Bio 723: Statistical Computing for Biologists

Paul M. Magwene

Modern biological data is. . .

- Heterogeneous – sequence data, species distributions, protein concentrations, . . .
- High-dimensional – genome-wide, GIS, . . .
- Copious – time series, population surveys, global monitoring stations, . . .

To be an effective analyst of biological data you must have...

- Biological knowledge and intuition – What does the data mean? What are the key questions? What are interesting patterns or findings in the data?
- Computational skills – sort, filter, transform, and transform data and work with in an efficient and reproducible manner
- Statistical skills – build and interpret quantitative statements (models) about patterns in your data, distinguish interesting "signal" from "noise" of natural biological variability and experimental design

Course Goals

1. Learn to visualize and explore complex biological data using R, a programming language and statistical computing environment
2. Introduce multivariate statistics from a geometric perspective, emphasizing the geometry of vector spaces.
3. Illustrate how to carry out common scientific computing tasks like simulation and building bioinformatics pipelines
4. Provide the tools and knowledge to conduct reproducible computational and statistical research.

Syllabus

1. R Basics, data munging, and visualization
2. Introduction to multivariate statistics from a geometric perspective
3. A survey of common machine learning methods – clustering, classification, dimensionality reduction
4. Building bioinformatics pipelines

Course mechanics

- Texts
- Grading
- Expectations and Policies

Class structure

1. Lectures

- Typically 60-75 minutes
  - Emphasize the mathematical basis of the methods/approaches from both a geometric and algebraic basis
  - Discuss algorithms underlying the methods

2. Hands-on

- Walk through some examples
  - Apply the techniques and concepts to real data
  - Highlight available R libraries

Goals for today's class session

- Review key topics covered in reading material for Lecture 0 and 1
- Introduce key R data structures – vectors, data frames
- Introduce the `dplyr` library for manipulating, filtering, and transforming data frames

Concepts covered in "Lecture 0" materials

- Installing R and RStudio
- Getting oriented in RStudio
- Working at the Console
- R Markdown and R Notebooks
- R Help System
  - `?` and `help.search`, `apropos`, `vignette`, etc
- Installing packages
- Loading packages
- Core data types
  - Numerical – Doubles, Integers, Complex
  - Logical (Boolean) – TRUE and FALSE
  - Character – strings

RStudio Tour

In-class demonstration of RStudio interface.

RMarkdown

In-class demonstration of creating and knitting an RMarkdown document.

Getting Help in R

- `help("topic")` or `?topic`
- `apropos()`
- `help.search("topic")` or `??topic` – "fuzzy search"
- `example("topic")`

Data types

- Refers to the types of values that can be represented in a computer program
- Determine the representation of values in memory
- Constrains the operations you can perform on those values

In R data types are usually inferred by the interpretter rather than specified by the user, though occassionally you may find it necessary to specify a data type.

Numeric data types in R

- Floating point values ("doubles") – represent real numbers (continuous values)

```
> x <- 10.0
> typeof(x)
```

- Integers – represent whole numbers. The default numeric type is double, so you must explicitly ask for integers.

```
> x <- as.integer(10)
> typeof(x)
```

- Complex numbers – numbers with a real and "imaginary" part. We won't be explicitly using complex numbers in this course, but they sometimes appear unexpectedly in some calculations.

```
> x <- 1+1i
> typeof(x)
```

Arithmetic operations on numerical data types

```
> 10 + 2 # addition
> 10 - 2 # subtraction
> 10 * 2 # multiplication
> 10 / 2 # division
> 10 ^ 2 # exponentiation
> 10 ** 2 # alternate exponentiation
> 10 %% 4 # modulus (remainder after division)
> 4-5 / 2 # operator precedence matters!
> (4-5)/2 # parentheses help you specify/disambiguate precedence.
```

Basic mathematical functions

```
> sqrt(10) # square root
> 10 ^ 0.5 # same as square root
> sqrt(-1) # NaN means "Not a Number"
> sqrt(-1 + 0i) # But works if we use complex type
> exp(1) # exponential function
> log(100)  # log base e
> log10(100) # log base 10
> log2(8) # log base 2
> factorial(5) # factorial function: 5 * 4 * 3 * 3 * 1
> pi  # R knows some useful constants
> cos(2*pi)  # cosine, also sin, tan, ...
```

Logical (Boolean) type

```
> x <- TRUE
> typeof(x)
> y <- FALSE
> typeof(FALSE)
> !TRUE # logical negation of TRUE
> !y # logical negation (NOT) of value in y
```

Numerical comparison operators return logical types

```
> 4 < 5  # less than
> 10 >= 9 # greater than or equal to
```

The logical results of comparison operations can be assigned to variables

```
> x <- 1 > 2
> y <- 2 >= 2
> isTRUE(x) # returns true if x is logical and true
> x & y # Logical AND
> x | y # Logical OR
```

Character (string) type

```
> x <- "Hello" # enclosed in double quotes
> y <- 'World' # or single quotes
> z <- 'You said "Hello World"' # allows nesting
```

Simple functions on characters

```
> paste(x, y)  # concatenate strings
> paste(x, y, sep = "") # concatenate with no space
> strsplit("Hello world!", split=" ") # split on space
```

The `stringr` package (part of the tidyverse) contains many useful string manipulation functions.

Missing values (NA)

`NA` ("not available") values represent missing data

```
> x <- NA
> x
> is.na(x)
```

Numerical computations involving NA usually "propagate" the NA values appropriately:

```
> y <- 1
> x + y
```

NA-aware functions

Many functions are "NA"-aware in that they include options to handle (often ignore) missing values if requested

```
> mean(c(2, 4, 6, NA, 8))
```

with optional `na.rm` argument:

```
> mean(c(2, 4, 6, 8), na.rm = TRUE)
```

NaN and Inf

`NaN` ("not a number") represent results of invalid numerical calculations

```
> z <- sqrt(-1)
> z
> is.nan(z)
```

`Inf` represents numerically infinite values

```
> x <- 1/0
> x
> is.infinite(x)
> is.finite(y)
```

Data structures

- Represent different ways collections of data are stored in memory or accessed by the user
- Different data structures are more efficient for particular modes of access or to represent different types operations

Vectors

- Vectors are homogeneous ordered list

The `c()` (combine) function can be used to create vectors "from scratch":

```
> x <- c(2, 4, 6, 8)
> y <- c("hello", "world", "how", "are", "you?")
```

Vectors always have a length (possibly zero) and a type

```
> length(x)
> typeof(x)
> length(y)
> typeof(y)
```

Elements of a vector will be coerced to be the same type

```r
> x <- c(1+1i, 2+1i, 'Fred', 10)
> x
> typeof(x)
```

Indexing vectors

Accessing the objects in a vector is accomplished by "indexing".

The elements of the vector are assigned indices $1 \ldots n$ where n is the length of the vector

```
> x <- c(2,4,6,8)
> length(x)
> x[1]
> x[2]
> x[4]
> x[length(x)] # why might this be preferred way to get last value?
```

Indexing past the end of a vector returns an NA value

```
> x[5]
```

Single objects of core data types in R are themselves vectors

```
> x <- 1
> length(x)
> x[1] # can be indexed like any other vectors
> is.vector(x)  # function to test whether something is vector
```

Vectors can be indexed by other vectors, including logical vectors

```
> x <- c(2, 4, 6, 8)
> x[c(2,4)]  # get 2nd and 4th elements of x
```

```
> x <- c(2, 4, 6, 8)
> y <- c(0, 1, 2, 10)
> x[x > y] # get elements of x where x > y
```

Arithmetic operators and most math functions work on numerical vectors

Arithmetic operators and comparison work element-by-element on vectors.

```
> x <- c(2, 4, 6, 8)
> y <- c(0, 1, 2, 10)
> x + y
> x * y
> x^2
> sqrt(x)
> x < y
```

Lists

Lists in R are like vectors but the elements of a list are arbitrary objects (even other lists). Lists are "heteregeneous".

```
> x <- list('Bob',27, 10, c(720,710))
> typeof(x)
> x
```

Indexing Lists

Items in lists are accessed in a different manner than vectors.

- Typically you use double brackets (`[[]]`)to return the element at index `i`
- Single brackets always return a list containing the element at index `i`

```
> x <- list('Bob', 27, 10, c(720,710))
> x[1]
> typeof(x[1])
> x[[1]]
> typeof(x[[1]])
```

List elements can have names

```
> x <- list(name='Bob',age=27, years.in.school=10)
> x
```

Named list objects can be accessed via the $ operator

```
> x$years.in.school
```

The names of list elements can be accessed with the `names()` function

```
> names(x)
```

Data frames

Data frames represent data tables (data sets)

- Each column in the table has the same number of rows
- Every item in a given column has to be of the same type (think of each column as a vector)
- Columns in a data frame must have names

```
> # constructing a data frame from scratch
> name <- c("Paul", "Maira", "Peter", "Beatriz")
> grade <- c("C", "A", "B", "A-")
> age <- c(40, 42, 28, 52)
>
> example.df <- data.frame(name = name,
+                          age = age,
+                          grade = grade)
> example.df
```

Data frames: shape and column indexing

Column names

```
> names(examples.df)
```

Shape

```
> dim(example.df)    # number of rows and columns
> nrow(example.df)   # number of rows
> ncol(example.df)   # number of columns
```

Indexing columns by name or position

```
> example.df["grade"]
> example.df[3]
```

Indexing with a vector to subset columns

```
> example.df[c("name","grade")]
```

$ operator

Indexing a column name with single brackets (previous slide) returns a new data frame.

```
> example.df["grade"]
```

The $ operator returns a vector rather than a data frame.

```
> example.df$grade
```

Data frames: indexing rows

To index one or more rows of a data frame, specifying the row number(s) to index on followed by a comma:

```
> example.df[1,]  # first row
```

```
> example.df[c(1,3),] # first and third row
```

You can simultaneously index both rows and columns:

```
> example.df[c(1,3), c("name", "age")]
```