A brief introduction to **R**

PRISM brownbag

Eleonora Mattiacci Jason Morgan

Department of Political Science
The Ohio State University

October 20, 2010

What is **R**?

- Generally speaking, "**R** is a language and environment for statistical computing and graphics" (source: **R** homepage).
- **R** is a programming language.
- **R** is a statistical computing environment.

Advantages to using **R**

- **R** is *free* and *open source* software (and always will be) and can be run on a variety of platforms including Apple Mac OS X, Microsoft Windows, GNU/Linux, and Unix.
- **R** is powerful and comes without limitations.
- **R** is easily extensible, and more than 2100 user-contributed packages are available—for free—on CRAN.
- **R**-core is under rapid and constant development.
- Detailed and extensive documentation is available (and free), and there are multiple mailing list dedicated to answering questions.

Obstacles to using **R**

- Learning curve: **R** is a programming language; consequently, there is something of a learning curve.
- The GUI is (intentionally) limited.
- There is limited commercial support.

Online manuals

Installation and administration manual can be found at the following site:

http://cran.r-project.org/doc/manuals/R-admin.html

Microsoft Windows and Apple Mac FAQs can be found at the following links:

http://cran.r-project.org/bin/windows/base/rw-FAQ.html
http://cran.r-project.org/bin/macosx/RMacOSX-FAQ.html

A list of available packages is available here:

http://cran.r-project.org/web/packages/

Installing new packages

Packages are most easily installed from the command line. For example

```
> install.packages("car")
```

will install the `car` package after prompting for you to select a mirror. (For best results, pick a mirror that's geographically close.) Multiple packages can be installed at the same time by providing `install.packages()` with a vector of package names. For example

```
> install.packages(c("car", "apsrtable", "Hmisc"))
```

will install the `car`, `apsrtable`, and `Hmisc` packages. Note that package names must *exactly* match the name provided on the CRAN site.

Updating installed packages

All installed packages can be updated with the following command:

```
> update.packages()
```

When package updates are available, you will be given the option of updating or leaving each package as-is. We recommend updating your packages once a month or whenever **R** is updated.

R-distributed packages

The following packages are distributed with **R** (i.e., no need to install them).

- `foreign`: Load data in STATA, SAS, SPSS, and other formats.
- `MASS`: One of the original **S** packages ported to **R**. Includes modeling functions for ordinal and dichotomous dependent variables as well as much more.
- `lattice`: Advanced graphing capabilities.

Recommended and useful packages

Following are several packages that are particularly useful. These do not come with the standard **R** distribution and will need to be installed.

- `car`: A package of very useful functions used in John Fox' *An R and S-PLUS Companion to Applied Regression*.
- `Hmisc`: A large set of useful functions by Frank Harrell Jr.
- `apsrtable`: The easy way to make LaTeX tables from model results.
- `plyr`: An extremely useful package for data management as well as modeling.
- `xtable`: Outputs LaTeX or HTML representations of **R** tables and other data structures.

Help! Accessing online documentation

The most important thing you need to know is how to get access the expansive documentation available for (nearly) all functions, data types, etc. The help page for functions can be accessed at the prompt with

```
> help(function.name)
```

or, equivalently,

```
> ?function.name
```

The interactive prompt and `.R` files

There are two primary ways to use **R**.

- The interactive command prompt. For quick "calculator-like" interaction. Not recommended for use in research.
- `.R` files: For larger projects that you want to be able to replicate, share with others, keep for reference, etc.

Functions and objects

- Objects: Objects are containers for data and functions. Almost everything in **R** is an object, from the simplest binary variable to, even, mathematical operators (e.g. `+`, `-`, `*`).
- Functions: **R** is also a functional programming language; i.e., you manipulate objects with functions.

  ```
  > diag(M)
  ```

  Here, `diag()` (a function) extracts the diagonal elements from a matrix `M` (an object).

Object assignment

- Assignment: The assignment of an object in R is done with the assignment, <-, operator. For instance, the following assigns the value 23 to some variable `x`.

```
> x <- 23
```

To see the value stored in a particular variable, type it in at the prompt.

```
> x
[1] 23
```

Function arguments

Arguments: Values for function arguments are specified using a *single* equals sign, $=$. For example, a 5×5 diagonal matrix with values 1 through 5 can be created as follows:

```
> X <- diag(1:5, ncol = 5, nrow = 5)
> X
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    0    0    0    0
[2,]    0    2    0    0    0
[3,]    0    0    3    0    0
[4,]    0    0    0    4    0
[5,]    0    0    0    0    5
```

Data types

There are several data types in **R** that you will often have to deal with. These include strings, integers, logical, numeric, vectors, matrices, data frames, lists, and factors.

```
## The basic data types.
a <- "foo"  # a string
b <- "bar"  # another string
x <- 1      # an integer
z <- 1.523  # a floating point number
q <- TRUE   # q is TRUE; a logical
```

Data types: vectors

Vectors can be concatenated using the `c()` function:

```
> (ab <- c(a, b))
[1] "foo" "bar"
> (xz <- c(x, z))
[1] 1.000 1.523
```

Data types: matrices

Matrices can be created with the `matrix()` function.

```
> (A <- matrix(0, nrow = 10, ncol = 10))
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    0    0    0    0    0    0    0    0    0     0
 [2,]    0    0    0    0    0    0    0    0    0     0
 [3,]    0    0    0    0    0    0    0    0    0     0
 [4,]    0    0    0    0    0    0    0    0    0     0
 [5,]    0    0    0    0    0    0    0    0    0     0
 [6,]    0    0    0    0    0    0    0    0    0     0
 [7,]    0    0    0    0    0    0    0    0    0     0
 [8,]    0    0    0    0    0    0    0    0    0     0
 [9,]    0    0    0    0    0    0    0    0    0     0
[10,]    0    0    0    0    0    0    0    0    0     0
```

Data types: matrices

Two or matrices can be joined row-wise with the `rbind()` function or column-wise with the `cbind()` function.

```
> B <- matrix(1:5, nrow = 10, ncol = 10)
> rbind(A, B)
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    0    0    0    0    0    0    0    0    0     0
 [2,]    0    0    0    0    0    0    0    0    0     0
 [3,]    0    0    0    0    0    0    0    0    0     0
         ...
[18,]    3    3    3    3    3    3    3    3    3     3
[19,]    4    4    4    4    4    4    4    4    4     4
[20,]    5    5    5    5    5    5    5    5    5     5
```

Data types: `data.frames`

A `data.frame` is a special data type in **R** and is analogous to how data is stored in a spreadsheet.

```
> (Data <- data.frame(gender, income, married, age, famsize))
   gender  income married age famsize
1    male   41200    TRUE  23       3
2  female 5360000    TRUE  55       2
3  female   34700    TRUE  46       1
   ...
15 female   82200    TRUE  63       3
```

Data types: special values

There are several special values you will run across frequently.

- `TRUE` and `FALSE`: The result returned when making a logical comparison; e.g., `x == 1`.
- `NULL`: The value `NULL` indicates the absence of a value or object; it **is not** equal to zero.
- `NA`: `NA` indicates a *missing* value. It is not the same as `NULL` and it is also **not** equal to zero.

Data types: special values

There are several special values you will run across frequently.

- `TRUE` and `FALSE`: The result returned when making a logical comparison; e.g., `x == 1`.
- `NULL`: The value `NULL` indicates the absence of a value or object; it **is not** equal to zero.
- `NA`: `NA` indicates a *missing* value. It is not the same as `NULL` and it is also **not** equal to zero.

Mathematical and logical operators

- Mathematical operators include the standard ones you would expect: `+`, `-`, `*`, `/`, `^`, and `%*%` (matrix multiplication).
- Logical operators include: `==` (equal to), `>`, `<`, `>=`, `<=`, `!`, `%in%` (matching operator, very handy), and others.
- *Warning*! `x = 1` and `x == 1` are **not** the same thing.

Mathematical and logical operators

- Mathematical operators include the standard ones you would expect: `+`, `-`, `*`, `/`, `^`, and `%*%` (matrix multiplication).
- Logical operators include: `==` (equal to), `>`, `<`, `>=`, `<=`, `!`, `%in%` (matching operator, very handy), and others.
- *Warning*! `x = 1` and `x == 1` are **not** the same thing.

Mathematical and logical operators

- Mathematical operators include the standard ones you would expect: `+`, `-`, `*`, `/`, `^`, and `%*%` (matrix multiplication).
- Logical operators include: `==` (equal to), `>`, `<`, `>=`, `<=`, `!`, `%in%` (matching operator, very handy), and others.
- *Warning*! `x = 1` and `x == 1` are **not** the same thing.

Mathematical and logical operators

Logical operators typically evaluate each element in a vector or matrix.

```
> xz
[1] 1.000 1.523
> (xz > 1)
[1] FALSE  TRUE
```

Accessing data: vectors

Vector elements are accessed by placing an index value between square brackets immediately following the vector name. Here we create a vector of values 0 through 9 and then select the third (i.e., $i = 3$) value in that vector.

```
> a <- 0:9
> a
 [1] 0 1 2 3 4 5 6 7 8 9
> a[3]
[1] 2
```

Accessing data: matrices

Matrix elements are accessed similarly, though a row, i, and column, j, have to be specified.

```
> M <- matrix(1:25, nrow = 5, ncol = 5)
> M
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8   13   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
> M[3, 4]
[1] 18
```

Accessing data: data.frames

data.frame columns, rows, and elements can be accessed in much the same way as matrices. However, you can also specify columns by name.

```
> Data[, c("age", "gender")]  # select the age and
                              # gender columns
   age gender
1   23   male
2   55 female
3   46 female
    ...
13  19 female
14  78   male
15  63 female
```

Accessing data: `data.frames`

To access a single column by name, use the $ operator.

```
> Data$gender
 [1]  male   female female male   male   male   female
 [8]  male   female female male   male   female male
 [15] female
Levels: female male
```

Altering data: vectors and matrices

Elements of vectors and matrices can easily be updated by simply specifying which element row or column that should be changed.

```
> a[5] <- 999; a
 [1]   0   1   2   3 999   5   6   7   8   9
> M[3, 3] <- 999; M
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    6   11   16   21
[2,]    2    7   12   17   22
[3,]    3    8  999   18   23
[4,]    4    9   14   19   24
[5,]    5   10   15   20   25
```

Data generation: sequences and replication

Sequences of integer values can be created with colon operator:

```
> 1:19
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19
```

The `seq()` function allows one to create a sequence with an arbitrary distance between elements; e.g.,

```
> (a <- seq(1, 19, by = 3))
 [1]  1  4  7 10 13 16 19
```

And `rep()` repeats a vector an arbitrary number of times:

```
> rep(a, 2)
 [1]  1  4  7 10 13 16 19  1  4  7 10 13 16 19
```

Data generation: sampling

The `sample()` function produces a random sample from a given vector, both with and without replacement.

```
> sample(1:10, 5, replace = FALSE)
 [1]  6  1  7  3 10
> sample(1:10, 5, replace = TRUE)
 [1]  8 10  3  8 10
```

Data generation: random variables

Random samples from the various distributions are provided by `r*()` functions; e.g., `rnorm()`, `rpois()`, `runif()`, `rt()`, etc.

```
> rnorm(10, mean = 0, sd = 1)
 [1]  0.627  2.616  0.543 -0.859 -0.791  0.377  0.945 ...
> rt(10, df = 100)
 [1] -2.185 -0.983 -1.109  0.199 -0.887  0.152  0.265 ...
> runif(10, min = 0, max = 1)
 [1] 0.199 0.922 0.488 0.410 0.007 0.385 0.569 0.715 ...
> rchisq(10, df = 4)
 [1] 5.663 1.419 3.189 2.390 2.903 4.649 0.417 3.220 ...
```

Loading data

Loading data from STATA, CSV, or other files formats is done using the `read()` family of functions. For example, loading the sample ISSP data set (saved in CSV format) can be done as follows.

```
> ISSP <- read.csv(file = "../data/ISSP-1996.csv")
```

We can look at the top several lines of the data with the `head()` function:

```
> head(ISSP)
```

EDA: crosstabs

The `table()` function can provide some basic crosstabs. For example, the number of observations by country and sex in the ISSP example dataset can be found as follows:

```
> with(ISSP, table(sex, country))
        country
sex      CZE HUN LVA POL SVN
  Female  50  75  74  72  57
  Male    50  60  70  47  45
```

EDA: crosstabs

More detailed crosstabs are provided by the `CrossTable` function in the `gmodels` library.

```
> library(gmodels)
> with(ISSP, CrossTable(empstat, sex))
```

EDA: summary statistics

Calculate some summary statistics after subsetting the original ISSP dataset into one containing only females and one containing only males. Keep only the income data.

```
> ## Subset income by sex.
> inc.f <- ISSP[ISSP$sex == "Female", "inc96"]
> inc.m <- ISSP[ISSP$sex == "Male", "inc96"]
> ## Get some summary statistics.
> length(inc.f)  # number of females
[1] 328
> mean(inc.f)    # mean income for females
[1] 3002.712
> sd(inc.f)      # standard deviation of income for females
[1] 22752.19
> range(inc.f)   # range of income for females
[1]      1 284500
```

EDA: summary statistics

Test whether the different between male and female earnings is statistically significant.

```
> (T <- t.test(inc.m, inc.f, var.equal = TRUE))

Two Sample t-test

data:  inc.m and inc.f
t = 0.4418, df = 598, p-value = 0.6588
alternative hypothesis: true difference in means is
                        not equal to 0
95 percent confidence interval:
 -2977.087  4705.156
sample estimates:
mean of x mean of y
 3866.746  3002.712
```

EDA: statistical tests

And a χ^2-test can be performed manually as follows:

```
> with(ISSP, chisq.test(sex, empstat))

Pearson's Chi-squared test

data:  sex and empstat
X-squared = 43.529, df = 5, p-value = 2.886e-08
```

EDA: scatter plots

A simple scatter plot between two variables can be produced with the `plot()` command.

```
> plot(ISSP$edyrs, ISSP$faminc)
```

This plot isn't that informative. Now take the natural log of *faminc* and then produce the same plot.

```
> plot(ISSP$edyrs, log(ISSP$faminc))
```

This may be problematic if any families in the dataset have $faminc = 0$.

EDA: scatter plots

Now update the plot with more informative labels for the x- and y-axes. Also, make the points red.

```
> plot(ISSP$edyrs, log(ISSP$faminc), col = "red",
+      xlab = "Years of education", ylab = "Family income (log)"
+      main = "Family income by respondent years of education")
```

EDA: scatter plots

PDFs can be created directly by **R** using the `pdf()` command.

```
> plot(ISSP$edyrs, log(ISSP$faminc), col = "red",
+      xlab = "Years of education", ylab = "Family income (log)"
+      main = "Family income by respondent years of education")
> dev.off()
```

Notice the `dev.off()` function, which turns the PDF device off. Strange things can happen if you forget to include this after you make the plot.

EDA: histograms

Histograms describing the distribution of a variable are created with the `hist()` function.

```
> hist(log(ISSP$faminc))
```

And the number of breaks can be controlled with `breaks` argument.

```
> hist(log(ISSP$faminc), breaks = 20)
```

EDA: density plots

Density plots are created by combining the `plot()` and `density()` functions; e.g.,

```
> plot(density(log(ISSP$faminc)))
```

And the bandwidth used to calculate the density can be changed with the `bw` argument.

```
> plot(density(log(ISSP$faminc), bw = 0.75))
```

EDA: combining histograms and density plots

Different plot types can be overlaid. Here is an example that overlays the histogram created above with the density plot.

```
> faminc.dens <- density(log(ISSP$faminc), bw = 0.75)
> hist(log(ISSP$faminc), freq = FALSE, xlab = "Family income (lo
+      main = "Distribution of family income (log)")
> lines(faminc.dens, col = "red")
```

Notice that the density calculations are saved in a variable, which is then used by the `lines()` function **after** the histogram is created. The `lines()` is one function used to add elements to an existing plot.

EDA: combining histograms and density plots

Finally, create a PDF of this plot.

```
> pdf(file = "./faminc-dens.pdf")
> hist(log(ISSP$faminc), freq = FALSE, xlab = "Family income (lo
+      main = "Distribution of family income (log)")
> lines(faminc.dens, col = "red")
> dev.off()
```

PRISM contacts

- Eleonora Mattiacci, Senior fellow (`mattiacci.1@osu.edu`).
- Jason Morgan, Junior fellow (`morgan.746@osu.edu`).