# Biostat 776: Other Topics in R

**Roger Peng**

**November 19, 2003**

## A Hodge-Podge of Stuff

- S4 Classes and Methods
- Lexical Scoping and Statistical Computing

## Classes and Methods

- A system for doing object oriented programming
- R is rare because it is both interactive *and* has a system for object orientation.
  - Other languages which support OOP: C++, Java, Lisp, Python, Perl
- In R, much of the code for supporting S4 classes/methods is written by John Chambers himself.
  - Chambers, J. (1998) *Programming with Data: A Guide to the S Language*, Springer, NY.

**R has two “styles” of classes and methods**

- S3 classes/methods
  - Included with version 3 of the S language.
  - Informal, a little kludgey
  - Sometimes called “old-style” classes/methods
- S4 classes/methods
  - more formal and rigorous
  - Included with S-PLUS 6, R $\geq$ 1.4.0
  - Also called “new-style” classes/methods

## Two worlds

- For now (and the forseeable future), S3 classes/methods and S4 classes/methods are *separate* systems.
- 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”.
- Oh well. . . .

## Object Oriented Programming in R

- A *class* is a description of an thing. A class can be defined using `setClass()`.
- An *object* is an instance of a class. Objects can be created using `new()`.
- A *generic function* is an R function which dispatches methods. A generic function typically encapsulates a "generic" concept.
  - e.g. `plot`, `mean`, `logLik`, `residuals`, `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.

Classes

Generic Functions

Methods

**Things to look up**

- The help files for the 'methods' package are extensive – do read them.
- Check out:
  - `?setClass`, `?setMethod`, `?setGeneric`, `?Methods`
- Some of it gets technical, but don't worry about that for now.

## Classes

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("asdf")
[1] "character"
>
```

## Classes (cont'd)

```
> x <- rnorm(100)
> y <- x + rnorm(100)
> fit <- lm(y ~ x)
> class(fit)
[1] "lm"
>
```

**Generics/Methods in R**

- S4 and S3 style generic functions look different but conceptually, they are the same (they play the same role).
- When you program you can
  1. Write new methods for an existing generic function
  2. Create your own generics and associated methods

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

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

**An S4 generic function (from the ‘methods’ package)**

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

function (object)
standardGeneric("show")
<environment: 0x8d7cdc8>
Methods may be defined for arguments: object

>
```

**The generic/method mechanism**

The first argument of a generic function is an *object* of a particular class (there may be a bunch of 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.

**Example 1**

```
> x <- rnorm(100)
> mean(x)
[1] -0.06846675
```

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

```
> mean.default
function (x, trim = 0, na.rm = FALSE, ...)
{
    ## ... Skip 18 lines ...
    if (is.integer(x))
        sum(as.numeric(x))/n
    else sum(x)/n
}
<environment: namespace:base>
>
```

**Example 2**

```
> df <- data.frame(x = rnorm(100), y = rnorm(100, 1))
> mean(df)
          x           y
0.002565053 0.972148319
```

1. The class of `df` is "data.frame".
2. There *is* a method for "data.frame" objects!
3. We call `mean.data.frame` on `df`.

```
> mean.data.frame
function (x, ...)
sapply(x, mean, ...)
<environment: namespace:base>
>
```

NOTE: Generally, you should *not* call methods directly. Rather, use the generic function and let the method be dispatched automatically.

**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`

You could write a new method for an existing class, but more likely you'll want to write a method for a class that *you* create.

**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
  - e.g. a fitted point process model, mixed-effects models
- To abstract 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).

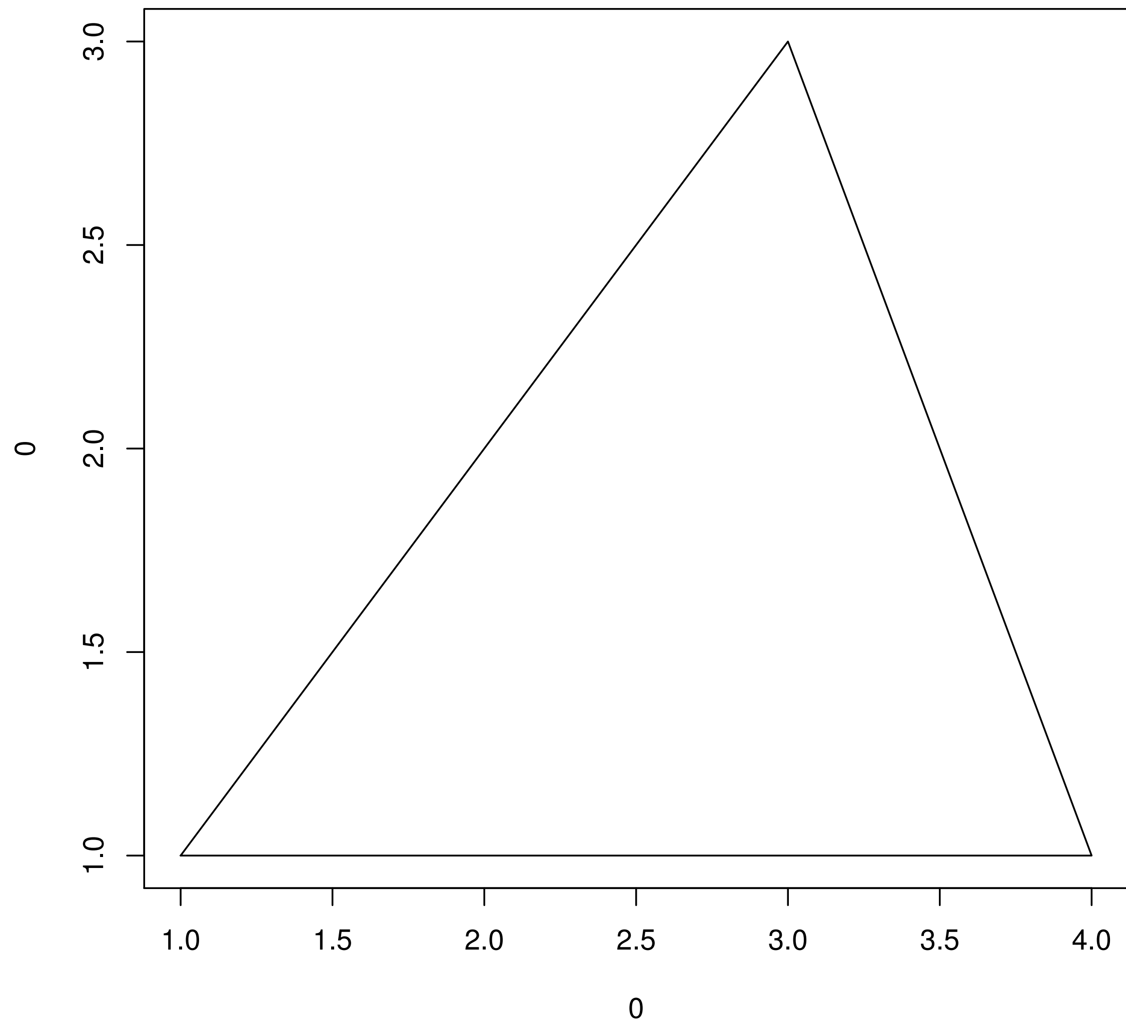
**Example: A Sparse Matrix**

```
# Sparse general matrix in triplet format
setClass("tripletMatrix",
         representation(i = "integer",
                        j = "integer",
                        x = "numeric",
                        Dim = "integer"))
setMethod("crossprod",
          signature(x = "tripletMatrix",
                    y = "tripletMatrix"),
                    ## code for cross products
          )
```

**Example: A polygon class**

```
setClass("polygon",
         representation(x = "numeric",
                        y = "numeric"))
setMethod("plot", "polygon",
          function(x, y, ...) {
              xlim <- range(x@x)
              ylim <- range(x@y)
              plot(0, 0, type = "n", xlim = xlim,
                   ylim = ylim , ...)
              xp <- c(x@x, x@x[1])
              yp <- c(x@y, x@y[1])
              lines(xp, yp)
          })
```

```
> setClass("polygon", [ ...OMITTED... ]
[1] "polygon"
>
> setMethod("plot", "polygon", [ ...OMITTED... ]
Creating a new generic function for "plot" in ".GlobalEnv"
[1] "plot"
> p <- new("polygon", x = c(1,2,3,4), y = c(1,2,3,1))
> plot(p)
```



3.0
2.5
2.0
1.5
1.0
0
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0

**Where to look, places to start**

- The best way to learn this stuff is to look at examples.
- Sadly, there aren't too many examples on CRAN which use S4 classes/methods.
- My suggestions:
  - 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 (poorly written), flexmix, its, lme4, orientlib, pixmap
  - Version 1.8.0 of the base R installation comes with a package 'mle' which use S4 classes/methods. It's a small package and is a good place to start.

Pause

**Lexical Scoping and Statistical Computing**

1. What is lexical scoping?
2. How can it help me with statistical computing?
3. Examples

**Scoping Rules**

- Rules for assigning values to *free variables*
- A free variable is a variable that is
  - Not a formal argument to a function
  - Not assigned inside a function (i.e. a local variable)

**Example 1**

```
f <- function(x) {
    a <- 3
    x + a
}
```

- `x` is a formal argument
- `a` is a local variable

```
> f(2)
????
```

**Example 2**

```
g <- function(x) {
    a <- 3
    x + a + y
}
```

- `x` is a formal argument
- `a` is a local variable
- `y` is a *free variable*

```
> g(2)
????
```

**Dynamic Scoping (old school)**

- Free variables are looked up in the environment in which the function was *called* (function call stack)
- In R, this is called the *parent frame*
  - can be accessed via `parent.frame()`
- e.g. If you call a function from the command line, the parent frame is the global workspace.

**Lexical Scoping (modern)**

- Free variables are looked up in the environment in which the function was *defined*.
- In R, this is called the *parent environment*
  - can be accessed via `parent.env()`
- In other words, free variables are looked up according to the *textual* description of the function

**Note**: If a function is defined in the global workspace and is also called from the global workspace, then the parent environment and the parent frame are the same.

## Languages that Support Lexical Scoping

- Scheme
- R (much like Scheme)
- Common Lisp
- Perl
- Python

**Example 2 (cont'd)**

```
> rm(list = ls(all = TRUE))  ## Clear workspace
> g <- function(x) {
+     a <- 3
+     x + a + y
+ }
> g(2)
Error in g(2) : Object "y" not found
> y <- 3
> g(2)
[1] 8
>
```

Here, the function `g()` is defined in the *global workspace*. Therefore, the parent environment is the global workspace.

**Example 2a**

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

## Moving along

Can a function have something *other* than the global workspace as the parent environment? Yes!

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

`make.pow` returns a *function* which takes a single argument `x`. The function returned by `make.pow` has a free variable, `n`.

## Example 3

```
> cube <- make.pow(3)
> cube
function(x) {
        x^n
    }
<environment: 0x8f39ce8>
> cube(4)  ## No error here!
[1] 64
```

**Example 3 (cont'd)**

- The function `cube` was defined inside the `make.pow` function. Therefore, the parent environment of `cube` is the body of the `make.pow` function, *not the global workspace.*
- Note that when the `cube` function is printed, the parent environment is printed at the bottom of the function body:

  ```
  <environment:  0x8f39ce8>
  ```
- If a function is defined somewhere besides the global workspace, the parent environment is printed along with the function body.

## Consequences of Lexical Scoping

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

**Why should I care?**

- Lexical scoping provides a convienent way to create *function closures*
- Can be used to maintain local state
- Extremely useful for plug 'n' play optimization routines

**Application: Optimization**

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

## Example: Maximum Likelihood for a Normal model

```
negloglik <- function(p, data) {
    mu <- p[1]
    sigma <- p[2]
    a <- -0.5 * length(data) * log(2 * pi * sigma^2)
    b <- -0.5 * sum((data - mu)^2) / (sigma^2)
    -(a + b)  ## Return negative LL
}
> normals <- rnorm(100)
> out <- optim(c(1, 2), negloglik, data = normals,
               method = "BFGS")
> out[["par"]]
[1] -0.001523056  0.963032909
```

**Note:** `optim()` and `nlm()` *minimize* functions by default, so you usually have to compute the negative log-likelihood.

## Example (cont'd): Using lexical scoping

Write a "constructor" function:

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

**Example (cont'd): Construct the likelihood function**

```
> set.seed(1); normals <- rnorm(100, 1, 2)
> nLL <- make.negloglik(normals)
> nLL
function(p) {
        op[!fixed] <- p
        mu <- op[1]
        sigma <- op[2]
        a <- -0.5 * length(data) * log(2 * pi * sigma^2)
        b <- -0.5 * sum((data - mu)^2) / (sigma^2)
        -(a + b)
    }
<environment: 0x8f78ccc>
> ls(environment(nLL))
[1] "data"  "fixed" "op"
```

## Example (cont'd): Estimate both parameters

```
> optim(c(mu=0,sigma=1), nLL, method="BFGS")[["par"]]
      mu    sigma
1.217758 1.787531
> c(mean(normals), sd(normals))
[1] 1.217775 1.796399
```

## Example (cont'd): Hold parameters fixed

Fixing $\sigma = 2$:

```
> nLL <- make.negloglik(normals, fixed=c(FALSE, 2))
> optimize(nLL, c(-1, 3))[["minimum"]]
[1] 1.217775
> mean(normals)
[1] 1.217775
```

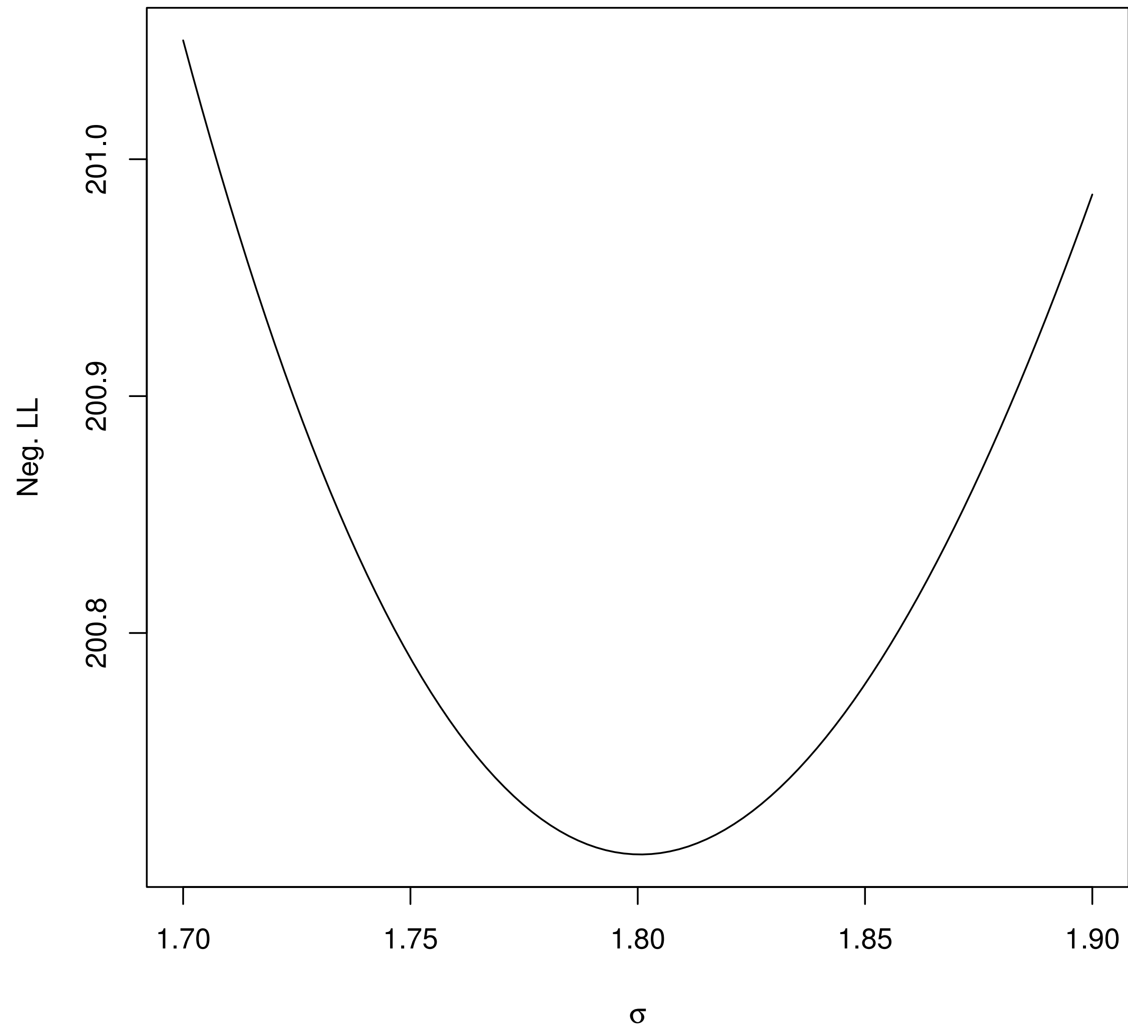
## Example (cont'd)

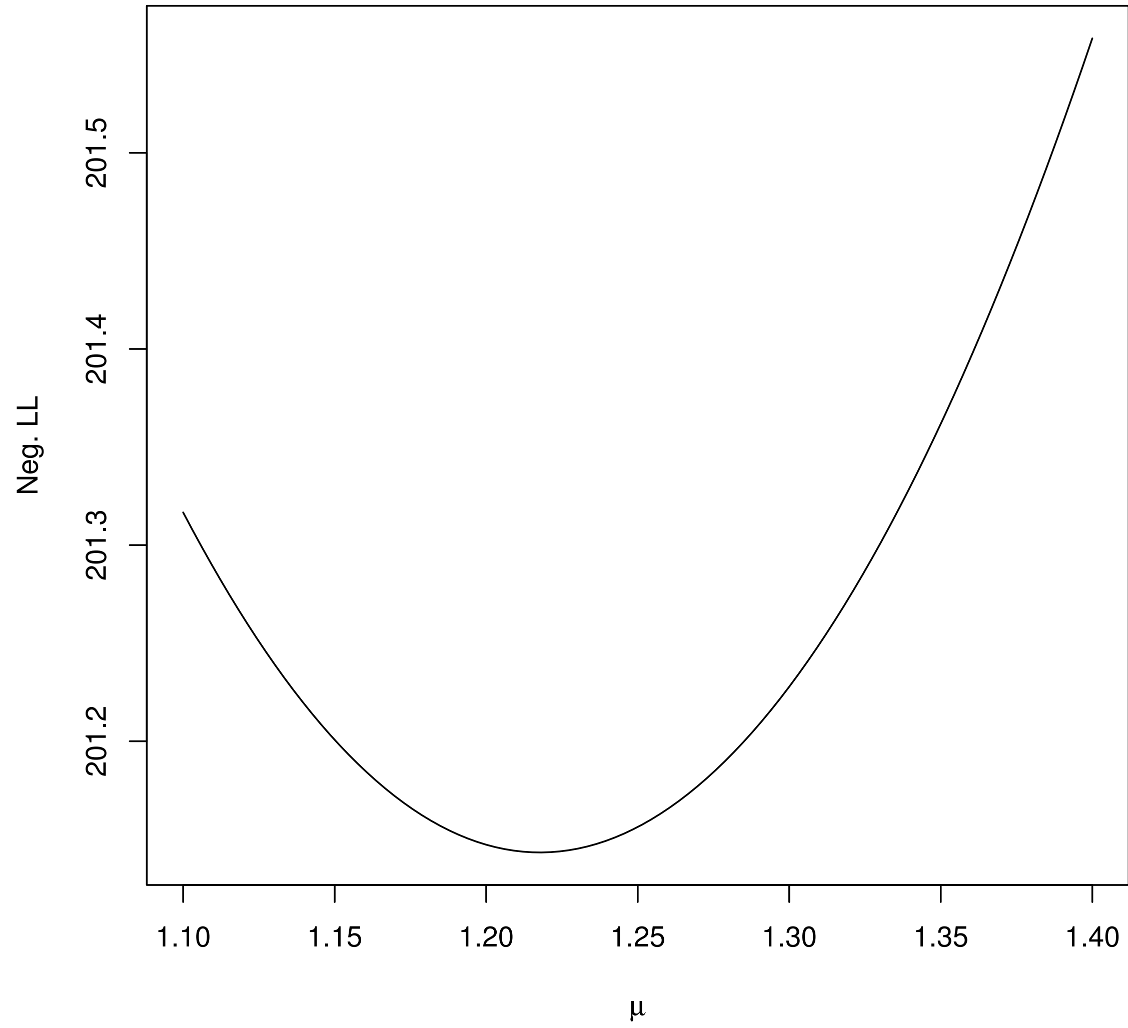
Fixing $\mu = 1$:

```
> nLL <- make.negloglik(normals, fixed=c(1, FALSE))
> optimize(nLL, c(1e-6, 5))[["minimum"]]
[1] 1.800620
> sd(normals)
[1] 1.796399
```

## Example (cont'd): Plot the likelihood function

```
nLL <- make.negloglik(normals, fixed=c(1, FALSE))
x <- seq(1.7, 1.9, len = 100)
y <- sapply(x, nLL)  ## nLL is not vectorized!
plot(x, y, type = "l",
     xlab= expression(sigma),
     ylab = "Neg. LL",
     main = expression(paste(mu, " = 1")))
```


μ = 1
Neg. LL
201.0
200.9
200.8
1.70
1.75
1.80
1.85
1.90
σ


σ = 2
Neg. LL
201.2
201.3
201.4
201.5
1.10
1.15
1.20
1.25
1.30
1.35
1.40
μ

**Lexical Scoping Summary**

- Objective functions can be "built" which contain all of the necessary data and other things.
- No need to carry around long argument lists – useful for interactive/exploratory work.
- Code can be simplified/cleaned up.

**Reference**


- Gentleman, R. and Ihaka, R. (2000), "Lexical Scope and Statistical Computing", JCGS, 9, 491-508.

**Use R!**

**Tell your friends!**