Computational Statistics
General Background Information

• Brief introduction to R.

• R functions.

• Monte Carlo methods in statistics.

• Uniform random number generation.

• Random number generation in R.
A Little Bit on R

R is based on S developed at Bell Labs in the 1970s by John Chambers. R (and S, and S-Plus, another packaged based on S) is an interactive, interpretive, function language.

R is a graphically-oriented data analysis system and object-oriented programming language.

Available on Unix, Linux, MS Windows, and Mac systems. Documentation exists in several volumes, and in an on-line help system.

Get it from http://www.r-project.org/
(Do a Google search for "R".)
R Fundamentals

Most statements are of the form

\texttt{variable <- function(...)}

or

\texttt{function(...)}

R is case sensitive.
R Fundamentals

R objects:

- variables,
- vectors,
- lists,
- matrices,
- arrays,
- formulas,
- factors (for statistical applications),
- data frames (for statistical applications),
- fits (for statistical applications)

R has an extensive set of functions, i.e. verbs. The specific meaning depends on the class of the object to which it is applied.
Objects are built by constructor functions:

```r
> A <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow=3)
```

This object is a matrix and can participate in matrix multiplication and so on. Typing A results in

```
> A =
1 4 7
2 5 8
3 6 9
```
R Syntax, Functions

All actions are “functions”. A function name is followed by a set of parentheses to enclose arguments.

help() or help(plot), e.g. or ?plot

q()

The syntax is somewhat similar to C++, except that there are no special delimiters for statements; statements continue as necessary to complete the function.

Statements may be grouped by { and }.

Blanks are significant only when it makes sense for them to be.

The language is not strongly typed.

Numbers are generally single precision floating point numbers.
Examples

> x <- c(2, 5, 3)
> x
> z1 <- 3 + 4*i
> z2 <- complex(real=3, imaginary=4)
> z <- scan(file="example.dat")
> y <- scan()
7: 23. 23. 28. 28. 27. 26.
13: 24. 22. 27. 26. 28. 28.
25:
> a <- 0:10
> b <- 0:10/10
> c <- seq(from=0, t=1, by=.1)
> plot(y,type="1")
General Design

R deals with functions. This dictates the syntax – no statement delimiters (but does use { and }).

Comments begin with #.

No fixed naming conventions; the wise user, however, adopts mnemonic conventions. Use periods to represent components.

Various ways of extracting components. $ component operator:
object$member

See objects with objects().
R Matrices

A <- matrix(c(1, 3, 5, 2, 4, 6), nrow=3)
x <- c(3, 4)
z <- A[2, ]
w <- A[2, 1:2]
v <- c(1, 2, 1)
b <- A%x;
cc <- v%*%A
D <- cbind(A, v)
E <- solve(D)
f <- solve(D, v)
H <- D%x
L <- D*D
M <- D^2
objects()
rm(...)
Other Operators and Functions

<, <=, >, >=, ==, !=

&, |, !

sin, cos, ...

abs, Arg, sqrt, Re, Im, Conj,
round, trunc, floor, ceiling, sign,
%%,
exp, log, log10

t, crossprod, solve,
sink("filename")
sink()

Plus lots of functions for statistical analyses.
Packages in R

There are many “packages” that extend the functionality of R.

Each package may contain several functions in a given area of application or of analysis.

These must first be “installed” using files that you usually get from some online respository. Once you choose a mirror site from which to obtain the files, you choose the specific package from a drop-down list.

Once a package is installed, it must be “loaded” in any R session in which you need a function from the package.

A package can be loaded by using the drop-down menu under the “Packages” button, or else it can be loaded by means of the library function in R.
Programming

Conditionals:

```plaintext
if(x >= 3)
{
    y <- 2
    z <- 4
} else if (x <= 1) ## note how the command must be continued
{
    y <- 1
} else
    y <- 5
```
**Conditionals**

Conditionals resolve to a numeric value of 0 for false and 1 for true.

They can be operated on.

```r
x <- 3
y <- 5*(x>2) + 4
```

Yields a value of 9 for y.
Loops

Suppose the vector \( x \) is given.

\[
\begin{align*}
n1 & \leftarrow \text{length}(x)-1 \\
y & \leftarrow \text{rep}(1,n1) \\
\text{for } (i \text{ in } 1:n1) \{ \\
 & \quad \text{if}(x[i+1]<x[i]) \quad y[i] \leftarrow -1 \\
\}
\end{align*}
\]

or

\[
\begin{align*}
y & \leftarrow \text{rep}(1,n1) \\
i & \leftarrow 1 \\
\text{while } (i \leq n1) \{ \\
 & \quad \text{if}(x[i+1]<x[i]) \quad y[i] \leftarrow -1 \\
 & \quad i \leftarrow i+1 \\
\}
\end{align*}
\]
Loops are inefficient in R.
Try to implement as statements involving vectors.

\[ y <- \text{ifelse}(x[-1] < x[1:n1], -1, 1) \]

Now, let A be a matrix. Instead of

\[
\begin{align*}
y & \leftarrow \text{numeric}(\text{dim}(A)[2]) \\
\text{for } (i \text{ in } 1:\text{dim}(A)[2]) \{ \\
\quad y[i] & \leftarrow \text{sum}(A[,i]) \\
\}
\end{align*}
\]

Use apply:

\[ y \leftarrow \text{apply}(A, 2, \text{sum}) \]
R Functions for Graphics

plot
plot.factor
pairs
brush
hist
stem
barplot
persp
faces
stars
matplot

And others. The actual appearance of the graph depends on the class of the object.

Arguments for functions may be required or optional. Most required ones are positional, many optional ones are keyword.
Annotating R Graphics

Titles and other text in R graphics can be formatted using `expression()`.

```r
plot(x[,1],x[,2],xlim=c(min(x[,1],x[,2]),max(x[,1],x[,2])),
     ylim=c(min(x[,1],x[,2]),max(x[,1],x[,2])),
     xlab=expression(italic(x)[1]),ylab=expression(italic(x)[2]),
     pch=3)

plot(1:10, 1:10)
text(4, 9, expression(hat(beta) == (X^t * X)^{-1} * X^t * y))
text(4, 8.4, "expression(hat(beta) == (X^t * X)^{-1} * X^t * y)",
     cex = .8)
text(4, 7, expression(bar(x) == sum(frac(x[i], n), i==1, n)))
text(4, 6.4, "expression(bar(x) == sum(frac(x[i], n), i==1, n))",
     cex = .8)
text(8, 5, expression(paste(frac(1, sigma*sqrt(2*pi)), " ",
     plain(e)^{frac(-(x-mu)^2, 2*sigma^2)})),
     cex = 1.2)
```
Controlling the Environment

The environment includes such things as the active objects (in a "library"), the working directory, and graphics layouts.

library(MASS)
oldpar<-par(no.readonly=T)
par(mfcol=c(2,3)) ### gives two rows and three columns
...  
par(oldpar)
Using R

Much of the functionality of R can be accessed from a GUI.

The GUI is platform-dependent.

It is generally better to used typed commands than to use the buttons and menus of a GUI.

A list of typed commands is called a “program”, a “script”, or a “chunk”. The advantage of typed commands is that they can be saved and reused.

They can also be read and inspected by someone else.
Using R; Writing R Scripts

There is an emacs-based editor that comes with the standard R distribution package.

The standard GUI provides the ability to execute a highlighted portion of R code in the R editor.

Alternatively, any text editor can be used to write R code. Execution, however, generally requires that text be moused into the R console window.

The advantage of a separate text editor is that they can be more intelligent and even more R-aware than the standard R editor. (Keywords can be colored, etc.)

RStudio is one of the best editors. (For many years, I have used Crimson Editor on MS Windows and emacs with ESS (“emacs speaks statistics”) on other platforms.)
Using R; Dynamic Documents

“Reproducible research” requires that all text, programs, and output be packaged.

An easy way of doing this is to write regular R code as “chunks” in the text of the research document, and use knitr to control execution of R and weave the output into the compiled document. (This practice first became easy using Sweave. knitr by Yihui Xie is a more recent implementation of the concept.)
Using R; Saving Graphics Files

There are various commands to save a graphics file. They are, however, platform-dependent.

Graphics files can be save in various formats, .jpeg, .eps, etc.

For example, in Microsoft Windows,

```r
graphdir<-'c:/MyGraphs/'
...
savePlot(filename=paste(graphdir,'plot02',sep=''),type='eps')
```

saves the active graphics window ("device") in a file called plot02.eps in the directory c:/MyGraphs/.
R Functions for Standard Distributions

Functions

• d – density

• p – cumulative probability

• q – quantile

• r – random number generation
R Functions for Standard Distributions

Distributions

- beta
- f
- gamma
- norm
- t
- unif
- etc.

Examples: \texttt{rnorm(25, 100, 8)} generates 25 N(100,64) numbers
\texttt{qf(.95, 5, 10)} the .95 quantile of an F with 5 and 10 degrees of freedom.
The Object Orientation of R

Most of the functions of R are overloaded and are data-driven.

A new class can be defined by defining a constructor function that gives an identifier to the class:

```r
factor <- function(x, levels = sort(unique(x)),
  labels = as.character(levels)) {
  y <- match(x, levels)
  names(y) <- names(x)
  levels(y) <- labels
  class(y) <- "factor"
  y
}
```

The function `factor` will construct an object of class "factor".
The Object Orientation of R

The function \texttt{mode} can be used to coerce an object to a particular type:

\begin{verbatim}
mode(x) <- "factor"
\end{verbatim}

The function \texttt{inherits} can be used to determine if an object is of a particular class. Functions specific to the class can be written.
Monte Carlo Studies

A Monte Carlo study uses a computer experiment to evaluate procedures.

Often the procedure is a statistical method for inference.

It may be an estimation method or a statistical test.

The performance of a statistical method depends on the underlying distribution.

Performance:
• bias
• variance
• power
• significance level
• etc.
Performance of Statistical Methods

In mathematical statistics, we work out such things as the bias, the variance, etc. of a statistical method, based on an assumed underlying distribution.

For complicated distributions, or for mixtures of distributions, or for distributions with outliers, the mathematical analysis may be extremely difficult or impossible.

There are many possible scenarios to investigate; that is, there are many possibilities for the underlying distributions.

Monte Carlo studies provide an alternative.

Monte Carlo studies use simulation.
An Example: A Problem for a Research Statistician

Given a random sample from a mixture of two normal distributions, \( p N(\mu, \sigma^2) \) and \( (1 - p) N(\mu, k^2 \sigma^2) \). What is the “best” estimate of \( \mu \)?

Two possibilities would be \( \bar{x} \), the average of the sample, and \( \bar{x}_W(\pi) \), the Winsorized sample mean (set the largest \( \pi\% \) of the values to the largest of the middle \( (1 - 2\pi)\% \) and the smallest \( \pi\% \) of the values to the smallest of the middle \( (1 - 2\pi)\%)\).

How could we decide which is better, \( \bar{x} \) or \( \bar{x}_W(\pi) \)?

If both estimators are unbiased, the one with smaller variance is the better one.

What are the variances?

They’re hard to work out.
Monte Carlo Experimentation

We estimate the variances by Monte Carlo methods.

Get a (pseudo-) random sample from the assumed distribution, compute \( \bar{x} \) and \( \bar{x}_W(\pi) \).

Do many times, so we get a sample of many \( \bar{x} \)'s and \( \bar{x}_W(\pi) \)'s.

We compute their sample variances. This is an estimate of the true variances of \( \bar{x} \) and \( \bar{x}_W(\pi) \) for the case in which the underlying population distribution is the one simulated.
A Monte Carlo study is an experiment.

The Monte Carlo study should adhere to the same high standards of any scientific experimentation:

- control
- reproducibility
- efficiency
- careful and complete documentation
Monte Carlo Experimentation

In simulation, *control*, among other things, relates to the fidelity of a *nonrandom* process to a *random* process. The experimental units are only simulated using computer programs.

Questions about the setup of the computer experiment must be addressed. (How good are the pseudo-random number generators? etc.)

Likewise, *reproducibility* is predicated on good RNG’s (or else on equally bad ones!)

The principles of good statistical design can improve the efficiency.
In experimentation, we have some *response* of interest.

We identify various *factors* that may affect the response.

Different values of the factors are called *levels*.

In statistical design, we often distinguish the factors as *blocking* factors or *treatment* factors.

The purpose may be to determine which treatment yields the best response for various blocking factors.

(Statistical design and analysis of experiments is one of the major fields of statistics. It is at the heart of analysis of variance (ANOVA), in which linear models are used to study the relationship of the factors to the response.)
Monte Carlo Experimentation

A statistical experiment involves a number of experimental units. They are generally chosen randomly.

Each combination of levels of factors constitute a cell in the experiment, and the number of experimental units used in each cell determines the precision with which the response may be measured for that combination of factor levels.

Often the numbers are chosen the same for all cells, but sometimes it makes more sense to have more units in one cell than in others.

The number of experimental units in each cell in a Monte Carlo experiment is Monte Carlo sample size.
Statistical Design and Analysis of Experiments

The principles of statistical design and analysis apply just as much to a Monte Carlo study as they do to any other scientific experiment.

Monte Carlo experimentation is one of the few chances that statisticians get to design an experiment, collect the data, and analyze the data.
A Monte Carlo Experiment to Study the Variances

In the study to compare the two estimators of $\mu$ in the

$$pN(\mu, \sigma^2) + (1 - p)N(\mu, k^2\sigma^2)$$

mixture distribution, the “treatments” are the estimators; the ordinary sample mean $\bar{x}$, and the Winsorized sample mean $\bar{x}_W(\pi)$. The Winsorized sample mean has a parameter, $\pi$.

The possible factors are $p$, $\mu$, $\sigma^2$, $k$, and $n$, the sample size.

Can these be standardized?

What other factors may be of interest?

The design of the experiment determines levels of the factors.
Reporting a Monte Carlo Experiment

Sometimes, because of all of the factors involved in the experiment, it is difficult to present a clear description of experiment and the results.

The description of the experiment must include all relevant information, including exact descriptions of what was computed, and what software was used.

Tables and graphs can help to present the results. They are not substitutes for clear summary descriptions, however.

The numeric results are estimates. Reports of statistical estimates must always be accompanied by some indication of their variances. In tables showing Monte Carlo results, a good way of doing this is to provide the sample standard deviation in parentheses beside each reported value.
Monte Carlo Studies in the Published Literature

Over half of the research articles in theoretical statistics include Monte Carlo studies of the performance of statistical methods.

One of the most important research journals in statistics is *Journal of the American Statistical Association* (JASA).

You can access this journal at the GMU library website.

I will go through this from the top just to show you how to get to articles.
Examples of Monte Carlo Studies in the Published Literature

I looked through some recent issues of *JASA* for an article with an Monte Carlo study that everyone should be able to understand.

There were many – most involved some more advanced statistics that I didn’t want to get into now (some topics such as bootstrapping, lasso, etc., we’ll cover later).

One that seems fairly simple happens to be on time series, but you don’t need to know much about time series to understand it.

The question is whether the mean of a time series $X_1, X_2, \ldots$ is constant.

It involves use of a “self-normalization” procedure compared to a KS-type procedure.

Let’s access it.
Random Numbers

Where can you get “random numbers”?

Start with any integer $x_0$ (a “random” one) and then let

$$x_i \equiv \alpha x_{i-1} \mod m,$$

with

$$0 \leq x_i \leq m - 1.$$

Then take

$$u_i = x_i / m.$$

If $\alpha$ and $m$ are properly chosen, the $u_i$'s will “look like” they are randomly and uniformly distributed between 0 and 1.
The method is called a linear congruential generator.

Note that the congruential method is

1. deterministic (hence, at best “pseudo-random”) and
2. cyclic (will repeat after at most $m - 1$ iterations).

The length of the cycle is called the period.

What makes a random number generator “good”?

There are many ways of testing generators for good distributional properties, but, unfortunately, bad generators often pass tests.

For any deterministic generator, a test could be constructed that it would fail.
Simple linear congruential generators will yield about $\sim 2 \times 10^9$ different ‘random numbers’, and then repeat.

Need methods with larger periods.
Other Uniform Generators

There are several other types of uniform generators.

- Lagged Fibonacci:

\[ x_i \equiv (x_{i-j} + x_{i-k}) \mod m. \]

Requires an initial sequence, rather than just a single seed. If \( j, k, \) and \( m \) are chosen properly, the lagged Fibonacci generator can perform well. If \( m \) is a prime and \( k > j \), the period can be as large as \( m^k - 1 \).

\[
\begin{align*}
  j &= 24 \\
  k &= 55 \\
  m &= 2^{32}
\end{align*}
\]

Period of about \( 10^{25} \)
Feedback Shift Register Methods

\[ \begin{array}{cccccccccccccccccccccc}
0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\
\end{array} \]
Feedback Shift Register Methods

The way the feedback register method works is as a polynomial mod 2 and the results are interpreted mod 1.

Feedback shift register methods and generalized feedback shift register (GFSR) methods are based on a sequence

\[ a_k \equiv (c_p a_{k-p} + c_{p-1} a_{k-p+1} + \cdots + c_1 a_{k-1}) \mod 2. \]

Successive \( p \)-tuples (or in a ‘generalized’ form, \( l \)-tuples formed from subsequences) are taken as the binary representation of the random numbers.
Properties of Feedback Shift Generators

The period can be very long.

Variations include the number of “taps” (multiple recursion) and “twisting” of bits by multiplication by a binary matrix.

In a common implementation with \( p = 521 \), with \( c_{521} = c_{32} = 1 \) and all other \( cs \) equal to zero, the period is \( 2^{521} \).
Twisting the Bits

Another way of addressing the problems of two-tap shift registers has been proposed by Matsumoto and Kurita (1992, 1994) and Matsumoto and Nishimura (1998, 2002).

They modify the GFSR by “twisting” the bit pattern in $x_{i-p+q}$.

This is done by viewing the $x$s as $l$-vectors of zeros and ones and multiplying $x_{i-p+q}$ by an $l \times l$ matrix $A$.

The recurrence then becomes

$$x_i = x_{i-p} \oplus A x_{i-p+q}. \tag{1}$$

They call this a twisted GSFR generator.

The choice of $l$, $p$, $q$, and $A$ determine the quality of the generator.
The Mersenne Twister

The twisted GSFR generator called the “Mersenne Twister” is probably the best uniform random number generator.

Uniform in very high dimensions.

The period is $2^{19937} - 1$.

Matsumoto and Nishimura maintain a webpage at http://www.math.sci.hiroshima-u.ac.jp/ m-mat/MT/emt.html
Random Number Generation in R or S-Plus

S-Plus and R do not use the same random number generators. *Monte Carlo studies conducted using built-in random number generators in one system cannot reliably be reproduced exactly in the other system with its built-in generators.*

Several choices for the basic uniform generator are available in R. The function

\[\text{RNGkind}\]

can be used to choose the generator.

The default is the Mersenne Twister.

The chosen (or default) basic uniform generator is used in the generation of nonuniform variates.
Controlling the State of the Generators

R uses an object called `.Random.seed` to maintain the state of the random number generators.

In R, `.Random.seed` also maintains an indicator of which of the basic uniform random number generators is the current choice.

Whenever random number generation is performed, if `.Random.seed` does not exist in the user’s working directory, it is created.

If it exists, it is used to initiate the pseudorandom sequence and then is updated after the sequence is generated.

Setting a different working directory will change the state of the random number generator.
Random Number Generation in R

The function `set.seed(i)` provides a convenient way of setting the value of the `.Random.seed` object in the working directory to one of a fixed number of values.

The argument `i` is an integer between 0 and 1023, and each value represents a state of the generator, which is “far away” from the other states that can be set in `set.seed`.

To save the state of the generator, just copy `.Random.seed` into a named object, and to restore, just copy the named object back into `.Random.seed`

```r
oldseed <- .Random.seed  # save it
y <- runif(1000)         # get sample, analyze, etc.
...                      # ...
.Random.seed <- oldseed  # restore seed
yagain <- rnorm(1000)    # will be the same as y
```
Random Number Generation in R

A common situation is one in which computations for a Monte Carlo study are performed intermittently and are interspersed with other computations, perhaps broken over multiple sessions.

In such a case, we may begin by setting the seed using the function `set.seed(i)`, save the state after each set of computations in the study, and then restore it prior to resuming the computations.

```
set.seed(10)  # set seed at beginning of study
...  # perform some computations for the Monte Carlo study
MC1seed <- .Random.seed  # save the generator state
...  # do other computations
.Random.seed <- MC1seed  # restore seed
...  # perform some computations for the Monte Carlo study
MC1seed <- .Random.seed  # save the generator state
```
Random Number Generation in R

In order to avoid the side effect of changing the state of the generator, when writing a function in R, the user can preserve the state upon entry to the function and restore it prior to exit.

The assignment

`.Random.seed <- oldseed`

does not work if it occurs within a user-written function in R.

Within a function, the assignment must be performed by the `<<-` operator.
Random Number Generation in R

A well-designed R function that invokes a random number generator would have code similar this.

```r
fun <- function(...){
  oldseed <- .Random.seed  # save seed on entry
  ...
  .Random.seed <<- oldseed  # restore seed on exit
}
```