=\"Content-Type\" content=\"text/html; charset=utf-8\">"
```

`apply` is used to evaluate a function (often an anonymous one) over the margins of an array.

- It is most often used to apply a function to the rows or columns of a matrix
- It can be used with general arrays, e.g. taking the average of an array of matrices
- It is not really faster than writing a loop, but it works in one line!

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

- X is an array
- MARGIN is an integer vector indicating which margins should be “retained”.
- FUN is a function to be applied
- ... is for other arguments to be passed to FUN

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 2, mean)
[1] 0.04868268 0.35743615 -0.09104379
[4] -0.05381370 -0.16552070 -0.18192493
[7] 0.10285727 0.36519270 0.14898850
[10] 0.26767260

> apply(x, 1, sum)
[1] -1.94843314 2.60601195 1.51772391
[4] -2.80386816 3.73728682 -1.69371360
[7] 0.02359932 3.91874808 -2.39902859
[10] 0.48685925 -1.77576824 -3.34016277
[13] 4.04101009 0.46515429 1.83687755
[16] 4.36744690 2.21993789 2.60983764
[19] -1.48607630 3.58709251
```

For sums and means of matrix dimensions, we have some shortcuts.

- `rowSums = apply(x, 1, sum)`
- `rowMeans = apply(x, 1, mean)`
- `colSums = apply(x, 2, sum)`
- `colMeans = apply(x, 2, mean)`

The shortcut functions are *much* faster, but you won't notice unless you're using a large matrix.

Other Ways to Apply

Quantiles of the rows of a matrix.

```
> x <- matrix(rnorm(200), 20, 10)
> apply(x, 1, quantile, probs = c(0.25, 0.75))
```

	[,1]	[,2]	[,3]	[,4]
25%	-0.3304284	-0.99812467	-0.9186279	-0.49711686
75%	0.9258157	0.07065724	0.3050407	-0.06585436
	[,5]	[,6]	[,7]	[,8]
25%	-0.05999553	-0.6588380	-0.653250	0.01749997
75%	0.52928743	0.3727449	1.255089	0.72318419
	[,9]	[,10]	[,11]	[,12]
25%	-1.2467955	-0.8378429	-1.0488430	-0.7054902
75%	0.3352377	0.7297176	0.3113434	0.4581150
	[,13]	[,14]	[,15]	[,16]
25%	-0.1895108	-0.5729407	-0.5968578	-0.9517069
75%	0.5326299	0.5064267	0.4933852	0.8868922
	[,17]	[,18]	[,19]	[,20]

Average matrix in an array

```
> a <- array(rnorm(2 * 2 * 10), c(2, 2, 10))
```

```
> apply(a, c(1, 2), mean)
```

```
      [,1]      [,2]  
[1,] -0.2353245 -0.03980211  
[2,] -0.3339748  0.04364908
```

```
> rowMeans(a, dims = 2)
```

```
      [,1]      [,2]  
[1,] -0.2353245 -0.03980211  
[2,] -0.3339748  0.04364908
```

Debugging

Computing for Data Analysis

Something's Wrong!

Indications that something's not right

- `message`: A generic notification/diagnostic message produced by the `message` function; execution of the function continues
- `warning`: An indication that something is wrong but not necessarily fatal; execution of the function continues; generated by the `warning` function
- `error`: An indication that a fatal problem has occurred; execution stops; produced by the `stop` function
- `condition`: A generic concept for indicating that something unexpected can occur; programmers can create their own conditions

Something's Wrong!

Warning

```
> log(-1)
```

```
[1] NaN
```

Warning message:

```
In log(-1) : NaNs produced
```

Something's Wrong

```
printmessage <- function(x) {  
  if(x > 0)  
    print("x is greater than zero")  
  else  
    print("x is less than or equal to zero")  
  invisible(x)  
}
```

Something's Wrong

```
printmessage <- function(x) {  
  if(x > 0)  
    print("x is greater than zero")  
  else  
    print("x is less than or equal to zero")  
  invisible(x)  
}  
> printmessage(1)  
[1] "x is greater than zero"  
> printmessage(NA)  
Error in if (x > 0) { : missing value where TRUE/FALSE needed
```

Something's Wrong!

```
printmessage2 <- function(x) {  
  if(is.na(x))  
    print("x is a missing value!")  
  else if(x > 0)  
    print("x is greater than zero")  
  else  
    print("x is less than or equal to zero")  
  invisible(x)  
}
```

Something's Wrong!

```
printmessage2 <- function(x) {  
  if(is.na(x))  
    print("x is a missing value!")  
  else if(x > 0)  
    print("x is greater than zero")  
  else  
    print("x is less than or equal to zero")  
  invisible(x)  
}  
  
> x <- log(-1)  
Warning message:  
In log(-1) : NaNs produced  
> printmessage2(x)  
[1] "x is a missing value!"
```


Something's Wrong!

How do you know that something is wrong with your function?

- What was your input? How did you call the function?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does what you get differ from what you were expecting?
- Were your expectations correct in the first place?
- Can you reproduce the problem (exactly)?

Debugging Tools in R

The primary tools for debugging functions in R are

- `traceback`: prints out the function call stack after an error occurs; does nothing if there's no error
- `debug`: flags a function for “debug” mode which allows you to step through execution of a function one line at a time
- `browser`: suspends the execution of a function wherever it is called and puts the function in debug mode
- `trace`: allows you to insert debugging code into a function at specific places
- `recover`: allows you to modify the error behavior so that you can browse the function call stack

These are interactive tools specifically designed to allow you to pick through a function. There's also the more blunt technique of inserting `print`/`cat` statements in the function.

```
> mean(x)
Error in mean(x) : object 'x' not found
> traceback()
1: mean(x)
>
```

```
> lm(y ~ x)
Error in eval(expr, envir, enclos) : object 'y' not found
> traceback()
7: eval(expr, envir, enclos)
6: eval(predvars, data, env)
5: model.frame.default(formula = y ~ x, drop.unused.levels = TRUE)
4: model.frame(formula = y ~ x, drop.unused.levels = TRUE)
3: eval(expr, envir, enclos)
2: eval(mf, parent.frame())
1: lm(y ~ x)
```

```
> debug(lm)
> lm(y ~ x)
debugging in: lm(y ~ x)
debug: {
  ret.x <- x
  ret.y <- y
  cl <- match.call()
  ...
  if (!qr)
    z$qr <- NULL
  z
}
Browse[2]>
```

```
Browse[2]> n
debug: ret.x <- x
Browse[2]> n
debug: ret.y <- y
Browse[2]> n
debug: cl <- match.call()
Browse[2]> n
debug: mf <- match.call(expand.dots = FALSE)
Browse[2]> n
debug: m <- match(c("formula", "data", "subset", "weights", "na.action",
  "offset"), names(mf), 0L)
```

```
> options(error = recover)
> read.csv("nosuchfile")
Error in file(file, "rt") : cannot open the connection
In addition: Warning message:
In file(file, "rt") :
  cannot open file 'nosuchfile': No such file or directory
```

Enter a frame number, or 0 to exit

```
1: read.csv("nosuchfile")
2: read.table(file = file, header = header, sep = sep, quote = quote, dec :
3: file(file, "rt")
```

Selection:

Summary

- There are three main indications of a problem/condition: message, warning, error; only an error is fatal
- When analyzing a function with a problem, make sure you can reproduce the problem, clearly state your expectations and how the output differs from your expectation
- Interactive debugging tools traceback, debug, browser, trace, and recover can be used to find problematic code in functions
- Debugging tools are not a substitute for thinking!

Introduction to the R Language

Functions

Computing for Data Analysis

Functions are created using the `function()` directive and are stored as R objects just like anything else. In particular, they are R objects of class “function”.

```
f <- function(<arguments>) {  
    ## Do something interesting  
}
```

Functions in R are “first class objects”, which means that they can be treated much like any other R object. Importantly,

- Functions can be passed as arguments to other functions
- Functions can be nested, so that you can define a function inside of another function

The return value of a function is the last expression in the function body to be evaluated.

Function Arguments

Functions have *named arguments* which potentially have *default values*.

- The *formal arguments* are the arguments included in the function definition
- The `formals` function returns a list of all the formal arguments of a function
- Not every function call in R makes use of all the formal arguments
- Function arguments can be *missing* or might have default values

Argument Matching

R functions arguments can be matched positionally or by name. So the following calls to `sd` are all equivalent

```
> mydata <- rnorm(100)
> sd(mydata)
> sd(x = mydata)
> sd(x = mydata, na.rm = FALSE)
> sd(na.rm = FALSE, x = mydata)
> sd(na.rm = FALSE, mydata)
```

Even though it's legal, I don't recommend messing around with the order of the arguments too much, since it can lead to some confusion.

Argument Matching

You can mix positional matching with matching by name. When an argument is matched by name, it is “taken out” of the argument list and the remaining unnamed arguments are matched in the order that they are listed in the function definition.

```
> args(lm)
function (formula, data, subset, weights, na.action,
          method = "qr", model = TRUE, x = FALSE,
          y = FALSE, qr = TRUE, singular.ok = TRUE,
          contrasts = NULL, offset, ...)
```

The following two calls are equivalent.

```
lm(data = mydata, y ~ x, model = FALSE, 1:100)
lm(y ~ x, mydata, 1:100, model = FALSE)
```

- Most of the time, named arguments are useful on the command line when you have a long argument list and you want to use the defaults for everything except for an argument near the end of the list
- Named arguments also help if you can remember the name of the argument and not its position on the argument list (plotting is a good example).

Function arguments can also be *partially* matched, which is useful for interactive work. The order of operations when given an argument is

- 1 Check for exact match for a named argument
- 2 Check for a partial match
- 3 Check for a positional match

Defining a Function

```
f <- function(a, b = 1, c = 2, d = NULL) {  
  
}
```

In addition to not specifying a default value, you can also set an argument value to NULL.

Lazy Evaluation

Arguments to functions are evaluated *lazily*, so they are evaluated only as needed.

```
f <- function(a, b) {  
  a^2  
}  
f(2)
```

This function never actually uses the argument `b`, so calling `f(2)` will not produce an error because the `2` gets positionally matched to `a`.

Lazy Evaluation

Another example

```
f <- function(a, b) {  
  print(a)  
  print(b)  
}
```

```
> f(45)
```

```
[1] 45
```

```
Error in print(b) : argument "b" is missing, with no default
```

```
>
```

Notice that “45” got printed first before the error was triggered. This is because `b` did not have to be evaluated until after `print(a)`. Once the function tried to evaluate `print(b)` it had to throw an error.

The “...” Argument

The ... argument indicate a variable number of arguments that are usually passed on to other functions.

- ... is often used when extending another function and you don't want to copy the entire argument list of the original function

```
myplot <- function(x, y, type = "l", ...) {  
  plot(x, y, type = type, ...)  
}
```

- Generic functions use ... so that extra arguments can be passed to methods (more on this later).

```
> mean  
function (x, ...)  
UseMethod("mean")
```

The “...” Argument

The ... argument is also necessary when the number of arguments passed to the function cannot be known in advance.

```
> args(paste)
function (... , sep = " ", collapse = NULL)

> args(cat)
function (... , file = "", sep = " ", fill = FALSE,
          labels = NULL, append = FALSE)
```

Arguments Coming After the “...” Argument

One catch with ... is that any arguments that appear *after* ... on the argument list must be named explicitly and cannot be partially matched.

```
> args(paste)
function (... , sep = " ", collapse = NULL)
```

```
> paste("a", "b", sep = ":")
[1] "a:b"
```

```
> paste("a", "b", se = ":")
[1] "a b :"
```

Introduction to the R Language

Loop Functions

Computing for Data Analysis

Looping on the Command Line

Writing `for`, `while` loops is useful when programming but not particularly easy when working interactively on the command line. There are some functions which implement looping to make life easier.

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

An auxiliary function `split` is also useful, particularly in conjunction with `lapply`.

`lapply` takes three arguments: a list `X`, a function (or the name of a function) `FUN`, and other arguments via its `...` argument. If `X` is not a list, it will be coerced to a list using `as.list`.

```
> lapply
function (X, FUN, ...)
{
  FUN <- match.fun(FUN)
  if (!is.vector(X) || is.object(X))
    X <- as.list(X)
  .Internal(lapply(X, FUN))
}
```

The actual looping is done internally in C code.

`lapply` always returns a list, regardless of the class of the input.

```
> x <- list(a = 1:5, b = rnorm(10))
```

```
> lapply(x, mean)
```

```
$a
```

```
[1] 3
```

```
$b
```

```
[1] 0.0296824
```

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))  
> lapply(x, mean)  
$a  
[1] 2.5  
  
$b  
[1] 0.06082667  
  
$c  
[1] 1.467083  
  
$d  
[1] 5.074749
```

```
> x <- 1:4  
> lapply(x, runif)  
[[1]]  
[1] 0.2675082  
  
[[2]]  
[1] 0.2186453 0.5167968  
  
[[3]]  
[1] 0.2689506 0.1811683 0.5185761  
  
[[4]]  
[1] 0.5627829 0.1291569 0.2563676 0.7179353
```

```
> x <- 1:4  
> lapply(x, runif, min = 0, max = 10)  
[[1]]  
[1] 3.302142  
  
[[2]]  
[1] 6.848960 7.195282  
  
[[3]]  
[1] 3.5031416 0.8465707 9.7421014  
  
[[4]]  
[1] 1.195114 3.594027 2.930794 2.766946
```

`lapply` and friends make heavy use of *anonymous functions*.

```
> x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
```

```
> x
```

```
$a
```

	[,1]	[,2]
[1,]	1	3
[2,]	2	4

```
$b
```

	[,1]	[,2]
[1,]	1	4
[2,]	2	5
[3,]	3	6

An anonymous function for extracting the first column of each matrix.

```
> lapply(x, function(elt) elt[,1])
```

```
$a
```

```
[1] 1 2
```

```
$b
```

```
[1] 1 2 3
```

sapply will try to simplify the result of lapply if possible.

- If the result is a list where every element is length 1, then a vector is returned
- If the result is a list where every element is a vector of the same length (> 1), a matrix is returned.
- If it can't figure things out, a list is returned

```
> x <- list(a = 1:4, b = rnorm(10), c = rnorm(20, 1), d = rnorm(100, 5))  
> lapply(x, mean)  
$a  
[1] 2.5  
  
$b  
[1] 0.06082667  
  
$c  
[1] 1.467083  
  
$d  
[1] 5.074749
```



```
> sapply(x, mean)
```

a	b	c	d
2.50000000	0.06082667	1.46708277	5.07474950

```
> mean(x)
```

```
[1] NA
```

Warning message:

In mean.default(x) : argument is not numeric or logical: returning NA

Introduction to the R Language

Control Structures

Computing for Data Analysis

Control structures in R allow you to control the flow of execution of the program, depending on runtime conditions. Common structures are

- `if`, `else`: testing a condition
- `for`: execute a loop a fixed number of times
- `while`: execute a loop *while* a condition is true
- `repeat`: execute an infinite loop
- `break`: break the execution of a loop
- `next`: skip an iteration of a loop
- `return`: exit a function

Most control structures are not used in interactive sessions, but rather when writing functions or longer expressions.

Control Structures: if

```
if(<condition>) {  
    ## do something  
} else {  
    ## do something else  
}
```

```
if(<condition1>) {  
    ## do something  
} else if(<condition2>) {  
    ## do something different  
} else {  
    ## do something different  
}
```

This is a valid if/else structure.

```
if(x > 3) {  
    y <- 10  
} else {  
    y <- 0  
}
```

So is this one.

```
y <- if(x > 3) {  
    10  
} else {  
    0  
}
```

Of course, the else clause is not necessary.

```
if(<condition1>) {  
  
}
```

```
if(<condition2>) {  
  
}
```

for loops take an iterator variable and assign it successive values from a sequence or vector. For loops are most commonly used for iterating over the elements of an object (list, vector, etc.)

```
for(i in 1:10) {  
    print(i)  
}
```

This loop takes the *i* variable and in each iteration of the loop gives it values 1, 2, 3, ..., 10, and then exits.

These three loops have the same behavior.

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

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

```
for(i in seq_along(x)) {  
  print(x[i])  
}
```

```
for(letter in x) {  
  print(letter)  
}
```

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


Nested for loops

for loops can be nested.

```
x <- matrix(1:6, 2, 3)
```

```
for(i in seq_len(nrow(x))) {  
  for(j in seq_len(ncol(x))) {  
    print(x[i, j])  
  }  
}
```

Be careful with nesting though. Nesting beyond 2–3 levels is often very difficult to read/understand.

while

While loops begin by testing a condition. If it is true, then they execute the loop body. Once the loop body is executed, the condition is tested again, and so forth.

```
count <- 0

while(count < 10) {
  print(count)
  count <- count + 1
}
```

While loops can potentially result in infinite loops if not written properly. Use with care!

while

Sometimes there will be more than one condition in the test.

```
z <- 5
```

```
while(z >= 3 && z <= 10) {  
  print(z)  
  coin <- rbinom(1, 1, 0.5)  
  
  if(coin == 1) { ## random walk  
    z <- z + 1  
  } else {  
    z <- z - 1  
  }  
}
```

Conditions are always evaluated from left to right.

repeat

Repeat initiates an infinite loop; these are not commonly used in statistical applications but they do have their uses. The only way to exit a repeat loop is to call `break`.

```
x0 <- 1
tol <- 1e-8

repeat {
  x1 <- computeEstimate()

  if(abs(x1 - x0) < tol) {
    break
  } else {
    x0 <- x1
  }
}
```

The loop in the previous slide is a bit dangerous because there's no guarantee it will stop. Better to set a hard limit on the number of iterations (e.g. using a for loop) and then report whether convergence was achieved or not.

next is used to skip an iteration of a loop

```
for(i in 1:100) {  
  if(i <= 20) {  
    ## Skip the first 20 iterations  
    next  
  }  
  ## Do something here  
}
```

return signals that a function should exit and return a given value

Summary

- Control structures like `if`, `while`, and `for` allow you to control the flow of an R program
- Infinite loops should generally be avoided, even if they are theoretically correct.
- Control structures mentioned here are primarily useful for writing programs; for command-line interactive work, the `*apply` functions are more useful.

mapply is a multivariate apply of sorts which applies a function in parallel over a set of arguments.

```
> str(mapply)
function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE,
         USE.NAMES = TRUE)
```

- FUN is a function to apply
- ... contains arguments to apply over
- MoreArgs is a list of other arguments to FUN.
- SIMPLIFY indicates whether the result should be simplified

The following is tedious to type

```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

Instead we can do

```
> mapply(rep, 1:4, 4:1)
```

```
[[1]]
```

```
[1] 1 1 1 1
```

```
[[2]]
```

```
[1] 2 2 2
```

```
[[3]]
```

```
[1] 3 3
```

```
[[4]]
```

```
[1] 4
```

Vectorizing a Function

```
> noise <- function(n, mean, sd) {  
+   rnorm(n, mean, sd)  
+ }  
> noise(5, 1, 2)  
[1]  2.4831198  2.4790100  0.4855190 -1.2117759  
[5] -0.2743532  
  
> noise(1:5, 1:5, 2)  
[1] -4.2128648 -0.3989266  4.2507057  1.1572738  
[5]  3.7413584
```

Instant Vectorization

```
> mapply(noise, 1:5, 1:5, 2)
[[1]]
[1] 1.037658

[[2]]
[1] 0.7113482 2.7555797

[[3]]
[1] 2.769527 1.643568 4.597882

[[4]]
[1] 4.476741 5.658653 3.962813 1.204284

[[5]]
[1] 4.797123 6.314616 4.969892 6.530432 6.723254
```

Which is the same as

```
list(noise(1, 1, 2), noise(2, 2, 2),  
      noise(3, 3, 2), noise(4, 4, 2),  
      noise(5, 5, 2))
```

Why is any of this information useful?

- Optimization routines in R like `optim`, `nlm`, and `optimize` require you to pass a function whose argument is a vector of parameters (e.g. a log-likelihood)
- However, an object function might depend on a host of other things besides its parameters (like *data*)
- When writing software which does optimization, it may be desirable to allow the user to hold certain parameters fixed

Maximizing a Normal Likelihood

Write a “constructor” function

```
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {  
  params <- fixed  
  function(p) {  
    params[!fixed] <- p  
    mu <- params[1]  
    sigma <- params[2]  
    a <- -0.5*length(data)*log(2*pi*sigma^2)  
    b <- -0.5*sum((data-mu)^2) / (sigma^2)  
    -(a + b)  
  }  
}
```

Note: Optimization functions in R *minimize* functions, so you need to use the negative log-likelihood.

Maximizing a Normal Likelihood

```
> set.seed(1); normals <- rnorm(100, 1, 2)
> nLL <- make.NegLogLik(normals)
> nLL
function(p) {
    params[!fixed] <- p
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma^2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a + b)
}
<environment: 0x165b1a4>
> ls(environment(nLL))
[1] "data"    "fixed"   "params"
```

Estimating Parameters

```
> optim(c(mu = 0, sigma = 1), nLL)$par  
      mu      sigma  
1.218239 1.787343
```

Fixing $\sigma = 2$

```
> nLL <- make.NegLogLik(normals, c(FALSE, 2))  
> optimize(nLL, c(-1, 3))$minimum  
[1] 1.217775
```

Fixing $\mu = 1$

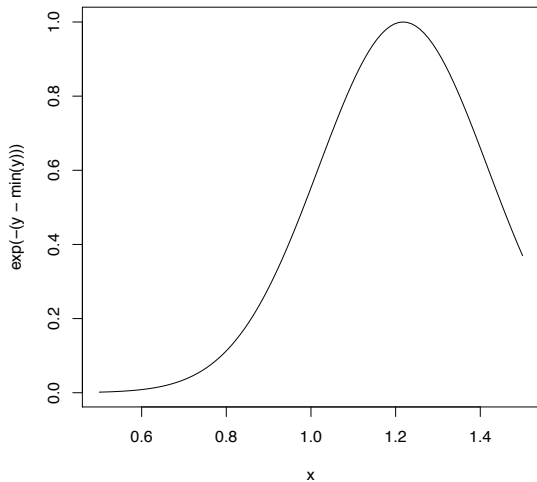
```
> nLL <- make.NegLogLik(normals, c(1, FALSE))  
> optimize(nLL, c(1e-6, 10))$minimum  
[1] 1.800596
```


Plotting the Likelihood

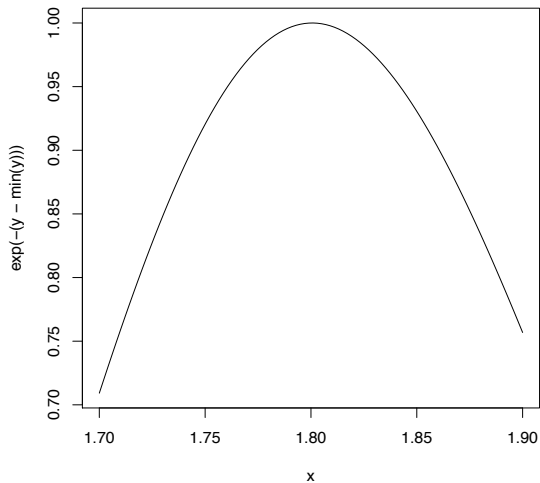
```
nLL <- make.NegLogLik(normals, c(1, FALSE))  
x <- seq(1.7, 1.9, len = 100)  
y <- sapply(x, nLL)  
plot(x, exp(-(y - min(y)))), type = "l")
```

```
nLL <- make.NegLogLik(normals, c(FALSE, 2))  
x <- seq(0.5, 1.5, len = 100)  
y <- sapply(x, nLL)  
plot(x, exp(-(y - min(y)))), type = "l")
```

Plotting the Likelihood



Plotting the Likelihood



- Objective functions can be “built” which contain all of the necessary data for evaluating the function
- No need to carry around long argument lists — useful for interactive and exploratory work.
- Code can be simplified and cleaned up
- Reference: Robert Gentleman and Ross Ihaka (2000). “Lexical Scope and Statistical Computing,” *JCGS*, 9, 491–508.

A Diversion on Binding Values to Symbol

How does R know which value to assign to which symbol? When I type

```
> lm <- function(x) { x * x }  
> lm  
function(x) { x * x }
```

how does R know what value to assign to the symbol `lm`? Why doesn't it give it the value of `lm` that is in the **stats** package?

A Diversion on Binding Values to Symbol

When R tries to bind a value to a symbol, it searches through a series of environments to find the appropriate value. When you are working on the command line and need to retrieve the value of an R object, the order is roughly

- 1 Search the global environment for a symbol name matching the one requested.
- 2 Search the namespaces of each of the packages on the search list

The search list can be found by using the `search` function.

```
> search()
[1] ".GlobalEnv"          "package:stats"      "package:graphics"
[4] "package:grDevices"  "package:utils"      "package:datasets"
[7] "package:methods"    "Autoloads"          "package:base"
```

Binding Values to Symbol

- The *global environment* or the user's workspace is always the first element of the search list and the **base** package is always the last.
- The order of the packages on the search list matters!
- User's can configure which packages get loaded on startup so you cannot assume that there will be a set list of packages available.
- When a user loads a package with `library` the namespace of that package gets put in position 2 of the search list (by default) and everything else gets shifted down the list.
- Note that R has separate namespaces for functions and non-functions so it's possible to have an object named `c` and a function named `c`.

The scoping rules for R are the main feature that make it different from the original S language.

- The scoping rules determine how a value is associated with a free variable in a function
- R uses *lexical scoping* or *static scoping*. A common alternative is *dynamic scoping*.
- Related to the scoping rules is how R uses the *search list* to bind a value to a symbol
- Lexical scoping turns out to be particularly useful for simplifying statistical computations

Consider the following function.

```
f <- function(x, y) {  
  x^2 + y / z  
}
```

This function has 2 formal arguments x and y . In the body of the function there is another symbol z . In this case z is called a *free variable*.

The scoping rules of a language determine how values are assigned to free variables. Free variables are not formal arguments and are not local variables (assigned inside the function body).

Lexical scoping in R means that

the values of free variables are searched for in the environment in which the function was defined.

What is an environment?

- An *environment* is a collection of (symbol, value) pairs, i.e. `x` is a symbol and `3.14` might be its value.
- Every environment has a parent environment; it is possible for an environment to have multiple “children”
- the only environment without a parent is the empty environment
- A function + an environment = a *closure* or *function closure*.

Searching for the value for a free variable:

- If the value of a symbol is not found in the environment in which a function was defined, then the search is continued in the *parent environment*.
- The search continues down the sequence of parent environments until we hit the *top-level environment*; this usually the global environment (workspace) or the namespace of a package.
- After the top-level environment, the search continues down the search list until we hit the *empty environment*.
- If a value for a given symbol cannot be found once the empty environment is arrived at, then an error is thrown.

Why does all this matter?

- Typically, a function is defined in the global environment, so that the values of free variables are just found in the user's workspace
- This behavior is logical for most people and is usually the “right thing” to do
- However, in R you can have functions defined *inside other functions*
 - Languages like C don't let you do this
- Now things get interesting — In this case the environment in which a function is defined is the body of another function!

Lexical Scoping

```
make.power <- function(n) {  
  pow <- function(x) {  
    x^n  
  }  
  pow  
}
```

This function returns another function as its value.

```
> cube <- make.power(3)  
> square <- make.power(2)  
> cube(3)  
[1] 27  
> square(3)  
[1] 9
```

Exploring a Function Closure

What's in a function's environment?

```
> ls(environment(cube))
```

```
[1] "n"    "pow"
```

```
> get("n", environment(cube))
```

```
[1] 3
```

```
> ls(environment(square))
```

```
[1] "n"    "pow"
```

```
> get("n", environment(square))
```

```
[1] 2
```

Lexical vs. Dynamic Scoping

```
y <- 10
```

```
f <- function(x) {  
  y <- 2  
  y^2 + g(x)  
}
```

```
g <- function(x) {  
  x * y  
}
```

What is the value of

`f(3)`

Lexical vs. Dynamic Scoping

- With lexical scoping the value of y in the function g is looked up in the environment in which the function was defined, in this case the global environment, so the value of y is 10.
- With dynamic scoping, the value of y is looked up in the environment from which the function was *called* (sometimes referred to as the *calling environment*).
 - In R the calling environment is known as the *parent frame*So the value of y would be 2.

Lexical vs. Dynamic Scoping

When a function is *defined* in the global environment and is subsequently *called* from the global environment, then the defining environment and the calling environment are the same. This can sometimes give the appearance of dynamic scoping.

```
> g <- function(x) {  
+       a <- 3  
+       x + a + y  
+ }  
> g(2)  
Error in g(2) : object "y" not found  
> y <- 3  
> g(2)  
[1] 8
```

Other languages that support lexical scoping

- Scheme
- Perl
- Python
- Common Lisp (all languages converge to Lisp)

Consequences of Lexical Scoping

- In R, all objects must be stored in memory
- All functions must carry a pointer to their respective defining environments, which could be anywhere
- In S-PLUS, free variables are always looked up in the global workspace, so everything can be stored on the disk because the “defining environment” of all functions is the same.

Why is any of this information useful?

- Optimization routines in R like `optim`, `nlm`, and `optimize` require you to pass a function whose argument is a vector of parameters (e.g. a log-likelihood)
- However, an object function might depend on a host of other things besides its parameters (like *data*)
- When writing software which does optimization, it may be desirable to allow the user to hold certain parameters fixed

Maximizing a Normal Likelihood

Write a “constructor” function

```
make.NegLogLik <- function(data, fixed=c(FALSE,FALSE)) {  
  params <- fixed  
  function(p) {  
    params[!fixed] <- p  
    mu <- params[1]  
    sigma <- params[2]  
    a <- -0.5*length(data)*log(2*pi*sigma^2)  
    b <- -0.5*sum((data-mu)^2) / (sigma^2)  
    -(a + b)  
  }  
}
```

Note: Optimization functions in R *minimize* functions, so you need to use the negative log-likelihood.

Maximizing a Normal Likelihood

```
> set.seed(1); normals <- rnorm(100, 1, 2)
> nLL <- make.NegLogLik(normals)
> nLL
function(p) {
    params[!fixed] <- p
    mu <- params[1]
    sigma <- params[2]
    a <- -0.5*length(data)*log(2*pi*sigma^2)
    b <- -0.5*sum((data-mu)^2) / (sigma^2)
    -(a + b)
}
<environment: 0x165b1a4>
> ls(environment(nLL))
[1] "data"    "fixed"   "params"
```

Estimating Parameters

```
> optim(c(mu = 0, sigma = 1), nLL)$par  
      mu      sigma  
1.218239 1.787343
```

Fixing $\sigma = 2$

```
> nLL <- make.NegLogLik(normals, c(FALSE, 2))  
> optimize(nLL, c(-1, 3))$minimum  
[1] 1.217775
```

Fixing $\mu = 1$

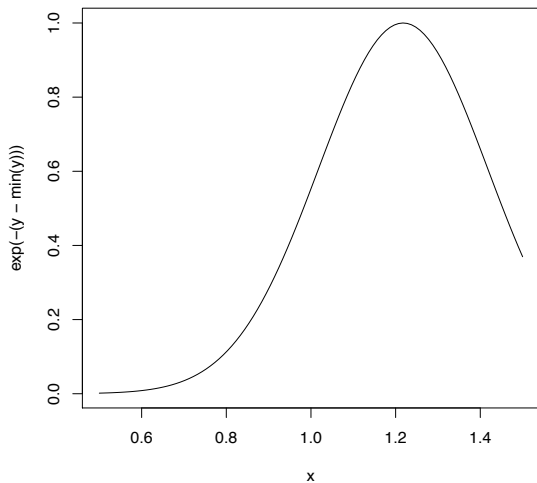
```
> nLL <- make.NegLogLik(normals, c(1, FALSE))  
> optimize(nLL, c(1e-6, 10))$minimum  
[1] 1.800596
```

Plotting the Likelihood

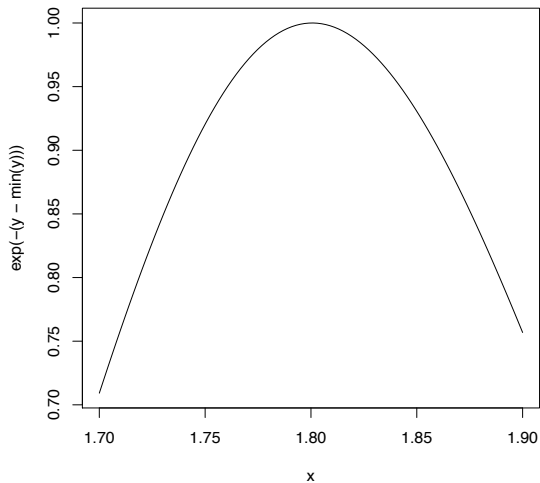
```
nLL <- make.NegLogLik(normals, c(1, FALSE))  
x <- seq(1.7, 1.9, len = 100)  
y <- sapply(x, nLL)  
plot(x, exp(-(y - min(y)))), type = "l")
```

```
nLL <- make.NegLogLik(normals, c(FALSE, 2))  
x <- seq(0.5, 1.5, len = 100)  
y <- sapply(x, nLL)  
plot(x, exp(-(y - min(y)))), type = "l")
```


Plotting the Likelihood



Plotting the Likelihood



- Objective functions can be “built” which contain all of the necessary data for evaluating the function
- No need to carry around long argument lists — useful for interactive and exploratory work.
- Code can be simplified and cleaned up
- Reference: Robert Gentleman and Ross Ihaka (2000). “Lexical Scope and Statistical Computing,” *JCGS*, 9, 491–508.

tapply is used to apply a function over subsets of a vector. I don't know why it's called tapply.

```
> str(tapply)
function (X, INDEX, FUN = NULL, ..., simplify = TRUE)
```

- X is a vector
- INDEX is a factor or a list of factors (or else they are coerced to factors)
- FUN is a function to be applied
- ... contains other arguments to be passed FUN
- simplify, should we simplify the result?

Take group means.

```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> f
 [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3
[24] 3 3 3 3 3 3 3
Levels: 1 2 3
> tapply(x, f, mean)
      1      2      3 
0.1144464 0.5163468 1.2463678
```

Take group means without simplification.

```
> tapply(x, f, mean, simplify = FALSE)
```

```
$'1'
```

```
[1] 0.1144464
```

```
$'2'
```

```
[1] 0.5163468
```

```
$'3'
```

```
[1] 1.246368
```

Find group ranges.

```
> tapply(x, f, range)
```

```
$'1'
```

```
[1] -1.097309  2.694970
```

```
$'2'
```

```
[1] 0.09479023 0.79107293
```

```
$'3'
```

```
[1] 0.4717443 2.5887025
```

`split` takes a vector or other objects and splits it into groups determined by a factor or list of factors.

```
> str(split)
function (x, f, drop = FALSE, ...)
```

- `x` is a vector (or list) or data frame
- `f` is a factor (or coerced to one) or a list of factors
- `drop` indicates whether empty factors levels should be dropped


```
> x <- c(rnorm(10), runif(10), rnorm(10, 1))
> f <- gl(3, 10)
> split(x, f)
$'1'
 [1] -0.8493038 -0.5699717 -0.8385255 -0.8842019
 [5]  0.2849881  0.9383361 -1.0973089  2.6949703
 [9]  1.5976789 -0.1321970

$'2'
 [1] 0.09479023 0.79107293 0.45857419 0.74849293
 [5] 0.34936491 0.35842084 0.78541705 0.57732081
 [9] 0.46817559 0.53183823

$'3'
 [1] 0.6795651 0.9293171 1.0318103 0.4717443
 [5] 2.5887025 1.5975774 1.3246333 1.4372701
```

A common idiom is `split` followed by an `lapply`.

```
> lapply(split(x, f), mean)
```

```
$'1'
```

```
[1] 0.1144464
```

```
$'2'
```

```
[1] 0.5163468
```

```
$'3'
```

```
[1] 1.246368
```

Splitting a Data Frame

```
> library(datasets)
> head(airquality)
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA       NA 14.3   56     5   5
6    28       NA 14.9   66     5   6
```

Splitting a Data Frame

```
> s <- split(airquality, airquality$Month)
> lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
$'5'
```

Ozone	Solar.R	Wind
NA	NA	11.62258

```
$'6'
```

Ozone	Solar.R	Wind
NA	190.16667	10.26667

```
$'7'
```

Ozone	Solar.R	Wind
NA	216.483871	8.941935

```
$'8'
```

Ozone	Solar.R	Wind
-------	---------	------

Splitting a Data Frame

```
> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

	5	6	7	8	9
Ozone	NA	NA	NA	NA	NA
Solar.R	NA	190.16667	216.483871	NA	167.4333
Wind	11.62258	10.26667	8.941935	8.793548	10.1800

```
> sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")],  
                                   na.rm = TRUE))
```

	5	6	7	8	9
Ozone	23.61538	29.44444	59.115385	59.961538	31.44828
Solar.R	181.29630	190.16667	216.483871	171.857143	167.43333
Wind	11.62258	10.26667	8.941935	8.793548	10.18000

Splitting on More than One Level

```
> x <- rnorm(10)
> f1 <- gl(2, 5)
> f2 <- gl(5, 2)
> f1
[1] 1 1 1 1 1 2 2 2 2 2
Levels: 1 2
> f2
[1] 1 1 2 2 3 3 4 4 5 5
Levels: 1 2 3 4 5
> interaction(f1, f2)
[1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
10 Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 ... 2.5
```

Splitting on More than One Level

Interactions can create empty levels.

```
> str(split(x, list(f1, f2)))  
List of 10  
 $ 1.1: num [1:2] -0.378  0.445  
 $ 2.1: num(0)  
 $ 1.2: num [1:2]  1.4066 0.0166  
 $ 2.2: num(0)  
 $ 1.3: num -0.355  
 $ 2.3: num 0.315  
 $ 1.4: num(0)  
 $ 2.4: num [1:2] -0.907  0.723  
 $ 1.5: num(0)  
 $ 2.5: num [1:2]  0.732  0.360
```

Empty levels can be dropped.

```
> str(split(x, list(f1, f2), drop = TRUE))
```

List of 6

```
$ 1.1: num [1:2] -0.378  0.445
```

```
$ 1.2: num [1:2]  1.4066 0.0166
```

```
$ 1.3: num -0.355
```

```
$ 2.3: num 0.315
```

```
$ 2.4: num [1:2] -0.907  0.723
```

```
$ 2.5: num [1:2]  0.732  0.360
```


Introduction to the R Language

Plotting

Computing for Data Analysis

The plotting and graphics engine in R is encapsulated in a few base and recommend packages:

- **graphics**: contains plotting functions for the “base” graphing systems, including `plot`, `hist`, `boxplot` and many others.
- **lattice**: contains code for producing Trellis graphics, which are independent of the “base” graphics system; includes functions like `xyplot`, `bwplot`, `levelplot`
- **grid**: implements a different graphing system independent of the “base” system; the **lattice** package builds on top of **grid**; we seldom call functions from the **grid** package directly
- **grDevices**: contains all the code implementing the various graphics devices, including X11, PDF, PostScript, PNG, etc.

The Process of Making a Plot

When making a plot one must first make a few choices (not necessarily in this order):

- To what device will the plot be sent? The default in Unix is `x11`; on Windows it is `windows`; on Mac OS X it is `quartz`
- Is the plot for viewing temporarily on the screen, or will it eventually end up in a paper? Are you using it in a presentation? Plots included in a paper/presentation need to use a file device rather than a screen device.
- Is there a large amount of data going into the plot? Or is it just a few points?
- Do you need to be able to resize the graphic?

The Process of Making a Plot

- What graphics system will you use: base or grid/lattice? These generally cannot be mixed.
- Base graphics are usually constructed piecemeal, with each aspect of the plot handled separately through a series of function calls; this is often conceptually simpler and allows plotting to mirror the thought process
- Lattice/grid graphics are usually created in a single function call, so all of the graphics parameters have to be specified at once; specifying everything at once allows R to automatically calculate the necessary spacings and font sizes.

Base graphics are used most commonly and are a very powerful system for creating 2-D graphics.

- Calling `plot(x, y)` or `hist(x)` will launch a graphics device (if one is not already open) and draw the plot on the device
- If the arguments to `plot` are not of some special class, then the *default method* for `plot` is called; this function has *many* arguments, letting you set the title, x axis label, y axis label, etc.
- The base graphics system has *many* parameters that can be set and tweaked; these parameters are documented in `?par`; it wouldn't hurt to memorize this help page!

Some Important Base Graphics Parameters

The `par` function is used to specify global graphics parameters that affect all plots in an R session. These parameters can often be overridden as arguments to specific plotting functions.

- `pch`: the plotting symbol (default is open circle)
- `lty`: the line type (default is solid line), can be dashed, dotted, etc.
- `lwd`: the line width, specified as an integer multiple
- `col`: the plotting color, specified as a number, string, or hex code; the `colors` function gives you a vector of colors by name
- `las`: the orientation of the axis labels on the plot

Some Important Base Graphics Parameters

- `bg`: the background color
- `mar`: the margin size
- `oma`: the outer margin size (default is 0 for all sides)
- `mfrow`: number of plots per row, column (plots are filled row-wise)
- `mfcol`: number of plots per row, column (plots are filled column-wise)

Some Important Base Graphics Parameters

Some default values.

```
> par("lty")  
[1] "solid"  
> par("lwd")  
[1] 1  
> par("col")  
[1] "black"  
> par("pch")  
[1] 1
```


Some Important Base Graphics Parameters

Some default values.

```
> par("bg")  
[1] "transparent"  
> par("mar")  
[1] 5.1 4.1 4.1 2.1  
> par("oma")  
[1] 0 0 0 0  
> par("mfrow")  
[1] 1 1  
> par("mfcol")  
[1] 1 1
```

Some Important Base Plotting Functions

- `plot`: make a scatterplot, or other type of plot depending on the class of the object being plotted
- `lines`: add lines to a plot, given a vector `x` values and a corresponding vector of `y` values (or a 2-column matrix); this function just connects the dots
- `points`: add points to a plot
- `text`: add text labels to a plot using specified `x`, `y` coordinates
- `title`: add annotations to `x`, `y` axis labels, title, subtitle, outer margin
- `mtext`: add arbitrary text to the margins (inner or outer) of the plot
- `axis`: adding axis ticks/labels

The list of devices is found in `?Devices`; there are also devices created by users on CRAN

- `pdf`: useful for line-type graphics, vector format, resizes well, usually portable
- `postscript`: older format, also vector format and resizes well, usually portable, can be used to create encapsulated postscript files, Windows systems often don't have a postscript viewer
- `xfig`: good if you use Unix and want to edit a plot by hand

- `png`: bitmapped format, good for line drawings or images with solid colors, uses lossless compression (like the old GIF format), most web browsers can read this format natively, good for plotting many many many points, does not resize well
- `jpeg`: good for photographs or natural scenes, uses lossy compression, good for plotting many many many points, does not resize well, can be read by almost any computer and any web browser, not great for line drawings
- `bitmap`: needed to create bitmap files (`png`, `jpeg`) in certain situations (uses Ghostscript), also can be used to create a variety of other bitmapped formats not mentioned
- `bmp`: a native Windows bitmapped format

Copying Plots

There are two basic approaches to plotting.

- 1 Launch a graphics device
- 2 Make a plot; annotate if needed
- 3 Close graphics device

Or

- 1 Make a plot on a screen device (default); annotate if needed
- 2 Copy the plot to another device if necessary (not an exact process)

Copying a plot to another device can be useful because some plots require a lot of code and it can be a pain to type all that in again for a different device.

- `dev.copy`: copy a plot from one device to another
- `dev.copy2pdf`: copy a plot to a Portable Document Format (PDF) file
- `dev.list`: show the list of open graphics devices
- `dev.next`: switch control to the next graphics device on the device list
- `dev.set`: set control to a specific graphics device
- `dev.off`: close the current graphics device

NOTE: Copying a plot is not an exact operation!

- `xyplot`: this is the main function for creating scatterplots
- `bwplot`: box-and-whiskers plots (“boxplots”)
- `histogram`: histograms
- `stripplot`: like a boxplot but with actual points
- `dotplot`: plot dots on “violin strings”
- `splo`m: scatterplot matrix; like pairs in base graphics system
- `levelplot`, `contourplot`: for plotting “image” data

Lattice Functions

Lattice functions generally take a formula for their first argument, usually of the form

$y \sim x \mid f * g$

- On the left of the \sim is the y variable, on the right is the x variable
- After the \mid are *conditioning variables* — they are optional; the $*$ indicates an interaction
- The second argument is the data frame or list from which the variables in the formula should be obtained.
- If no data frame or list is passed, then the parent frame is used.
- If no other arguments are passed, there are defaults that can be used.

Lattice functions behave differently from base graphics functions in one critical way.

- Base graphics functions plot data directly the graphics device
- Lattice graphics functions return an object of class `trellis`.
- The print methods for lattice functions actually do the work of plotting the data on the graphics device.
- Lattice functions return “plot objects” that can, in principle, be stored (but it’s usually better to just save the code + data).
- On the command line, `trellis` objects are *auto-printed* so that it appears the function is plotting the data

Lattice Panel Functions

Lattice functions have a `panel` function which controls what happens inside each panel of the entire plot.

```
x <- rnorm(100)
y <- x + rnorm(100, sd = 0.5)
f <- gl(2, 50, labels = c("Group 1", "Group 2"))
xyplot(y ~ x | f)
```

plots y vs. x conditioned on f .

Lattice Panel Functions

```
xyplot(y ~ x | f,  
       panel = function(x, y, ...) {  
         panel.xyplot(x, y, ...)  
         panel.abline(h = median(y),  
                      lty = 2)  
       })
```

plots y vs. x conditioned on f with horizontal (dashed) line drawn at the median of y for each panel.

Adding a regression line

```
xyplot(y ~ x | f,  
       panel = function(x, y, ...) {  
         panel.xyplot(x, y, ...)  
         panel.lmline(x, y, col = 2)  
       })
```

fits and plots a simple linear regression line to each panel of the plot.

R can produce \LaTeX -like symbols on a plot for mathematical annotation. This is very handy and is useful for making fun of people who use other statistical packages.

- Math symbols are “expressions” in R and need to be wrapped in the `expression` function
- There is a set list of allowed symbols and this is documented in `?plotmath`
- Plotting functions that take arguments for text generally allow expressions for math symbols

Some examples.

```
plot(0, 0, main = expression(theta == 0),  
     ylab = expression(hat(gamma) == 0),  
     xlab = expression(sum(x[i] * y[i], i==1, n)))
```

Pasting strings together.

```
x <- rnorm(100)  
hist(x,  
     xlab=expression("The mean (" * bar(x) * ") is " *  
                      sum(x[i]/n,i==1,n)))
```

Substituting

What if you want to use a computed value in the annotation?

```
x <- rnorm(100)
y <- x + rnorm(100, sd = 0.5)
plot(x, y,
      xlab=substitute(bar(x) == k, list(k=mean(x))),
      ylab=substitute(bar(y) == k, list(k=mean(y)))
)
```

Or in a loop of plots

```
par(mfrow = c(2, 2))
for(i in 1:4) {
  x <- rnorm(100)
  hist(x, main=substitute(theta==num,list(num=i)))
}
```

Summary of Important Help Pages

- ?par
- ?plot
- ?xyplot
- ?plotmath
- ?axis

Simulation

Computing for Data Analysis

Generating Random Numbers

Functions for probability distributions in R

- `rnorm`: generate random Normal variates with a given mean and standard deviation
- `dnorm`: evaluate the Normal probability density (with a given mean/SD) at a point (or vector of points)
- `pnorm`: evaluate the cumulative distribution function for a Normal distribution
- `rpois`: generate random Poisson variates with a given rate

Generating Random Numbers

Probability distribution functions usually have four functions associated with them. The functions are prefixed with a

- d for density
- r for random number generation
- p for cumulative distribution
- q for quantile function

Generating Random Numbers

Working with the Normal distributions requires using these four functions

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
```

```
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
```

```
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
```

```
rnorm(n, mean = 0, sd = 1)
```

If Φ is the cumulative distribution function for a standard Normal distribution, then $\text{pnorm}(q) = \Phi(q)$ and $\text{qnorm}(p) = \Phi^{-1}(p)$.

Generating Random Numbers

Generating random Normal variates

```
> x <- rnorm(10)
> x
[1] 1.38380206 0.48772671 0.53403109 0.66721944
[5] 0.01585029 0.37945986 1.31096736 0.55330472
[9] 1.22090852 0.45236742
> x <- rnorm(10, 20, 2)
> x
[1] 23.38812 20.16846 21.87999 20.73813 19.59020
[6] 18.73439 18.31721 22.51748 20.36966 21.04371
> summary(x)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 18.32   19.73   20.55   20.67   21.67   23.39
```

Generating Random Numbers

Setting the random number seed with `set.seed` ensures reproducibility

```
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
> rnorm(5)
[1] -0.8204684  0.4874291  0.7383247  0.5757814
[5] -0.3053884
> set.seed(1)
> rnorm(5)
[1] -0.6264538  0.1836433 -0.8356286  1.5952808
[5]  0.3295078
```

Always set the random number seed when conducting a simulation!

Generating Random Numbers

Generating Poisson data

```
> rpois(10, 1)
[1] 3 1 0 1 0 0 1 0 1 1
> rpois(10, 2)
[1] 6 2 2 1 3 2 2 1 1 2
> rpois(10, 20)
[1] 20 11 21 20 20 21 17 15 24 20

> ppois(2, 2)  ## Cumulative distribution
[1] 0.6766764  ## Pr(x <= 2)
> ppois(4, 2)
[1] 0.947347  ## Pr(x <= 4)
> ppois(6, 2)
[1] 0.9954662  ## Pr(x <= 6)
```

Generating Random Numbers From a Linear Model

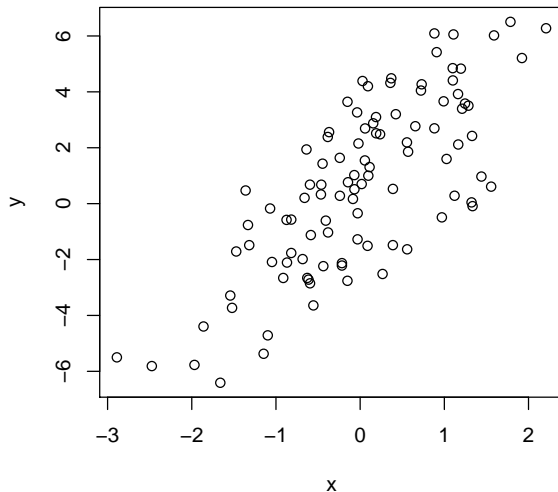
Suppose we want to simulate from the following linear model

$$y = \beta_0 + \beta_1 x + \varepsilon$$

where $\varepsilon \sim \mathcal{N}(0, 2^2)$. Assume $x \sim \mathcal{N}(0, 1^2)$, $\beta_0 = 0.5$ and $\beta_1 = 2$.

```
> set.seed(20)
> x <- rnorm(100)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-6.4080 -1.5400  0.6789  0.6893  2.9300  6.5050
> plot(x, y)
```


Generating Random Numbers From a Linear Model

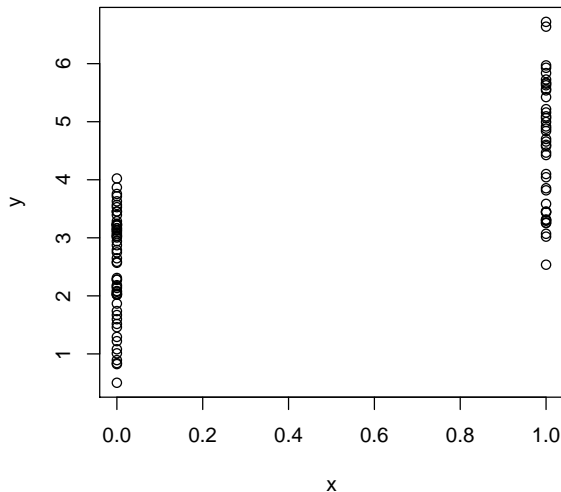


Generating Random Numbers From a Linear Model

What if x is binary?

```
> set.seed(10)
> x <- rbinom(100, 1, 0.5)
> e <- rnorm(100, 0, 2)
> y <- 0.5 + 2 * x + e
> summary(y)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-3.4940 -0.1409  1.5770  1.4320  2.8400  6.9410
> plot(x, y)
```

Generating Random Numbers From a Linear Model



Generating Random Numbers From a Generalized Linear Model

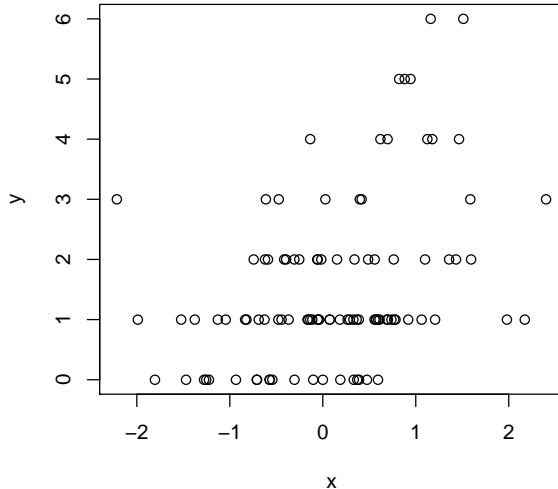
Suppose we want to simulate from a Poisson model where

$$Y \sim \text{Poisson}(\mu)$$
$$\log \mu = \beta_0 + \beta_1 x$$

and $\beta_0 = 0.5$ and $\beta_1 = 0.3$. We need to use the `rpois` function for this

```
> set.seed(1)
> x <- rnorm(100)
> log.mu <- 0.5 + 0.3 * x
> y <- rpois(100, exp(log.mu))
> summary(y)
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.00   1.00   1.00   1.55   2.00   6.00
> plot(x, y)
```

Generating Random Numbers From a Generalized Linear Model



Random Sampling

The `sample` function draws randomly from a specified set of (scalar) objects allowing you to sample from arbitrary distributions.

```
> set.seed(1)
> sample(1:10, 4)
[1] 3 4 5 7
> sample(1:10, 4)
[1] 3 9 8 5
> sample(letters, 5)
[1] "q" "b" "e" "x" "p"
> sample(1:10) ## permutation
[1] 4 7 10 6 9 2 8 3 1 5
> sample(1:10)
[1] 2 3 4 1 9 5 10 8 6 7
> sample(1:10, replace = TRUE) ## Sample w/replacement
[1] 2 9 7 8 2 8 5 9 7 8
```

Summary

- Drawing samples from specific probability distributions can be done with `r*` functions
- Standard distributions are built in: Normal, Poisson, Binomial, Exponential, Gamma, etc.
- The `sample` function can be used to draw random samples from arbitrary vectors
- Setting the random number generator seed via `set.seed` is critical for reproducibility

Programming Assignment 4

Computing for Data Analysis

Introduction

This Programming Assignment will focus on regular expressions and the various regular expression functions implemented in R. The ability to handle and construct regular expressions is important when dealing with unstructured text data, often obtained from the Web or other online source.

First download the `Baltimore_homicides.zip` file from Courseplus and unzip it into your working directory. You can read the file `homicides.txt` with `readLines` via

```
> homicides <- readLines("homicides.txt")
```

The data were downloaded from the Baltimore Sun web site (<http://goo.gl/hSofH>) and contain information about homicides occurring between 2007 and the middle of 2012. There is one homicide record per line of text. The data contain various HTML and JavaScript markup and so are not easily read into R without some manipulation/processing.

1 How many of each cause of homicide?

The goal of this problem is to count the number of different types of homicides that are in this dataset. In each record there is a field with the word “Cause” in it indicating the cause of death (e.g. “Cause: shooting”). The basic goal is to extract this field and count the number of instances of each cause.

Write a function named `count` that takes one argument, a character string indicating the cause of death. The function should then return an integer representing the number of homicides from that cause in the dataset. If no cause of death is specified, then the function should return an error message via the `stop` function.

- Your function should read the homicides dataset in the manner indicated above.
- The options for cause of death are “asphyxiation”, “blunt force”, “other”, “shooting”, “stabbing”, “unknown”. No other causes are allowed. If a cause of death is specified that is not one of these, then the function should throw an error with the `stop` function.
- Note that some homicides in the dataset do not have a cause of death listed and those records should be ignored.
- Your function should deal with some irregularities in the dataset like capitalization. For example “Shooting” and “shooting” should be counted as the same cause of death.

- Do not worry about spelling errors in the dataset (records with spelling errors can be ignored)

The function should use the following template.

```
count <- function(cause = NULL) {
  ## Check that "cause" is non-NULL; else throw error
  ## Check that specific "cause" is allowed; else throw error

  ## Read "homicides.txt" data file

  ## Extract causes of death

  ## Return integer containing count of homicides for that cause
}
```

The function should execute as follows:

```
> count("other")

[1] 6

> num <- count("unknown")
> print(num)

[1] 10
```

Save your code for this function to a file named `count.R`.

Use the submit script provided to submit your solution to this part. There are 3 tests that need to be passed for this part of the assignment.

2 Ages of homicide victims

The goal of this part is to write a function called `agecount` that returns the number of homicide victims of a given age. For most (but not all) records there is an indication of the age of the victim. Your function should take one argument, the age of the victim(s), extract the age of the victim from each record and then return a count of the number of victims of the specified age.

- The argument passed to `agecount` should be a positive integer, but you do not need to check for this.
- If a record does not contain any age information, the record should be ignored.
- The function should return an integer indicating the number of victims of a given age.
- Your function should read the homicides dataset in the manner indicated above.

The function should use the following template.

```

agecount <- function(age = NULL) {
  ## Check that "age" is non-NULL; else throw error

  ## Read "homicides.txt" data file

  ## Extract ages of victims; ignore records where no age is
  ## given

  ## Return integer containing count of homicides for that age
}

```

The function should execute as follows:

```

> agecount(3)

[1] 0

> num <- agecount(21)
> print(num)

[1] 60

```

Save your code for this function to a file named `agecount.R`.

Use the submit script provided to submit your solution to this part. There are 2 tests that need to be passed for this part of the assignment.

Classes and Methods in R

Computing for Data Analysis

- A system for doing object oriented programming
- R was originally quite interesting because it is both interactive *and* has a system for object orientation.
 - Other languages which support OOP (C++, Java, Lisp, Python, Perl) generally speaking are not interactive languages
- In R much of the code for supporting classes/methods is written by John Chambers himself (the creator of the original S language) and documented in the book *Programming with Data: A Guide to the S Language*
- A natural extension of Chambers' idea of allowing someone to cross the user → programmer spectrum
- Object oriented programming is a bit different in R than it is in most languages — even if you are familiar with the idea, you may want to pay attention to the details

Two styles of classes and methods

S3 classes/methods

- Included with version 3 of the S language.
- Informal, a little kludgy
- Sometimes called *old-style* classes/methods

S4 classes/methods

- more formal and rigorous
- Included with S-PLUS 6 and R 1.4.0 (December 2001)
- Also called *new-style* classes/methods

Two worlds living side by side

- For now (and the foreseeable future), S3 classes/methods and S4 classes/methods are separate systems (but they can be mixed to some degree).
- Each system can be used fairly independently of the other.
- Developers of new projects (you!) are encouraged to use the S4 style classes/methods.
 - Used extensively in the Bioconductor project
- But many developers still use S3 classes/methods because they are “quick and dirty” (and easier).
- In this lecture we will focus primarily on S4 classes/methods
- The code for implementing S4 classes/methods in R is in the **methods** package, which is usually loaded by default (but you can load it with `library(methods)` if for some reason it is not loaded)

Object Oriented Programming in R

- A *class* is a description of an thing. A class can be defined using `setClass()` in the **methods** package.
- An *object* is an instance of a class. Objects can be created using `new()`.
- A *method* is a function that only operates on a certain class of objects.
- A generic function is an R function which dispatches methods. A generic function typically encapsulates a “generic” concept (e.g. `plot`, `mean`, `predict`, ...)
 - The generic function does not actually do any computation.
- A *method* is the implementation of a generic function for an object of a particular class.

Things to look up

- The help files for the 'methods' package are extensive — do read them as they are the primary documentation
- You may want to start with ?Classes and ?Methods
- Check out ?setClass, ?setMethod, and ?setGeneric
- Some of it gets technical, but try your best for now—it will make sense in the future as you keep using it.
- Most of the documentation in the **methods** package is oriented towards developers/programmers as these are the primary people using classes/methods

All objects in R have a class which can be determined by the class function

```
> class(1)
```

```
[1] "numeric"
```

```
> class(TRUE)
```

```
[1] "logical"
```

```
> class(rnorm(100))
```

```
[1] "numeric"
```

```
> class(NA)
```

```
[1] "logical"
```

```
> class("foo")
```

```
[1] "character"
```

Data classes go beyond the atomic classes

```
> x <- rnorm(100)
> y <- x + rnorm(100)
> fit <- lm(y ~ x)  ## linear regression model
> class(fit)

[1] "lm"
```

- S4 and S3 style generic functions look different but conceptually, they are the same (they play the same role).
- When you program you can write new methods for an existing generic OR create your own generics and associated methods.
- Of course, if a data type does not exist in R that matches your needs, you can always define a new class along with generics/methods that go with it

An S3 generic function (in the 'base' package)

The mean function is generic

```
> mean
```

```
function (x, ...)
UseMethod("mean")
<bytecode: 0x7fc25c27afc0>
<environment: namespace:base>
```

So is the print function

```
> print
```

```
function (x, ...)
UseMethod("print")
<bytecode: 0x7fc25bd8ee00>
<environment: namespace:base>
```

```
> methods("mean")  
  
[1] mean.data.frame mean.Date  
[3] mean.default    mean.diffftime  
[5] mean.POSIXct    mean.POSIXlt
```

An S4 generic function (from the 'methods' package)

The S4 equivalent of `print` is `show`

```
> show
```

```
standardGeneric for "show" defined from package "methods"
```

```
function (object)
```

```
standardGeneric("show")
```

```
<bytecode: 0x7fc25b5ced08>
```

```
<environment: 0x7fc25c51aea0>
```

Methods may be defined for arguments: `object`

Use `showMethods("show")` for currently available ones.

(This generic function excludes non-simple inheritance; see `?setIs`)

The `show` function is usually not called directly (much like `print`) because objects are auto-printed

S4 methods

There are many different methods for the `show` generic function

```
> showMethods("show")
```

```
Function: show (package methods)
```

```
object="ANY"
```

```
object="classGeneratorFunction"
```

```
object="classRepresentation"
```

```
object="envRefClass"
```

```
object="function"
```

```
    (inherited from: object="ANY")
```

```
object="genericFunction"
```

```
object="genericFunctionWithTrace"
```

```
object="MethodDefinition"
```

```
object="MethodDefinitionWithTrace"
```

```
object="MethodSelectionReport"
```

```
object="MethodWithNext"
```

```
object="MethodWithNextWithTrace"
```

```
object="namedList"
```

The first argument of a generic function is an object of a particular class (there may be other arguments)

- 1 The generic function checks the class of the object.
- 2 A search is done to see if there is an appropriate method for that class.
- 3 If there exists a method for that class, then that method is called on the object and we're done.
- 4 If a method for that class does not exist, a search is done to see if there is a default method for the generic. If a default exists, then the default method is called.
- 5 If a default method doesn't exist, then an error is thrown.

Examining Code for Methods

Examining the code for an S3 or S4 method requires a call to a special function

- You cannot just print the code for a method like other functions because the code for the method is usually hidden.
- If you want to see the code for an S3 method, you can use the function `getS3method`.
- The call is `getS3method(<generic>, <class>)`
- For S4 methods you can use the function `getMethod`
- The call is `getMethod(<generic>, <signature>)` (more details later)

S3 Class/Method: Example 1

What's happening here?

```
> set.seed(2)
> x <- rnorm(100)
> mean(x)
```

```
[1] -0.03069816
```

- 1 The class of `x` is “numeric”
- 2 But there is no `mean` method for “numeric” objects!
- 3 So we call the default function for `mean`.

S3 Class/Method: Example 1

```
> head(getS3method("mean", "default"))

1 function (x, trim = 0, na.rm = FALSE, ...)
2 {
3     if (!is.numeric(x) && !is.complex(x) && !is.logical(x)) {
4         warning("argument is not numeric or logical: returning NA")
5         return(NA_real_)
6     }
7
8     if (na.rm) {
9         x = x[!is.na(x)]
10    }
11
12    if (trim > 0) {
13        lo <- floor(n * trim) + 1
14        hi <- n + 1 - lo
15        x <- sort.int(x, partial = unique(c(lo, hi)))[lo:hi]
16    }
17
18    .Internal(mean(x))
19 }
20
21
22
23
24
```

S3 Class/Method: Example 2

What happens here?

```
> set.seed(3)
> df <- data.frame(x = rnorm(100), y = 1:100)
> sapply(df, mean)
```

```
           x           y
0.01103557 50.50000000
```

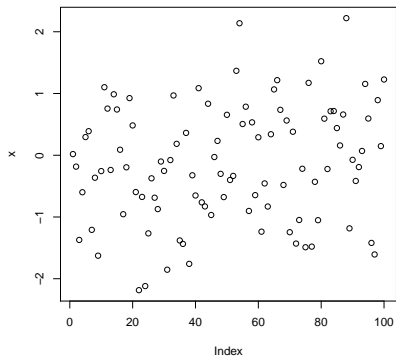
- 1 The class of `df` is “`data.frame`”; in a data frame each column can be an object of a different class
- 2 We `sapply` over the columns and call the `mean` function
- 3 In each column, `mean` checks the class of the object and dispatches the appropriate method.
- 4 Here we have a `numeric` column and an `integer` column; in both cases `mean` calls the default method

NOTE: Some methods are visible to the user (i.e. `mean.default`), but you should **never** call methods directly. Rather, use the generic function and let the method be dispatched automatically.

S3 Class/Method: Example 3

The `plot` function is generic and its behavior depends on the object being plotted.

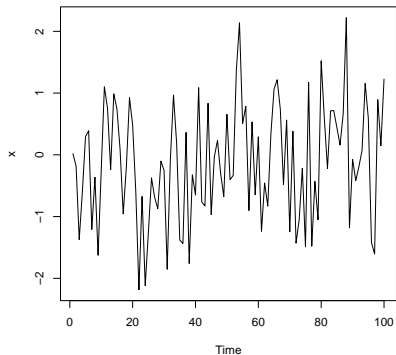
```
> set.seed(10)
> x <- rnorm(100)
> plot(x)
```



S3 Class/Method: Example 3

For time series objects, plot connects the dots

```
> set.seed(10)
> x <- rnorm(100)
> x <- as.ts(x)  ## Convert to a time series object
> plot(x)
```



Write your own methods!

If you write new methods for new classes, you'll probably end up writing methods for the following generics:

- `print/show`
- `summary`
- `plot`

There are two ways that you can extend the R system via classes/methods

- Write a method for a new class but for an existing generic function (i.e. like `print`)
- Write new generic functions and new methods for those generics

Why would you want to create a new class?

- To represent new types of data (e.g. gene expression, space-time, hierarchical, sparse matrices)
- New concepts/ideas that haven't been thought of yet (e.g. a fitted point process model, mixed-effects model, a sparse matrix)
- To abstract/hide implementation details from the user

I say things are “new” meaning that R does not know about them (not that they are new to the statistical community).

S4 Class/Method: Creating a New Class

A new class can be defined using the `setClass` function

- At a minimum you need to specify the name of the class
- You can also specify data elements that are called *slots*
- You can then define methods for the class with the `setMethod` function
- Information about a class definition can be obtained with the `showClass` function

S4 Class/Method: Polygon Class

Creating new classes/methods is usually not something done at the console; you likely want to save the code in a separate file

```
setClass("polygon",  
        representation(x = "numeric",  
                       y = "numeric"))
```

The slots for this class are x and y. The slots for an S4 object can be accessed with the @ operator.

S4 Class/Method: Polygon Class

A plot method can be created with the `setMethod` function.

- For `setMethod` you need to specify a generic function (`plot`), and a *signature*.
- A signature is a character vector indicating the classes of objects that are accepted by the method. In this case, the `plot` method will take one type of object—a polygon object.

```
setMethod("plot", "polygon",  
  function(x, y, ...) {  
    plot(x@x, x@y, type = "n", ...)  
    xp <- c(x@x, x@x[1])  
    yp <- c(x@y, x@y[1])  
    lines(xp, yp)  
  })
```

Notice that the slots of the polygon (the x- and y-coordinates) are accessed with the `@` operator.

S4 Class/Method: Polygon Class

Create a new class

```
> setClass("polygon",  
+         representation(x = "numeric",  
+                        y = "numeric"))
```

Create a plot method for this class

```
> setMethod("plot", "polygon",  
+          function(x, y, ...) {  
+              plot(x@x, x@y, type = "n", ...)  
+              xp <- c(x@x, x@x[1])  
+              yp <- c(x@y, x@y[1])  
+              lines(xp, yp)  
+          })
```

```
[1] "plot"
```

If things go well, you will not get any messages or errors and nothing useful will be returned by either `setClass` or `setMethod`.

S4 Class/Method: Polygon Class

After calling `setMethod` the new `plot` method will be added to the list of methods for `plot`.

```
> showMethods("plot")
```

```
Function: plot (package graphics)
```

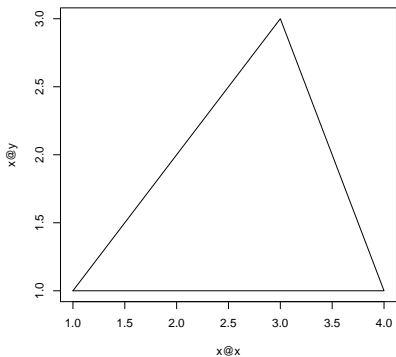
```
x="ANY"
```

```
x="polygon"
```

Notice that the signature for class `polygon` is listed. The method for `ANY` is the default method and it is what is called when now other signature matches

S4 Class/Method: Polygon class

```
> p <- new("polygon", x = c(1, 2, 3, 4), y = c(1, 2, 3, 1))  
> plot(p)
```



Where to Look, Places to Start

- The best way to learn this stuff is to look at examples (and try the exercises for the course)
- There are now quite a few examples on CRAN which use S4 classes/methods.
- Bioconductor (<http://www.bioconductor.org>) — a rich resource, even if you know nothing about bioinformatics
- Some packages on CRAN (as far as I know) — SparseM, gpclib, flexmix, its, lme4, orientlib, pixmap
- The stats4 package (comes with R) has a bunch of classes/methods for doing maximum likelihood analysis.

Dates and Times in R

Computing for Data Analysis

January 23, 2013

Dates and Times in R

R has developed a special representation of dates and times

- ▶ Dates are represented by the `Date` class
- ▶ Times are represented by the `POSIXct` or the `POSIXlt` class
- ▶ Dates are stored internally as the number of days since 1970-01-01
- ▶ Times are stored internally as the number of seconds since 1970-01-01

Dates in R

Dates are represented by the Date class and can be coerced from a character string using the `as.Date()` function.

```
x <- as.Date("1970-01-01")
```

```
x
```

```
## [1] "1970-01-01"
```

```
unclass(x)
```

```
## [1] 0
```

```
unclass(as.Date("1970-01-02"))
```

```
## [1] 1
```

Times in R

Times are represented using the `POSIXct` or the `POSIXlt` class

- ▶ `POSIXct` is just a very large integer under the hood; it use a useful class when you want to store times in something like a data frame
- ▶ `POSIXlt` is a list underneath and it stores a bunch of other useful information like the day of the week, day of the year, month, day of the month

There are a number of generic functions that work on dates and times

- ▶ `weekdays`: give the day of the week
- ▶ `months`: give the month name
- ▶ `quarters`: give the quarter number ("Q1", "Q2", "Q3", or "Q4")

Times in R

Times can be coerced from a character string using the `as.POSIXlt` or `as.POSIXct` function.

```
x <- Sys.time()
```

```
x
```

```
## [1] "2013-01-23 15:19:11 EST"
```

```
p <- as.POSIXlt(x)
```

```
names(unclass(p))
```

```
## [1] "sec"    "min"    "hour"   "mday"   "mon"
```

```
## [6] "year"   "wday"   "yday"   "isdst"
```

```
p$sec
```

```
## [1] 11.86
```

Times in R

You can also use the POSIXct format.

```
x <- Sys.time()
x ## Already in 'POSIXct' format

## [1] "2013-01-23 15:19:11 EST"

unclass(x)

## [1] 1358972352

x$sec

## Error: $ operator is invalid for atomic vectors

p <- as.POSIXlt(x)
p$sec

## [1] 11.88
```

Times in R

Finally, there is the `strptime` function in case your dates are written in a different format

```
datestring <- c("January 10, 2012 10:40", "December 9, 2011 09:10")
x <- strptime(datestring, "%B %d, %Y %H:%M")
x
```

```
## [1] "2012-01-10 10:40:00" "2011-12-09 09:10:00"
```

```
class(x)
```

```
## [1] "POSIXlt" "POSIXt"
```

I can *never* remember the formatting strings. Check `?strptime` for details.

Operations on Dates and Times

You can use mathematical operations on dates and times. Well, really just + and -. You can do comparisons too (i.e. ==, <=)

```
x <- as.Date("2012-01-01")  
y <- strptime("9 Jan 2011 11:34:21", "%d %b %Y %H:%M:%S")  
x - y
```

```
## Warning: Incompatible methods ("-.Date",  
## "-.POSIXt") for "-"
```

```
## Error: non-numeric argument to binary operator
```

```
x <- as.POSIXlt(x)  
x - y
```

```
## Time difference of 356.3 days
```


Operations on Dates and Times

Even keeps track of leap years, leap seconds, daylight savings, and time zones.

```
x <- as.Date("2012-03-01")  
y <- as.Date("2012-02-28")  
x - y
```

Time difference of 2 days

```
x <- as.POSIXct("2012-10-25 01:00:00")  
y <- as.POSIXct("2012-10-25 06:00:00", tz = "GMT")  
y - x
```

Time difference of 1 hours

Summary

- ▶ Dates and times have special classes in R that allow for numerical and statistical calculations
- ▶ Dates use the `Date` class
- ▶ Times use the `POSIXct` and `POSIXlt` class
- ▶ Character strings can be coerced to `Date/Time` classes using the `strptime` function or the `as.Date`, `as.POSIXlt`, or `as.POSIXct`

Regular Expressions in R

Computing for Data Analysis

Regular Expression Functions

The primary R functions for dealing with regular expressions are

- `grep`, `grep1`: Search for matches of a regular expression/pattern in a character vector; either return the indices into the character vector that match, the strings that happen to match, or a TRUE/FALSE vector indicating which elements match
- `regexpr`, `gregexpr`: Search a character vector for regular expression matches and return the indices of the string where the match begins and the length of the match
- `sub`, `gsub`: Search a character vector for regular expression matches and replace that match with another string
- `regexec`: Easier to explain through demonstration.

Here is an excerpt of the Baltimore City homicides dataset:

```
> homicides <- readLines("homicides.txt")
> homicides[1]
[1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon
Nelson</dt><dd class=\"address\">3400 Clifton Ave.<br />Baltimore, MD
21216</dd><dd>black male, 17 years old</dd>
<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd></dl>'"

> homicides[1000]
[1] "39.33626300000, -76.55553990000, icon_homicide_shooting, 'p1200', ..."
```

How can I find the records for all the victims of shootings (as opposed to other causes)?

```
> length(grep("iconHomicideShooting", homicides))  
[1] 228  
> length(grep("iconHomicideShooting|icon_homicide_shooting", homicides))  
[1] 1003  
> length(grep("Cause: shooting", homicides))  
[1] 228  
> length(grep("Cause: [Ss]hooting", homicides))  
[1] 1003  
> length(grep("[Ss]hooting", homicides))  
[1] 1005
```

```
> i <- grep("[cC]ause: [Ss]hooting", homicides)
> j <- grep("[Ss]hooting", homicides)
> str(i)
  int [1:1003] 1 2 6 7 8 9 10 11 12 13 ...
> str(j)
  int [1:1005] 1 2 6 7 8 9 10 11 12 13 ...
> setdiff(i, j)
integer(0)
> setdiff(j, i)
[1] 318 859
```

```
> homicides[859]
[1] "39.33743900000, -76.66316500000, icon_homicide_bluntforce,
'p914', '<dl><dt><a href=\"http://essentials.baltimoresun.com/
micro_sun/homicides/victim/914/steven-harris\">Steven Harris</a>
</dt><dd class=\"address\">4200 Pimlico Road<br />Baltimore, MD 21215
</dd><dd>Race: Black<br />Gender: male<br />Age: 38 years old</dd>
<dd>Found on July 29, 2010</dd><dd>Victim died at Scene</dd>
<dd>Cause: Blunt Force</dd><dd class=\"popup-note\"><p>Harris was
found dead July 22 and ruled a shooting victim; an autopsy
subsequently showed that he had not been shot,...</dd></dl>'"
```


By default, `grep` returns the indices into the character vector where the regex pattern matches.

```
> grep("^New", state.name)
[1] 29 30 31 32
```

Setting `value = TRUE` returns the actual elements of the character vector that match.

```
> grep("^New", state.name, value = TRUE)
[1] "New Hampshire" "New Jersey"      "New Mexico"      "New York"
```

`grepl` returns a logical vector indicating which element matches.

```
> grepl("^New", state.name)
[1] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25] FALSE FALSE FALSE FALSE TRUE TRUE TRUE TRUE FALSE FALSE FALSE FALSE
[37] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[49] FALSE FALSE
```

Some limitations of `grep`

- The `grep` function tells you which strings in a character vector match a certain pattern but it doesn't tell you exactly where the match occurs or what the match is (for a more complicated regex).
- The `regexpr` function gives you the index into each string where the match begins and the length of the match for that string.
- `regexpr` only gives you the first match of the string (reading left to right).
`gregexpr` will give you all of the matches in a given string.

How can we find the date of the homicide?

```
> homicides[1]
[1] "39.311024, -76.674227, iconHomicideShooting, 'p2', '<dl><dt>Leon
Nelson</dt><dd class=\"address\">3400 Clifton Ave.<br />Baltimore,
MD 21216</dd><dd>black male, 17 years old</dd>
<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd></dl>'"
```

Can we just 'grep' on "Found"?

The word 'found' may be found elsewhere in the entry.

```
> homicides[954]
```

```
[1] "39.30677400000, -76.59891100000, icon_homicide_shooting, 'p816',  
'<dl><dd class=\"address\">1400 N Caroline St<br />Baltimore, MD 21213</dd>  
<dd>Race: Black<br />Gender: male<br />Age: 29 years old</dd>  
<dd>Found on March 3, 2010</dd><dd>Victim died at Scene</dd>  
<dd>Cause: Shooting</dd><dd class=\"popup-note\"><p>Wheeler\\'s body  
was&nbsp;found on the grounds of Dr. Bernard Harris Sr.&nbsp;Elementary  
School</p></dd></dl>'"
```

Let's use the pattern

```
<dd>[F|f]ound(.*)</dd>
```

What does this look for?

```
> regexpr("<dd>[F|f]ound(.*)</dd>", homicides[1:10])
[1] 177 178 188 189 178 182 178 187 182 183
attr(,"match.length")
[1] 93 86 89 90 89 84 85 84 88 84
attr(,"useBytes")
[1] TRUE
> substr(homicides[1], 177, 177 + 93 - 1)
[1] "<dd>Found on January 1, 2007</dd><dd>Victim died at Shock
Trauma</dd><dd>Cause: shooting</dd>"
```

The previous pattern was too greedy and matched too much of the string. We need to use the ? metacharacter to make the regex “lazy”.

```
> regexpr("<dd>[F|f]ound(.*)</dd>", homicides[1:10])
[1] 177 178 188 189 178 182 178 187 182 183
attr(,"match.length")
[1] 33 33 33 33 33 33 33 33 33 33
attr(,"useBytes")
[1] TRUE

> substr(homicides[1], 177, 177 + 33 - 1)
[1] "<dd>Found on January 1, 2007</dd>"
```

One handy function is `regmatches` which extracts the matches in the strings for you without you having to use `substr`.

```
> r <- regexpr("<dd>[F|f]ound(.*)</dd>", homicides[1:5])
> regmatches(homicides[1:5], r)
[1] "<dd>Found on January 1, 2007</dd>" "<dd>Found on January 2, 2007</dd>"
[3] "<dd>Found on January 2, 2007</dd>" "<dd>Found on January 3, 2007</dd>"
[5] "<dd>Found on January 5, 2007</dd>"
```

Sometimes we need to clean things up or modify strings by matching a pattern and replacing it with something else. For example, how can we extract the data from this string?

```
> x <- substr(homicides[1], 177, 177 + 33 - 1)
> x
[1] "<dd>Found on January 1, 2007</dd>"
```

We want to strip out the stuff surrounding the “January 1, 2007” piece.

```
> sub("<dd>[F|f]ound on |</dd>", "", x)
[1] "January 1, 2007</dd>"
```

```
> gsub("<dd>[F|f]ound on |</dd>", "", x)
[1] "January 1, 2007"
```


sub/gsub can take vector arguments

```
> r <- regexpr("<dd>[F|f]ound(.*?)</dd>", homicides[1:5])
> m <- regmatches(homicides[1:5], r)
> m
[1] "<dd>Found on January 1, 2007</dd>" "<dd>Found on January 2, 2007</dd>"
[3] "<dd>Found on January 2, 2007</dd>" "<dd>Found on January 3, 2007</dd>"
[5] "<dd>Found on January 5, 2007</dd>"
> gsub("<dd>[F|f]ound on |</dd>", "", m)
[1] "January 1, 2007" "January 2, 2007" "January 2, 2007" "January 3, 2007"
[5] "January 5, 2007"
> as.Date(d, "%B %d, %Y")
[1] "2007-01-01" "2007-01-02" "2007-01-02" "2007-01-03" "2007-01-05"
```

The `regexec` function works like `regexpr` except it gives you the indices for parenthesized sub-expressions.

```
> regexec("<dd>[F|f]ound on (.*)</dd>", homicides[1])  
[[1]]  
[1] 177 190  
attr(,"match.length")  
[1] 33 15
```

```
> regexec("<dd>[F|f]ound on .*?</dd>", homicides[1])  
[[1]]  
[1] 177  
attr(,"match.length")  
[1] 33
```

Now we can extract the string in the parenthesized sub-expression.

```
> regexec("<dd>[F|f]ound on (.*?)</dd>", homicides[1])  
[[1]]  
[1] 177 190  
attr("match.length")  
[1] 33 15  
  
> substr(homicides[1], 177, 177 + 33 - 1)  
[1] "<dd>Found on January 1, 2007</dd>"  
  
> substr(homicides[1], 190, 190 + 15 - 1)  
[1] "January 1, 2007"
```

Even easier with the `regmatches` function.

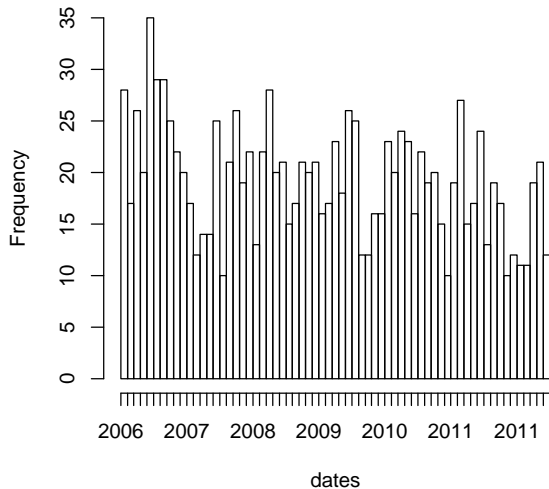
```
> r <- regexec("<dd>[F|f]ound on (.*)</dd>", homicides[1:2])
> regmatches(homicides[1:2], r)
[[1]]
[1] "<dd>Found on January 1, 2007</dd>" "January 1, 2007"

[[2]]
[1] "<dd>Found on January 2, 2007</dd>" "January 2, 2007"
```

Let's make a plot of monthly homicide counts

```
> r <- regexec("<dd>[F|f]ound on (.*)</dd>", homicides)
> m <- regmatches(homicides, r)
> dates <- sapply(m, function(x) x[2])
> dates <- as.Date(dates, "%B %d, %Y")
> hist(dates, "month", freq = TRUE)
```

Histogram of dates



The primary R functions for dealing with regular expressions are

- `grep`, `grep1`: Search for matches of a regular expression/pattern in a character vector
- `regexpr`, `gregexpr`: Search a character vector for regular expression matches and return the indices where the match begins; useful in conjunction with `regmatches`
- `sub`, `gsub`: Search a character vector for regular expression matches and replace that match with another string
- `regexec`: Gives you indices of parenthesized sub-expressions.

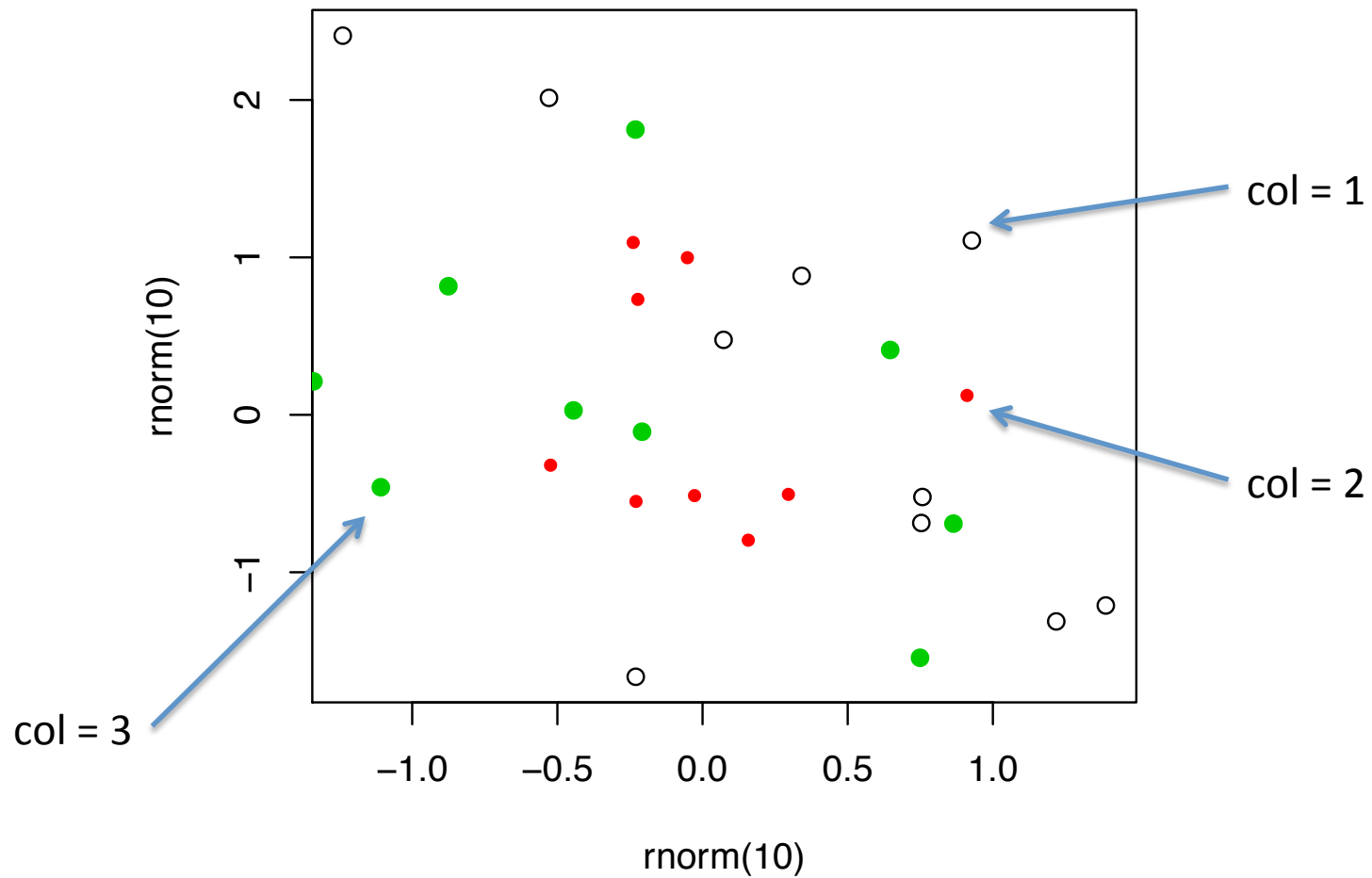
Plotting and Color in R

Computing for Data Analysis

Plotting and Color

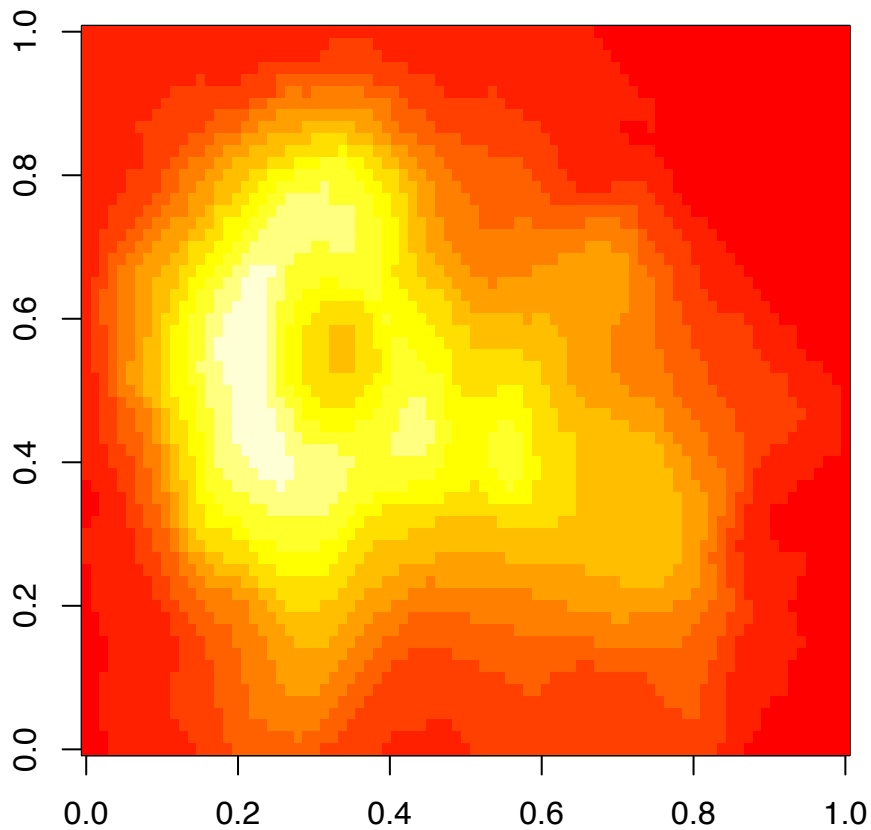
- The default color schemes for most plots in R are horrendous
 - I don't have good taste and even I know that
- Recently there have been developments to improve the handling/specification of colors in plots/graphs/etc.
- There are functions in R and in external packages that are very handy

Colors 1, 2, and 3

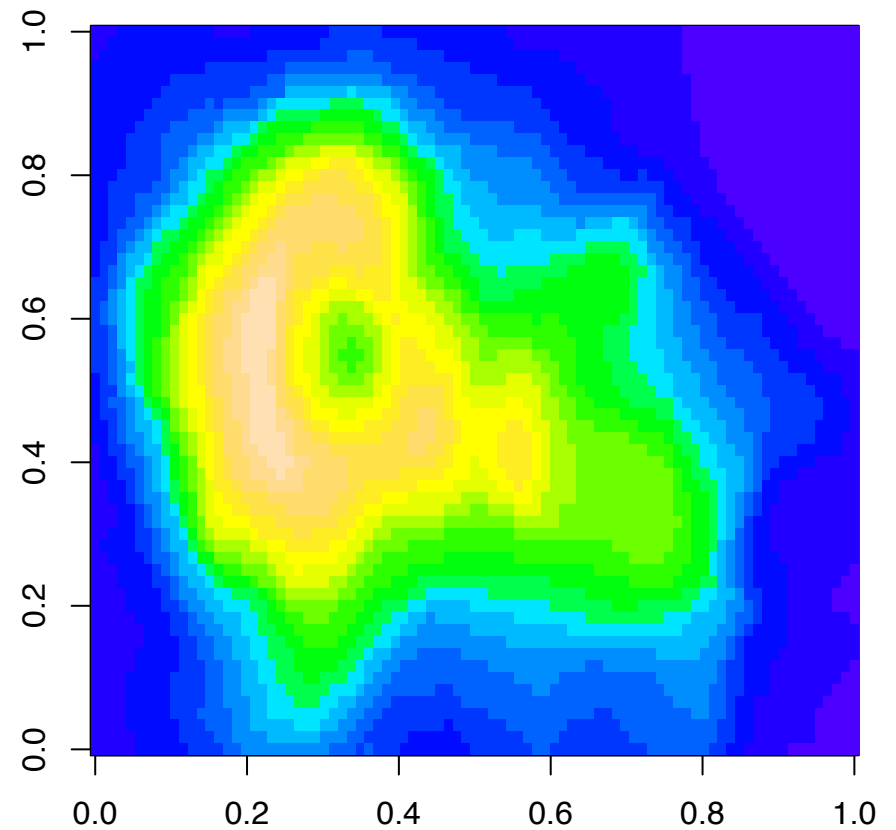


Default Image Plots in R

heat.colors()



topo.colors()



Color Utilities in R

- The **grDevices** package has two functions
 - `colorRamp`
 - `colorRampPalette`
- These functions take palettes of colors and help to interpolate between the colors
- The function `colors ()` lists the names of colors you can use in any plotting function

Color Palette Utilities in R

- `colorRamp`: Take a palette of colors and return a function that takes values between 0 and 1, indicating the extremes of the color palette (e.g. see the 'gray' function)
- `colorRampPalette`: Take a palette of colors and return a function that takes integer arguments and returns a vector of colors interpolating the palette (like `heat.colors` or `topo.colors`)

colorRamp

```
> pal <- colorRamp(c("red", "blue"))
```

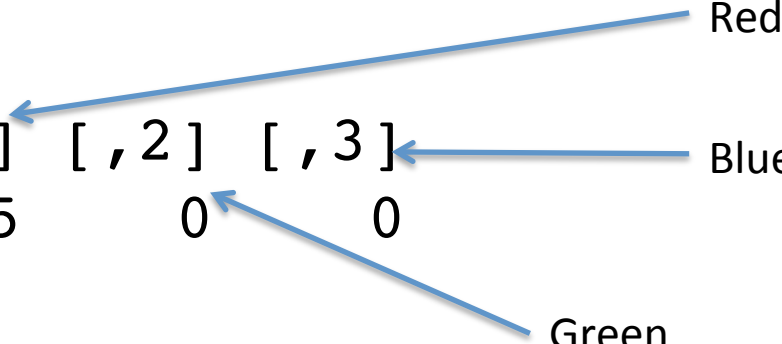
```
> pal(0)
```

	[,1]	[,2]	[,3]
[1,]	255	0	0

Red

Blue

Green



```
> pal(1)
```

	[,1]	[,2]	[,3]
[1,]	0	0	255

```
> pal(0.5)
```

	[,1]	[,2]	[,3]
[1,]	127.5	0	127.5

colorRamp

```
> pal(seq(0, 1, len = 10))  
      [,1] [,2]      [,3]  
[1,] 255.00000      0  0.00000  
[2,] 226.66667      0  28.33333  
[3,] 198.33333      0  56.66667  
[4,] 170.00000      0  85.00000  
[5,] 141.66667      0 113.33333  
[6,] 113.33333      0 141.66667  
[7,]  85.00000      0 170.00000  
[8,]  56.66667      0 198.33333  
[9,]  28.33333      0 226.66667  
[10,]   0.00000      0 255.00000
```

colorRampPalette

```
> pal <- colorRampPalette(c("red", "yellow"))
```

```
> pal(2)
```

```
[1] "#FF0000" "#FFFF00"
```

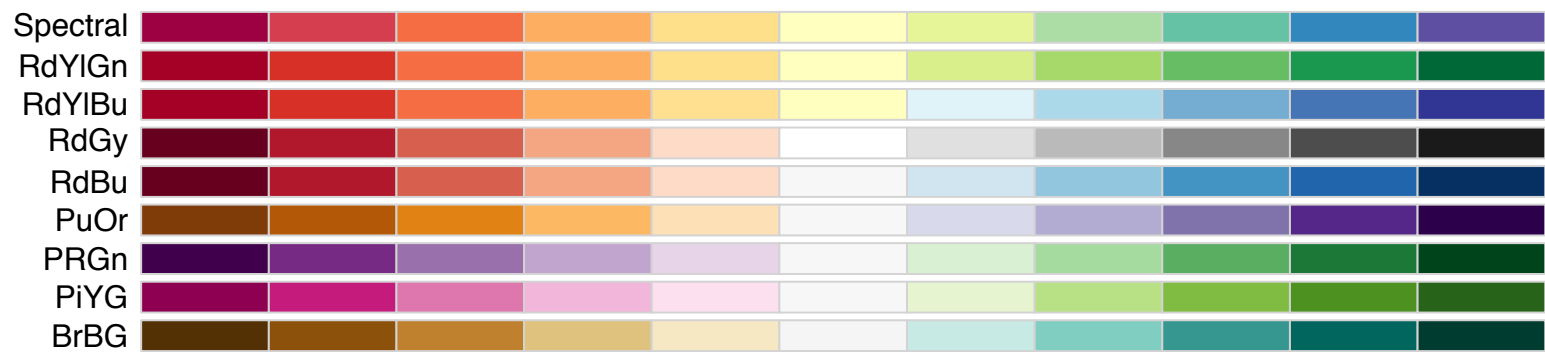
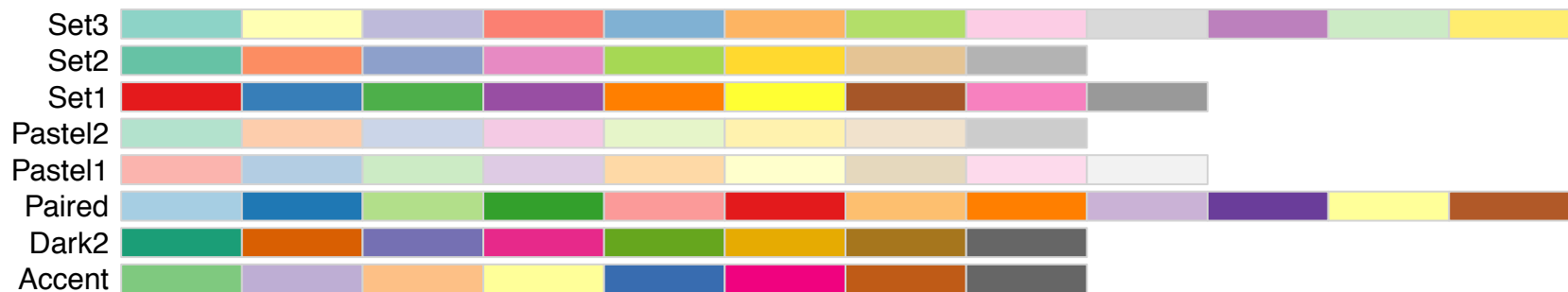
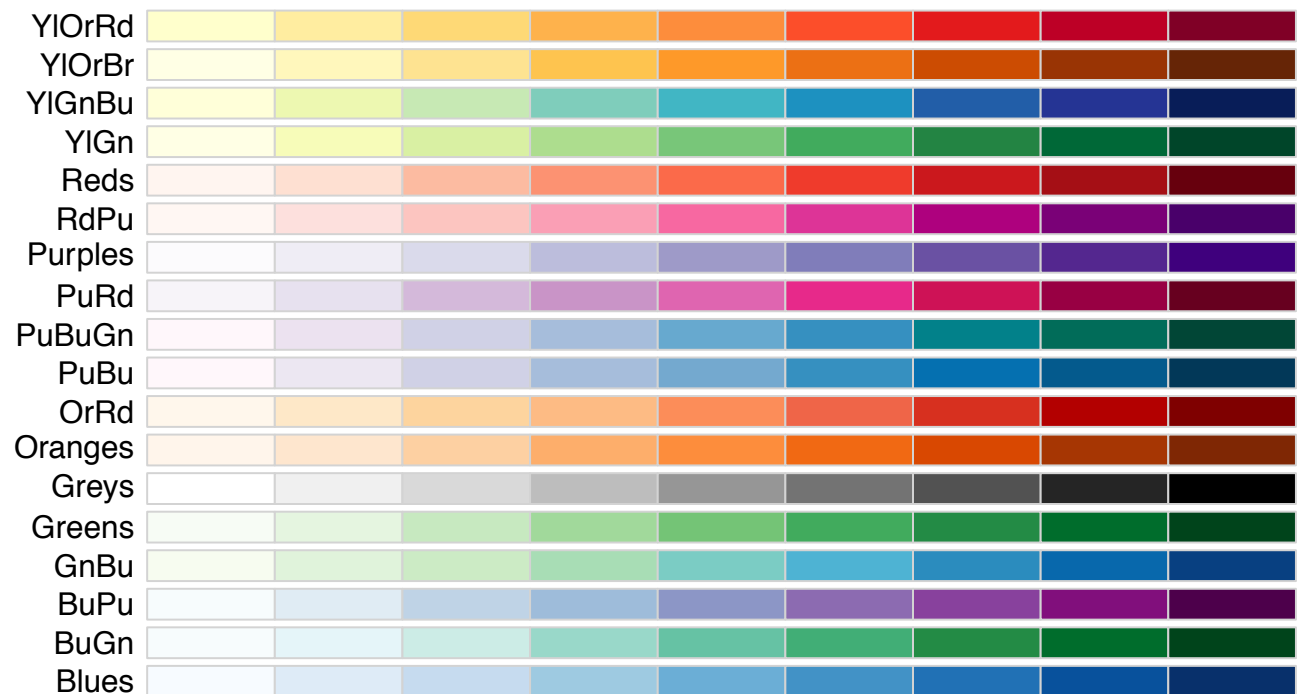
```
> pal(10)
```

```
[1] "#FF0000" "#FF1C00" "#FF3800" "#FF5500" "#FF7100"
```

```
[6] "#FF8D00" "#FFAA00" "#FFC600" "#FFE200" "#FFFF00"
```


RColorBrewer Package

- One package on CRAN that contains interesting/useful color palettes
- There are 3 types of palettes
 - Sequential
 - Diverging
 - Qualitative
- Palette information can be used in conjunction with the `colorRamp()` and `colorRampPalette()`



RColorBrewer and colorRampPalette

```
> library(RColorBrewer)

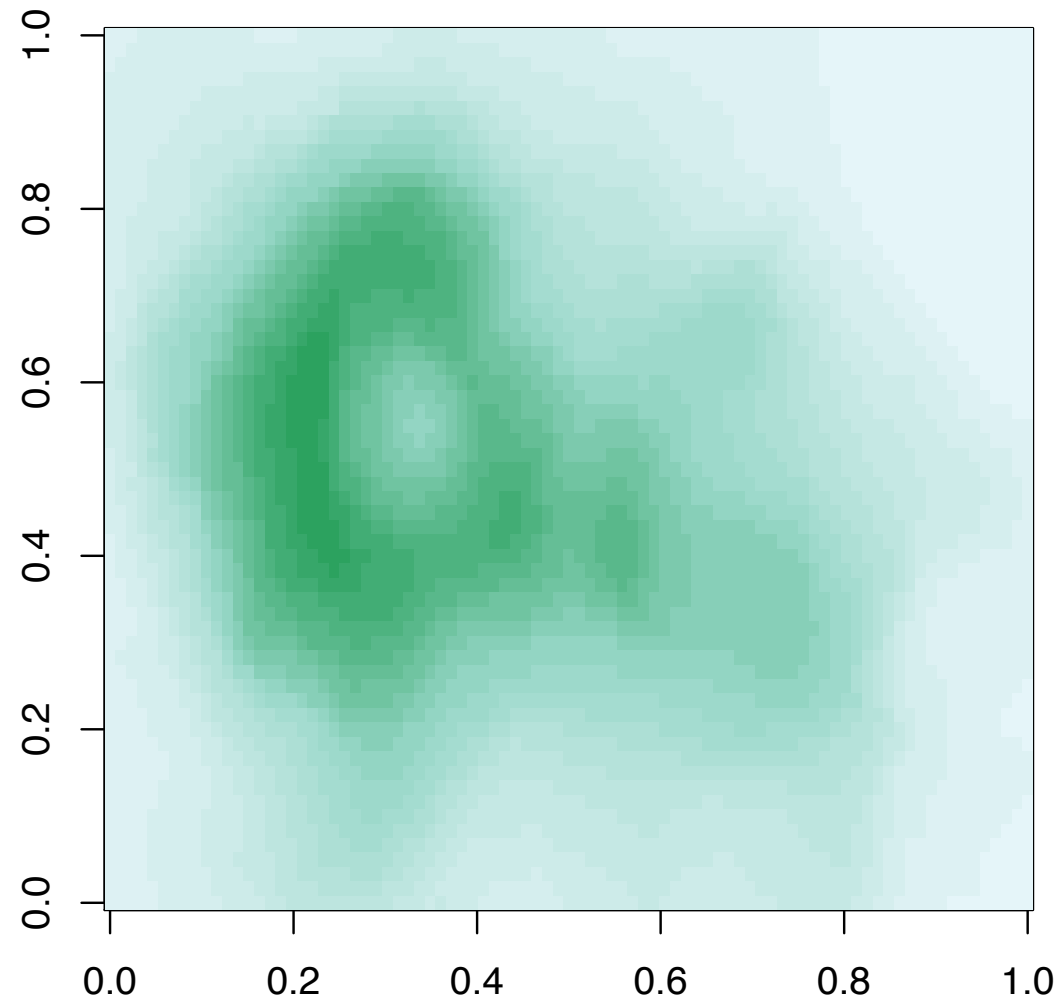
> cols <- brewer.pal(3, "BuGn")

> cols
[1] "#E5F5F9" "#99D8C9" "#2CA25F"

> pal <- colorRampPalette(cols)

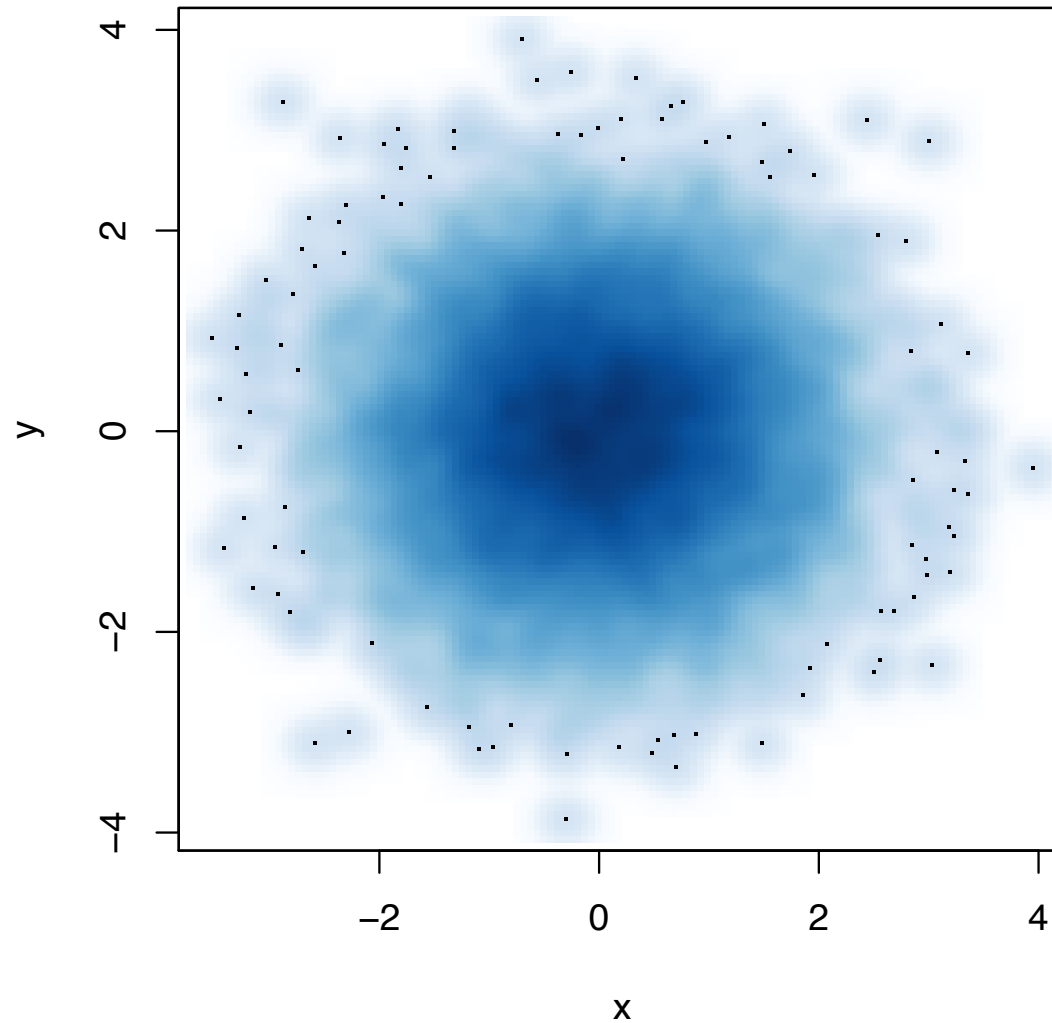
> image(volcano, col = pal(20))
```

RColorBrewer and colorRampPalette



The smoothScatter function

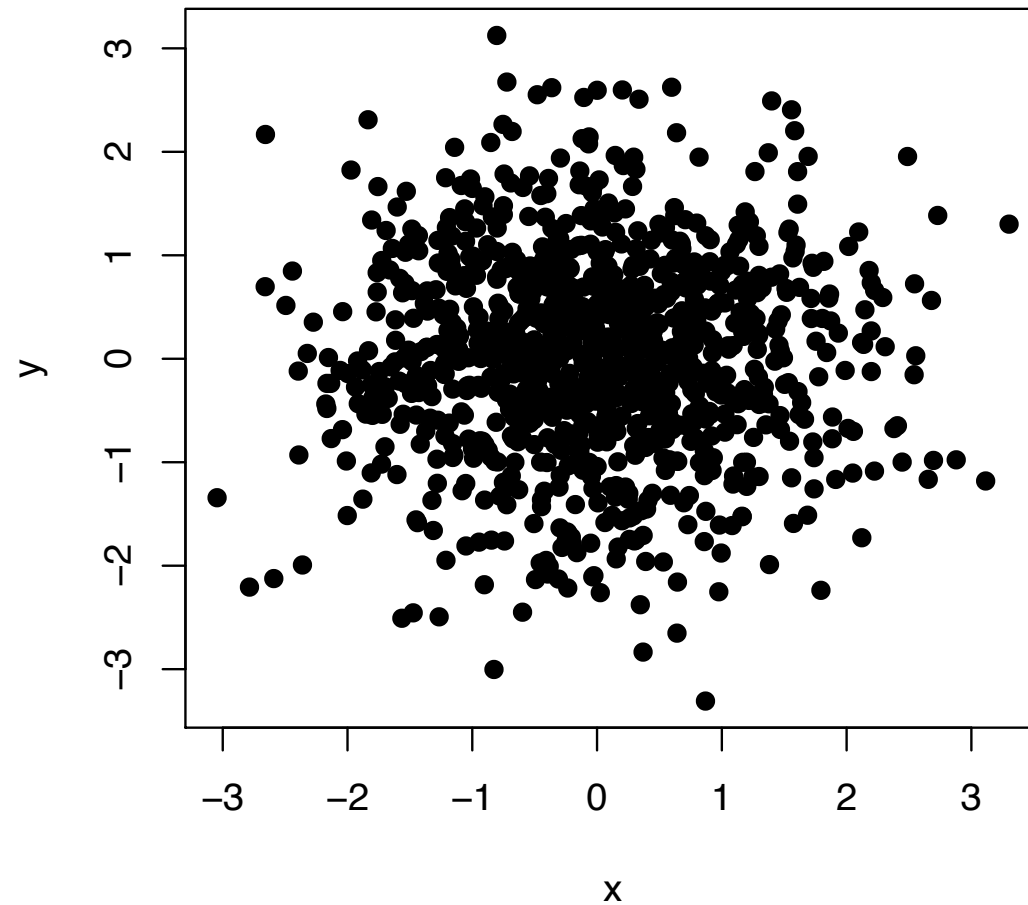
```
x <- rnorm(10000)  
y <- rnorm(10000)  
smoothScatter(x, y)
```



Some Other Plotting Notes

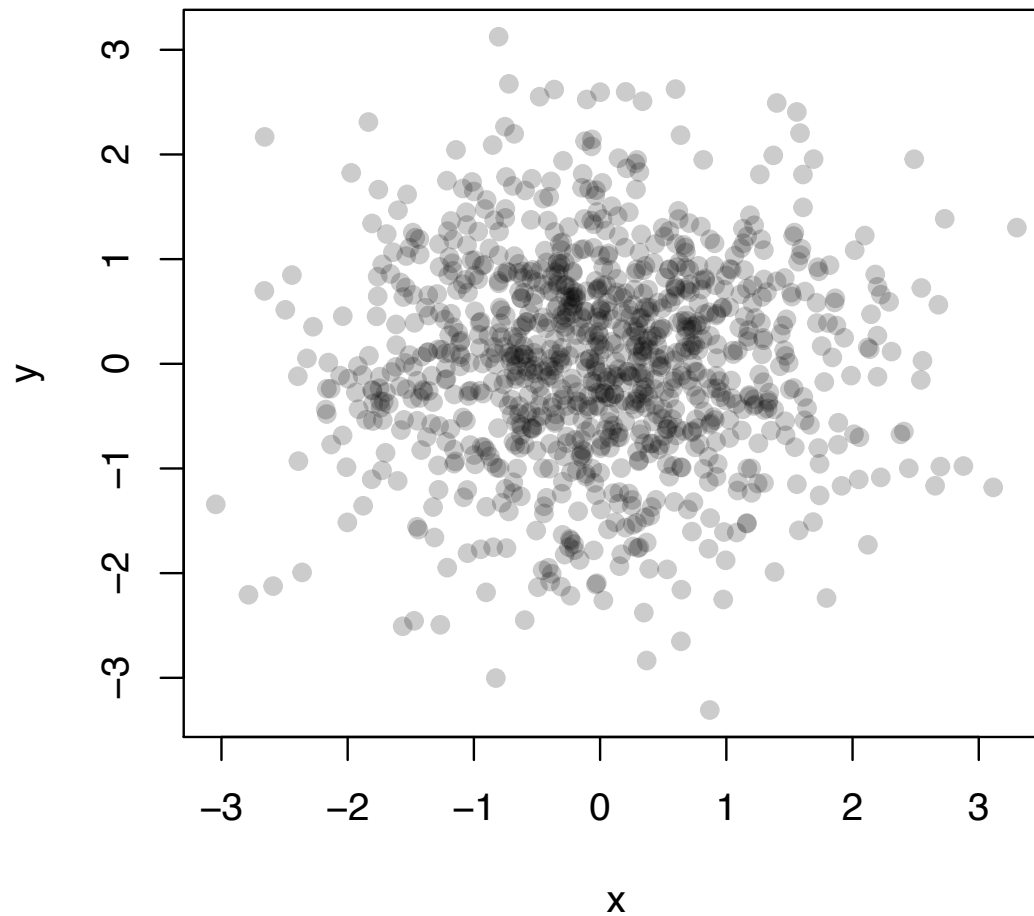
- The `rgb` function can be used to produce any color via red, green, blue proportions
- Color transparency can be added via the `alpha` parameter to `rgb`
- The **colorspace** package can be used for a different control over colors

Scatterplot with no transparency



`plot(x, y, pch = 19)`

Scatterplot with transparency



```
plot(x, y, col = rgb(0, 0, 0, 0.2), pch = 19)
```


Summary

- Careful use of colors in plots/maps/etc. can make it easier for the reader to get what you're trying to say (why make it harder?)
- The **RColorBrewer** package is an R package that provides color palettes for sequential, categorical, and diverging data
- The `colorRamp` and `colorRampPalette` functions can be used in conjunction with color palettes to connect data to colors
- Transparency can sometimes be used to clarify plots with many points

Regular Expressions

Computing for Data Analysis

- Regular expressions can be thought of as a combination of literals and *metacharacters*
- To draw an analogy with natural language, think of literal text forming the words of this language, and the metacharacters defining its grammar
- Regular expressions have a rich set of metacharacters

Literals

Simplest pattern consists only of literals. The literal “nuclear” would match to the following lines:

```
Ooh. I just learned that to keep myself alive after a
nuclear blast! All I have to do is milk some rats
then drink the milk. Aweosme. :}
```

```
Laozi says nuclear weapons are mas macho
```

```
Chaos in a country that has nuclear weapons -- not good.
```

```
my nephew is trying to teach me nuclear physics, or
possibly just trying to show me how smart he is
so I'll be proud of him [which I am].
```

```
lol if you ever say "nuclear" people immediately think
DEATH by radiation LOL
```

The literal “Obama” would match to the following lines

Politics r dum. Not 2 long ago Clinton was sayin Obama
was crap n now she sez vote 4 him n unite? WTF?
Screw em both + McCain. Go Ron Paul!

Clinton concedes to Obama but will her followers listen??

Are we sure Chelsea didn't vote for Obama?

thinking ... Michelle Obama is terrific!

jetlag..no sleep...early mornig to starbux..Ms. Obama
was moving

- Simplest pattern consists only of literals; a match occurs if the sequence of literals occurs anywhere in the text being tested
- What if we only want the word “Obama”? or sentences that end in the word “Clinton”, or “clinton” or “clinto”?

We need a way to express

- whitespace word boundaries
- sets of literals
- the beginning and end of a line
- alternatives (“war” or “peace”)

Metacharacters to the rescue!

Some metacharacters represent the start of a line

```
^i think
```

will match the lines

```
i think we all rule for participating
i think i have been outed
i think this will be quite fun actually
i think i need to go to work
i think i first saw zombo in 1999.
```


Metacharacters

\$ represents the end of a line

morning\$

will match the lines

```
well they had something this morning
then had to catch a tram home in the morning
dog obedience school in the morning
and yes happy birthday i forgot to say it earlier this morning
I walked in the rain this morning
good morning
```

Character Classes with []

We can list a set of characters we will accept at a given point in the match

```
[Bb] [Uu] [Ss] [Hh]
```

will match the lines

```
The democrats are playing, "Name the worst thing about Bush!"  
I smelled the desert creosote bush, brownies, BBQ chicken  
BBQ and bushwalking at Molonglo Gorge  
Bush TOLD you that North Korea is part of the Axis of Evil  
I'm listening to Bush - Hurricane (Album Version)
```

Character Classes with []

`^[Ii] am`

will match

i am so angry at my boyfriend i can't even bear to
look at him

i am boycotting the apple store

I am twittering from iPhone

I am a very vengeful person when you ruin my sweetheart.

I am so over this. I need food. Mmmm bacon...

Character Classes with []

Similarly, you can specify a range of letters [a-z] or [a-zA-Z]; notice that the order doesn't matter

```
^[0-9][a-zA-Z]
```

will match the lines

```
7th inning stretch
```

```
2nd half soon to begin. OSU did just win something
```

```
3am - cant sleep - too hot still.. :(
```

```
5ft 7 sent from heaven
```

```
1st sign of starvagtion
```

Character Classes with []

When used at the beginning of a character class, the “^” is also a metacharacter and indicates matching characters NOT in the indicated class

```
[^?.]$
```

will match the lines

```
i like basketballs
```

```
6 and 9
```

```
dont worry... we all die anyway!
```

```
Not in Baghdad
```

```
helicopter under water? hmmm
```

“.” is used to refer to any character. So

9.11

will match the lines

```
its stupid the post 9-11 rules
```

```
if any 1 of us did 9/11 we would have been caught in days.
```

```
NetBios: scanning ip 203.169.114.66
```

```
Front Door 9:11:46 AM
```

```
Sings: 0118999881999119725...3 !
```

This does not mean “pipe” in the context of regular expressions; instead it translates to “or”; we can use it to combine two expressions, the subexpressions being called alternatives

```
flood|fire
```

will match the lines

```
is firewire like usb on none macs?
```

```
the global flood makes sense within the context of the bible
```

```
yeah ive had the fire on tonight
```

```
... and the floods, hurricanes, killer heatwaves, rednecks, gun nuts, etc.
```

We can include any number of alternatives...

```
flood|earthquake|hurricane|coldfire
```

will match the lines

```
Not a whole lot of hurricanes in the Arctic.
```

```
We do have earthquakes nearly every day somewhere in our State
```

```
hurricanes swirl in the other direction
```

```
coldfire is STRAIGHT!
```

```
'cause we keep getting earthquakes
```


The alternatives can be real expressions and not just literals

```
^[Gg]ood|[Bb]ad
```

will match the lines

```
good to hear some good knews from someone here
```

```
Good afternoon fellow american infidels!
```

```
good on you-what do you drive?
```

```
Katie... guess they had bad experiences...
```

```
my middle name is trouble, Miss Bad News
```

More Metacharacters: (and)

Subexpressions are often contained in parentheses to constrain the alternatives

```
^([Gg]ood|[Bb]ad)
```

will match the lines

```
bad habbit
```

```
bad coordination today
```

```
good, becuae there is nothing worse than a man in kinky underwear
```

```
Badcop, its because people want to use drugs
```

```
Good Monday Holiday
```

```
Good riddance to Limey
```

More Metacharacters: ?

The question mark indicates that the indicated expression is optional

```
[Gg]eorge( [Ww]\.)? [Bb]ush
```

will match the lines

```
i bet i can spell better than you and george bush combined
```

```
BBC reported that President George W. Bush claimed God told him to invade
```

```
a bird in the hand is worth two george bushes
```

One thing to note...

In the following

```
[Gg]eorge( [Ww]\.)? [Bb]ush
```

we wanted to match a “.” as a literal period; to do that, we had to “escape” the metacharacter, preceding it with a backslash In general, we have to do this for any metacharacter we want to include in our match

More metacharacters: * and +

The * and + signs are metacharacters used to indicate repetition; * means “any number, including none, of the item” and + means “at least one of the item”

`(.*)`

will match the lines

`anyone wanna chat? (24, m, germany)`

`hello, 20.m here... (east area + drives + webcam)`

`(he means older men)`

`()`

More metacharacters: * and +

The * and + signs are metacharacters used to indicate repetition; * means “any number, including none, of the item” and + means “at least one of the item”

```
[0-9]+ (.*) [0-9]+
```

will match the lines

```
working as MP here 720 MP battallion, 42nd birgade
so say 2 or 3 years at colleage and 4 at uni makes us 23 when and if we fir
it went down on several occasions for like, 3 or 4 *days*
Mmmm its time 4 me 2 go 2 bed
```

More metacharacters: { and }

{ and } are referred to as interval quantifiers; they let us specify the minimum and maximum number of matches of an expression

```
[Bb]ush( +[^\s]+ +){1,5} debate
```

will match the lines

Bush has historically won all major debates he's done.

in my view, Bush doesn't need these debates..

bush doesn't need the debates? maybe you are right

That's what Bush supporters are doing about the debate.

Felix, I don't disagree that Bush was poorly prepared for the debate.

indeed, but still, Bush should have taken the debate more seriously.

Keep repeating that Bush smirked and scowled during the debate

More metacharacters: `*` and `+`

- `m,n` means at least `m` but not more than `n` matches
- `m` means exactly `m` matches
- `m,` means at least `m` matches

More metacharacters: (and) revisited

- In most implementations of regular expressions, the parentheses not only limit the scope of alternatives divided by a “|”, but also can be used to “remember” text matched by the subexpression enclosed
- We refer to the matched text with \1, \2, etc.

More metacharacters: (and) revisited

So the expression

```
+([a-zA-Z]+) +\1 +
```

will match the lines

```
time for bed, night night twitter!
```

```
blah blah blah blah
```

```
my tattoo is so so itchy today
```

```
i was standing all all alone against the world outside...
```

```
hi anybody anybody at home
```

```
estudiando css css css css.... que desastritooooo
```

More metacharacters: (and) revisited

The `*` is “greedy” so it always matches the *longest* possible string that satisfies the regular expression. So

```
^s(.*)s
```

matches

sitting at starbucks

setting up mysql and rails

studying stuff for the exams

spaghetti with marshmallows

stop fighting with crackers

sore shoulders, stupid ergonomics

More metacharacters: (and) revisited

The greediness of `*` can be turned off with the `?`, as in

```
^s(.*)s$
```

- Regular expressions are used in many different languages; not unique to R.
- Regular expressions are composed of literals and metacharacters that represent sets or classes of characters/words
- Text processing via regular expressions is a very powerful way to extract data from “unfriendly” sources (not all data comes as a CSV file)

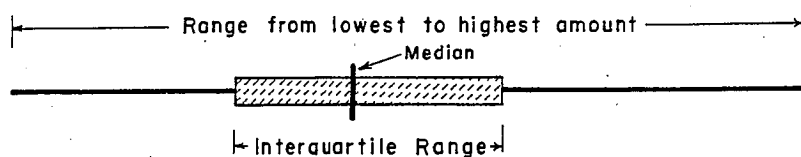
(Thanks to Mark Hansen for some material in this lecture.)

6 *Data-Ink Maximization and Graphical Design*

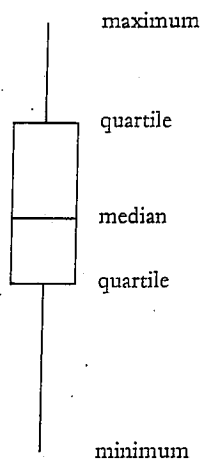
So far the principles of maximizing data-ink and erasing have helped to generate a series of choices in the process of graphical revision. This is an important result, but can the ideas reach beyond the details and particularities of editing? Is it possible to do what a theory of graphics is supposed to do, that is, to derive new graphical forms? In this chapter the principles are applied to many graphical designs, basic and advanced, including box plots, bar charts, histograms, and scatterplots. New designs result.

Redesign of the Box Plot

Mary Eleanor Spear's "range bar"



and John Tukey's "box plot"



Mary Eleanor Spear, *Charting Statistics* (New York, 1952), p. 166; and John W. Tukey, *Exploratory Data Analysis* (Reading, Mass., 1977).

can be mostly erased without loss of information:



The revised design, a *quartile plot*, shows the same five numbers. It is easy to draw by hand or computer and, most importantly, can replace the conventional scatterplot frame. The straightedge need only be placed on the paper once to draw the quartile plot, compared to six separate placings for the box plot. An alternative is

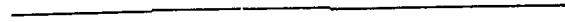


but this design will not work effectively to frame a scatterplot. Nor does it look very good.

Perhaps special emphasis should be given to the middle half of the distribution, however, as in the box plot. This can be done by changing line weights



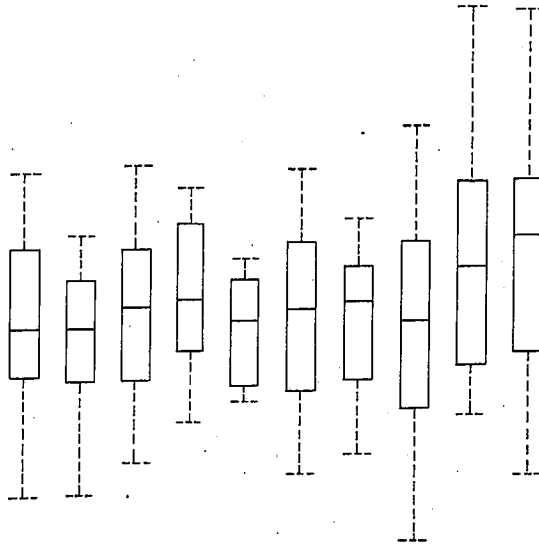
or, even better, by offsetting the middle half:



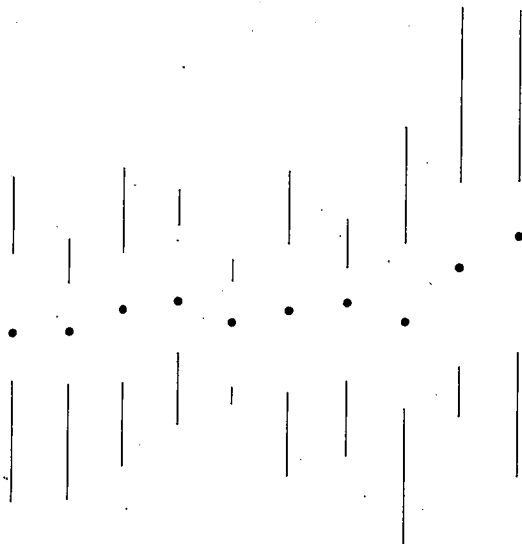
This latter design is the preferred form of the quartile plot. It uses the ink effectively and looks good.

In these revisions of the box plot, the principle of maximizing data-ink has suggested a variety of designs, but the choice of the best overall arrangement naturally also rests on statistical and aesthetic criteria—in other words, the procedure is one of *reasonable* data-ink maximizing.

The same logic applies to many similar designs, such as this "parallel schematic plot." The original required 80 separate placings of the straightedge, 50 horizontals and 30 verticals:



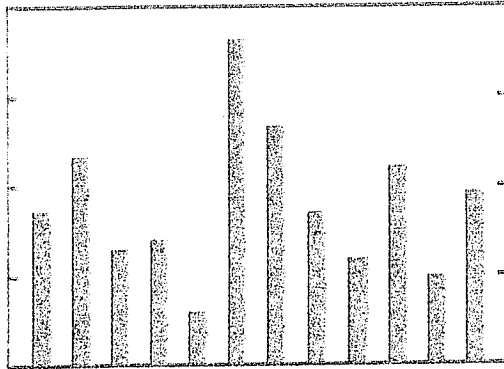
An erased version requires only 10 verticals to show the same information:



The large reduction in the amount of drawing is relevant for the use of such designs in informal, exploratory data analysis, where the research worker's time should be devoted to matters other than drawing lines.

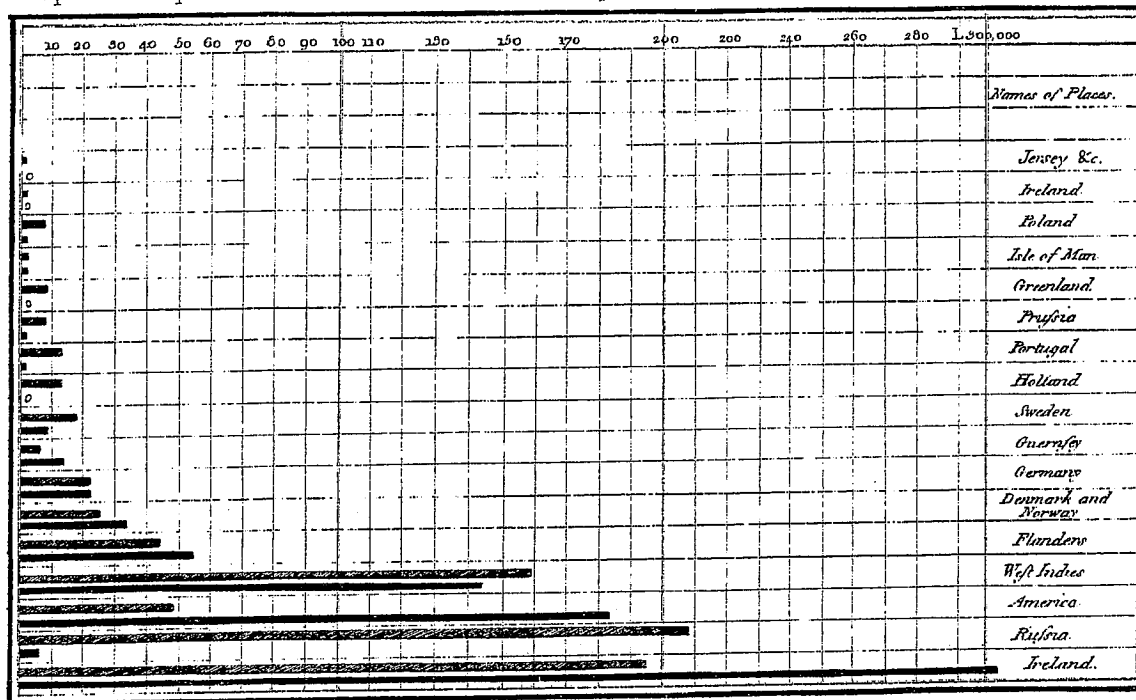
Redesign of the Bar Chart/Histogram

Here is the standard model bar chart, with the design endorsed by the practices and the style sheets of many statistical and scientific publications:



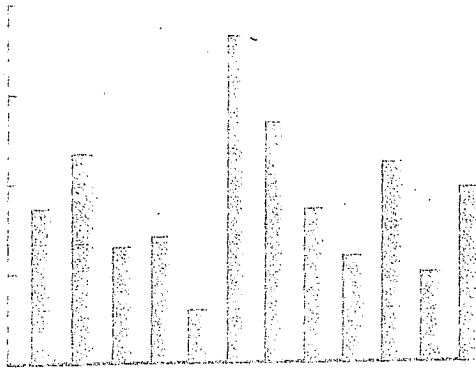
Its architecture differs little from Playfair's original design:

Exports and Imports of SCOTLAND to and from different parts for one Year from Christmas 1780 to Christmas 1781

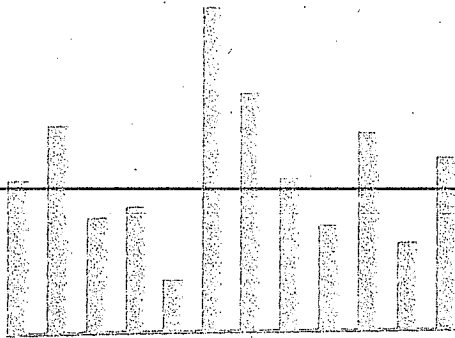


The Upright divisions are Ten Thousand Pounds each. The Black Lines are Exports the Ribbed lines Imports
 Published as the Act directs June 7th 1786 by W^m Playfair
 No. 10. St. Paul's Church-yard, London.

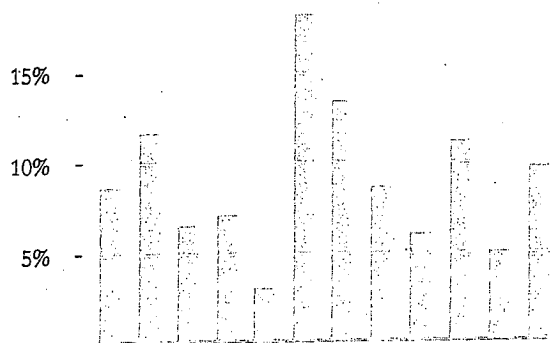
The box can be erased:



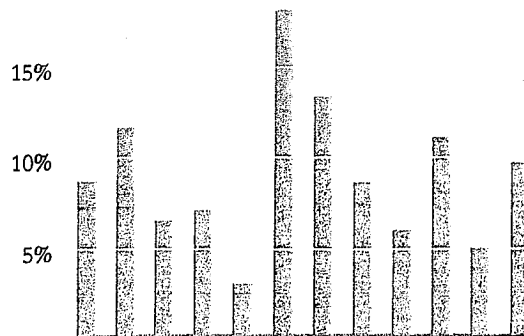
And the vertical axis, except for the ticks:



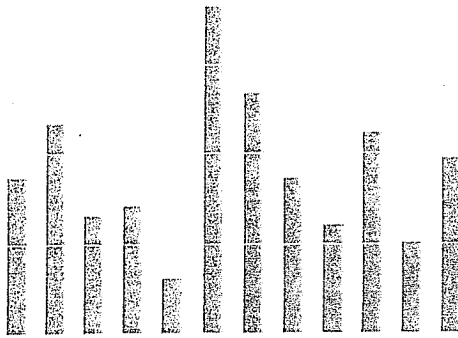
Even part of the data measures can be erased, making a *white grid*, which shows the coordinate lines more precisely than ticks alone:



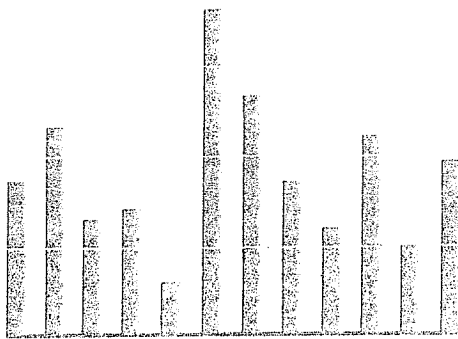
The white grid eliminates the tick marks, since the numerical labels on the vertical are tied directly to the white lines:



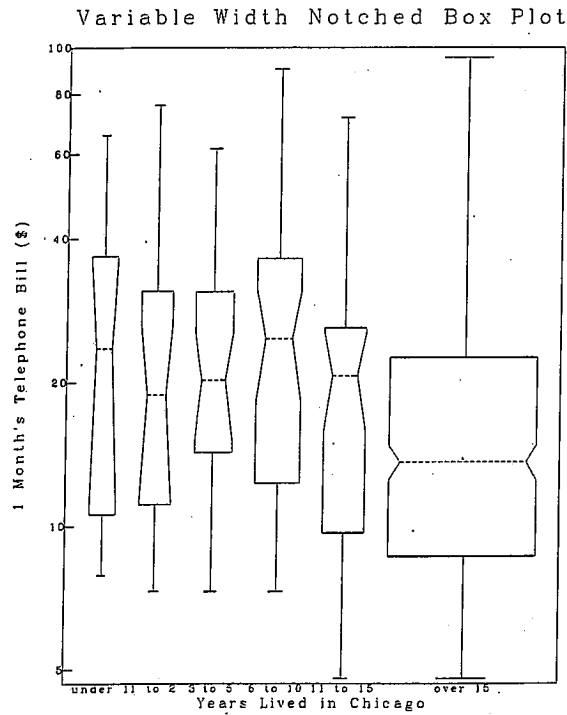
Although the intersection of the thicker bar with the thinner baseline creates an attractive visual effect (but also the optical illusion of gray dots at the intersections), the baseline can be erased since the bars define the end-point at the bottom:



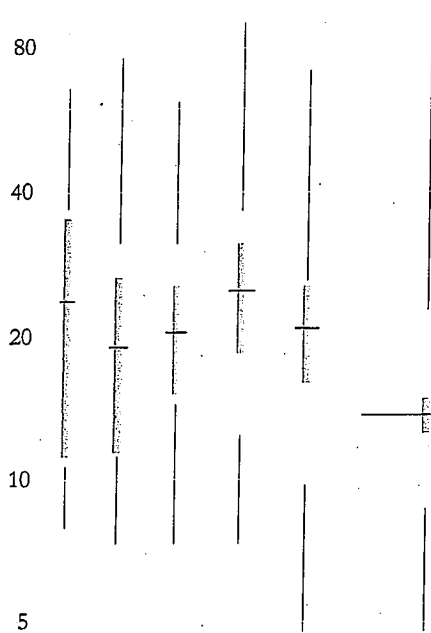
Still, a thin baseline looks good:



Erasing and data-ink maximizing have induced changes in the plain old bar chart. The techniques—no frame, no vertical axis, no ticks, and the white grid—apply to other designs:

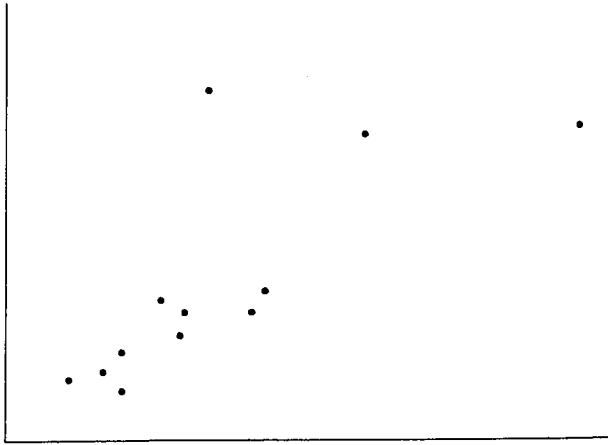


Robert McGill, John W. Tukey, and Wayne A. Larsen, "Variations of Box Plots," *American Statistician*, 32 (1978), 12-16.

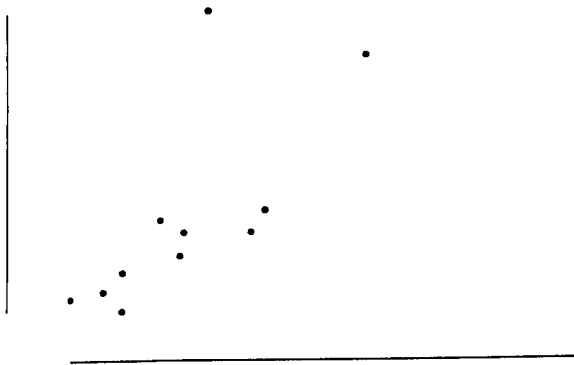


Redesign of the Scatterplot

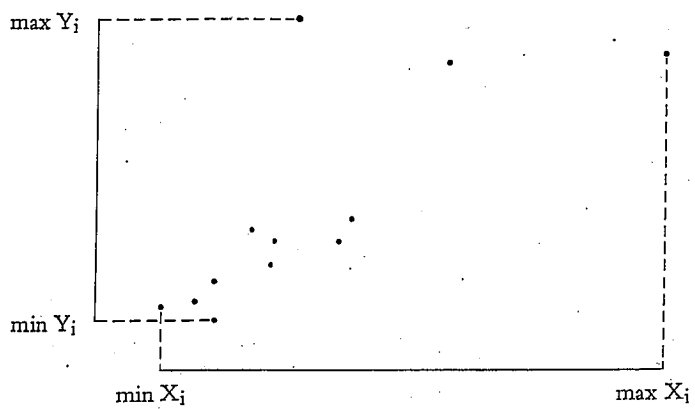
Consider the standard bivariate scatterplot:



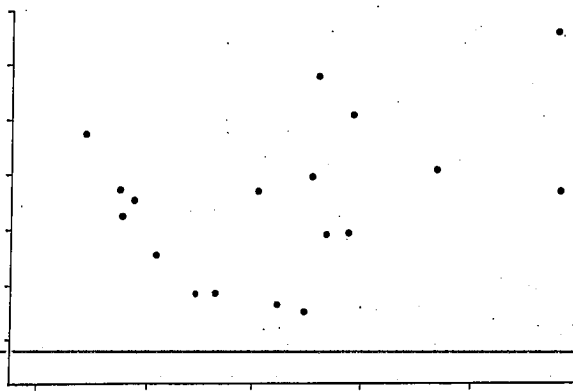
A useful fact, brought to notice by the maximization and erasing principles, is that the frame of a graphic can become an effective data-communicating element simply by erasing part of it. The frame lines should extend only to the measured limits of the data rather than, as is customary, to some arbitrary point like the next round number marking off the grid and grid ticks of the plot. That part of the frame exceeding the limits of the observed data is trimmed off:



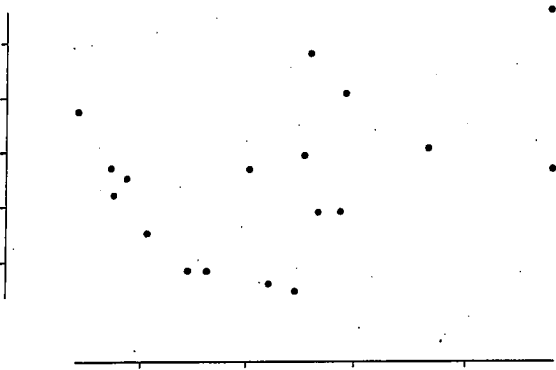
The result, a *range-frame*, explicitly shows the maximum and minimum of both variables plotted (along with the range), information available only by extrapolation and visual estimation in the conventional design. The data-ink ratio has increased: some non-data-ink has been erased, and the remainder of the frame, now carrying information, has gone over to the side of data-ink.



Nothing but the tails of the frame need change:



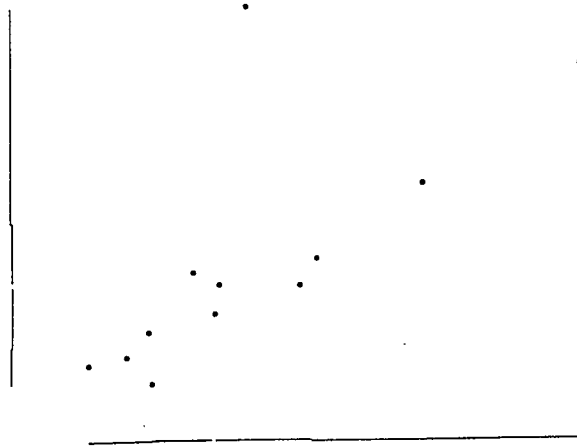
Conventional Scatterplot



Range-Frame

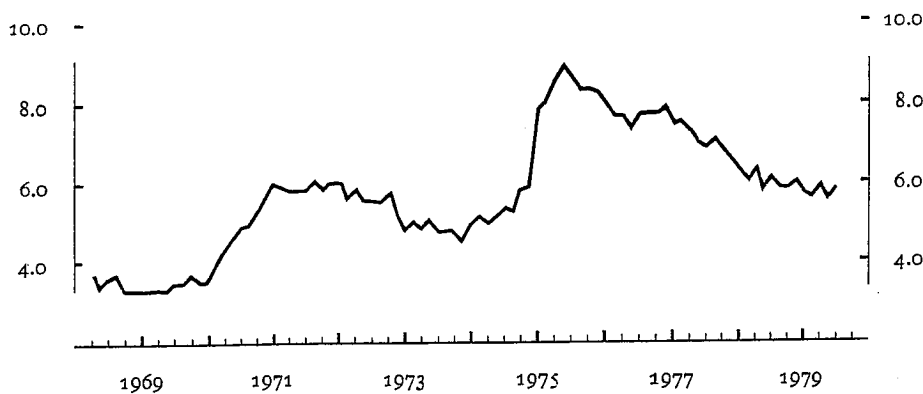
A range-frame does not require any viewing or decoding instructions; it is not a graphical puzzle and most viewers can easily tell what is going on. Since it is more informative about the data in a clear and precise manner, the range-frame should replace the non-data-bearing frame in many graphical applications.

A small shift in the remaining ink turns each range-frame into a quartile plot:



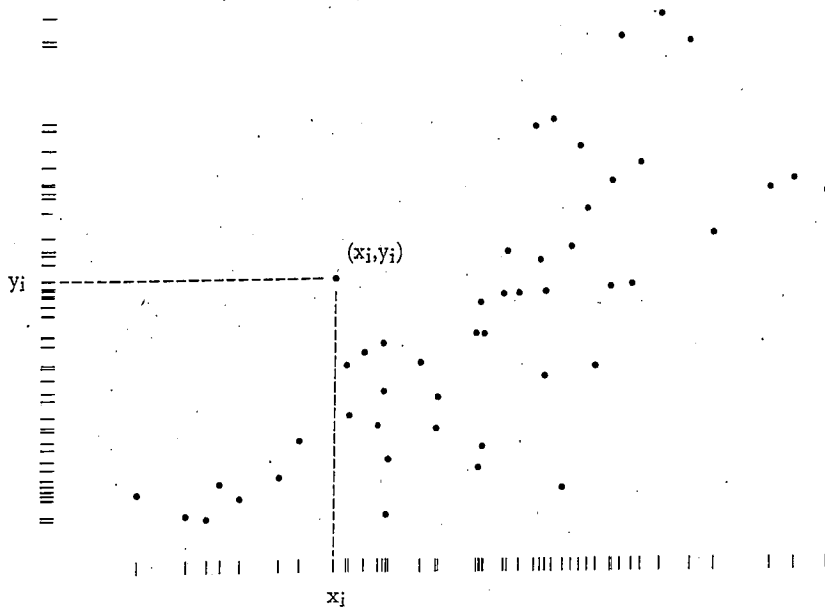
Erasing and editing has led to the display of ten extra numbers (the minimum, maximum, two quartiles, and the median for both variables). The design is useful for analytical and exploratory data analysis, as well as for published graphics where summary characterizations of the marginal distributions have interest. The design is nearly always better than the conventionally framed scatterplot.

Range-frames can also present ranges along a single dimension. Here the historical high and low is shown in the vertical frame:



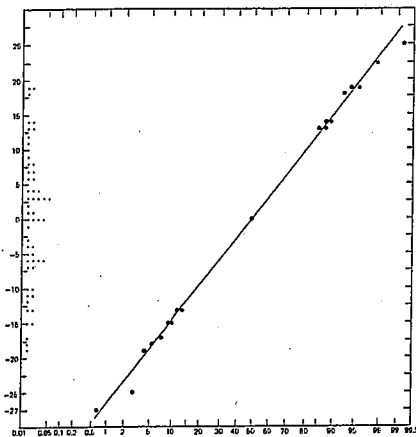
Finally, the entire frame can be turned into data by framing the bivariate scatter with the marginal distribution of each variable. The *dot-dash-plot* results.¹

¹ The terminology follows tradition, for scatterplots were once called "dot diagrams"—for example, in R. A. Fisher's *Statistical Methods for Research Workers* (Edinburgh, 1925).



The dot-dash-plot combines the two fundamental graphical designs used in statistical analysis, the marginal frequency distribution and the bivariate distribution. Dot-dash-plots make routine what good data analysts do already—plotting marginal and joint distributions together.

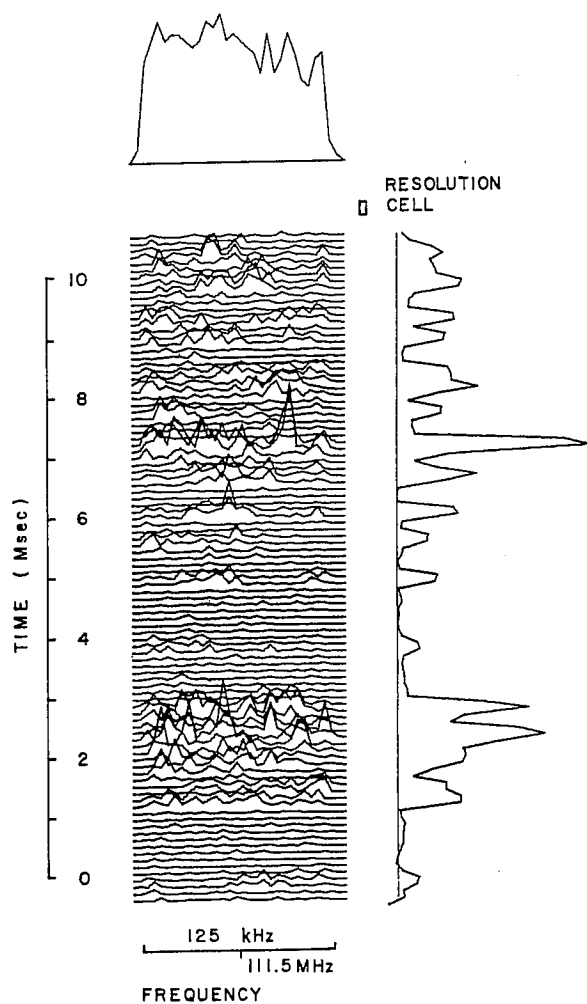
An empirical cumulative distribution of residuals on a normal grid shows the outer 16 terms plus the 30th term, with all 60 points plotted in the marginal distribution:



Cuthbert Daniel, *Applications of Statistics to Industrial Experimentation* (New York, 1976), p. 155.

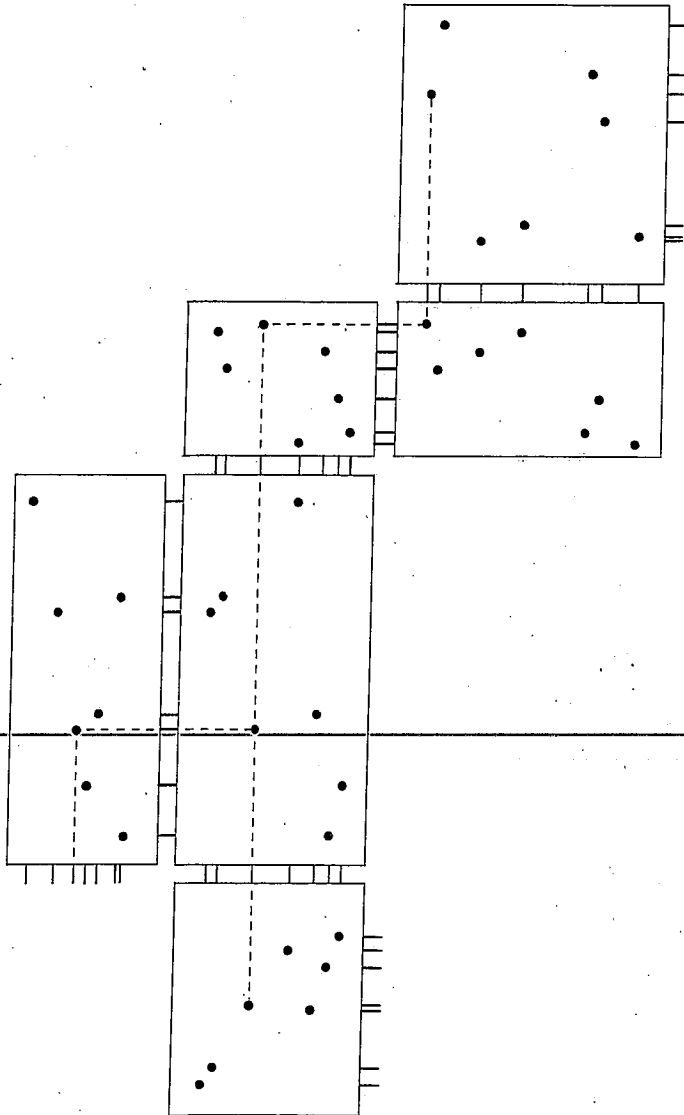
Similarly, this data-rich graphic of signals from pulsars shows both marginal distributions:

Timothy H. Hankins and Barney J. Rickett, "Pulsar Signal Processing," in Berni Alder, et al., eds., *Methods in Computational Physics, Volume 14: Radio Astronomy* (New York, 1975), p. 108.



Narrowband spectra of individual subpulses. Each point of the intensity $I_e(t)$ plotted on the right is the sum of the distribution of intensities across the receiver bandwidth shown in the center. At the top is plotted the spectrum averaged over the pulse. In the limit of many thousands of pulses this would show the receiver bandpass shape.

The fringe of dashes in the dot-dash-plot can connect a series of bivariate scatters in a *rugplot* (since it resembles a set of fringed rugs—and covers the statistical ground):



Reflecting the one-dimensional projections from each scatter, the dashes encourage the eye to notice how each plot filters and translates the data through the scatter from one adjacent plot to the next. Sometimes it is useful to think of each bivariate scatter as the imperfect empirical representation of an underlying curve that transforms one variable into another. In the rugplot, the sequence of variables can wander off as appropriate. The quantitative history of a single observation can be traced through a series of one- and two-dimensional contexts.

Conclusion

The first part of a theory of data graphics is in place. The idea, as described in the previous three chapters, is that most of a graphic's ink should vary in response to data variation. The theory has something to say about a great variety of graphics—workaday scientific charts, the unique drawings of Roger Hayward, the exemplars of graphical handbooks, newspaper displays, computer graphics, standard statistical graphics, and the recent inventions of Chernoff and Tukey.

The observed increases in efficiency, in how much of the graphic's ink carries information, are sometimes quite large. In several cases, the data-ink ratio increased from .1 or .2 to nearly 1.0. The transformed designs are less cluttered and can be shrunk down more readily than the originals.

But, are the transformed designs *better*?

(1) They are necessarily better within the principles of the theory, for more information per unit of space and per unit of ink is displayed. And this is significant; indeed, the history of devices for communicating information is written in terms of increases in efficiency of communication and production.

(2) Graphics are almost always going to improve as they go through editing, revision, and testing against different design options. The principles of maximizing data-ink and erasing generate graphical alternatives and also suggest a direction in which revisions should move.

(3) Then there is the audience: will those looking at the new designs be confused? Some of the designs are self-explanatory, as in the case of the range-frame. The dot-dash-plot is more difficult, although it still shows all the standard information found in the scatterplot. Nothing is lost to those puzzled by the frame of dashes, and something is gained by those who do understand. Moreover, it is a frequent mistake in thinking about statistical graphics to underestimate the audience. Instead, why not assume that if you understand it, most other readers will, too? Graphics should be as intelligent and sophisticated as the accompanying text.

(4) Some of the new designs may appear odd, but this is probably because we have not seen them before. The conventional designs for statistical graphics have been viewed thousands of times by nearly every reader of this book; on the other hand, the range-frame, the dot-dash-plot, the white grid, the quartile plot, the rugplot, and the half-face just a few times. With use, the new designs will come to look just as reasonable as the old.

Maximizing data ink (within reason) is but a single dimension of a complex and multivariate design task. The principle helps conduct experiments in graphical design. Some of those experiments will succeed. There remain, however, many other considerations in the design of statistical graphics—not only of efficiency, but also of complexity, structure, density, and even beauty.

A Brief History of S

Richard A. Becker

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

INTRODUCTION

The S language has been in use for more than 15 years now, and this appears to be a good time to recollect some of the events and influences that marked its development. The present paper covers material on the design of S that has also been addressed by Becker and Chambers (1984b), but the emphasis here is on historical development and less on technical computing details. Also, many important new ideas have come about since that work. Similarly, parts of Chambers (1992) discuss the history of S, but there the emphasis is on very recent developments and future directions.

Why should anyone care about the history of S? This sounds like the question people ask of history in general, and the answer is much the same — the study of the flow of ideas in S, in particular the introduction and dropping of various concepts and their origins, can help us to understand how software in general evolves. Some of the ideas that are so clear today were murky yesterday; how did the insights arise? What things were done well? What conditions helped S to grow into the strong system that it is now? Could we have anticipated the changes that took place and have implemented years ago something more like the current version of S?

This history of S should also be of interest to people who use the commercially available S-PLUS language, because S and S-PLUS are very closely related.

THE EARLY DAYS (1976-80 VERSION OF ‘S’)

S was not the first statistical computing language designed at Bell Laboratories, but it was the first one to be implemented. The pre-S language work dates from 1965 and is described in Becker and Chambers (1989). However, because it did not directly influence S, we will not elaborate here.

S grew up in 1975-1976 in the statistics research departments at Bell Laboratories. Up to that time, most of our statistical computing, both for research and routine data analysis work, was carried out using a large, well-documented Fortran library known as SCS (Statistical Computing Subroutines). The SCS library had been developed in-house over the years to provide a flexible base for our research. Because the statistics research departments were charged with developing new methodology, often to deal with non-standard data situations, the ability to carry out just the right computations was paramount. Routine analysis was rare.

Commercial software did not fit well into our research environment. It often used a “shotgun” approach — print out everything that might be relevant to the problem at hand because it could be several hours before the statistician could get another set of output. This was reasonable when computing was done in a batch mode. However, we wanted to be able to interact with our data, using Exploratory Data Analysis (Tukey, 1971) techniques. In addition, commercial statistical software usually didn’t compute what we wanted and was not set up to be modified. The SCS library provided excellent support for our simulations, large problems, monte-carlo, and non-standard analyses.

On the other hand, we did occasionally do simple computations. For example, suppose we wanted to carry out a linear regression given 20 x, y data points. The idea of writing a Fortran program that called library routines for something like this was unappealing. While the actual regression was done in a single subroutine call, the program had to do its own input and output, and time spent in I/O often dominated the

actual computations. Even more importantly, the effort expended on programming was out of proportion to the size of the problem. An interactive facility could make such work much easier.

It was the realization that routine data analysis should not require writing Fortran programs that really got S going. Our initial goals were not lofty; we simply wanted an interactive interface to the algorithms in the SCS library. In the Spring of 1976, John Chambers, Rick Becker, Doug Dunn and Paul Tukey held a series of meetings to discuss the idea. Graham Wilkinson, who was a visitor at the time, also participated in these early discussions. Gradually, we came to the conclusion that a full language was needed. The initial implementation was the work of Becker, Chambers, Dunn, along with Jean McRae and Judy Schilling.

The environment at Bell Laboratories in 1976 was unusual and certainly influenced the way S came together. Our computer center provided a Honeywell 645 computer running GCOS. Almost all statistical computing was done in Fortran, but there was a wide range of experience in using Fortran to implement portable algorithms, as well as non-numeric software tools. For example, the PFORT Verifier (Ryder & Hall, 1974) was a program written in Fortran to verify the portability of Fortran programs. Portability was a big concern then, since the Honeywell configuration was unlikely to be available to others and since the investment we made in software was considerable.

Bell Laboratories in 1976 also provided a unique combination of people, software tools, and ideas. John Chambers had been at Bell Laboratories for 10 years, had experimented with many approaches to statistical computing, and had codified his thoughts in a book (Chambers, 1977). Rick Becker came to Bell Labs in 1974 with experience in interactive computing from work with the Troll (NBER, 1974) system. The statistics research departments were heavily influenced by John Tukey and his approach to Exploratory Data Analysis (Tukey, 1971). Computer science research at Bell Laboratories was still actively working on the UNIX® Operating System (Ritchie and Thompson, 1978) and many of the tools from that work were available on GCOS. In particular, we had access to the C language, based on the Portable C Compiler (Ritchie, et. al. 1978), the YACC and LEX tools for parsing and lexical analysis (Johnson and Lesk, 1978), the Ratfor language for writing structured Fortran (Kernighan, 1975), Struct, for turning Fortran into Ratfor (Baker, 1977), and the M4 macro processor (Kernighan and Ritchie, 1977).

An important precondition for an interactive computing system was a good graphics facility. Becker and Chambers (1976, 1977) (and Chambers, 1975) wrote a subroutine library called GR-Z to provide flexible, portable, and device-independent graphics. It used a computer-center supplied library, Grafpac, to produce output on batch devices such as microfilm recorders and printer/plotters. Interactive devices, such as Tektronix graphics terminals, line printers, and others were supported by device driver subroutines written for GR-Z. Its graphical parameters controlled graph layout and rendering, and were an extension of those provided by Grafpac. The system provided for graphic input as well as output, and became the cornerstone for graphics in S.

Another enabling experiment was one in separate compilation of portions of a system. Development of a statistical computing system under GCOS required the use of overlays, where portions of the instructions were brought into memory when they were needed. Unlike modern virtual memory systems, space on the Honeywell machine was severely limited, and compilation and linking was slow. Development of the system was likely to require lots of work on the various S functions, and it would be unacceptably slow and expensive to recompile or relink the entire system for every change in one of the functions. To get around this, we developed a pair of assembler language routines that implemented transfer vectors. The main program would ensure that jump instructions to various subroutines were placed at known fixed memory locations. Overlays could then be compiled and linked separately, knowing exactly where they could find various needed subroutines. Similarly, the overlays had a vector of transfer instructions to the overlay's functions stored in another set of fixed addresses. Thus, the main program could always be sure of the address to use when calling the subroutines. This separate compilation/linkage process fit right in with the notion that S would be based on a collection of functions; each function or related group of functions could reside in a single overlay.

A feature of the computing environment on GCOS was the QED text editor (Ritchie, 1970). It was used as a command interpreter, doing textual computations to build an operating system command and finally executing it. Thus, QED provided a primitive analog to the Unix system shell command interpreter. QED was used extensively behind the scenes, to control compilations and overlay loading; it provided the scaffolding that allowed us to build S.

Initial S Concepts

What were the basic ideas involved in S? Our primary goal was to bring interactive computing to bear on statistics and data analysis problems. S was designed as an interactive language based entirely on functions. A few of these functions would be implemented as prefix or infix operators (arithmetic, subscripting, creating sequences), but most would be written with a function name followed by a parenthesized argument list. Functions would be allowed an arbitrary number of positional and keyword arguments and would return a data structure that would allow a collection of named results. Arguments that were left out would have default values. The basic data structures would be vectors and a hierarchical structure, a combination of vectors and/or other such structures (Chambers, 1978). Entire collections of data would be referred to by a single name.

The power of S and its functions would never be used, however, unless the system were extremely flexible and provided the operations that our colleagues wanted. In order to access the computations that were presently available as subroutines in the SCS library (and those that would later be written) we planned to import subroutines written in a lower-level programming language (Fortran) and we planned to allow ordinary *users* to do the same thing. Of course, to import an algorithm would require some sort of interface, to translate between the internal S data structures and those of the language, as well as to regularize the calling sequence. Our model for this was a diagram that represented an algorithm as a circle, with a square interface routine wrapped around it, allowing it to fit into the square slot provided for S functions.

Another important notion that came early was that the language should be defined generally, based on a formal grammar, and have as few restrictions as possible. (We were all tired of Fortran's seemingly capricious restrictions, on the form of subscripts, etc.) English-like syntax had been tried in many other contexts and had generally failed to resemble a natural language in the end, so we were happy to have a simple, regular language. Other applications languages at the time seemed too complicated — although a formal grammar might describe the basic expression syntax, there were generally many other parts. We wanted the expression syntax to cover everything. General functions and data structures were the key to this simplicity. Functions that could take arbitrary numbers of arguments as input, and produce an arbitrarily complex data structure as output should be all the language would need. Infix operators for arithmetic and subscripting provided very natural expressions, but were simply syntactic sugar for underlying arithmetic and subscripting functions.

From the beginning, one of the most powerful operations in S was the subscripting operator. It comes in several flavors. First, a vector of numeric subscripts selects corresponding elements of an object. Even here there is a twist, because the vector of subscripts can contain repeats, making the result longer than the original. Negative subscripts may be an idea that originated with S: they describe which elements should *not* be selected. Logical subscripts select elements corresponding to TRUE values, and empty subscripts select everything. All of these subscripting operations generalize to multi-way arrays. In addition, any data structure can be subscripted as if it were a vector.†

S began with several notions about data. The most common side effect in S is accomplished by the assignment function; it gives a name to an S object and causes it to be stored. This storage is persistent — it lasts across S sessions.

The basic data structure in S is a vector of like-elements: numbers, character strings, or logical values. Although the notion of an *attribute* for an S object wasn't clearly implemented until the 1988 release, from the beginning S recognized that the primary vector of data was often accompanied by other values that described special properties of the data. For example, a matrix is just a vector of data along with an auxiliary vector named `Dim` that tells the dimensionality (number of rows and columns). Similarly, a time series has a `Tsp` attribute to tell the start time, end time, and number of observations per cycle. These vectors with attributes are known as *vector structures*, and this distinguishes S from most other systems. For example, in APL, everything is a multi-way array, while in Troll, everything is a time series. The LISP notion of a property list is probably the closest analogy to S attributes. The general treatment of vectors with other attributes in S makes data generalizations much easier and has naturally extended to an object-oriented

†Later, for arrays, we added one further twist: a matrix subscript with k columns can be used to select individual elements of a k -way array.

implementation in recent releases (but that's for later).

Vector structures in S were treated specially in many computations. Most notably, subscripting was handled specially for arrays, with multiple subscripts allowed, one for each dimension. Similarly, time series were time aligned (influenced by Troll) for arithmetic operations.

Another data notion in the first implementation of S was the idea of a hierarchical structure, which contained a collection of other S objects.[†] Early S structures were made up of named *components*; they were treated specially, with the “\$” operator selecting components from structures and functions for creating and modifying structures. (There was even a nascent understanding of the need to iterate over all elements of a structure, accomplished by allowing a number instead of a name when using the “\$” operator.) By the 1988 release we recognized these structures as yet another form of vector, of mode *list*, with an attribute vector of names. This was a very powerful notion, and it integrated the treatment of hierarchical structures with that of ordinary vectors.

Implementation

Persistent data in S was stored in a system known as *scatter storage*, implemented in routines on the SCS library by Jack Warner. Earlier projects in statistics research had needed what amounted to a large flat file system, held in one operating-system file. Scatter storage provided that: give it a hunk of data, its length, and a name and it would store the data under the name. Given the name, it could retrieve the data.

In memory, S objects were stored in dynamically allocated space; S used SCS library Fortran-callable utilities that enabled a program to request space from the operating system and use that space for holding Fortran arrays.

The implementation was done in Fortran for many reasons. Primarily, there was little other choice. C was only beginning to get a start on the GCOS system and at the time, very little numerical work had been done in C. Assembler language was obviously difficult and non-portable. Also, by using Fortran, we were able to make good use of the routines already available on the SCS library. S was already a formidable implementation job and in the beginning we wanted to make the best use of the already-written components. (As S has been re-implemented, all of the reliance on SCS and most of the Fortran has gone away.) On the other hand, even though we were using Fortran, we were loath to give up modern control structures (IF-ELSE) and data structures. As a compromise, we used Ratfor along with the M4 macro processor to give us an implementation language that was quite reasonable, especially before the advent of Fortran 77; we called it the *algorithm language*.

As S was being built, there was a long-held belief at Bell Labs in the importance of software portability and we made a number of attempts to provide it. The Fortran code we used was intended to be portable, not only for numerical software, but also for character handling, where M4 was used to parameterize machine-specific constants such as the number of characters that could be stored in an integer. The GCOS environment certainly encouraged thoughts of portability, because it provided, on one machine, two different character sets (BCD and ASCII) and two different sets of system calls (for batch and time-sharing). To provide even more for portability, we tried to write code that would encapsulate operating-system dependencies: file opening, memory management, printing.

Because S is based so heavily on functions that carry out particular computational tasks, we realized as soon as we had a parser and executive program that it would be necessary to implement lots of functions. Although many of the functions had at their core an SCS subroutine, it soon became apparent that implementation of the necessary interface routines was repetitive, error-prone, and boring. As a result, a set of M4 macros, the *interface macros*, was developed to help speed the implementation of interface routines. We also developed some new algorithms for S; many of them, involving for example, linear algebra and random numbers, reflected ideas described in Chambers, 1977.

From the beginning, S was designed to provide a complete environment for data analysis. We believed that the raw data would be read into S, operated on by various S functions, and results would be

[†] In this discussion we will refer to S data structures as *objects*, even though this nomenclature came later. The connotations of the word are appropriate: an S object can contain arbitrary data, is self-describing, and stands on its own without requiring a name.

produced. Users would spend much more time computing on S objects than they did in getting them into or out of S. This is reflected in the relatively primitive input/output facilities for S. Basically, one function was provided to read a vector of data values into S from a file, and no provisions were made for accommodating Fortran-style formatted input/output.

From the beginning, S included interactive graphics. This was implemented by the GR-Z software. A separate overlay area was provided for the graphics device functions, so that the S user could perform appropriate graphics on a graphics terminal or at least line-printer style graphics on a character terminal.

Another early decision was that S should have documentation available online. A *detailed documentation* file was written for each function and described arguments, default values, usage, details of the algorithm, an example, and cross references to other documentation.

The S executive program was a very basic loop: read user input, parse it according to the S grammar, walk the parse tree evaluating the arguments and functions, and then print the result. The S executive used 20K 36-bit words, and the functions were executed in a 6K-word overlay area. This gave a reasonable time-sharing response time on GCOS for datasets of less than 10,000 data items.

Internal data structures were based on a 4-word header for S vectors, that gave a pointer to a character string name, a numeric code for the mode of the data, its length, and a pointer to a vector of appropriate values (Chambers, 1974). These headers are still around in recent implementations, although they no longer reflect the true data structure, since they make no explicit provision for attributes.

A Working Version of 'S'

By July, 1976, we decided to name the system. Acronyms were in abundance at Bell Laboratories, so it seemed sure that we would come up with one for our system, but no one seemed to be able to agree with any one else's suggestion: Interactive SCS (ISCS), Statistical Computing System (too confusing), Statistical Analysis System (already taken), etc. In the end, we decided that all of these names contained an 'S' and, with the C programming language as a precedent, decided to name the system 'S'. (We dropped the quotes by 1979).

By January, 1977, there was a manual describing Version 1.0 of S that included 30 pages of detailed documentation on the available functions. People who computed on the GCOS machine at Murray Hill were invited to use S, and a user community gradually began to form.

Work on S progressed quickly and by May, 1977, we had a revised implementation that used a new set of database management routines. Version 2.0 was introduced about a year later and included new features, such as missing values, looping, broader support for character data, and many more functions.

The earliest versions of S worked reasonably well and were used for statistical computing at Bell Labs. However, there were a number of problems that we hoped to address in the future. We saw the need for an increasing number of functions and, even with the interface macros, it was too hard to interface new subroutines to S. A goal was to enable users to interface their own subroutines. Our reliance on the QED editor to provide basic maintenance scripts for S was painful, because QED programs were difficult to write and almost impossible to read.

As S developed, the transfer vectors that pointed into the S executive would occasionally need updating. This necessitated a long process of relinking all of the S functions.

The basic parse-eval-print loop was successful and insulated users from most problems involving operating system crashes; when the computer went down, all computations carried out to that time were already saved. However, the master scatter storage file was sometimes corrupted by a crash that happened in the middle of updates, making the system somewhat fragile.

THE UNIX GENERATION (1981 BROWN BOOK)

Although the GCOS implementation of S proved its feasibility, it soon became apparent that growth of S required a different environment. Many of our colleagues in statistics research were using S for routine data analysis, but were still writing stand-alone Fortran programs for their research work. At the same time, word of S was getting to other researchers, both within the Bell System and at universities around the country. The time had come for S to expand its horizons.

Hardware Changes

In 1978, an opportunity came to perform a portability experiment with S. Portability was considered an important notion for S, so when the computer science research departments got an Interdata 8/32 computer to use in portability experiments for the UNIX operating system (Johnson and Ritchie, 1978), S naturally went along. The Interdata machine was a byte-oriented computer with 32-bit words. Prior to this, S work on UNIX was hampered by the 16-bit address space of machines that ran the UNIX system and by the lack of a good Fortran compiler. The latter problem was solved by a new Fortran 77 compiler (Feldman and Weinberger, 1978). As the UNIX research progressed on the Interdata machine, all sorts of useful tools and languages (C in particular) became available for use with S.

Gradually, we realized the advantages of using the UNIX system as the base for S. By the time that we had parallel implementations of S on GCOS and UNIX, it became apparent to us that porting S to a new operating system was going to be painful, no matter how much work was done to isolate system dependencies. That is because the utilities, the scaffolding needed to build S, would have to be rewritten for each new operating system. That was a big task. By writing those utilities once for the UNIX system and making use of the shell, the loader, *make*, and other tools, S could travel to any hardware by hitching a ride with UNIX. It was obvious even then that there would be far more interest in porting the UNIX operating system to new hardware than in porting S to a new system.

The link with the UNIX system provided other benefits as well. S could use the C language for implementation, since C was likely to be available and well debugged on any UNIX machine. Since the *f77* Fortran-77 compiler used the same code generator as C, inter-language calls were easy. S could rely on a consistent environment, with the shell command language (a real programming language to replace QED scripts), the *make* command for describing the system build process, known compilers and compiler options, a reasonably stable linker and libraries, and a hierarchical file system with very general file names.

The benefits of linking S and UNIX were obvious, but the Interdata environment was a one-time experiment and not hardware that would be widely used. The DEC PDP-11 machines were the natural home of the UNIX system at the time and we wanted to try S there. Unfortunately, these machines had a 16-bit architecture and the squeeze proved to be too much until the PDP-11/45 machine came out with 16-bit addressing for both instructions and data. S functions naturally ran as separate processes, but to fit on the PDP-11 we even made the parser into a separate process. Similarly, graphics was done by running the device function as a separate process linked by two-way pipes to the main S process. S put a rather severe strain on the resources of such a small machine, so it was obvious that the PDP-11 was not the ideal S vehicle, either.

Soon we had another opportunity to test our new model of piggyback portability. The UNIX system was ported to the DEC VAX computers, a new generation of 32-bit machines, where the size constraints of the PDP-11 wouldn't chafe so much. Freed from process size constraints, we tried a new implementation, where all of the functions were glued together into one large S process. That allowed the operating system to use demand paging to bring in from disk just the parts of S that were needed for execution. This worked well with S, because it is unlikely that a single S session will use more than a fraction of the S functions. The 32-bit architecture also allowed S to take on much bigger problems, with size limited only by the amount of memory the operating system would allow S to dynamically allocate. By eliminating all of the separately executable functions, disk usage was also much less, and the single S process could easily be relinked when basic S or system library routines changed.

By October, 1979, we had made a concerted effort to move all of our S functions to the UNIX version of S, making it our primary platform.

Changes to S

Perhaps the biggest change made possible by running S on the VAX machines was the ability to distribute it outside of Bell Labs research. Our first outside distribution was in 1980, but by 1981, source versions of S were made widely available. At the time, the Bell System was a regulated monopoly, and as such would often distribute software for a nominal fee for educational use. Source distribution not only allowed others access to S, but provided the statistics research departments a new vehicle for disseminating the results of our research. As we began wider distribution of S, we dropped version numbering in favor of

dates; for example an S release might be identified as “June, 1980”. Major releases were accompanied by completely new books; updates had revised versions of the online documentation.

As S became more widely used, it also gained some new features. One major addition was that of a portable macro processor, implemented by Ed Zayas, based on the macro processor from *Software Tools* (Kernighan and Plaugher, 1976).[†] Macros allowed users to extend the S language by combining existing functions and macros into entities designed specifically to carry out their own tasks. Once a macro was completed, it would carry out the computation and hide the implementation details.

As users began to use S for larger problems, it became apparent that the implicit looping that S used (iterating over all elements of a vector), was unable to handle every problem. It was necessary to add explicit looping constructs to the language. We did so with a general `for` expression:

```
for(index in values) expression
```

where the `index` variable would take on the series of values in turn, each time executing the `expression`.

Since one very common use of `for` loops was to iterate over the rows or columns of a matrix, the `apply` function was introduced. This function takes a matrix and the name of a function and applies that function in turn to each of the rows or columns of the matrix (or to general slices of an array) and does not require an explicit loop. We were able to implement `apply` so that the implicit looping was done with much greater efficiency than could be achieved by a loop in the S language.

Graphics were enhanced considerably at this time. New devices were supported, including pen plotters from Hewlett-Packard and Tektronix. Graphic input was available, to allow interaction with a plot. The user would manipulate the cursor or pen of the plotting device to signal a position. Underlying *graphical parameters* allowed control over both graphical layout and rendering. These parameters originated with the GR-Z graphics library, but extended naturally to S use. Layout parameters allowed specification of the number of plots per page (or screen), the size and shape of the plot region; these parameters were all set by the `par` function. Rendering parameters, controlling such things as line width, character size, and color, were allowed as arguments to any graphics functions, and their values were reset following the call to the graphics function.

Another innovation was the `diary` function, that allowed the user to activate (or deactivate) a recording of the expressions that were typed from the terminal. This recognized the importance of keeping track of what analysis had been performed and of the unique position of the system to provide a machine-readable record. This facility also made S comments more important. Syntactically, the remainder of a line following a “#” was ignored, thus allowing comments, but now, the comments could also be recorded in the diary file, giving a way of making annotations. In the interests of conserving disk space, the diary file was off by default.

At this time, S provided 3 directories[‡] for holding datasets: the *working* directory, the *save* directory, and the *shared* directory. Any datasets created by assignments were stored in the working directory. Although it provided storage that persisted from one S session to another, the thought was that the user would clean up the working directory from time to time, or perhaps even clear it out altogether. In order to preserve critical data (the raw data for an analysis and important computed results), the *save* directory was available. The function `save` was used to record datasets on the *save* directory.[‡] Finally, the *shared* directory was the repository for some datasets that S provided — generally example datasets that were used in the manual.

A new data structure was introduced at this time: the category. It was designed to hold discrete (categorical) data and consisted of a vector of labels and an integer vector that indexed those labels. Computationally, the category acted like a vector of numbers, but it still retained the names of the original levels.

As S became more popular, demand for users to add their own algorithms began to grow, and we also

[†] A macro facility was described in documentation for the earlier version of S, but it was seldom used.

[‡]The name “directory” replaced the earlier usage “database” to emphasize that the S datasets were stored in a UNIX system directory.

[‡]We should have realized that people have a very hard time at cleaning up; the working data tended to be used for everything and the *save* directory was seldom used.

felt the need to expand the basic pool of algorithms available within S. This led to the creation of the *interface language*. This language was designed to describe new S functions, specifying the S data structures for arguments, their default values, simple computations, and the output data structure. It was built from several components. First, there was a “compiler” for the language, that used a *lex* program to turn the input syntax into a set of *m4* macro calls. This compiler was written by Graham McRae, a summer employee of Bell Labs. The macro calls emitted by the compiler were run through *m4* with a large collection of macro definitions, the interface macros, whose job was to produce the code that dealt with argument and data structure processing for S functions, producing a file in the algorithm language (described above). The algorithm language was then processed by *m4*, *ratfor*, and the Fortran compiler, in sequence, in order to produce executable code. As might be expected, the interface language had a number of quirks, each inherited from one of the many tools required to process it.

The macro processor was very popular, yet it was also the source of confusion for many users. While simple macros could be used by rote, a real understanding of macro processing was required for more complex tasks. Because macro processing was carried out as a preliminary to the S parser, users often needed to understand the difference between operations that were carried out at macro time by the macro processor and those that were carried out during execution by the S functions. The macro processor built the expressions that were given to the S parser and evaluator, but in so doing, occasionally needed its own special operations. Thus, the macro processor had separate assignments (the `?DEFINE` macro), conditionals (the `?IFELSE` macro), and looping (the `?FOR` macro). Although the initial “?” and the all-capital-letter names were meant as a cue that these operations were different from ordinary S “functions”, the notion confused many users. While in many ways macros were able to provide a convenient way for users to store sequences of S expressions, there were annoying problems: macros could not be used by the `apply` function, for example.

Another example of the problems with macros was the use of storage. Often, a computation encapsulated in a macro required intermediate datasets, but it was important that those datasets not conflict with ones that the user had created explicitly. A naming convention, that named these temporaries with an initial “T”, was adopted, but even this was insufficient. Suppose one macro called another macro. If the first macro had a temporary named “Tx”, it was important that the second macro not use the same name. Eventually, we developed a macro “?T” that would include a level number in the generated name, to avoid these problems. Thus, “?T(x)” would produce the name “T1x” in the first-level macro and “T2x” in the next level. Finally, some way was needed to clean up all of these temporary datasets at the end, so the function that removed objects was made to return a value so that the last step of a macro could remove temporaries and yet still return a value.

Overall, this first UNIX-based version of S worked well and its availability to a wider audience helped S grow. The next step was to polish it up, provide better documentation, and make it more available.

S TO THE WORLD (1984 BROWN BOOK)

In 1984, the book *S: An Interactive Environment for Data Analysis and Graphics* was published by Wadsworth (and later reviewed by Larntz, 1986). This book, along with an article in the *Communications of the ACM*, (Becker and Chambers, 1984b) described S as of 1984. The following year, Wadsworth published *Extending the S System*, describing the interface language and the process of building new S functions in detail.

Prior to 1984, S was distributed directly from the Computer Information Service at Bell Labs in New Jersey. Beginning in 1984, S source code was licensed, both educationally and commercially, by the AT&T Software Sales organization in Greensboro, North Carolina. S was always distributed in source form, in keeping with the small group of people working directly with it, i.e. Becker and Chambers (and, in 1984, Wilks). Especially after our experiences with the divergent GCOS and UNIX versions of S, we realized the importance of keeping only one copy of the source code. It was the only way that our small group could manage the software. We also knew that a source code distribution would allow people to make their own adjustments as they installed S on a variety of machines and we also knew that there was no way that we could provide binary distributions for even a handful of the hardware/software combinations that S users had. Our focus was on portability within the UNIX system (Becker, 1984). Also, remember that the S

work was being done by researchers; S was not viewed as a commercial venture with support by Bell Laboratories. Thus, sites that used S needed source code to give them a way to administer and maintain the system.

While a number of S features were expanded and cleaned up in the 1984 release, perhaps the more important influences were happening outside of that version. John Chambers moved to a different organization at Bell Laboratories and began work on a system known as QPE, the Quantitative Programming Environment (Chambers, 1987). The QPE work was based on S, but was intended to make it into a more general programming language for applications involving data. Up to now, S was known as a “statistical” language, primarily because of its origins in the statistics research departments, but in reality, most of its strengths were in general data manipulation, graphics, and exploratory data analysis. While millions of people with personal computers began using spreadsheets for understanding their own data, few of them thought they were doing anything like “statistics”. The actual statistical functions in S were primarily for regression. There were, of course, functions to compute probabilities and quantiles for various distributions, but these were left as building blocks, not integrated with statistical applications. QPE was a natural notion to move S away from a perception that it was best used by a statistician.

For some time, Chambers worked on QPE by himself, rewriting the executive in C and generalizing the language. In 1986 QPE was ready to be used by others, and the question was how should QPE and S relate to one another? At the time, S had thousands of users, while QPE was still a research project; S had hundreds of functions, while QPE had few. A marriage was proposed: we would integrate the functions of S with the executive of QPE, gaining the best of both worlds.

NEW S (1988 BLUE BOOK)

The combination of QPE and S produced a language that was called “New S”. Because it was different from the 1984 version of S in many ways, there was a need for a new name.[†] In the Fall of 1984, just after publication of the Brown book, Allan Wilks joined Bell Laboratories and became a full partner in the S effort. The blue book, *The New S Language*, was authored by Becker, Chambers, and Wilks, as was the New S software. Major changes to S occurred during the 1984-1988 period.

Functions and Syntax

The transition to New S was perhaps the most radical that S users experienced. The macro processor was gone, replaced by a new and general way of defining functions in the S language itself. This made functions into truly first-class S objects. Functions were named by assignment statements[‡]:

```
square <- function(x) x^2
```

S functions could now be passed to other functions (like `apply`), and could be the output of other computations. Because S data structures could contain functions, it no longer seemed appropriate to use the name “dataset”; from this point on, we referred to S “objects” (a suggestion of David Lubinsky).

One nice characteristic of S functions is that they are able to handle arbitrary numbers of arguments by a special argument named “...”. Any arguments not otherwise matched are matched by ... and can be passed down to other functions.

Internally, S functions are stored as parse trees, and explicit functions `parse` and `deparse` are provided to produce parse trees from text and to turn those trees back into character form. Parse trees are S objects, and can be manipulated by S functions. For example, the blue book contains a function that takes symbolic derivatives of S expressions.

Many functions were rewritten in the S language when New S was introduced. Functions like `apply` became easy to express in the S language itself, although these implementations were somewhat less efficient than the versions they replaced. In general, as functions are implemented in S, the users have an

[†] Eventually, the new S language was called simply S; the 1984 version became known as Old S and now it is hardly mentioned.

[‡]The initial idea of functions in QPE did not include this form of function as an S object. Instead, the syntax was more akin to the way we defined macros: `FUNCTION square(x) x^2`.

easier time of modifying and understanding them, because debugging is done in the interpreted S language (with the help of interactive browsing).

Unlike the transition from the 1981 to 1984 versions of S, the transition to New S was difficult for users. Their macros would no longer work. To help in this change, we provided a conversion utility, `MAC.to.FUN` that would attempt to convert straightforward macros to S functions.

New S provided a very uniform environment for functions. All S functions were represented by S objects, even those whose implementation was done internally in C code. An explicit notion of interfaces provided the key: there were functions to interface to internal code (`.Internal`), to algorithms in Fortran (`.Fortran`), to algorithms in C (`.C`), and even to old-S routines (`.S`). Along these same lines, the function `unix` provided an interface to programs that run in the UNIX system—the standard output of the process was returned as the value of the function. This was an outgrowth of successful early experiments in interfacing S and GLIM (Chambers and Schilling, 1981); we found it was quite feasible to have a stand-alone program carry out computations for S.

From the user's point of view, the most noticeable syntax change was in the way very simple expressions were interpreted. For example, with old-S, the expression

```
foo
```

would print the dataset named “foo” if one existed or else execute the function “foo” with no arguments. With new S, this expression would always print the object named “foo”, even if it happened to be a function. To execute the function required parentheses:

```
foo()
```

In old-S, it had been possible to omit parentheses from even complicated expressions:

```
foo a,b
```

was once equivalent to

```
foo(a,b)
```

but it is now a syntax error.

Some functions familiar to old-S users had their names changed or were subsumed by new functions. These changes were described in the detailed documentation of the blue book under the name `Deprecated`. In some instances, the operations that the functions carried out were no longer needed; in others the operation changed enough that it was deemed reasonable to change the name to force the user to realize that something had changed. In other cases, fat was trimmed. For example, there was no need for the Fortran-style “**” operator for superscripts when “^” was available and more suggestive.

Various functions were introduced in order to help in producing and debugging S functions. Most important was the `browser`, that allowed the user interactively to examine S data structures. It became an invaluable tool for understanding the state of a computation and was often sprinkled liberally throughout code under development. The function `trace` allowed selected functions to be monitored, with user-defined actions when the functions were executed. There was also an option that specified the functions to be called when S encountered an error or interrupt. Implementation of `trace` is particularly elegant; a traced function is modified to do the user actions and is placed in the session frame (retained for the duration of an S session) where it will be earlier on the search path when the S executive looks for it. Both the `browser` and `trace` functions are written entirely in S.

Data and Data Management

New S regularized the treatment of directories, by having a list of them, rather than the fixed sequence of working, save, and shared directories of Old S. The functions `get` and `assign` were extended to allow retrieval and storage of S objects on any of the attached directories. An S object, `.Search.list` was used to record the names of the directories on the search list.

A major change in data structures was in the separation of the notions of hierarchical structures and attributes of vectors. A vector of mode “list” represented a vector of other S objects. Such vectors could be subscripted and manipulated like other, atomic mode, vectors. Now, any vectors could have attributes:

names to name the elements of the vector, `dim` to give dimensioning information for treating a vector as an array, etc. The notion of a vector structure was recognized as simply a vector with other attributes. Vectors were also allowed to have zero length, a convenience that regularizes some computations.[†]

A first hint at the object-oriented features of S came about with the 1988 release. The function `print` was made to recognize an attribute named `class`; if this attribute was present with a value, say, `xxx` and if there was a function named `print.xxx`, that function would be called to handle the printing.

Another feature of New S was the notion that S objects could be used to control the operation of S itself. In particular, there were four such objects: `.Program` expressed the basic parse/eval/print loop in S itself and could be replaced for special purposes; `.Search.list` gave a character vector describing the directories to be searched for S objects, `.Options` was a list controlling optional features such as the S prompt string, default sizes and limits for execution parameters, etc. `.Random.seed` recorded the current state of the random number generator. In addition, various functions named beginning with `sys.` contained information about the state of the S executive and its current evaluation. This allowed the `browser` to be applied to evaluation frames of running functions.

The diary file of old-S was replaced by an audit file. The audit file furnishes both an audit trail by which the entire set of expressions given to S can be recreated, and it also is the basis for a history function that allows re-execution of expressions and for an externally implemented auditing facility (Becker and Chambers, 1988). There have been occasional complaints about the presence of the audit file, about its size, and all of the information therein describing which S objects were read and written by each expression, but generally it is recognized that the file contains important information about the path of the analysis. The external audit file processor can be used to recreate analyses, to find the genesis of objects, or to decide what should be executed following changes to data objects.

S data directories inherited a feature from commercial databases: commitment and backout upon error. As in the past, each S expression typed by the user is evaluated as a whole and the system starts afresh with each new expression. With the database backout facilities, S objects that are created during an expression are not written to the database until the expression successfully completes. This provides several advantages. First, because objects are kept in memory during expression execution, multiple use or assignments to one object do not require multiple disk access. Second, if an error occurs, the old data that was present before the error is preserved.

Another feature is that of in-memory databases, called *frames*, which hold name/value pairs in memory. A new frame is opened for each function that executes, providing a place to store temporary data until that function exits. When a function exits, its frame and all that was stored there is deallocated. There is also a frame associated with each top-level expression as well as one that lasts as long as the entire S session.

Implementation

New S was implemented primarily in C, as was the QPE executive. Internal routines, too, such as arithmetic and subscripting, were generally rewritten in C. The rewrite was necessary to support double-precision arithmetic. Code to support complex numbers was added. Many routines that in old-S had been implemented in the interface language were re-implemented in the S language, although graphics functions were largely left in the interface language and invoked through the `.s` function. Most of the Fortran that was present in S was rewritten, although basic numerical algorithms, such as those from LINPACK (Dongerra, et. al, 1979) or EISPACK (Smith et. al, 1976) were left in Fortran.

The advent of New S also brought about a basic change in underlying algorithms and data. Now, everything is done in double precision, although the user is presented with a unified view of a data mode named `numeric`, which includes integer and floating point values. For special purposes (most notably for passing values down to C or Fortran algorithms), it is possible to specify the representation of numbers (integer, single, or double precision), but this is normally far from the user's mind.

S uses lazy evaluation. This means that an expression is not evaluated until S actually needs the

[†]The APL language is especially good at treating boundary cases such as zero-length vectors and we found that proper treatment of boundary cases often makes writing S functions easier.

value. For example, when a function is called, the values of its actual arguments are not computed until, somewhere in the function body, an expression uses the value.

Along with the ability to call C or Fortran algorithms, on some machines S is capable of reading object files (such as those produced by the C or Fortran compilers) and loading them directly into memory while S is executing. This means that new algorithms can be incorporated into S on the fly, with recompilation of only the changed program, and without relinking the S system itself. Dynamic loading like this is machine dependent, requiring a knowledge of object file formats and relocation procedures, so it only works on a select group of computers. It is always possible to link external subroutines with S by using the UNIX system loader.

Graphics functions in New S are very similar to those in earlier versions (for the most part, they are still implemented by old-S interface routines), however they have been regularized in many ways. A single function `plot.xy` is used to determine the x - and y -coordinates for a scatterplot. Utilizing the parse-tree for the plotting expression, S can provide meaningful labels for the plot axes, based on the expressions used to produce the x - and y -coordinates. Graphics devices are implemented by C routines that fill a data structure with pointers to primitive routines: draw a line, plot a string, clear the screen, etc.

As pen plotters and graphics terminals became less prevalent, new graphics device functions were brought into S. In particular, the most commonly used devices are now one for the X window system (`x11`), and for Adobe's PostScript™ language (`postscript`).

A novel portion of the blue book describes the semantics of S execution in the S language itself. We found that it was often easier to implement new ideas using the S language in order to try them out, and only after deciding that they were worthwhile would we recode them more efficiently in C.

The advent of New S and the blue book gave S a solid foundation as a computational language. The time was ripe for a major thrust into statistics.

STATISTICAL SOFTWARE IN S (1991 WHITE BOOK)

Because of its relatively recent publication, it is harder to reflect on the book, *Statistical Models in S* (Chambers and Hastie, 1992). Certainly, a number of important enhancements to the core of S were made in that time, although, for the most part, the Statistical Models work was a very large implementation of software primarily in the S language itself. Notice that statistics is not properly part of S itself; S is a computational language and environment for data analysis and graphics. The Statistical Models work is implemented in S and is not S itself.

The models work was done by ten authors, coordinated by John Chambers and Trevor Hastie. In some ways, this was a departure from the small-team model of S development. On the other hand, the work was based upon a common core and was carried out by researchers with intense personal interests in their particular portions. In addition, formal code reviews were held for all of the models software, helping to ensure that it was well-implemented and fit in with the rest of S.

The most fundamental contribution of the Models work is probably the modeling language, and it was achieved with a minimal change to the S language — the addition of the “~” operator to create formula objects (Chambers, 1993a). The modeling language provided a way of unifying a wide range of statistical techniques, including linear models, analysis of variance, generalized linear models, generalized additive models, local regression models, tree-based models, and non-linear models.

The major change to S itself was the strong reliance on classes and methods. Generic functions were introduced; they call the function `UseMethod`, which tells the S executive to look for the `class` attribute of the generic function's argument and to dispatch the appropriate *method* function, the one whose name is made by concatenating the generic function name with the class name. By installing the object-oriented methods in this form much of S could remain as it was, with generics and methods being installed where they were needed. A form of inheritance was supplied by allowing the `class` attribute to be a character vector rather than just a single character string. If no method can be found to match the first class, S searches for a method that matches one of the later classes. Chambers (1993a) describes the ideas in much greater detail.

Another major addition to the language concerns data: the *data frame* class of objects provides a

matrix-like data structure for variables that are not necessarily of the same mode (matrices must have a single mode). This enables a variety of observations to be recorded on the subjects: for example, data on medical patients could contain, for each patient, the name (a character string), sex (a factor), age group (an ordered factor), height and weight (numeric).

Another interesting inversion happened with S data directories; we once again refer to them as databases. This is because the notion of a UNIX directory became too confining. Now, there are several sorts of S databases: 1) UNIX system directories; 2) S objects that are attached as part of the search list; 3) a compiled portion of S memory; 4) a user-defined database. Because of this, we now think of methods that operate on S databases. There are methods to attach databases to the search list, to read and write objects, to remove them, to generate a list of objects in a database, and to detach them from the search list. The search list itself outgrew its early character-string implementation; it is now a true list.

THE FUTURE

This paper has described S up to the present, but will soon be out of date, as S continues to evolve. What is likely to happen?

We intend to address efficiency issues. New ideas in internal data structures may allow significant savings at run time by allowing earlier binding of operations to data. A better memory management model may use reference counts or other techniques to avoid extraneous copies of data, improving memory usage.

We hope to eliminate another of our side effects: graphics. Instead of calling graphics functions to produce output, graphics functions would instead produce a graphics object. The (automatically called) print method for a graphics object would plot it. This concept would allow plots to be saved, passed to other functions, modified, composed with one another, etc.

One general area that we would like to integrate into S is the area of high-interaction graphics with linked windows. Difficult research issues remain with this, including the problem of determining when a set of points on one plot corresponds to a set on another plot. There seems to be a conflict with this application and the general independence of S expressions.

In the long run, for portability reasons, we would like to eliminate Fortran from S, relying on C or perhaps C++ as the implementation language. One less language would make it easier (and less expensive) for people to compile S. But without Fortran, how do we make use of broadly available numerical algorithms written in Fortran? The *f2c* tool (Feldman, et. al, 1990) holds some promise here, since it has been used successfully to install S on machines without Fortran.

When we built S, we intentionally did not provide a user interface with added features such as command editing or graphical input. We preferred instead to build a simple system that could support a variety of specialized user interfaces. The X window system (Scheifler and Gettys, 1986) may now provide a portable way to implement graphical user interfaces. Perhaps the time has come for serious experiments along these lines.

With the Statistical Models work, we did not press the object-oriented methodology to its logical conclusion. For example, we did not convert current time series, matrices, and categories into objects with `class` attributes, although we plan to do so in the future. This seems obvious in retrospect: many of the special cases in arithmetic computations and subscripting are caused by the need to deal with time-series, matrices, and categories. By using classes, we could handle these special cases in separate functions. Factors are one step in this direction; a factor is essentially a category with a class attribute and a variety of methods to provide appropriate operations on factors.

The future of S is increasingly influenced by people in the academic and business communities. The commercial success of the S-PLUS system from StatSci (1993) has brought the S language to thousands of people, including users of personal computers (reviewed by Therneau, 1990). Software related to S is distributed electronically via the Mike Meyer's Statlib archive, statlib@lib.stat.cmu.edu, and an electronic user group, S-news@stat.toronto.edu, actively discusses S issues. Härdle (1990) has written a book describing smoothing with an implementation in S, and Venables and Ripley (1994) have written a statistics text based on S.

CONCLUSIONS

Early on, we hinted that a study of the history of S could be helpful in understanding the software development process. Now, the time has come to try to draw some of these conclusions.

Much of the work in getting S to its current state followed an evolutionary process. Couldn't we have gotten there more quickly, more easily, and with less user pain by designing the language better in the first place? I think the answer is "No.", we could not have avoided some of the excursions we made.

S in 1976 was remarkably like S of 1993 — many of the basic ideas were in place. However, even if we had a blueprint for the full version of S in 1976, it would have been overwhelming, both to us as implementors and to the computing hardware of the day. Aiming too high may have caused us to give up in advance. It was useful to keep the project at a level that could be sustained by two or three people, because in that way, S evolved in a way that was aesthetically pleasing to all of us. S was not designed by committee. There is nothing like individual motivation — wanting to produce something for your personal use — to get it done right.

S was also the result of an evolution with plenty of feedback from our own and others' real use. The iteration of design, use, and feedback was important. It seemed that each time we completed a new revision to S and had a little while to rest, we then felt new energy that enabled us to think of forging ahead again.

Each version of S had limited goals. When we started, our initial idea was to produce an interactive version of the SCS library. Because it was limited, the goal also seemed attainable. As Brian Kernighan has stated "I never start on a project that I think will require more than a month." Of course, the projects do grow to require more time than that, but especially for research, it is good to feel like the time horizon is short and that there will be time for feedback and changes.

Another important concept is that we always emphasized the way the user would see the system. Our initial efforts gave only a vague thought toward machine efficiency and concentrated instead on human efficiency. It is a mistake to let hardware dominate the design of software; yes, it does have to be implemented and actually work, but it need not be totally controlled by current hardware. In the years that S has been around, machine speed has increased by several orders of magnitude, disk space is far more available, and costs have come down dramatically. It would have been unfortunate if we had allowed the performance of now-ancient hardware to affect the overall design of S. A good idea is to first make it work and then make it fast.

Efficiency in S has always been an elusive goal and has often been the part of the code that changes most from one version of S to another. Work on efficiency must be continuous because, as the system gets more efficient and runs on more powerful computers, our tendency and that of users is to implement more and more in the S language itself, causing efficiency to be a constant battle. We have also found more than once that it is not obvious which changes will improve efficiency. Our best advice is to implement and then measure. Chambers (1992) discusses some recent attempts to measure performance and Becker, Chambers, and Wilks (1993) describes a tool we devised to monitor memory usage.

Along these same lines, we tried from the beginning to make S as broad as possible, by eliminating constraints. S was built to use whatever cpu or memory resources the operating system would allow; nothing in S itself restricts the problem size. Similarly, S has no restrictions in its language — everything is an expression, built up from combinations of other expressions. Data structures are also general combinations of vectors, with no restrictions on how they can be put together.

A general principle is that the language should have minimal side effects. In S, the major side effects are assignments (that create persistent objects), printing, and graphics. Minimal side effects provide an important property: each expression is independent of the expressions that preceded it, aside from the data objects that it uses.

S also succeeded by its basic building-block approach. Within the broad context of the S language, we furnished a number of functions to carry out primitive computations, relying on the user to put them together in a way most appropriate to the problem at hand. Although this does not make for a system that can be used without thought, it does provide a powerful tool for those who know how to get what they want.

We have always *used* S ourselves. Although that may seem like a trite observation, it is very

important to the way S has evolved. Often, the impetus for change in S is the result of using it to address a new problem. Without new problems to solve, S wouldn't grow. If we hadn't considered S our primary research tool, we would probably not have kept it up-to-date with the latest algorithms and statistical methods. S has always had new methodology available quickly, at least in part because it was designed to be extended by *users* as well as its originators.

S has done well, too, because of the synergy that comes from integrating tools. When developing an algorithm, the S user can use the data management and graphical routines to generate test data, to plot results, and generally to support the activity.

One of the characteristics that made S into what it has become is that each part of S is "owned" by someone. There has been no "market driven" component to S — each function is there because one of us wanted it there for our own use or because someone else convinced us that a particular function was necessary. Decisions about S have generally been by consensus (a necessity with two authors and highly desirable with three). Thus, S has the imprint of a few individuals who cared deeply and is not the bland work of a committee.

Because S came from a research environment, we have felt free to admit our mistakes and to fix them rather than propagate them forever. There have been a few painful conversions, most notably the transition from old-S to new S, but the alternative is to carry the baggage of old implementations along forever. It is better to feel the momentary pain of a conversion than to be burdened forever by older, less informed decisions. Perhaps, had S been a commercial venture, we would not have had the luxury of doing it again and again until we got it right.

A tools approach was certainly beneficial to the development of S. The software available within the UNIX system allowed us to build S quickly and portably. For example, the interface language compiler was a fairly easy project because of *lex*, *m4*, and *ratfor*. On the other hand, a tool built from other tools tends to inherit the idiosyncrasies of each of those tools. It was difficult to describe to users of the interface language just why there were so many reserved words in the language (see Appendix A of Becker and Chambers, 1985).

One way of assessing the evolution of a language is to watch for concepts that have been dropped over time. These notions are important clues to what was done wrong and later improved. In S, the notion of a dataset was dropped in favor of the idea of an object. This recognized the much more general role that S objects had in the language, holding not only data but functions and expressions. Similarly, our idea of a vector structure was approximately right, but was clarified by two more precise concepts: the (vector) list of S objects and the notion that any object could have attributes.

Steve Johnson once said that programming languages should never incorporate an untried feature. While we didn't slavishly follow that advice, most of the features present in S were influenced by some other languages. We often found a good concept in an existing system and adapted it to our purposes.

As an example, the basic interactive operation of S, the parse/eval/print loop, was a well-explored concept, occurring in APL, Troll, LISP, and PPL, among others. From APL we borrowed the concept of the multi-way array (although we did not make it our basic data structure), and the overall consistency of operations. The notion of a typeless language (with no declarations) was also present in APL and PPL. As the PPL manual (Taft, 1972) stated, "*type* is an attribute, not of *variables*, but rather of *values*."

We wanted the basic syntax of the language to be familiar, so we used standard arithmetic expressions such as those found in Fortran or C. We wanted to emphasize that the assignment operation was different from an equality test so, like Algol, we used a special operator. The specific "<-" operator came from PPL (and was a single character on some old terminals).

The basic notion that everything in S is an expression fits in very nicely with the formal grammar that describes S — a language statement is just an expression. Certainly C and Ratfor contributed the notion that a braced expression was an expression, thus allowing S to get by without need for special syntax like BEGIN/END or IF/FI or CASE/ESAC.

Some of the more innovative ideas in data structuring came from LISP: the lambda calculus form of function declarations, the storage of functions as objects in the language, the notion of functions as first-class objects, property lists attached to data.

Some pieces of syntax come from languages that are pretty far removed from the applications we foresaw with S. In particular, the IBM 360 Assembler and Job Control (JCL) languages helped to contribute the notions of arguments given by either positional or keyword notation, along with defaults for the arguments. The common thread between S and these languages was the need for functions that had many arguments, although a user would only specify some of them for any particular application.

More recently, we borrowed ideas about classes and inheritance from languages like C++, CLOS, and Smalltalk.

The Troll system contributed in many ways: the need for a time-series data type and special operations on time-series; interactive graphics for data analysis; persistent storage of datasets; the power of functions as a basic way of expressing computations.

The UNIX operating system also contributed a few ideas (as well as an excellent development platform): the use of a leading “!” to indicate that a line should be executed by the operating system and the notion of having simple, well defined tools as the basis for computing. The UNIX shell also contributed to the notion of continuation lines. Like S, it knows when an expression is incomplete and responds with a special prompt to indicate this.

The origins of a few concepts are more mysterious. Perhaps they are unique to S in their precise implementation, but they were probably the result of some external influence. In particular, the syntax for the `for` loop: `for(i in x)` that allows looping over the elements of various data structures and doesn’t force the creation of an integer subscript. Other programming languages have functions that can cope with arbitrary numbers of arguments, but the use of `...` in S has its own unique qualities.

One important non-technical concept was very important to the evolution of S. Almost from the beginning, S was made readily available in source form to AT&T organizations, to educational institutions, and to outside companies. This distribution of source allowed an S user community to grow and also gave the research community a vehicle for distributing software. At present there is an active S mailing list, consisting of hundreds of users around the world, asking questions, providing software, and contributing to the evolution of S. We hope that evolution continues for a long time.

ACKNOWLEDGEMENTS

I would like to thank John Chambers and Allan Wilks for comments on this manuscript. Over the years many people have been contributed to S in one way or another. Some of them have been mentioned here, but there are many others whose often considerable efforts could not be mentioned for lack of space. Among them are Doug Bates, Ron Baxter, Ray Brownrigg, Linda Clark, Bill Cleveland, Dick De Veaux, Bill Dunlap, Guy Fayet, Nick Fisher, Anne Freeny, Colin Goodall, Rich Heiberger, Ross Ihaka, Jean Louis Charpentreau, John Macdonald, Doug Martin, Bob McGill, Allen McIntosh, Mike Meyer, Wayne Oldford, Daryl Pregibon, Matt Schiltz, Del Scott, Ritei Shibata, Bill Shugard, Ming Shyu, Masaai Sibuya, Kishore Singhal, Terry Therneau, Rob Tibshirani, Luke Tierney, and Alan Zaslavsky. Thank you. Unfortunately, I’m sure that other names that should appear here have been omitted. My apologies.

REFERENCES

- Baker, Brenda S. (1977), “An Algorithm for Structuring Flowgraphs”, *J. Assoc. for Comp. Machinery*, Vol 24, No. 1, pp. 98-120.
- Becker, Richard A. (1978), “Portable Graphical Software for Data Analysis”, *Proc. Computer Science and Statistics: 11th Annual Symposium on the Interface*, pp 92-95.
- Becker, Richard A. (1984), “Experiences with a Large Mixed-Language System Running Under the UNIX Operating System”, *Proc. USENIX Conference*.
- Becker, Richard A. and John M. Chambers (1976), “On Structure and Portability in Graphics for Data

- Analysis'', *Proc. 9th Interface Symp. Computer Science and Statistics*.
- Becker, Richard A. and John M. Chambers (1977), "GR-Z: A System of Graphical Subroutines for Data Analysis'', *Proc. 10th Interface Symp. on Statistics and Computing*, 409-415.
- Becker, Richard A. and John M. Chambers (1978), '*S*: A Language and System for Data Analysis, Bell Laboratories, September, 1978. (described "Version 2" of S).
- Becker, Richard A. and John M. Chambers (1981), *S: A Language and System for Data Analysis*, Bell Laboratories, January, 1981.
- Becker, Richard A. and John M. Chambers (1984), *S: An Interactive Environment for Data Analysis and Graphics*, Wadsworth Advanced Books Program, Belmont CA.
- Becker, Richard A. and John M. Chambers (1984b), "Design of the S System for Data Analysis'', *Communications of the ACM*, Vol. 27, No. 5, pp. 486-495, May 1984.
- Becker, Richard A. and John M. Chambers (1985), *Extending the S System*, Wadsworth Advanced Books Program, Belmont CA.
- Becker, Richard A. and John M. Chambers (1988), "Auditing of Data Analyses'', *SIAM J. Sci. and Stat. Comp.*, pp. 747-760, Vol 9, No 4.
- Becker, Richard A. and John M. Chambers (1989), "Statistical Computing Environments: Past, Present and Future'', *Proc. Am. Stat. Assn. Sesquicentennial Invited Paper Sessions*, pp. 245-258.
- Becker, Richard A., John M. Chambers and Allan R. Wilks (1988), *The New S Language*, Chapman and Hall, New York.
- Becker, Richard A., John M. Chambers and Allan R. Wilks (1993), "Dynamic Graphics as a Diagnostic Monitor for S'', AT&T Bell Laboratories Statistics Research Report.
- Chambers, John M. (1974), "Exploratory Data Base Management Programs'', Bell Laboratories Internal Memorandum, July 15, 1974.
- Chambers, John M. (1975), "Structured Computational Graphics for Data Analysis'', Intl. Stat. Inst. Invited Paper, 1-9 IX 1975, Warszawa.
- Chambers, John M. (1977), *Computational Methods for Data Analysis*, Wiley, New York.
- Chambers, John M. (1978), "The Impact of General Data Structures on Statistical Computing'', Bell Laboratories Internal Memorandum, May 20, 1978.
- Chambers, John M. (1980), "Statistical Computing: History and Trends'', *The American Statistician*, Vol 34, pp 238-243.
- Chambers, John M. (1987), "Interfaces for a Quantitative Programming Environment'', *Comp. Sci. and Stat, 19th Symp. on the Interface*, pp. 280-286.
- Chambers, John M. (1992), "Testing Software for (and with) Data Analysis'', AT&T Bell Laboratories Statistics Research Report.
- Chambers, John M. (1993a), "Classes and Methods in S. I: Recent Developments'', *Computational Statis-*

tics, to appear.

Chambers, John M. (1993b), “Classes and Methods in S. II: Future Directions”, *Computational Statistics*, to appear.

Chambers, John M. and Trevor Hastie, eds. (1992), *Statistical Models in S*, Chapman and Hall, New York.

Chambers, John M. and Judith M. Schilling (1981), “S and GLIM: An Experiment in Interfacing Statistical Systems”, Bell Laboratories Internal Memorandum, January 5, 1981.

Dongerra, J. J., J. R. Bunch, C. B. Moler, and G. W. Stewart (1979), *LINPACK Users Guide*, Society for Industrial and Applied Mathematics, Philadelphia.

Feldman, S. I., David M. Gay, Mark W. Maimone, N. L. Schryer (1990), “A Fortran-to-C Converter, Computing Science Technical Report No. 149, AT&T Bell Laboratories.

Feldman, S. I. and P. J. Weinberger (1978), “A Portable Fortran 77 Compiler”, Bell Laboratories Technical Memorandum.

Härdle, Wolfgang (1990) *Smoothing Techniques With Implementation in S*, Springer Verlag, New York.

Iverson, K. E. (1991), “A Personal View of APL”, *IBM Systems Journal*, Vol 30, No. 4, pp 582-593. (The entire issue of this journal is devoted to APL.)

Johnson, S. C. and M. E. Lesk (1978), “Language Development Tools”, *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, pp. 2155-2175, July-August 1978.

Johnson, S. C. and D. M. Ritchie (1978), “Portability of C Programs and the UNIX System”, *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, pp. 2021-2048, July-August 1978.

Kernighan, Brian W. (1975), “RATFOR—A Preprocessor for a Rational Fortran”, *Software—Practice and Experience*, Vol. 5, pp 395-406.

Kernighan, Brian W. and P. J. Plauger (1976), *Software Tools*, Addison Wesley.

Kernighan, Brian W. and Dennis M. Ritchie (1977), “The M4 Macro Processor”, Bell Laboratories Technical Memorandum, April, 1977.

Larntz, Kinley (1986), “Review of S: An Interactive Environment for Data Analysis and Graphics”, *J. of the American Statistical Association*, Vol. 81, pp. 251-252.

National Bureau of Economic Research (NBER, 1974), *Troll Reference Manual*, Installment 5, August 1974.

Ritchie, D. M. and K. Thompson (1970), “QED Text Editor”, Bell Laboratories Technical Memorandum.

Ritchie, D. M. and K. Thompson (1978), “The UNIX Time-Sharing System”, *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, pp. 1905-1929, July-August 1978.

Ritchie, D. M., S. C. Johnson, M. E. Lesk, and B. W. Kernighan (1978), “The C Programming Language”, *The Bell System Technical Journal*, Vol. 57, No. 6, Part 2, pp. 1991-2019, July-August 1978.

- Ryder, B. G. (1974), “The PFORT Verifier”, *Software — Practice and Experience*, Vol. 4, pp. 359-377.
- Scheifler, R. W. and J. Gettys (1986), “The X Window System”, *ACM Transactions on Graphics*, Vol. 5, No. 2, pp. 79-109.
- Smith, B. T., J. M. Boyle, J. J. Dongerra, B. S. Garbow, Y. Ikebe, V. C. Klema, and C. B. Moler (1976), *Matrix Eigensystem Routines—EISPACK Guide*, Second Edition, Lecture Notes in Computer Science 6, Springer-Verlag, Berlin.
- Standish, T. A. (1969), “Some Features of PPL — A Polymorphic Programming Language”, *Proc. of Extensible Language Symposium*, Christensen and Shaw, eds., *SIGPLAN Notices*, Vol 4, Aug. 1969.
- StatSci (1993), *S-PLUS Programmer’s Manual, Version 3.1*, Statistical Sciences, Seattle.
- Taft, Edward A. (1972), *PPL User’s Manual* Center for Research in Computing Technology, Harvard University, September, 1972.
- Therneau, Terry M. (1990), “Review of S-PLUS”, *The American Statistician*, Vol. 44, No. 3, pp. 239-241.
- Tukey, John W. (1971), *Exploratory Data Analysis*, Limited Preliminary Edition. Published by Addison-Wesley in 1977.
- Venables, W. N., and B. D. Ripley (1994), *Statistics with S*, Springer-Verlag.

