R Tips for STAT 515 at Penn State:

Stochastic Processes and Monte Carlo Methods

Baojun Qiu
CSE Dept., Penn State University
http://qiubaojun.googlepages.com/
May 1, 2008

There is a lot of more information in the R manuals (http://cran.us.r-project.org/manuals.html). Which give an introduction (http://cran.us.r-project.org/doc/manuals/R-intro.pdf) to the language, explain how to extend R (http://cran.us.r-project.org/doc/manuals/R-exts.pdf), and describe ways to get data in and out of R (http://cran.us.r-project.org/doc/manuals/R-data.pdf). And you will get more if you search on WWW.

1. Things to Keep in Mind

1) In R console, the arrow (up, down) keys can be used to explore your command history; the Escape key serves as your abort button.
2) R is case sensitive. Keep that in mind when you name objects or call functions.
3) There are lots of built-in objects and functions. You'd better to name your objects or functions without collisions with the system.
4) To get help in R console, e.g. `help(rnorm)`,Or use `?rnorm`
5) Good Coding Style
   a) Code is formatted properly (indented etc.). This will happen automatically if you use a sensible editor. See the course website for ways to obtain such editors. Dr. Haran recommends emacs + ESS.
   b) Add comments to important lines or difficult parts to make sure it is readable. In R, comments are started with the pound sign.
6) Try Google and Wikipedia for answers.
7) Feel free to update it and make it more helpful for students in STAT 515 class.

2. Import and Export Data or Information

1) Import and export a data frame

```
aDataFrame <-
read.table("http://www.stat.psu.edu/~mharan/515/hwdir/donner.txt",
header = FALSE) #Import a file in table format and creates a data frame
from it. The file can be on the web as well as in local disk.
aVector1 <- dataFrame$V1 # get the first column, V1 is the default header
for the first column.
aVector1 <- dataFrame[ , 1] # get the first column in another way.
write.table(aDataFrame, "afile.txt") #save the table frame to a file.
```

2) Import and export a matrix

```
aVector <-
scan("http://www.stat.psu.edu/~mharan/515/hwdir/donner.txt") # read in
the data into a vector
aMatrix <- matrix(aVector, 45, 3, byrow=TRUE)  # Save the data to a
matrix(45*3)
aMatrix <- matrix(aVector, , 3, byrow=TRUE)  # Omit one of the size index,
it also works.
write.table(aMatrix, "t_file.txt") #save a matrix to a files in table
format
library(MASS) # write.matrix( ) is in package MASS
write.matrix(aMatrix, file ="m_file.txt", sep = ",")
```

Comments: `read.table( )` and `write.table( )` are designed to read and write data frames which may have columns of very different classes. To read and write large matrices, especially those with many columns, use `scan( )` and `write.matrix( )` instead.

3) Save information to a file

```
notes1 <- "a string"
aNumber <- 2.5
cat(notes1,"\n","The value of the number:", aNumber,"\n",
file="afile.txt", append=TRUE)# cat is useful for producing output in
user-defined functions. If append=TRUE, output will be appended to file;
otherwise, it will overwrite the contents of file
```

3. Data Operation

1) Basic operators and functions

`+, -, *, /,`
`&&` (logic AND), `||` (logic OR)
`&` (logic AND), `|` (logic OR)
Comment: `&` and `|` apply element-wise to vectors; `&&` and `||` apply to vectors of length one, and only evaluate their second argument if necessary.

`^ (power), sqrt, abs, exp` (exponential function), `log,`
`floor` (largest integer less than or equal), `ceiling,`
`%%` or `%/%` (mod),
`<-` or `=` (assignment operator)
`pi` (It's not a operator, but the mysterious number)
Examples:

```
3+2.5 #5.5
sqrt(4) # 2
pi # 3.141593
```

2) Vector
a) Create vectors

```
aVector <- c(1,2,3,4) # create a vector with 3 components
```

```
aVector<- 1:4 #as same as above
aVector <- seq(1,4, by=1) #as same as above
aVector <- seq(1,4, length=4) #as same as above
bVector <- rep(2, 4) # create a vector (2, 2, 2, 2)
rep(c(1,2),2) # create a vector (1,2,1,2), the first parameter is the
pattern; the second parameter is the times to repeat the pattern
array(1:2,4) # create a vector (1,2,1,2), the second parameter is the
length of the vector
```

b) Access elements: `VectorName[index]`, `index` can be a number or a **vector**, but be sure the `index` is not out of bound

```
# bVector <- c(2,2,2,2)
bVector[1:2] = c(1,3) # change the first 2 elements to 1 and 3: the result
is (1,3,2,2)
bVector[c(1,3)] # return the first and the third elements: (1 2)
```

c) Component-wise operations

```
#aVector <- c(1,2,3,4)
#bVector <- c(1,3,2,2)
aVector + 1 # (2,3,4,5)
aVector * 2 # (2,4,6,8)
aVector^2 # (1,4,9,16)
2^aVector # (2,4,8,16)
aVector + bVector #(2,5,5,6);  -,*,/ are similar: component-wise
computations
```

d) Other operations and useful functions

```
#aVector <- c(1,2,3,4)
aVector %*% bVector # inner product
sum(aVector*bVector) #inner product: * is component-wise operation and
sum() sums up all elements
cumsum(aVector) #cumulative sum vector (1,3,6,10)
max(aVector) #return the maximum element: 4
min(aVector) #minimum: 1
mean(aVector) #returns 2.5
ave(aVector) #return (2.5, 2.5, 2.5, 2.5)
length(aVector) # number of elements
sort(aVector) # increasing order the vector
sort(aVector, decreasing=T)  # decreasing order
aVector>2 # Logic operation: returns (FALSE FALSE  TRUE  TRUE)
sum(aVector>2) # number of elements great than 2
```

Comment: If we have a vector `v` and its elements are samples from a distribution `X`, `Pr(X>a and X<b) = sum(v>a & v<b)/length(v)`. Please notice use `&` instead of `&&` since we are doing element-wise operations.

```
aVector[aVector>2] # list of the elements greater than 2
var(aVector) #returns the sample variance
sd(aVector) #returns the sample standard deviation
cor(aVector, bVector)#returns the sample correlation coefficient between
two vectors
```

3) Matrix

a) Create Matrix

```
cM <- matrix(0, 3, 2) # create 3 by 2 matrix with each element=0
#or create matrix from vectors like following code
aV <- c(1,2,3)
bV <- c(2,4,5)
aM <- cbind(aV, bV) #column bind two vectors to a matrix
     aV bV
[1,]  1  2
[2,]  2  4
[3,]  3  5
bM <- rbind(aV, bV)#row bind two vectors to a matrix
    [,1] [,2] [,3]
aV    1    2    3
bV    2    4    5
matrix(c(1,2,3,2,4,5),ncol=3, byrow=T) # as same as bM
matrix(c(1,2,3,2,4,5),,3, byrow=T) # as same as above
```

b) Access elements: `MatrixName[index1, index2]`, `index1` and `index2` can be a number or a vector, but be sure the index is not out of bound

```
bM[1,] #first row
bM[ ,2:3] #2nd and 3rd columns
bM[,c(1,3)] #1st and 3rd columns
bM[-1,] #submatrix without the first row
aM[c(1,3),2]#returns (2,5) the 2nd elements of the 1st and 3rd rows
cM[,1] <- aV #set the first column of cM as aV
cM[,2] <- bV # Now cM is identical to aM
```

c) Component-wise operations and functions

```
#dM <- matrix(1:4,2,2)
eM <- dM+dM  # -, *, / are used in a similar way, all of them are
component-wise computations
     [,1] [,2]
[1,]    2    6
[2,]    4    8
```

`sapply(X, FUN, ...)`: returns a list of the same length as X, each element of which is the result of applying FUN to the corresponding element of X

```
#X the array to be used.
#FUN the function to be applied: see 'Details'. In the case of functions
like +, %*%, etc., the function name must be backquoted or quoted.
#... optional arguments to FUN.
sapply(dM,log)#return a vector:(0.0000000 0.6931472 1.0986123
1.3862944)
```

d) Other operations and useful functions

```
t(aM) # transpose
dim(aM) # return (3,2) since it's 3 by 2 matrix
dM %*% eM  # matrix multiplication
```

```
solve(dM)  #inverse matrix of dM
sum(dM) # sum of all elements in dM, mean(.), max(.), min(.) are used
in the similar way
```

`apply(X, MARGIN, FUN, ...)` : Returns a vector or array or list of values obtained by applying a function to margins of an array

```
#X the array to be used.
#MARGIN a vector giving the subscripts which the function will be applied
over. 1 indicates rows, 2 indicates columns, c(1,2) indicates rows and
columns.
#FUN the function to be applied: use help(apply) to see more details.
In the case of functions like +, %*%, etc., the function name must be
backquoted or quoted.
#... optional arguments to FUN.
apply(dM, 1, mean) # calculate mean for each row of dM
apply(dM, 2, max) # find max for each column of dM
```

4. Control statements

```
if (expr_1) expr_2 else expr_3
# expr 1 must evaluate to a single logical value and the result of the
entire expression is then evident. The "short-circuit" operators && and
|| are often used as part of the condition in an if statement.
for (name in expr_1) expr_2
# name is the loop variable. expr 1 is a vector expression, (often a
sequence like 1:20), and expr 2 is often a grouped expression
while (condition) expr
repeat expr
break # used to terminate any loop, possibly abnormally. This is the only
way to terminate repeat loops.
next # used to discontinue one particular cycle and skip to the "next".
```

An example from Dr. Haran:

```
NUMREP=100
foo = rep(0, NUMREP)
for (i in 1:NUMREP) # repeat code below NUMREP times
{
   mysim=runif(50) # generate 50 random unif(0,1) and put it into mysim
   foo[i]=sum(mysim<0.3) # count number of values in mysim < 0.3
}
mean(foo)
```

5. Functions

1) Define new functions

```
### Function: exs
# input: a vector xs
# output: a vector, the ith element is the mean of xs[1:i]
exs <- function(xs){ # xs is a vector
   return(cumsum(xs)/1:length(xs))
}
exs(c(1,2,3,4)) #output (1, 1.5, 2, 3)
```

Comments: Make sure the name of your function have not been used by system.

2) Some Density Functions

```
rnorm( 1, 0, 2) # random generation function: draw a sample from Norm(0, 4), notice the third parameter is std. deviation.
qnorm(0.5, 0, 2 ) # quantile function: return x where CDF(x)=0.5
pnorm(0.5, 0, 2) # distribution function: return CDF(0.5)
dnorm(0.5, 0, 2) #density function: return Norm(0.5, 0, 2)
```

Prefix the names given in the following table by ‘`d`’ for the **density**, ‘`p`’ for the **CDF**, ‘`q`’ for the **quantile** function and ‘`r`’ for **simulation** (to draw samples).

Distribution	R name	Additional arguments
normal	`norm`	mean, sd
uniform	`unif`	min, max
Poisson	`pois`	lambda
Student’s t	`t`	df, ncp
geometric	`geom`	prob
binomial	`binom`	size, prob
gamma	`gamma`	shape, scale
beta	`beta`	shape1, shape2, ncp
F	`f`	df1, df2, ncp
Cauchy	`cauchy`	location, scale
chi-squared	`chisq`	df, ncp
exponential	`exp`	rate
hypergeometric	`hyper`	m, n, k
log-normal	`lnorm`	meanlog, sdlog
logistic	`logis`	location, scale
negative binomial	`nbinom`	size, prob
Weibull	`weibull`	shape, scale
Wilcoxon	`wilcox`	m, n

3) Some Other useful functions

`rm(obj1, obj2,…)` to get rid of objects that you don’t need any more(to save memory).
Use help() to get help, e.g. `help(rnorm)`,Or use `?rnorm`
`is.na(x)` to check whether `x` is `NA`. Other similar functions include, `is.matrix()`, `is.vector()…`
`as.integer(x)` to change `x` to integer(s), e.g, `as.integer(c(“1”,”2.2”))`. Other similar functions include, `as.character()`

6. Graphics

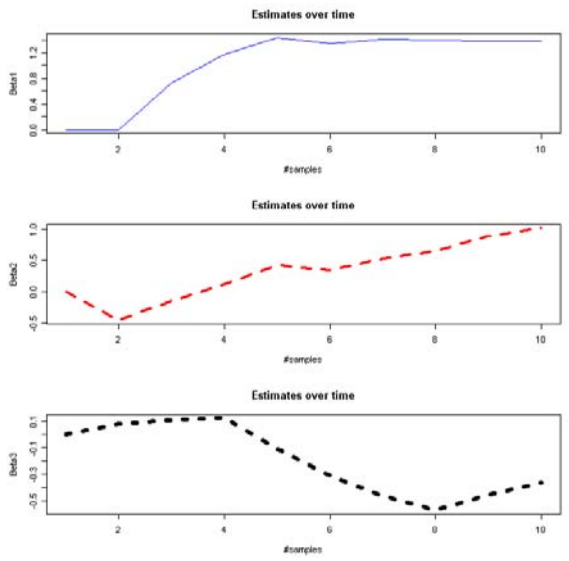
Suppose `sps` is an 11 by 3 matrix. The ith column is the samples from `π(βi|.)`, `i=1,2,3.` In real situation, the #samples may be very large, e.g. 1,000,000.

```
sps =
        [,1]          [,2]          [,3]
```

```
 [1,] 0.0000000  0.00000000  0.0000000
 [2,] 0.0000000  -0.90161301 0.1680712
 [3,] 2.1861774  0.46999325  0.1680712
 [4,] 2.5001347  0.92582000  0.1680712
 [5,] 2.5001347  1.61137820  -1.0257220
 [6,] 0.8315968  -0.00342978  -1.3438016
 [7,] 1.8384789  1.57492596  -1.3438016
 [8,] 1.2928159  1.57492596  -1.3438016
 [9,] 1.2350112  2.73327697  0.4794286
[10,] 1.4832795  2.23306257  0.4794286
[11,] 2.3580557  2.64352557  2.0544835
```

1) Estimates vs. #samples: plots to show the estimates change over time

```
par(mfcol=c(3,1)) # split the graphical window into 3 rows
#Parameters: col:color; lty: line type; lwd: line width; main: title;
plot.ts(exs(sps[,1]),col="blue", lty=1, lwd=1, main="Estimates over
time", xlab="#samples", ylab="Beta1")
plot.ts(exs(sps[,2]),lty=2,col="red", lwd=3, xlab="#samples",
ylab="Beta2")
title("Estimates over time") #same as main="Estimates over time" inside
the plot.ts( )
plot.ts(exs(sps[,3]),lty=3,col="black",lwd=5,main="Estimates over
time", xlab="#samples", ylab="Beta3")
savePlot("plot_file","jpg") # save plot to plot_file.jpg
```



You may also choose to use the function `estvssamp()` at http://www.stat.psu.edu/~mharan/batchmeans.R.

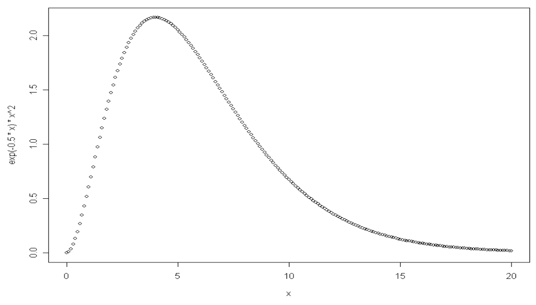
Notice: the other function `imse()` in *batchmeans.R* is also very useful. It applies consistent batchmeans procedure to estimate Markov chain standard errors (MCSE): `imse(`sampleVector`)`. Usually, people also use plots of “Markov chain standard errors (MCSE) vs. #samples to get more sense on convergence.

Comments: `plot()` can also draw plots for functions and provide you some sense of the shape of the functions. For example, in Monte Carlo sampling, we always have to make decision about proposal distribution for a target distribution, e.g. f(x) = C h(x). Usually C is an unknown normalization constant.

We can draw a plot for h(x) and choose a known distribution (usually, a distribution that is easy to draw samples) with the similar shape as proposal.

An example: check the shape of a function $g(x) \sim e^{-0.5x}x^2$, $(x>0)$.

```
x <- seq(0, 20, by=0.1)
fx <- plot(x, exp(-0.5*x)*x^2)
```

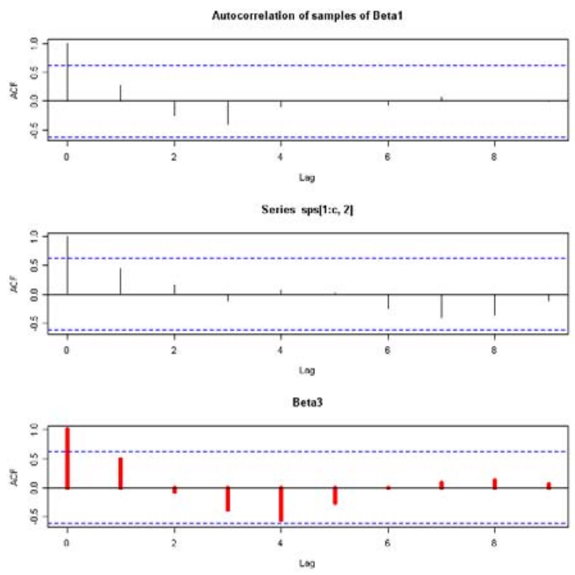


2) Autocorrelation plots

`acf()` computes (and by default plots) estimates of the autocovariance or autocorrelation. It is often being used to check the dependence of samples from Markov Chain (of course, the smaller is the better because we hope the draws are independent identical distributions (i.i.d)).

See definition of autocorrelation: http://en.wikipedia.org/wiki/Autocorrelation

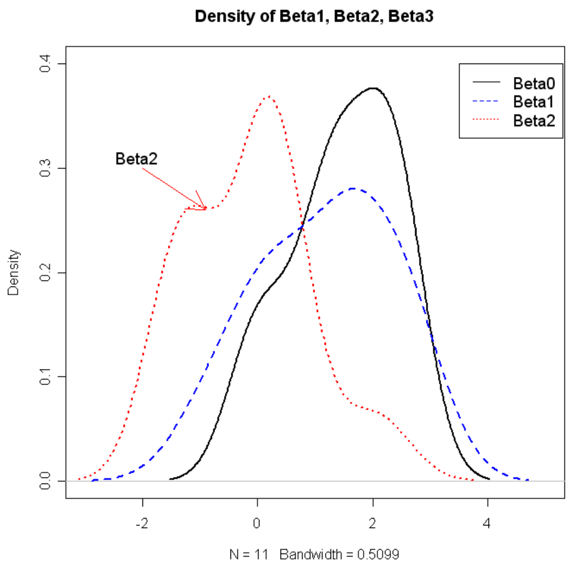
```
par(mfcol=c(3,1))
acf(sps[,1], main="Autocorrelation of samples of Beta1")
acf(sps[,2], xlab="Lag(Autocorrelation of samples of Beta2)")
acf(sps[,3], col="red", lty=1, lwd=5)     # parameters like col, lty are
still available
```



3) Smoothed estimated density plots

```
#Draw the smoothed estimated densities in one plot
```

```
par(mfrow=c(1,1))
###density: a function to get smoothed density from a vector
#xlim:x limits of the plot; ylim: y limits of the plot
plot(density(sps[,1]), xlim=c(-3,5),ylim=c(0,0.4), lty=1, lwd=2,
main="Density of Beta1, Beta2, Beta3")
lines(density(sps[,2]), col="blue", lty=2, lwd=2)
lines(density(sps[,3]), col="red", lty=3, lwd=2)
###set legend for the plot
#the first two parameters decide the left-up position of the legend
#cex: character expansion, to decide the font size, default=1
legend(3.5,0.4, c("Beta0","Beta1", "Beta2"), lty=c(1,2,3),
col=c("black","blue","red"), cex=1.2)
arrows(-2,0.3,-0.9,0.26, col="red")  # draw an arrow
text(-2.1,0.31,"Beta2", cex=1.2) # set text on the plot, the first two
parameters decide the position of text
```



4) Graphical windows

By default, there is only one graphical window, so high level graphic commands (`hist()`, `plot()`, `boxplot()`, `...`) produce a plot which replaces the previous one. To avoid this, use `win.graph( )` to open a separate graphic window. When you want to draw on a window, you need to set the window as active by using `dev.set(window ID)`

```
win.graph( )# opens a new graphic window and set it active.
dev.cur( )# gives the current active window,
dev.list( ) #lists all available graphics windows,
dev.set( ) #changes the active window,
dev.off( ) #closes the current graphic window,
graphics.off( ) #closes all the open graphics windows.
```

An example:

```
for (i in 1:3) win.graph()#open three graphic windows
dev.list()
windows windows windows
  2     3      4
```

```
dev.cur()
windows
4
dev.set(3) #change the current window to window 3
```

7. Performance Issue

1) Clock time

`Rprof():` a useful tool for profiling your code (determining which parts of your program are taking the most time. This can let you find bottlenecks (and errors) in your program).

```
Rprof() #begin tracking function time
Rprof(NULL) #stop tracking function time
summaryRprof() #Print summary or save to a file
```

An example:

```
Rprof()
## some code to be profiled
Rprof(NULL)
## some code NOT to be profiled
Rprof(append=TRUE)
## some code to be profiled
Rprof(NULL)
summaryRprof(filename = "Rprof.out") # save information to a file
```

2) Whenever possible, operating on the vector or matrix instead of operating on their elements is much faster.

In R(as well as in Matlab), please take advantage of the operation on vector and matrix. An example: `X` is an N by 2 matrix, X_{i1} and X_{i2} are the first and second elements in the ith row of `X`.
`Y` is a vector with length=N

Suppose we want to compute $\sum_{i=1}^{N} Y_i*(b_0 + b_1*X_{i1} + b_2*X_{i2})$, use the following code:

```
Y %*%(b0 + b1*X[,1] + b2*X[,2]) #It is fast and elegant
```

3) When using Markov chain standard error (MCSE) as a condition for stopping sampling, e.g.

```
sampleVector <- rep( 0, MAX_NUMIT )
it <- 1
while ( mcse >= 0.01 &&  it<=MAX_NUMIT ) { #stop sampling when MCSE<0.01 
or #samples>=MAX_NUMIT
    …
    mcse <- getMCSE( sampleVector ) # compute MCSE
    it <- it + 1
}
```

Since computation of MCSE is time-consuming, you can compute it every 100 or 1000 iterations instead of calculating it in each of iterations. E.g:

```
sampleVector <- rep( 0, MAX_NUMIT )
it <- 1
while( mcse > 0.01 && it<=MAX_NUMIT ) {#stop sampling when MCSE<0.01 or 
#samples=MAX_NUMIT
    …
    if( it %% 100 ==0 ) # compute MCSE every 100 iterations
       mcse <- getMCSE( sampleVector )
    it <- it + 1
}
```

4) Save and reuse previous results if you need them later instead of re-computing them, especially for those results taking long time to get.

Continue the example in 4): suppose you want to draw a plot to show the trends of MCSE over time after getting your samples. Instead of computing MCSE again, you can record the results in the iteration to a vector.

```
sampleVector <- rep( 0, MAX_NUMIT )
mcses <- rep(0, floor ( MAX_NUMIT / 100 ) ) # save MCSE to this vector
it <- 1
while( mcse > 0.01 && it<=MAX_NUMIT ) {#stop sampling when MCSE<0.01 or
#samples=MAX_NUMIT
    …
    if( it %% 100 ==0 ) {# compute MCSE every 100 iterations
       mcse <- getMCSE( sampleVector[1:it] )
       mcses[ it / 100 ] <- mcse  # save current MCSE to a vector
    }
    it <- it + 1
}
sn <- it -1 # calculate #samples
plot.ts( mcses[1:floor(sn / 100) ] ) #plot mcses to see its trends
```

Notice:

~~`plot( mcses)`~~ `# May be` **`WRONG`**`. If it stops sampling (when` ***`mcse<0.01`***`) before drawing` ***`MAX_NUMIT`*** `samples. That means some element(s) in the bottom of` ***`mcses`*** `haven't been assigned values (still 0, the initial value)`

~~`plot( mcses[1:(sn / 100) ] )`~~ `# Without floor(), it still works but not good`

5) Use log to make sure the intermediate results we get are not too large or too small to cause round-off error. For example, when computing the likelihood: `L=p(y1|betas)P(y2|betas)...p(yN|betas)`, if N is large, `L` could be too small to be handled by the computer accurately. For better accuracy, use log instead.

8. Frequent Bugs

1. When you use `while(it<MUNIT){…}`, make sure you increase `it` in each iteration. If you find your program doesn't stop at all, you may check this bug.
2. Make sure the precedence of operations, e.g., `samples[1:it-1]` is different from `samples[1: (it-1)]`, `":"` has a very high precedence(higher than `*` `/`, lower than `^`)
3. … (more from students)