Writing Functions in R

JEFF GILL
Division of Biostatistics
Washington University, St. Louis

Motivation

Software	Stack Overflow Discussions	Cross Validated Discussions
R	10,606	1,612
SAS	509	90
SPSS	95	145
Stata	55	95
All others	0	<10

Software	Number of Blogs
R	365
SAS	40
Stata	8
Others	0-3

Table 3. Number of blogs devoted to each software package on March 13, 2012.

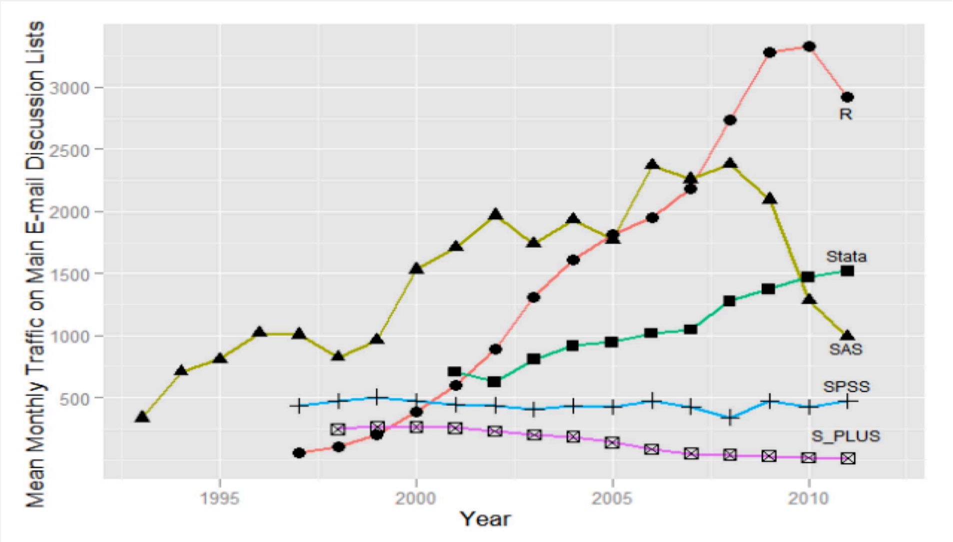


Figure 1. Plot of listserv discussion traffic by year (through December 31, 2011).

Introductory Notes

- Functions are an important part of using R because they allow you to customize and extend the language.
- Functions make you more productive over time.
- Functions can be shared.
- Existing functions can be extended.
- The rules for writing functions are pretty simple.
- Note on *lexical scoping*: variables created within functions are temporary, but variables in your R environment are not and can be read inside the function (although inside names have precedence).
- "Everything is an object in S, and all objects are dynamic and self-defining." -Chambers (1998, 168): function objects vs. data objects.

Illustrative Beginning Example

- To test memory retrieval Kail and Nippold (1984) asked 8, 12, and 21 year olds to name as many animals and pieces of furniture as possible in separate seven minute intervals.
- They find that this number increases across the tested age range but that the rate of retrieval slows down as the period continues.
- In fact, the responses often came in "clusters" of related responses ("lion," "tiger," "cheetah," etc.), where the relation of time in seconds to cluster size is fitted to be

  $$cs(t) = at^3 + bt^2 + ct + d,$$

  where time is t, and the others are estimated parameters (which differ by topic, age group and subject).
- There are strong theoretical reasons that $b = -18a$ from the literature.
- The researchers were very interested in the inflection point of this function since it suggests a change of cognitive process.

Illustrative Beginning Example

dog . . . (1 sec) . . . *cat* . . . (3 sec) . . . *bird* . . . (8 sec) . . . *lion* . . . (2 sec) . . . *tiger.* If a pause time of 1 sec or less is taken to reflect items retrieved from the same cluster (i.e., $t \leq 1$), then *dog/cat* would be from the same cluster; the remaining words would represent different clusters. In this case, $cf(1) = 1$ and $N = 5$, so the mean cluster size is $5/(5 - 1)$, or 1.25, reflecting three one-word clusters and one two-word cluster. Continuing the analysis, $cf(2) = 2$, so the mean cluster size is $5/(5 - 2) = 1.67$. Again verifying this result, with $t \leq 2$ sec as a criterion, clusters consist of *dog/cat, bird,* and *lion/tiger*. $cf(3) = cf(4) = cf(5) = cf(6) = cf(7) = 3$, hence the mean cluster size for $t = 3$–7 is $5/(5 - 3) = 2.5$. Finally, $cf(8) = 4$, so the mean cluster size is $5/(5 - 4) = 5$.

Cluster sizes computed in this manner are depicted in the right panel of Figure 2 as a function of t for the cumulative frequency data depicted in the left-hand panel of that figure. The cluster size function, like the cumulative frequency distribution, has a plateau between 5 and 7 sec. As before, this plateau corresponds to the break between the two distributions of pause times.

The final issue to be considered is how to identify the precise point at which the initial decelerating curve begins to accelerate, for this value differentiates the longer pause times associated with retrieval of clusters from the briefer pause times associated with rapid emission of items. In fact, functions like those depicted in Figure 2 are well described by a third-order polynomial of the type

$$cs(t) = at^3 + bt^2 + ct + d, \qquad (2)$$

where cs refers to cluster size and t is time in seconds. Further, the second derivative of this polynomial, $-b/3a$, corresponds to the inflection point at which the function stops decelerating and starts accelerating. Once this inflection point is known, pauses in the retrieval protocol can be identified unambiguously as reflecting either search for additional clusters or emission of items from within a cluster. Then one can derive the number of clusters as well as the average size of clusters in the retrieval protocol.

Cluster sizes were calculated for each individual's retrieval protocol for t ranging from 2 to 10 sec. These cluster values were then fit to equation (2) with STEPIT. The estimated values of a and b were used to calculate the inflection point of the cluster size function. Of the 39 individuals, six had at least one protocol that included either a negative number or an extraordinarily large

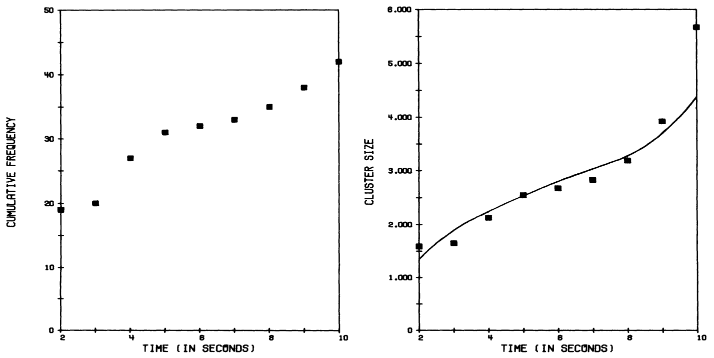


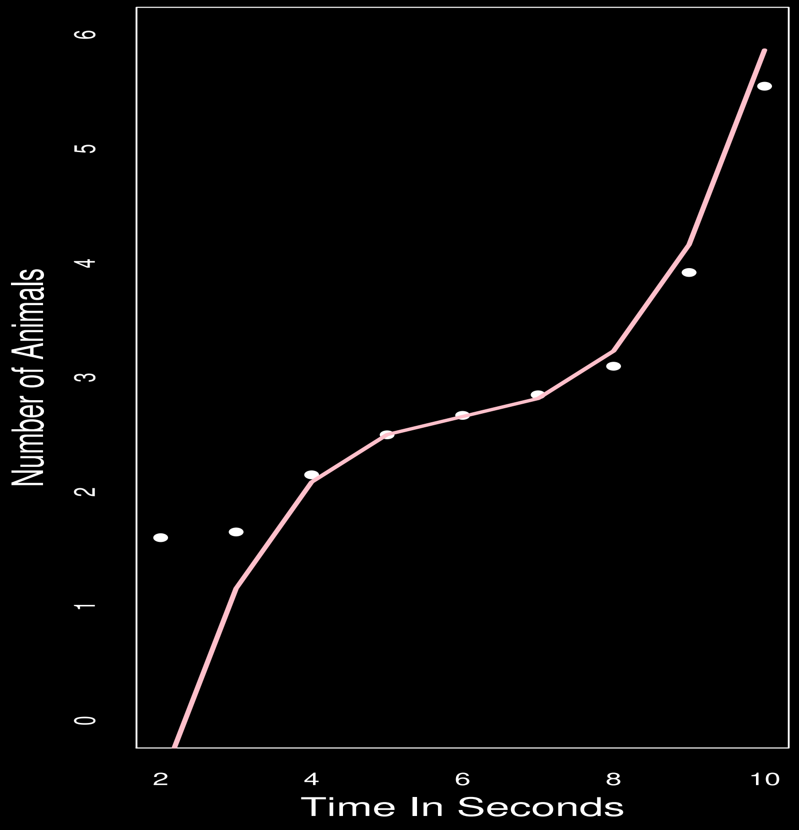
FIG. 2.—Cumulative frequency of pause times (left panel) and cluster size (right panel) as a function of time for one 8-year-old. The function in the right panel is derived from the best-fitting values of the a and b parameters from equation (2).

Illustrative Beginning Example

- We can specify hard-coded values of the parameters (below) by trial and error.

```
cs <- c(1.6,1.65,2.15,2.5,2.67,2.85,3.1,3.92,5.55)
seconds <- 2:10

cog <- function(a,c,d,t) a*t^3 + (-18*a)*t^2 + c*t + d

par(mfrow=c(1,1),mar=c(5,5,3,3),col.axis="white",col.lab="white",
    col.sub="white",col="white",bg="black")
plot(seconds,cs,pch=19,ylim=c(0,6),xlab="",ylab="")
cs.vals <- cog(a=0.04291667,c=4.75,d=-7.3,t=seconds)
lines(seconds,cs.vals,col="pink",lwd=3)
mtext(side=1,line=2.5,cex=1.5,"Time In Seconds")
mtext(side=2,line=2.5,cex=1.5,"Number of Animals")
```

Nonlinear (Weighted) Least-Squares



Illustrative Beginning Example

- We can also use the R function `nls` to estimate these by minimizing residuals:

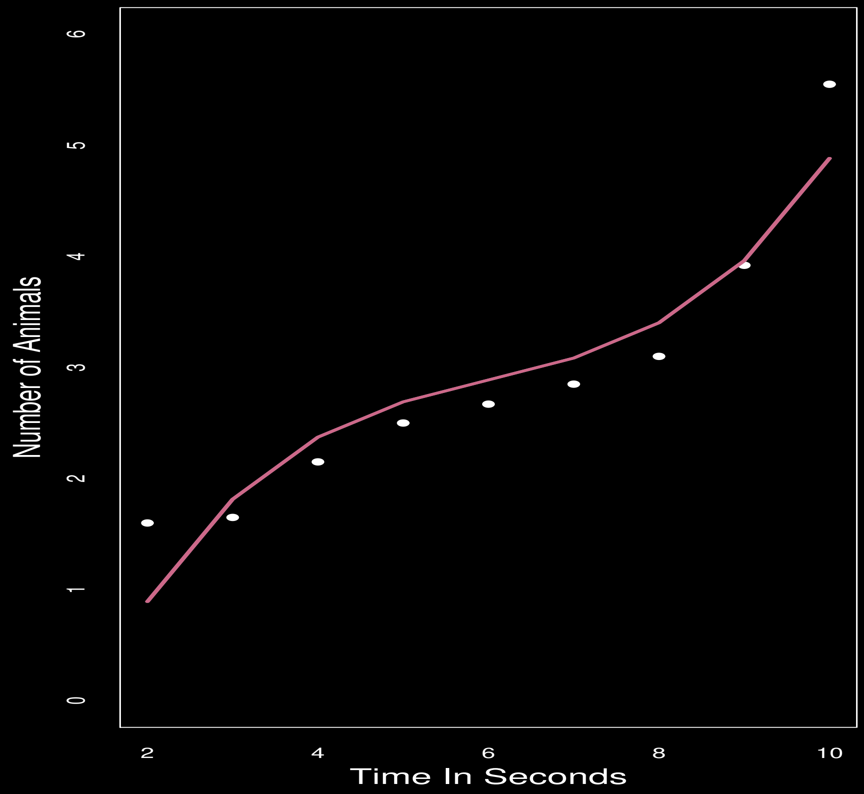
```
cog.df <- data.frame(seconds=seconds,cs=cs)
cog.nls <- nls(cs ~ a*seconds^3 + (-18*a)*seconds^2 + c*seconds + d,
               start=c(a=10,c=10,d=-10),trace=TRUE); summary(cog.nls)

  Estimate Std. Error t value Pr(>|t|)
a  0.02013    0.01211   1.662   0.1476
c  2.35048    1.16637   2.015   0.0905
d -2.52056    1.79998  -1.400   0.2109
Residual standard error: 0.4572 on 6 degrees of freedom
```

Illustrative Beginning Example

- And then plot the results:

```
par(mfrow=c(1,1),mar=c(4,4,4,4),oma=c(3,3,3,3),col.axis="white",col.lab="white",
    col.sub="white",col="white",bg="black")
plot(seconds,cs,pch=19,ylim=c(0,6),xlab="",ylab="")
cs.vals <- cog(a=summary(cog.nls)$parameters[1,1],
               c=summary(cog.nls)$parameters[2,1],
               d=summary(cog.nls)$parameters[3,1],
               t=seconds)
lines(seconds,cs.vals,col="palevioletred3",lwd=3)
mtext(side=1,line=2.5,cex=1.5,"Time In Seconds")
mtext(side=2,line=2.5,cex=1.5,"Number of Animals")
```

Illustrative Beginning Example



General Form

- Functions need a name that you provide (make them intuitive!).
- The function name is followed by the assignment operator.
- Then list the input parameters in parenthesis.
- Open commands with an open curly brace.
- Type commands one-per-line or semi-colon separated.
- The last line of commands is what you return to the user, either with the `return()` command with the object inside the parentheses or just by typing an object.
- Finish with a close clurly brace.
- Complete form:

```
my.fun <- function(in.parameters) { commands; return(output) }
```

Millions of Functions Already Exist in R

```
.First
function() cat("\n   Welcome to R!\n\n")
.Last
function()  cat("\n   Goodbye!\n\n")

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

ls
table
mode
```

Defining Functions

```
dam
Error: object "dam" not found

dam <- function(in.vec)  median(abs(in.vec - median(in.vec)))
dam(runif(100,0,23))
[1] 5.975548

my.binom <- function(max,p)  {
    out.probs <- NULL
    for (i in 0:max)
    out.probs <- c( out.probs,choose(max,i)*(p^i)*((1-p)^(max-i)) )
    return(out.probs)
}

binom.probs <- my.binom(3,0.5)
[1] 0.125 0.375 0.375 0.125
```

Note indentations.

Multiple Arguments

- You can pass multiple arguments to a function, but be careful about the order if the context is not obvious.
- Example using `X.Vals <- rchisq(100,df=3)` data:

```
mean(X.Vals)
mean(x=X.vals)
mean(x=X.vals, na.rm=FALSE)
mean(X.vals, na.rm=FALSE)
mean(na.rm=FALSE, x=X.vals)
mean(na.rm=FALSE, X.vals)
mean(FALSE,X.Vals)      # WILL FAIL:
 Error in mean.default(FALSE, X.Vals) :
   'trim' must be numeric of length one
```

- To see what arguments a functions needs use `args()`.
- Some functions use defaults for specific arguments, so you do not have to type them if the default is okay.

More On Arguments

- Arguments in R are evaluated "lazily" meaning that if not needed, they are ignored:

```
simple.fun <- function(x,y)  {
    return(log(x))
}
simple.fun(2)        [1] 0.69315
```

  but the reverse is not true: extra parameters in the function call cause failure;

```
simple.fun(1,3,9,12)
Error in simple.fun(1, 3, 9, 12) : unused arguments (9, 12) # 3 OKAY FROM VARIABLE
```

- Argument defaults can also be set to `NULL`.

- The `...` argument has two main functions: when you expect an unknown number of other functions to be called by this function, and when modifying an existing function and you don't care about the last set of arguments:

```
simple.fun <- function(x,y,...) {
    return(log(x))
}
simple.fun(1,3,9,12)        [1] 0
```

Naming Your Functions

- Use an intuitive name that is original, bad: `ZBR.139.v23`, good: `kernel.fit`.
- Do not use the name of an existing R function, although this is not fatal.
- If you do that R will give your function precedence over the built-in function, so important functions like `mean`, `lm`, `seq` will not be available until you delete yours.
- Most common case: name a variable `c`, for example:

```
c <- function(x) x^3
c(3)
[1] 27
c(1,2,3)
Error in c(1, 2, 3) : unused arguments (2, 3)
rm(c)
c(1,2,3)
[1] 1 2 3
```

Defining Functions, Defaults

- Default values can be very useful, such as `alpha=0.05`.
- Users can override defaults with explict values.
- For example,

```
my.binom <- function(num,p=0.5)  {
    out.probs <- rep(NA,num)
    for (i in 0:num)
       out.probs[i] <- choose(num,i)*p^i * (1-p)^(num-i)
    return(out.probs)
}
my.binom(5)
[1] 0.03125 0.15625 0.31250 0.31250 0.15625 0.03125
my.binom(5,0.1)
[1] 0.59049 0.32805 0.07290 0.00810 0.00045 0.00001
```

- It is also convenient to nest functions within other functions:

```
mean(my.binom(5))
[1] 0.19375
```

A Simple Function For Matrices

- Functions also work on matrices.

```
tr <- function(in.mat) sum(diag(in.mat))
tr
       function(in.mat) sum(diag(in.mat))

clement.mat <- matrix(c(0,1.732051,0,0,1.732051,0,2.0,0,0,2.0,0,1.732051,
                        0,0,1.732051,0), nrow=4)
clement.mat
              [,1]   [,2]   [,3]   [,4]
       [1,] 0.0000 1.7321 0.0000 0.0000
       [2,] 1.7321 0.0000 2.0000 0.0000
       [3,] 0.0000 2.0000 0.0000 1.7321
       [4,] 0.0000 0.0000 1.7321 0.0000

tr(clement.mat)
[1] 0
```

Logit and Inverse-Logit (Logistic) Functions

```
logit <- function(mu)  log(mu/(1-mu))
inv.logit <- function(Xb)  1/(1+exp(-Xb))

X <- matrix(rnorm(10000,0,5),ncol=10)
beta <- rt(10,df=5)

inv.logit(X%*%beta)

[,1]
[1,] 9.7054e-01
[2,] 2.0785e-06
[3,] 9.9983e-01
[4,] 1.8339e-01
[5,] 9.9999e-01
[6,] 9.9999e-01
 :
```

Loops In Functions

- Two basic kinds: `for` and `while` (see also `repeat`).
- Loops make functions powerful through, possibly many, iterations of some work.
- Makes less work for humans!
- A simple R function for Newton-Raphson mode-finding to get a square root:

```
newton.raphson.ex <- function(mu,x,iterations)  {
    for (i in 1:iterations)
        x <- 0.5*(x + mu/x)
    return(x)
}

newton.raphson.ex(99,2,3)
[1] 10.74386
newton.raphson.ex(99,2,6)
[1] 9.949874
```

Termination

- Sometimes it's handy to write-in termination criteria to functions.
- For example, our Newton-Raphson root-finding algorith should be stopped when further iterations provide trivial changes.
- Now define a tolerance for this parameter, which is a default, and rewrite according to:

```
newton.raphson.ex <- function(mu,x,tol=1e-06)  {
    diff <- 1
    while (diff > tol)  {
        x.new <- 0.5*(x + mu/x)
        diff <- abs(x.new - x)
        x <- x.new
    }
    return(x)
}

newton.raphson.ex(99,2)
[1] 9.949874
```

Lab Assignment

- Run the following function with the commands afterwards (and possibly more).
- Determine what this function does.

```
myFunction <- function(x){
    out <- TRUE
    checker <- function(a, b){
        if(b>a) {TRUE} else {FALSE}
    }
    for(i in 1:(length(x)-1)){
        out <- (checker(x[i], x[i+1])*out)
    }
    return(as.logical(out))
}

 myFunction(c(4,3,2,1))
 myFunction(c(1,2,2,4))
 myFunction(c(1,2,3,4))
 myFunction(2)
 myFunction(c(TRUE, FALSE))
```

A Function for a New Distribution

- The Witch's Hat Distribution:

$$p(\theta|\mathbf{x}) = (1-\delta)[2\pi\sigma^2]^{-d/2}\exp\left[-\sum_{i=1}^{d}\frac{1}{2\sigma^2}(x_i-\theta_i)^2\right] + \delta I_{(0,1)}(x_i),$$

- Implementing with a function:

```
witch.hat <- function(t1,t2,y,sigma,delta,T=1)  {
        theta <- c(t1,t2)
        normalizer <- (1-delta) * ( 1/(sqrt(2*pi)*sigma) )^length(theta)
        exponent   <- exp(-sum( ((y-theta)/sigma)^2/2 )/T)
        (normalizer * exponent + delta)
}
```

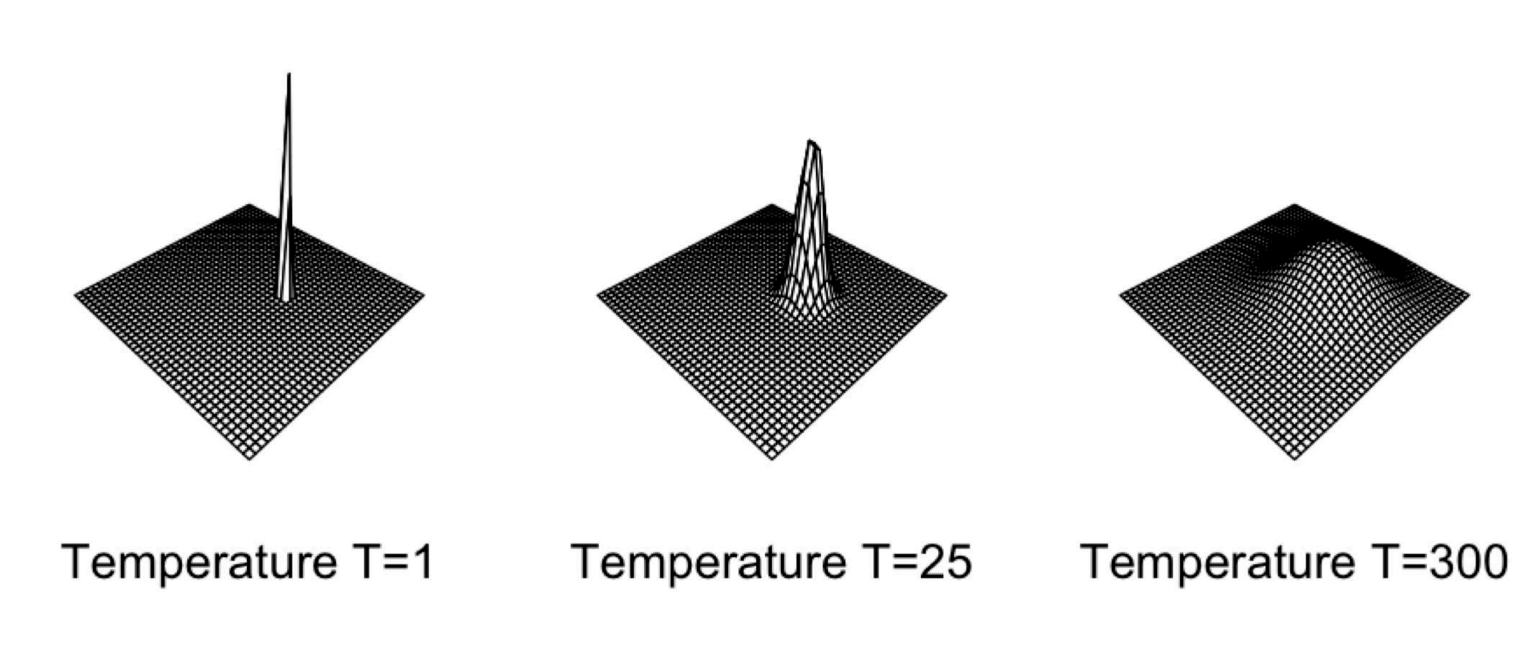
A Function for a New Distribution

```
delta <- 0.0000000001
sigma <- 0.01
y <- c(0.4,0.6)
theta1 <- seq(0,1,length=40)
theta2 <- seq(0,1,length=40)
witch.dens <- matrix(NA,length(theta1),length(theta2))

par(mfrow=c(1,3),mar=c(0.5,0.5,0.5,0.5),oma=c(0.01,0.01,0.01,0.01))
for(i in 1:length(theta1))
    for(j in 1:length(theta2))
            witch.dens[i,j] <- witch.hat(theta1[i],theta2[j],y,sigma,delta,T=1)
persp(theta1,theta2,witch.dens, theta = 135, phi = 30,box=F,zlim=c(0,600))
mtext(side=1,cex=1.3, line=-15,"Temperature T=1")
```

A Function for a New Distribution

```
for(i in 1:length(theta1))
    for(j in 1:length(theta2))
            witch.dens[i,j] <- witch.hat(theta1[i],theta2[j],y,sigma,delta,T=25)
persp(theta1,theta2,witch.dens, theta = 135, phi = 30,box=F,zlim=c(0,2200))
mtext(side=1,cex=1.3, line=-15,"Temperature T=25")

for(i in 1:length(theta1))
    for(j in 1:length(theta2))
            witch.dens[i,j] <- witch.hat(theta1[i],theta2[j],y,sigma,delta,T=300)
persp(theta1,theta2,witch.dens, theta = 135, phi = 30,box=F,zlim=c(0,6000))
mtext(side=1,cex=1.3, line=-15,"Temperature T=300")
```

A Function for a New Distribution



Example Function That Finds Primes From 1 To Given Max

```
find.primes <- function(max)  {
    num.vec <- seq(1,max,by=2)                                # SETUP VECTOR TO EVALUATE
    if(max > 5) primes <- 3  else(stop("min of max = 5"))  # START PRIMES VECTOR,
                                                              # CHECK FOR VALID INPUT
    for (i in 3:length(num.vec))  {                          # START LOOPING VECTOR
        if (min( num.vec[i] %% primes != 0 ))                # EVALUATE CURRENT VALUE
            primes <- c(primes,num.vec[i])                   # ADD TO PRIMES VECTOR
    }
    return(c(1,2,primes))                                     # RETURN TO USER
}

find.primes(200)
 [1]   1   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67
[21]  71  73  79  83  89  97 101 103 107 109 113 127 131 137 139 149 151 157 163 167
[41] 173 179 181 191 193 197 199
```

Lab Assignment

- Write a function to produce the first `max` Fibanacci numbers.
- The series starts with: $0, 1, 1, 2, 3, 5, 8, 13, \ldots$.
- Modify your function to return only the primes from this series.

The Gibbs Sampler

◇ Consider two exponential pdfs with parameters conditional on each other:

$$f(x|y) \propto y\exp[-yx], \quad f(y|x) \propto x\exp[-xy], \quad 0 < x, y < B < \infty.$$

where we want to describe the marginal distributions of x and y.

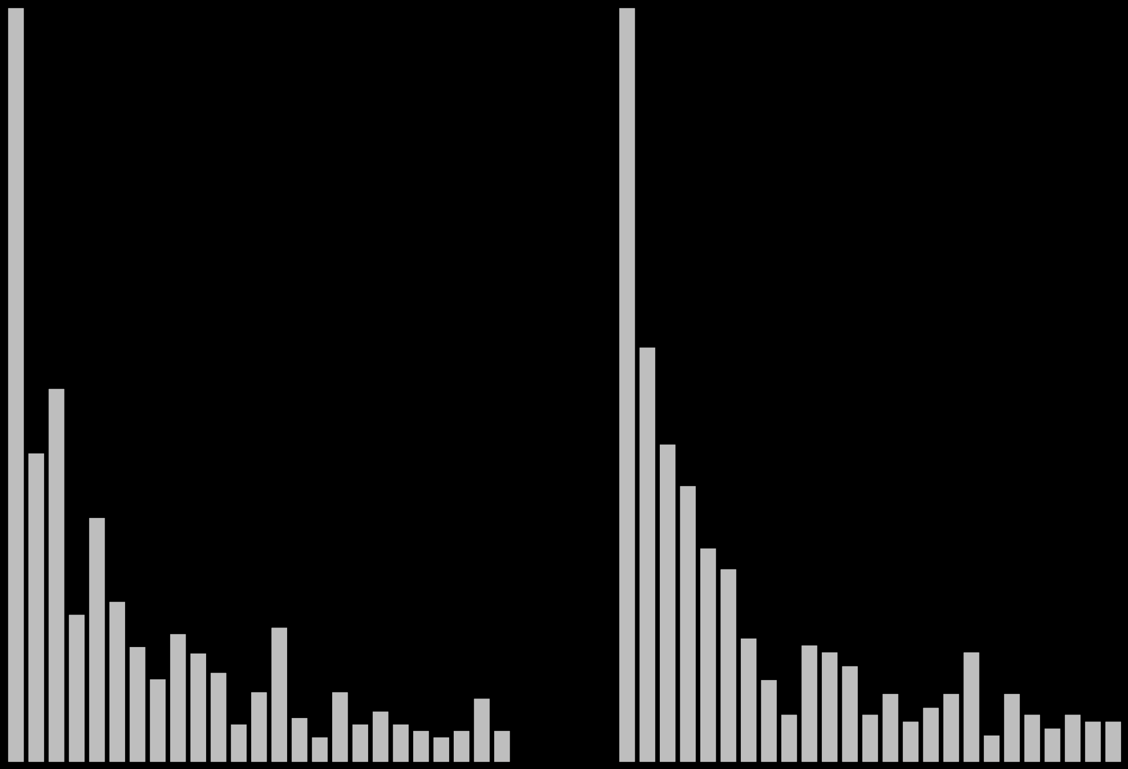
◇ For two parameters, x and y, this involves a starting point, $[x_0, y_0]$, and the cycles defined by drawing random values from the conditionals according to:

$$\begin{aligned}
x_1 &\sim f(x|y_0), & y_1 &\sim f(y|x_1) \\
x_2 &\sim f(x|y_1), & y_2 &\sim f(y|x_2) \\
x_3 &\sim f(x|y_2), & y_3 &\sim f(y|x_3) \\
&\vdots & &\vdots \\
&\vdots & &\vdots \\
x_m &\sim f(x|y_{m-1}), & y_m &\sim f(y|x_m).
\end{aligned}$$

The Gibbs Sampler, Conditional Exponential Distributions

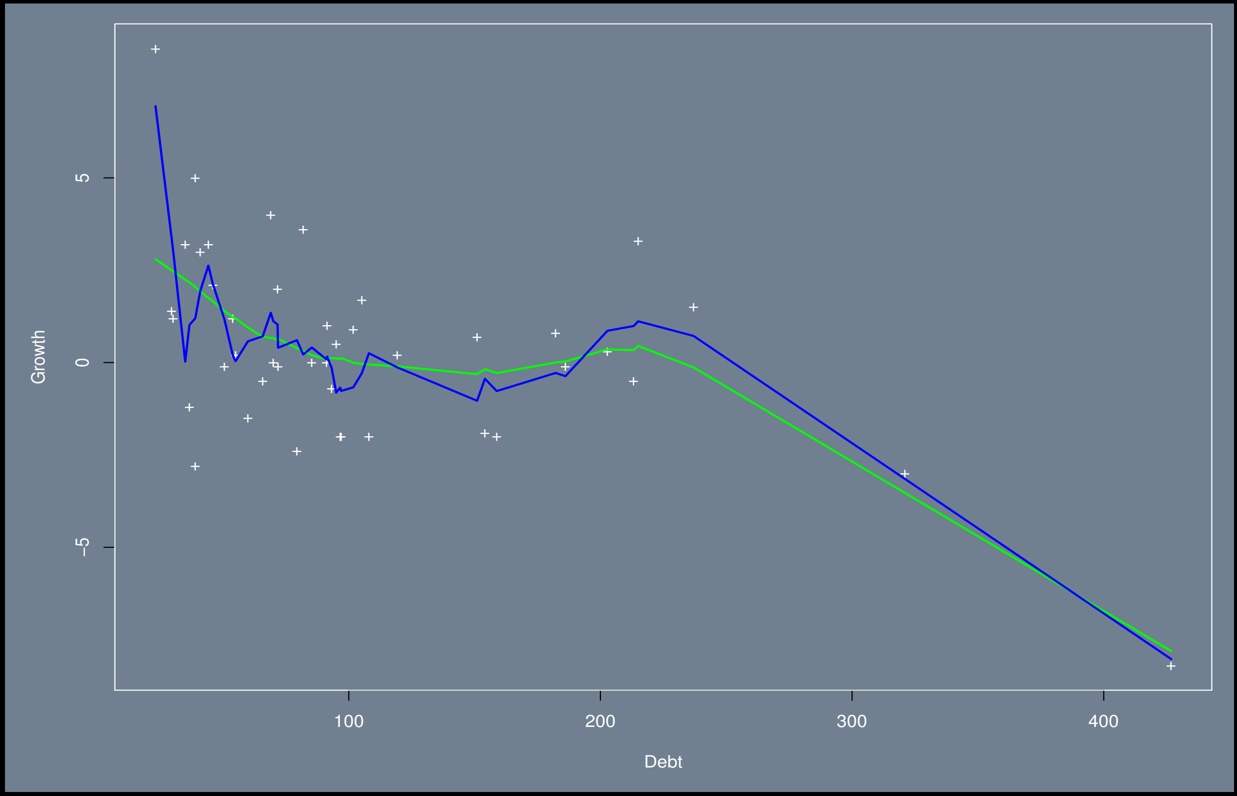
```
gibbs.expo <- function(B,m) {
    x <- c(runif(1,0,B),rep((B+1),length=(m-1)))
    y <- c(runif(1,0,B),rep((B+1),length=(m-1)))
    for (i in 2:m)  {
        while(x[i] > B) x[i] <- rexp(1,y[i-1])
        while(y[i] > B) y[i] <- rexp(1,x[i])
    }
    return(cbind(x,y))
}

gibbs.expo(B=5, m=500)
```

The Gibbs Sampler (cont.)



A Smoothers As Nonparametric Displays of Bivariate Relationships

- Smoothers are graphical ways to show generally non-linear relationships in data without having to give a parametric or functional form.
- This is a very active area in research statistics.



Kernel Smoothers

- An extension of the running-line smoother where explicit weights are included in the smoothing function $s()$.
- Standard idea: weight the points closer to x_i more than remote points.
- Pick k points to the left and k points to the right of the x_i point, producing a neighborhood size of $2k+1$.
- Define the jth weight for the ith point as the function:

$$\omega_{ij} = cd\left(\left|\frac{x_i - x_j}{\lambda}\right|\right) \qquad \text{for } j \in N_{2k+1},\ 0 \text{ otherwise}$$

where:

  - c is a normalizing constant such that $\int s(u)du = 1$:

$$c = \left[\sum_{i=1}^{2k+1} d\left(\left|\frac{x_i - x_j}{\lambda}\right|\right)\right]^{-1}$$

  - λ is the window width: $2k+1$,
  - and $d(t)$ is a decreasing function in $t = |x_i - x_j|/\lambda$.

Kernel Smoothers

- So the smoothed y-axis point is:

$$\hat{y}_i = s(y_i|\mathbf{X}) = \sum_{j=1}^{2k+1} \omega_{ij} y_i.$$

- Common forms:

$$d(t) = \phi(t) \qquad \text{[Gaussian]}$$

$$d(t) = \begin{cases} \frac{3}{4}(1 - t^2), & |t| \leq 1 \\ 0, & \text{otherwise} \end{cases} \qquad \text{[Epanechnikov]}$$

$$d(t) = \begin{cases} \frac{3}{8}(3 - 5t^2), & |t| \leq 1 \\ 0, & \text{otherwise} \end{cases} \qquad \text{[minimum variance]}$$

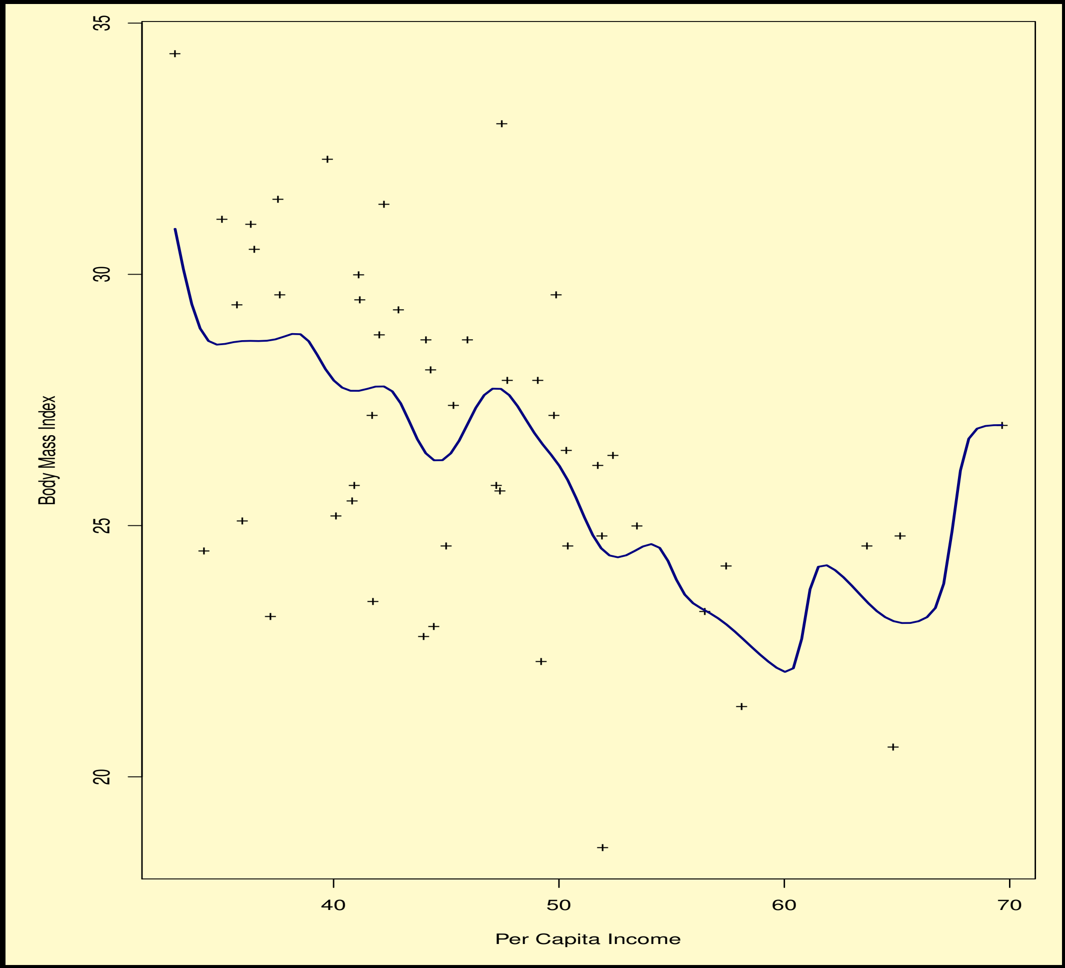
- Others: box, triangle, Parzen, miscellaneous polynomials.

Kernel Smoothers

- Using the R function `ksmooth`:

```
par(mfrow=c(1,1),mar=c(5,5,5,5),bg="lemonchiffon")
plot(x0,y0,pch="+",lwd=1,xlab="Per Capita Income",ylab="Body Mass Index")
lines(ksmooth(x0,y0,kern="normal",bandwidth=4),col="navy")
mtext(side=3,cex=1.3,line=2,"Gaussian Kernel Smoother")
```

Kernel Smoothers



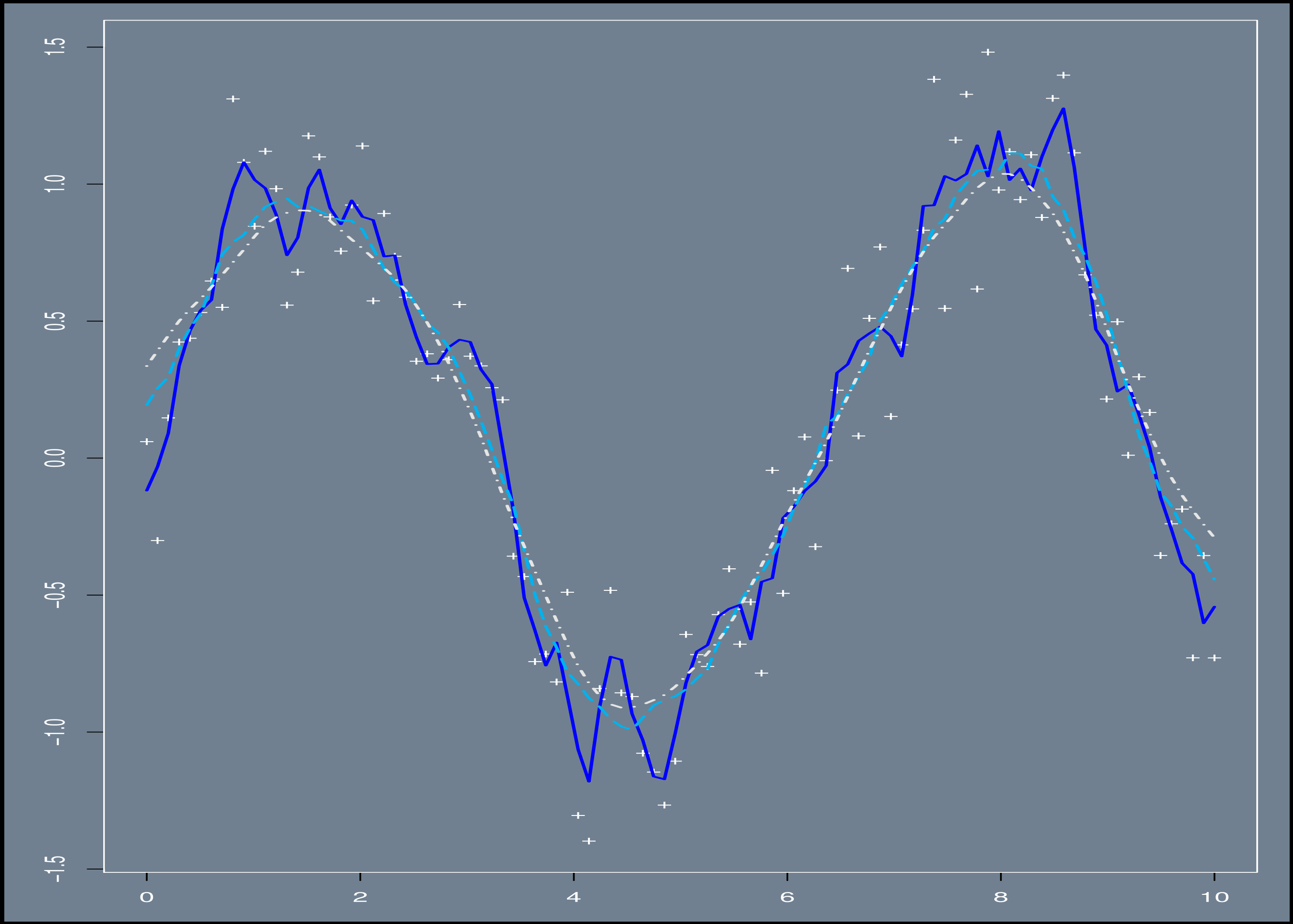
Kernel Smoothers, Rolling Our Own

```
epan <- function(x) ifelse(abs(x) <= 1, 0.75*(1-x^2), 0)

k.sm <- function(x,y,k)  {
    s.y <- y
    for (i in 1:length(x))  {
        lo <- ifelse(i-k >= 1, i-k, 1)
        hi <- ifelse(i+k <= length(x), i+k, length(x))
        w <- epan(x[i] - x[lo:hi])/sum(epan(x[i] - x[lo:hi]))
        s.y[i] <- y[lo:hi] %*% w
    }
    s.y
}
```

Kernel Smoothers, Rolling Our Own

```
X <- seq(0,10,length=100)
Y <- sin(X) + rnorm(length(X),0,0.3)
par(mar=c(3,3,1,1),col.axis="white",col.lab="white",col.sub="white",
    col="white",bg="slategray")
plot(X,Y,pch="+")
for (j in c(1,5,10)) lines(X,k.sm(X,Y,j),col=colors()[3+24*j],lty=j,lwd=2)
```

Kernel Smoothers, Rolling Our Own



Lab Assignment

- Write a minimum variance function:

$$d(t) = \frac{3}{8}(3 - 5t^2), \;\; |t| \leq 1$$

  instead of the Epanechnikov function by modifying it.

- Modify the `k.sm` function to to call your mininum variance function.

- Produce a graph with the fake data just given.

Different Ways To Write the Same Function

```
frobenius <- function(in.mat)  {
    Frob <- 0                                  # START WITH ZERO
    for (i in 1:nrow(in.mat))                  # LOOP THROUGH ROWS
        for (j in 1:ncol(in.mat))              # LOOP THROUGH COLUMNS
            Frob <- Frob + in.mat[i,j]^2       # ADD SQUARED VALUES
    Frob^(1/2)                                 # RETURN SQUARE ROOT
}

frobenius <- function(in.mat)  (sum(diag(in.mat%*%t(in.mat))))^(1/2)

X <- matrix(c(5,2,3,2.99,2,1,2,1,5,2,3,3),4,3)
frobenius(X)
[1] 10.195
```

General Guidance For Writing Functions

- Insert comments, even if your instructor doesn't very much.
- First solve a basic core problem, particularly for complex settings.
- Modularize as much as possible: functions calling other functions.
- Corollary 1: avoid rewriting the same code.
- Corollary 2: test sub-functions first.
- Contrive fake data where you know the answer to test your code.
- Use print statements when problems occur.
- Use meaningful variable and function names.

Writing Functions For Others To Use (including yourself in 6 months)

- Make the call and the variable definitions as clear as possible.
- Use checking: `stop`, `stopifnot()`, `warning()`.
- Consider writing an `R` package if it is something really useful and unique.
- Don't expect others (even coauthors!) to understand the inner guts of your code without help.

Lab Assignment

- Using the hemodialysis data, write a function that summarizes the data for tobacco users.
- First search on the `tobacco` variable for 1 rather than 0.
- Create a new data frame to store these cases.
- Summarize each variable across columns for these cases.