Statistical Programming with R

Connor Harris
(connorharris@college.harvard.edu)

CS50, Harvard University

October 27, 2015

If you want to follow along with the demos, download R at cran.r-project.org or from your Linux package manager

What is R?

- Programming language designed for **statistics** and **data analysis**
- Imperative, weakly typed, interpreted
- Broadly C-like syntax
- Extensive library and native facilities for data mining and statistical analysis

Recommended references (and free legal downloads)

- Venables et al.: *An Introduction to R*
  - "Official" R beginner's guide, maintained by core developers
  - `https://cran.r-project.org/doc/manuals/R-intro.pdf`
- C. R. Shalizi: *Advanced Data Analysis from an Elementary Point of View*
  - Statistics textbook with hundreds of R figures and code samples, plus appendix with useful advice on R programming
  - `http://www.stat.cmu.edu/ cshalizi/ADAfaEPoV/`

General caveats

- This discussion only scratches the surface of R's capabilities.
- I have simplified my discussion of many things, especially the capabilities and call syntax of certain functions. I have presented enough for the most common, simple use-cases; consult the documentation for the full detail.

Why use R?

Consider using R for components of your project that involve:

- Mining of large data sets
- Automated or complicated statistical analysis
- Data visualization or graphing

Type ontology

- Atomic types: `numeric` (64-bit floating point), `character` (strings), `logical` (Boolean); coercion (and `scanf()` equivalents) with `as.numeric` et sim.
- Vectors, matrices, higher-dimensionalarrays of the above
- "Lists" (type of associative array; vectors of lists behave a bit oddly)
- No real "pure" atomic types: single values treated as arrays of length 1
- No mixed-type arrays; if you try you'll get implicit string conversions

Unusual syntactic features

- No variable declarations!
- Assignment with `<-`
- Comments with `#`
- Modular residues and integer division with `%%` and `%/%`
- Ranges with colon (`2:5` is a vector `[2 3 4 5]`)
- One-indexing!
- For-loops: `for (value in vector) { ... }`
- Function syntax: `foo <- function(args) { ... }`
- (Forgot to mention this one in the talk: semicolons after statements are optional at the ends of lines)

Vectors

- Constructed with `c(`$datum_1$`, ..., `$datum_n$`)` (arguments can also be vectors, though resulting array is flattened)
- Cannot be of mixed type
- Behave as if padded infinitely with value `NA` ("missing value")
- Unary functions map over arrays
- Binary functions are applied entry by entry (cycling shorter array if necessary)
- Access with square brackets containing (one-indexed!) indices. Can pass vector of indices to get vector of corresponding elements, negative indices to remove elements (NB: differs from Python)
- Summary statistics with `summary()`

Matrices

- Initialized with `matrix(`*data*`, nrow=`*rows*`, ncol=`*columns*`)`; `data` (a vector) fills matrix first up to down, then left to right
- Excellent facilities for matrix multiplication (`a %*% b`), spectral decomposition (`eigen(a)`), and other common tasks
- Arrays (higher-dimensional matrices) can be initialized with `array(dim=c(`dim_1`, ..., `dim_n`))`
- Access columns with `foo[`*rownum*`,]`, columns with `foo[,`*colnum*`]`

Lists

- Type of associative array
- Initialized as `list(`key_1`=`val_1`, ..., `key_n`=`val_n`)`
- Access and set values with `foo$key`
- Access individual key–value pairs with integer indices or as `foo["key"]`
- Reading from a nonexistent key returns NULL, not an error; this can trip you up

Data frames

- Type of list in which every value is a vector of the same length
- Used for representing data table
- Initialized with `data.frame([`*column-name*$_1$`=]`*column-data*$_1$`, ...,` *column-name*$_n$`=]`*column-data*$_n$`, [row.names=`*string-vector*`])`
- Access columns with `foo["column-name"]` or `foo[`*column-index*`]`
- Access rows with `foo[`*row-index*`,]` (note trailing comma)
- Row and column names accessible with `rownames(foo)` and `colnames(foo)`
- Print header and first few rows with `head(foo)`

Functions

- `foo <- function(`arg_1`[=`$default_1$`], ..., `arg_n`[=`$default_n$`]) { ... }`
- Called as `foo([`arg_1`=]`$value_1$`, ..., [`arg_n`=]`$value_n$`)`
- No need for explicit `return()` statement (note parentheses): last statement evaluated is return value—but explicit `return()` is often better style
- Keyword arguments in function calls can be included in any order

Data import and export

- Read tabular data into data frames with `read.table()` (text files), `read.xls()` (Excel spreadsheets), `read.csv()` (CSV files), et sim.
- Write data frames out as tables with `write.table()`, etc.
- Save arbitrary R objects to binary files and reload them with `save(`$object_1$`, ..., `$object_n$`, file=`$file$`)` and `load(`$file$`)`
- Files are written and read by default into R's working directory, readable and modifiable with `getwd()` and `setwd(`dir`)`

Multilinear regression

- Syntax: `model <- lm(y ~ `x_1`[+`x_2`[...[+`x_n`]...]][, `*dataframe*`])`
- `y` is the dependent variable, x_1, ..., x_n the independent variables; can be either vectors or column headers of the data frame specified in the optional second argument
- Possible to specify more complex formulae, e.g. `model <- lm(y^2 + 1 ~ log(x)`
- Print summaries of the model (with line-of-best-fit parameters, etc.) with `summary`
- For calculating simple correlations, use `cor(`*vec1*`, `*vec2*`[, method=`*method*`])`

Plotting

- Workhorse function is `plot(x, y, ...)`, plus many variants and specializations
- Takes two vectors of the same length; precede with `attach(`*dataframe*`)` to use column headers instead of separate vectors
- Myriad optional arguments for controlling various details of the plot. Some common ones: `type` (`"p"` for points [i.e. scatter plot], `"l"` for lines, et sim.); `main` (overall title); `xlab`, `ylab` (axis labels), `col` (default color)
- Can add fit best-fit lines and local regression curves with `abline(`*regression-model*`)` and `lines(lowess(x, y))`
- Default graphical output is a pop-up window; write to files (in, e.g., PNG format) with `png(`*filename*`)`; close devices with `dev.off()`
- Facilities for 3D and contour plotting, but I won't go into these now
- Some third-party libraries for making animation, but a better choice is to use R to generate the frames for animations and then combine them with a third-party program like FFmpeg or ImageMagick

Time for a demo

- Unix and OS X users: open a terminal window and type "`R`" at the command prompt
- Windows users: find R in the list of programs in the start menu

Foreign-function interface I

- Allows R code to call C functions
- Why would you want to do this?
  - Higher speed in inner loops
  - Reuse of existing C libraries
- Interface is slightly arcane and existing tutorials are confusing
- Following instructions are for Unix-like systems (e.g. Linux, OS X)—I don't know about Windows
  - Please do not write your final project on Windows

Foreign-function interface II

- Must take all arguments as pointers (NB: for arrays, this is a pointer to the first element)
- Floating-point type is `double` (64-bit)
- Must return `void`; tells result by modifying arguments

```
void dotprod(double* vec1, double vec2, int* n, double* out) {
    *out = 0;
    for (int i = 0; i < *n; i++) {
        *out += vec1[i] * vec2[i];
    }
}
```

Foreign-function interface III

Good practice to write a function in C that takes arguments without pointers and then a “wrapper function” that handles the FFI requirements:

```
double dotprod_internal(double* vec1, double* vec2, int n) {
    double result = 0;
    for (int i = 0; i < n; i++) {
        result += vec1[i] * vec2[i]
    }
    return result;
}
void dotprod(double* vec1, double* vec2, int* n, double* out) {
    *out = dotprod_internal(vec1, vec2, *n);
}
```

Foreign-function interface IV

- Compile C code with `R CMD SHLIB foo.c` (in the OS shell, not R)—creates library foo.so
- Use library in R code with `dyn.load("foo.so")`
- Call using .C() function; takes name of C routine and type-coerced arguments (using `as.integer`, `as.double`, `as.character`, `as.logical`)
- Returns list (associative array) of parameter names and modified values

```
result <- .C("dotprod", as.double(vec1), as.double(vec2),
             as.integer(length(vec1)), as.double(0))
product <- result$out
```

Bad practices: Explicit loops

- Slow, inelegant
- Unnecessary because of R's facile vector handling
- Replace with higher-order functions (Map, Reduce, Find, Filter) or apply functions; see Shalizi for other methods

Bad practices: Appending to vectors

Several equivalent syntaxes, e.g.

```r
vec[length(vec)+1] <- newvalue
vec <- c(vec, newvalue)
```

- Vectors must be completely reallocated when resized
- Pre-allocate vectors to the necessary size

  ```r
  vec <- vector(length=1000)
  ```

- Changing iterated reallocations to preallocation caused a **thousandfold speedup** in one of my own projects (numerical differential-equation solver, vectors of length $10^4 \sim 10^5$)

Error handling

- R prefers continuing after possible errors to stopping, which can produce unexpected behavior in hard-to-predict places
- Two easy mistakes to make: vector values where single numbers are expected, and NULL values—cause functions to behave strangely, but don't throw clean errors
- Sanity-check function inputs with `stopifnot()` (equivalent to C's `assert()`)

End

- I am happy to take questions by e-mail: `connorharris@college.harvard.edu`
- I am also happy to serve as an unofficial adviser for anyone using R in a final project; talk to your TF and write me an e-mail if so