MCMC Notes - Module 1 Introduction to R

Professor Duncan

January 24, 2014

based on Introducing Monte Carlo Methods with R by Robert and Casella

1 Introduction

1.1 What is R?

An open source statistical programming language

- Fast
- Free
- Basic commands and syntax easy to pick up
- Sharing packages and functions is easy, making it a favorite of the research community

2 Getting Started

See additional content or learning guide on Getting Started for most recent download/installation and companion program instructions.

```
> getwd()  #Shows the working directory, where objects will be stored

[1] "D:/My Documents/STAT 701/Module1R"

>
> #setwd("D:\\Documents and Settings\\My Documents")
```

Sets the working directory to a desired location. You should change this if you want to save your work By default, work is saved in a folder within the R program, but you can also control this through the save command on the workspace.

Note that R ignores anything it encounters after a pound symbol on a given line

3 R Objects

The assignment operator in R is either a left arrow = or an equal sign. Both will be used in this course

3.1 vectors

contain elements of just 1 type, either numeric, logical, or character

3.1.1 Numeric

```
> y = 18
>        y = c(-3.6,1.58)
>        y = 1:9
>        y = seq(from=0,to=10,by=0.02)
>           #seq() a function w/ args from, to,..
>                 #by argument gives distance
>        y = seq(0,10,by=0.02)
>                 #if know where args are, no need to specify name
>
>        y = seq(-3,25,length=6)           #length option
>        x = c(3,3,9,12,1)
>        y = seq(0,10,along=x)             #will return seq w/ length(x)
>
```

3.1.2 Character Vectors

```
>        y ="San Diego"
>        w = c("LA","NY","San Fran")
>
```

3.1.3 Logical Vectors

```
>        y =T
>        w = c(F,F,T)
>        x = 1:7
>        z = (x<5)
>
```

3.1.4 Selecting Portions of Vectors

- position in vector as positive integer

```
> y =c(18,32,15,-7,12,19)
> y[2]

[1] 32

> y[3:5]

[1] 15 -7 12

>
```

- excluding elements, position as negative integers

```
> y[-c(1,5,6)]

[1] 32 15 -7
```

- by element name

```
> names(y) = c("Joe","Bill","Karen","Helen","Ray","Paul")
>       y[c("Helen","Ray")]

Helen   Ray
   -7    12
```

```
>
```

- by logical conditions

```
> y[y<16]

Karen Helen   Ray
   15    -7    12
```

3.2 matrices, arrays, and factors

Matrices by default get filled down columns. When defining a matrix, specify the number of rows and the number of columns. Data can be filled in later.

```
> y =c(1:6)
> x =matrix(data=y,nrow=2,ncol=3)     #fill columns first (column major order) the default
> x

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

> z =matrix(data=y,nrow=2,ncol=3,byrow=T)   #fill rows first
> z

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

> dimnames(x) = list(c("r1","r2"),c("a","b","c"))
```

Matrix operators include +, -, and * which happen element-wise. For matrix multiplication, use %*%. Transpose is a t. I try not to name things t or c since those are commands.

```
> x*z

   a  b  c
r1 1  6 15
r2 8 20 36

> x%*%t(z)

   [,1] [,2]
r1   22   49
r2   28   64
```

vectors can be converted to matrices and vice versa

```
> vmat=as.matrix(y)
> dim(vmat)

[1] 6 1

> as.vector(z)

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

vectors can be bound together to form matrices

```
> v1=c(1:10)
> v2=rep(c(1,2),5)
> v3=rep(c(1,2),each=5)
> bindmat=cbind(v1,v2,v3)
> bindmat

      v1 v2 v3
 [1,]  1  1  1
 [2,]  2  2  1
 [3,]  3  1  1
 [4,]  4  2  1
 [5,]  5  1  1
 [6,]  6  2  2
 [7,]  7  1  2
 [8,]  8  2  2
 [9,]  9  1  2
[10,] 10  2  2
```

Arrays are extensions of matrices to higher dimensions

```
> y = c(1:8, 11:18, 111:118)
> x = array(y, dim = c(2,4,3))   # creates a 2 by 4 by 3 array
> x

, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    3    5    7
[2,]    2    4    6    8

, , 2

     [,1] [,2] [,3] [,4]
[1,]   11   13   15   17
[2,]   12   14   16   18

, , 3

     [,1] [,2] [,3] [,4]
[1,]  111  113  115  117
[2,]  112  114  116  118
```

Factors are categorical variables with a character descriptor.

```
> color =c("red", "red", "red", "green", "blue")
>                                    #character valued vector
> colors = factor(color, c("red","green","blue"))
> table(colors)  # table counting occurrences of colors

colors
  red green  blue
    3     1     1

>                #one use: when building models where "dummy" variables needed
>
```

every vector (and object) has 2 ”implicit” attributes, mode and length some modes, e.g. ”numeric”, ”list”, ”logical”, ”NULL” other attributes name, dimension,levels, class

```
> attributes(y)

NULL

> attributes(x)

$dim
[1] 2 4 3

> attributes(colors)

$levels
[1] "red"   "green" "blue"

$class
[1] "factor"
```

simple vector doesn't report anything with attributes

3.3 lists and data frames

Lists are simply collections of objects. Accessing values in them can be tricky

1. access to components
2. access to elements within components. Use dollar sign to preceed named componenents
3. unlist() converts a list to a vector, handy for printing out returned values from function
4. attach(what,...) makes components of list directly accessible detach undoes this

```
> x=list(one=c(18:36),two=c("AK","AL","AZ"),three=c(T,T,F,T),four=matrix(1:12,3,4))
>     x[[1]]          #by order

 [1] 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

>     x$one           #by name

 [1] 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

>     x[[1]][3:6]

[1] 20 21 22 23

>     x$one[3:6]

[1] 20 21 22 23

>     unlist(x)

   one1    one2    one3    one4    one5    one6    one7    one8    one9   one10
   "18"    "19"    "20"    "21"    "22"    "23"    "24"    "25"    "26"    "27"
  one11   one12   one13   one14   one15   one16   one17   one18   one19    two1
   "28"    "29"    "30"    "31"    "32"    "33"    "34"    "35"    "36"    "AK"
   two2    two3  three1  three2  three3  three4   four1   four2   four3   four4
   "AL"    "AZ"  "TRUE"  "TRUE" "FALSE"  "TRUE"     "1"     "2"     "3"     "4"
  four5   four6   four7   four8   four9  four10  four11  four12
    "5"     "6"     "7"     "8"     "9"    "10"    "11"    "12"
```

```
>     attach(x)
>     one

 [1] 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36

>     detach(x)
>
```

data frames- a special kind of list object; number of elements must be the same for all components. What we typically think of as a data set with several variables for each case. handy for regression modeling

```
>       muscle = rnorm(n=10,mean=3,sd=1)
>       sex = factor(rep(c("M","F"),c(6,4)))
>       speed  = rep(0,10)
>       speed[1:6] = rnorm(6,30-2*muscle[1:6],2)
>       speed[7:10] = rnorm(4,40-2*muscle[7:10],2)
>       mydata = data.frame(y=speed,x1=muscle,x2=sex)
>       temp = lm(y~x1+x2,data=mydata)
>       #summary(temp)
>       mydata$speed[8]

NULL
```

3.4 Applying functions to vectors and lists

some functions allow you to put in a vector as an argument and return a vector as output. Note that if you put two (or more) vectors in as arguments, they get used element by element.

```
> qnorm(c(.95, .975, .99))

[1] 1.644854 1.959964 2.326348

> qnorm(c(.95, .95, .99, .99), 0, c(1,2,1,2))

[1] 1.644854 3.289707 2.326348 4.652696
```

To apply a function to each row or column of a matrix, use apply. Note that for means and sums, rowMeans and rowSums are faster

```
> x=array(data=rnorm(120),dim=c(10,4,3))
> apply(x,2,max)

[1] 1.728790 1.766193 1.973377 2.468890

> apply(x,3,max)

[1] 1.973377 2.468890 1.695721

> mymat=matrix(data=sample(c(1:10), 800000, replace=T), nrow=10000)
> apply(mymat,2,sum)

 [1] 54917 54765 54430 54895 55643 55477 54505 55300 55148 55254 54899 55312
[13] 54780 54568 54998 54879 54858 55350 54916 54797 54826 55119 54891 55072
[25] 55235 55106 55192 54749 54766 54880 55421 55019 54868 55340 54951 55210
[37] 55436 55241 54811 54604 54926 55144 55141 55067 55058 55092 54908 54493
[49] 54632 54742 54960 55331 55299 55154 54909 55531 55166 55673 54917 55273
[61] 55196 55435 54604 54869 55170 54847 54723 55208 55367 55326 54518 55876
[73] 55388 54742 55203 55036 54227 55012 54918 54595
```

```
> colSums(mymat)

 [1] 54917 54765 54430 54895 55643 55477 54505 55300 55148 55254 54899 55312
[13] 54780 54568 54998 54879 54858 55350 54916 54797 54826 55119 54891 55072
[25] 55235 55106 55192 54749 54766 54880 55421 55019 54868 55340 54951 55210
[37] 55436 55241 54811 54604 54926 55144 55141 55067 55058 55092 54908 54493
[49] 54632 54742 54960 55331 55299 55154 54909 55531 55166 55673 54917 55273
[61] 55196 55435 54604 54869 55170 54847 54723 55208 55367 55326 54518 55876
[73] 55388 54742 55203 55036 54227 55012 54918 54595
```

To see that one is faster than the other, we can use system.time. The book instructs you to put the whole expression to be tested in parentheses, but I have had difficulty with multi-line programs without writing a wrapper, so taking a before and after time is fine.

```
> system.time(apply(mymat,2,sum))

   user  system elapsed
      0       0       0

> system.time(colSums(mymat))

   user  system elapsed
      0       0       0

> gc()

         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 218199 11.7     467875 25.0   350000 18.7
Vcells 752406  5.8    1873732 14.3  1872873 14.3

> start=Sys.time()
> apply(mymat,2,sum)

 [1] 54917 54765 54430 54895 55643 55477 54505 55300 55148 55254 54899 55312
[13] 54780 54568 54998 54879 54858 55350 54916 54797 54826 55119 54891 55072
[25] 55235 55106 55192 54749 54766 54880 55421 55019 54868 55340 54951 55210
[37] 55436 55241 54811 54604 54926 55144 55141 55067 55058 55092 54908 54493
[49] 54632 54742 54960 55331 55299 55154 54909 55531 55166 55673 54917 55273
[61] 55196 55435 54604 54869 55170 54847 54723 55208 55367 55326 54518 55876
[73] 55388 54742 55203 55036 54227 55012 54918 54595

> Sys.time()-start

Time difference of 0.01600003 secs

> gc()

         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 219029 11.7     467875 25.0   350000 18.7
Vcells 753879  5.8    1873732 14.3  1872873 14.3

> start=Sys.time()
> colSums(mymat)

 [1] 54917 54765 54430 54895 55643 55477 54505 55300 55148 55254 54899 55312
[13] 54780 54568 54998 54879 54858 55350 54916 54797 54826 55119 54891 55072
[25] 55235 55106 55192 54749 54766 54880 55421 55019 54868 55340 54951 55210
[37] 55436 55241 54811 54604 54926 55144 55141 55067 55058 55092 54908 54493
[49] 54632 54742 54960 55331 55299 55154 54909 55531 55166 55673 54917 55273
[61] 55196 55435 54604 54869 55170 54847 54723 55208 55367 55326 54518 55876
[73] 55388 54742 55203 55036 54227 55012 54918 54595
```

```
> Sys.time()-start

Time difference of 0.006999969 secs
```

For lists, lapply is the command you need

```
> mylist=list(c("Red", "Green", "Red", "Yellow", "Blue", "Blue", "Yellow"), c("Brown", "Brown", "Blue",
> lapply(mylist,table)

[[1]]

  Blue  Green    Red Yellow
     2      1      2      2

[[2]]

 Blue Brown Green
    1     3     1
```

When you want to call a function multiple times with the same input, use replicate

```
> myprops=replicate(100, rbinom(100,1000,prob=.48))/100
```

4 Probability distributions in R

We have already seen some examples of generating random variables in R. Binomial, uniform, and normal are commonly used. Let's look at the prefixes.

- d gives the density, the height of the pdf for continuous variables, the pmf for discrete variables
- p gives the cumulative distribution; the left tail area
- q gives the value with a particular left tail area
- r generates random values

The collection of distributions available in the base package include beta, binom, cauchy, chisq, exp, f, gamma, geom, hyper, lnorm, logis, norm, pois, t, unif, weibull. Check the help menu to determine which arguments are needed and how the distribution is parameterized. For instance, normal takes sd, not variance.

```
> help(rnorm)
> dnorm(seq(-3,3,by=.2))

 [1] 0.004431848 0.007915452 0.013582969 0.022394530 0.035474593 0.053990967
 [7] 0.078950158 0.110920835 0.149727466 0.194186055 0.241970725 0.289691553
[13] 0.333224603 0.368270140 0.391042694 0.398942280 0.391042694 0.368270140
[19] 0.333224603 0.289691553 0.241970725 0.194186055 0.149727466 0.110920835
[25] 0.078950158 0.053990967 0.035474593 0.022394530 0.013582969 0.007915452
[31] 0.004431848

> pnorm(-2.6)

[1] 0.004661188

> qnorm(.99,100,15)

[1] 134.8952

> rnorm(10,50,10)

 [1] 50.50997 52.09389 61.70814 68.29775 44.43680 43.85873 43.24223 44.52984
 [9] 47.50102 30.03142
```

5 Basic Statistics

The base package has built-in functions for most well-established statistical procedures and tests.

For example, to do a chi-square test of association, we use chisq.test. What is returned is a list.

```
> x=matrix(data=c(26,20,50,15),nrow=2)
> x
```

```
     [,1] [,2]
[1,]   26   50
[2,]   20   15
```

```
> mytest=chisq.test(x)
> #mytest
> myfisher=fisher.test(x)
> #myfisher
```

When looking at output, a function that is useful is round.

```
> x=matrix(data=rnorm(100),nrow=10)
> x
```

```
             [,1]        [,2]        [,3]        [,4]       [,5]        [,6]
 [1,] -0.21357484  1.22863727  0.68680024 -0.70397405  0.0738881 -0.53349268
 [2,] -0.83356824  0.53234659  1.86091310  0.08740758  0.9723068  1.11512665
 [3,]  0.02730271  0.07208491  0.21429711  0.31185128  0.4595209  0.07469838
 [4,]  0.35313824 -1.57387447  0.69735173 -0.79244859 -0.6104283 -0.09492775
 [5,]  1.97042147 -0.98975011  0.07476312  0.22819217 -1.4867894 -0.81201233
 [6,]  0.10987866  0.45965625 -0.41836960  0.44337083 -0.5215420  0.08370412
 [7,]  1.26597849  0.36613139  0.72355719 -0.78234821 -0.1640852 -0.16283037
 [8,] -1.38895147  1.05589112 -0.52275909 -1.66314928 -0.4122639 -0.01965933
 [9,] -0.32391591 -0.32545257  0.16439201  1.03324067 -0.1934290 -1.49577334
[10,]  0.77648659  1.82675258  0.56528412 -0.93901354  1.0658094 -0.70271669
             [,7]       [,8]       [,9]      [,10]
 [1,] -0.46714502  0.3762638  0.1827928  1.0904863
 [2,] -1.01894271 -0.1497105 -0.1966202 -0.4785819
 [3,]  0.27057598 -0.8383322  0.4190293  0.7055529
 [4,] -1.20810639 -1.0335433 -1.7252401 -0.1712577
 [5,]  1.04125152 -0.0295851  0.3652443 -0.2488008
 [6,]  0.39381631  1.2454407  2.0490606  0.4654192
 [7,]  0.08061379  0.5576337 -0.9414589 -1.4669438
 [8,]  0.34500149 -0.2492008  0.6208786 -1.1959525
 [9,] -1.87853032  0.6193493 -0.9126042 -0.4318428
[10,] -0.69231086 -0.5874113  0.3869259 -1.1540523
```

```
> print(round(x,3))
```

```
       [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]   [,9]  [,10]
 [1,] -0.214  1.229  0.687 -0.704  0.074 -0.533 -0.467  0.376  0.183  1.090
 [2,] -0.834  0.532  1.861  0.087  0.972  1.115 -1.019 -0.150 -0.197 -0.479
 [3,]  0.027  0.072  0.214  0.312  0.460  0.075  0.271 -0.838  0.419  0.706
 [4,]  0.353 -1.574  0.697 -0.792 -0.610 -0.095 -1.208 -1.034 -1.725 -0.171
 [5,]  1.970 -0.990  0.075  0.228 -1.487 -0.812  1.041 -0.030  0.365 -0.249
 [6,]  0.110  0.460 -0.418  0.443 -0.522  0.084  0.394  1.245  2.049  0.465
 [7,]  1.266  0.366  0.724 -0.782 -0.164 -0.163  0.081  0.558 -0.941 -1.467
 [8,] -1.389  1.056 -0.523 -1.663 -0.412 -0.020  0.345 -0.249  0.621 -1.196
 [9,] -0.324 -0.325  0.164  1.033 -0.193 -1.496 -1.879  0.619 -0.913 -0.432
[10,]  0.776  1.827  0.565 -0.939  1.066 -0.703 -0.692 -0.587  0.387 -1.154
```

regression is performed using the lm (linear model) command

```
> help(lm)
> x=seq(10,20,by=.5)
> y=20+3.5*x+rnorm(21)
> mylm=lm(x~y)
> #mylm
> #summary(mylm)
```

6 Graphical Facilities

RStudio makes the management of plots much easier. It keeps a plot history and provides a menu for exporting plots. Simple scatter plots are made with a call to plot(x,y)

```
> wellvisitt=c(0,2,4,6,9.75,12,18,24)
> childweight=c(7.5,13.25,16.25,17+11/16,20+1/16,20+15.5/16,23+1/8 ,25+9/16)
> childweight2=c(9+1/8, 13+9/16, 17+33/16, 19+15/16, 22+6/16, 23+13.5/16, 25+6/16, 29)
> plot(wellvisitt,childweight)
```

To add another set of points to the same plot, use points or lines
Some plotting commands/parameters to know

- pch, plotting characters
- type, l for lines
- lty, lwd, line type and line width
- col, color. For basic colors, pie(rep(1,8), col=c(1:8)) To see a list of all color names, type colors()
- xlim, ylim IF you will be plotting other things, you may need a different window than what is chosen by default for your first plot
- cex enlarges plot features. Available separately for axes, margins
- main, xlab, ylab Titles and axis lables
- legend, added later. I usually choose bty=”n” to not have a box

```
> plot(wellvisitt,childweight, type="l", lty=1, col=4, lwd=3, ylim=c(5,30), xlab="Months", ylab="Weight"
> lines(wellvisitt, childweight2, lty=2, col=6, lwd=3)
> legend(x="topleft", col=c(6,4), lty=c(2,1), lwd=c(3,3), legend=c("IMD", "MJD"))
```

7 Writing new R functions

This example illustrates how to compute body mass index (BMI) using height (in inches) and weight (in pounds) as arguments. The input arguments are given in parentheses and the commands to be carried out within the brackets. Output of the function appears in the return statement; this is not necessary for single-line functions.

```
> ## BMI is measured in kilograms per square meter
> ## convert height and weight to metric and divide weight by squared height
> BMI=function(h,w){0.45455*w/(.0254*h)^2}
> BMI(71,180)

[1] 25.15765
```

An alternative formulation of this function

```
> BMI2=function(h,w){
+   mh=h*.0254
+   mw=w*.45455
+   return(mw/mh^2)
+ }
> BMI2(71,180)

[1] 25.15765
```

If you want to return a named list, provide names before return

```
> BMI3=function(h,w){
+   mh=h*.0254
+   mw=w*.45455
+   bmi=mw/mh^2
+   output=c(mh, mw, bmi)
+   names(output)=c("Metric Height", "Metric Weight", "BMI")
+   return(output)
+ }
> BMI3(71,180)

Metric Height Metric Weight           BMI
      1.80340      81.81900      25.15765
```

8 Input and Output and Administration of R objects in R

If you have a vector of numbers copied to the clipboard, you can read it in with scan. Press CTRL-C to paste and CTRL-Z to end the reading

```
> x=scan()
```

The commands read.table and read.csv are handy for bringing in data sets. In R Studio, you can also choose the Workspace menu and import dataset from text file to bring a data set into your workspace.

write, save, and dput are ways to save files outside of the workspace. Save keeps the objects in R-readable only format. To interface with other programs use write or dput. You may find it handy to keep a whole project/workspace saved together. All objects can be loaded together from the workspace menu. A saved workspace can be loaded from the workspace menu too.

9 The mcsm package

The authors wrote a package for their books. To access those data sets and functions, go to the packages menu and check the box next to mcsm. In scripts, if you want to load a package that you have installed, use the command library(mcsm). To use a data set from the package, use the attach command.

10 Number Storage and Representation

Underflow - Too small to be distinguished from zero Overflow - too large for the computer to store