AN INTRODUCTION TO R

DEEPAYAN SARKAR

Language Overview II

In this tutorial session, we will learn more details about the R language.

Objects. Objects in R are anything that can be assigned to a variable. They could be

- Constants: 2, 13.005, "January"
- Special symbols: NA, TRUE, FALSE, NULL, NaN
- Things already defined in R: seq, c (functions), month.name, letters (character), pi (numeric)
- New objects we can create using existing objects (this is done by evaluating *expressions* — for example, 1 / sin(seq(0, pi, length = 50)))

Mode and class. R objects come in a variety of types. Given an object, the functions mode and class tell us about its type. mode(object) has to do with how the object is stored. class(object) gives the *class* of an object. The main purpose of the class is so that generic functions (like print and plot) know what to do with it. Often, objects of a particular class can be created by a function with the same name as the class.

```
> obj <- numeric(5)
> obj
[1] 0 0 0 0 0
> mode(obj)
[1] "numeric"
> class(obj)
[1] "numeric"
```

The mode of an object tells us how it is stored. Two objects may be stored in the same manner but have different class. How the object will be printed will be determined by its class, not its mode.

```
> x <- 1:12
> y <- as.table(matrix(1:12, 2, 6))
> class(x)
[1] "integer"
> mode(x)
[1] "numeric"
> class(y)
[1] "table"
> mode(y)
[1] "numeric"
> print(x)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12
> print(y)
```

Date: November 2015.

```
  A B C D  E  F
A 1 3 5 7  9 11
B 2 4 6 8 10 12
```

Classes and generic functions. This is achieved by the following mechanism: When `print(y)` is called, the `print()` function determines that `class(y)` is `"table"`, so it looks for a function named `print.table()`. Such a function exists, so the actual printing is done by `print.table(y)`. When `print(x)` is called, the `print()` function determines that `class(x)` is `"integer"`, so it looks for a function named `print.integer()`. There is no such function, so instead the fallback is used, and the actual printing is done by `print.default()`. This happens only for *generic functions* (those that call `UseMethod`).

This is actually a simplified version of what really happens, but it is close enough for our purposes.

Functions. Functions in R are simply objects of a particular type (of mode `"function"`). Like other objects, they can be assigned to variables. Most of the time, they actually are assigned to a variable, and we refer to the function by the name of the variable it is assigned to. All standard functions (like `print()`, `plot()`, `c()`) are actually variables, whose value is an R object of mode `"function"`. When we refer to the `seq()` function, we actually mean the value of the variable `seq`.

> `class(seq)`

```
[1] "function"
```

> `mode(seq)`

```
[1] "function"
```

> `print(seq)`

```
function (...)
UseMethod("seq")
<bytecode: 0x55968f500490>
<environment: namespace:base>
```

Calling functions. To call a function, the function object is followed by a list of arguments in parentheses: `fun.obj(arglist)`. Every function has a list of formal arguments (displayed by `args(fun.obj)`). Arguments can be matched by

- position: e.g., `plot(v1, v2)`
- name: e.g., `plot(x = v1, y = v2)`
- default: many arguments have default values which are used when the arguments are not specified. For example, `plot()` can be given arguments called `col`, `pch`, `lty`, `lwd`. Since they have not been specified in the calls above, the defaults are used.

This is most easily understood by looking at examples.

Functions are objects. Functions are regular R objects, just like other objects like vectors, data frames, and lists. So, variables can be assigned values which are functions, and functions can be passed as arguments to other functions. We have already seen one example of this in the last tutorial, when using the `by()` function, where one of the arguments was a function object. More common examples are `lapply()` and `sapply()`, which we will encounter later in this tutorial.

What happens when a function is called? Most functions return a new R object, which is typically assigned to a variable or used as an argument to another function. Some functions are more useful for their *side-effects*, e.g., `plot()` produces a graphical plot, `write.table()` writes some data to a file on disk.

Expressions. Before we discuss functions further, we need to know a bit about *expressions*. Expressions are statements which are evaluated to create an object. They consist of operators (`+`, `*`, `^`) and other objects (variables and constants). e.g., `2 + 2`, `sin(x)^2`.

```
> a <- 2 + 2
> print(a)

[1] 4

> b <- sin(a)^2
> print(b)

[1] 0.57275
```

R allows *expression blocks* which consist of multiple expressions. All individual expressions inside this composite block are evaluated one by one. All variable assignments are done, and any `print()` or `plot()` call have the appropriate side-effect. But most importantly: this whole composite block *can be treated as a single expression* whose value on evaluation will be the value of the *last expression evaluated inside the block*.

```
> a <- {
      tmp <- 1:50
      log.factorial <- sum(log(tmp))
      sum.all <- sum(tmp)
      log.factorial
  }
> print(a)

[1] 148.4778

> print(sum.all)

[1] 1275
```

Defining a function. A new function is defined / created by a construct of the form

```
fun.name <- function( arglist ) expr
```

where

- `fun.name` is a variable where we store the function. This variable will be used to call the function later.
- `arglist` is a list of formal arguments. This list
  - can be empty (in which case the function takes no arguments);
  - can have just some names (in which case these names become variables inside the function, whose values have to be supplied when the function is called);
  - can have some arguments in `name = value` form, in which case the names are variables available inside the function, and the values are their default values.
- `expr` is an expression (typically an *exression block*) (which can make use of the variables defined in the argument list). This part is also referred to as the *body* of a function, and can be extracted by `body(fun.obj)`.
- Inside functions, there can be special `return(val)` calls which exits the function and returns the value `val`.

Variables and Scope. When an expression involves a variable, we must know how the value of that variable is determined. For example, most languages have *local* and *global* variables. In R, the situation is a bit more complicated, and involves the concept of *environments*.

Environments. Environments consist of a *frame* (a collection of named objects) and a pointer to an *enclosing environment*. Expressions are (usually) evaluated in the context of an environment. Each variable name in the expression is searched for in the evaluation environment. If it is not found there, it is searched for in the enclosing environment. If it is not found there, it is searched for in the next enclosing environment, and so on until there are no other environments to search in. If a variable with that name is not found in any of these environments, an error is generated.

Expressions entered at the top-level prompt are evaluated in a special environment called `.GlobalEnv`, also known as the *workspace*. The environments enclosing `.GlobalEnv` correspond to the various packages that are currently attached, ending in the so-called "base" environment, which has no further enclosing environment. The current chain of evaluation environments is listed by

```
> search()
 [1] ".GlobalEnv"        "package:tools"     "package:stats"
 [4] "package:graphics"  "package:grDevices" "package:utils"
 [7] "package:datasets"  "package:methods"   "Autoloads"
[10] "package:base"
```

This is also referred to as the *search path*, as variables are searched for here. The location(s) of a variable in the search path can be obtained using `find()`.

```
> find("pi")
[1] "package:base"
> pi
[1] 3.141593
> pi <- 22/7 ## A cruder approximation of pi
> find("pi")
[1] ".GlobalEnv"    "package:base"
> cos(2 * pi)
[1] 0.9999968
> with(baseenv(), cos(2 * pi))
[1] 1
```

Variable scope inside functions. Whenever a function is executed, a new environment is created within which the body of the function is evaluated. The enclosure of this environment is the environment where the function was defined. The frame of the environment consists of the variables local to the function call, which consists of variables defined as arguments of that function and further variables defined inside the function. These local variables remain in effect only inside the function; they can no longer be accessed once the function has returned (unless special effort is made to retain access to the environment).

Thus, variables inside a function are first searched for in the local environment. Variables not found there are searched for in the enclosure, and then further up the chain. Note that this means that if two variables of the same name are defined both outside and inside the function, the *one inside will be used*.

```
> myvar <- 1
> fun1 <- function() {
      myvar <- 5
      print(myvar)
  }
> fun2 <- function() {
      print(myvar)
  }
> fun1()
[1] 5
```

```
> myvar # has not changed
[1] 1
> fun2()
[1] 1
> myvar <- 10
> fun2()
[1] 10
```

Note that assignments will still create variables in the local environment, even if a variable with the same name exists outside. For example, suppose we want to keep track of the number of times a function has been called. The following attempt will fail:

```
> ncall <- 0
> fun3 <- function() {
      ncall <- ncall + 1
      cat("fun3 called", ncall, "time(s).", fill = TRUE)
  }
> fun3()
fun3 called 1 time(s).
> fun3()
fun3 called 1 time(s).
> ncall
[1] 0
```

Exercise 1. *Read* `help("<-")` *and accordingly modify the example above to give the desired result.*

Environments share many properties with lists. In particular, elements can be extracted and set using the `$` operator, and they can be used as the first argument in functions like `with()`. Environments offer an alternative solution for the problem above.

```
> store <- new.env() ## creates a new empty environment
> store$ncall <- 0
> fun3 <- function() {
      store$ncall <- store$ncall + 1
      cat("fun3 called", store$ncall, "time(s).", fill = TRUE)
  }
> fun3()
fun3 called 1 time(s).
> fun3()
fun3 called 2 time(s).
```

Exercise 2. *In the above example, replace* `store <- new.env()` *with* `store <- list()`. *Can you figure out why the code no longer works?*

Programming constructs. R has the standard programming constructs:

- `if`
- `else`
- `for`

- while
- etc.

As most R functions are already vectorized, the `for` construct is usually unnecessary, but can be useful in some situations. The `for` keyword is always followed by an expression of the form (`variable in vector`). The block of statements that follow this is executed once for every value in `vector`, with that value being stored in `variable`.

```
> for (i in 1:5) {
      print(i^2)
  }
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

`while` and `if` statements work more or less as they do in C, as an example will illustrate. The following function computes members of the Fibonacci sequence (whose first two entries are 0 and 1, and subsequent entries are computed as the sum of the previous two entries).

```
> fibonacci <- function(length.out) {
      if (length.out < 0) {
          warning("length.out cannot be negative")
          return(NULL)
      }
      else if (length.out < 2)
          x <- seq(length = length.out) - 1
      else {
          x <- c(0, 1)
          while (length(x) < length.out) {
              x <- c(x, sum(rev(x)[1:2]))
          }
      }
      x
  }
> fibonacci(-1)

NULL

> fibonacci(1)

[1] 0

> fibonacci(10)

 [1]  0  1  1  2  3  5  8 13 21 34
```

Note that a function returns the last expression it evaluates (in this case `x`). An explicit `return()` statement is necessary only when the function must return before reaching the end.

Exercise 3. *Although usually defined through a recurrence relation, the Fibonacci numbers have a closed-form solution as well. The n-th number in the sequence is given by*

$$F(n) = \frac{\phi^n - (1 - \phi)^n}{\sqrt{5}}$$

where $\phi = \frac{1+\sqrt{5}}{2}$ is the so-called "golden ratio". Use this relationship to write a function that given an argument n produces the first n Fibonacci numbers.

Exercise 4. *A finite-precision computer cannot represent an irrational number like $\sqrt{5}$ exactly. Consequently, we would not expect the numbers computed by the closed-form expression above to be exact integers, even though the Fibonacci numbers are actually integers. Do the numbers reported by your function appear to be integers? Are they really integers?*

What happens to objects? An object is created whenever an expression is evaluated. Unless the result is assigned to some variable (or used as part of another expression), this object is lost.

In the following example, all the `i^2` values are calculated, but nothing is stored and nothing is printed.

```
> for (i in 1:10) i^2
```

What to do with an object after creating it depends on what the purpose is. If the only intent is to see the value, the object can be printed using the `print()` function.

```
> for (i in 1:5) print(i^2)
[1] 1
[1] 4
[1] 9
[1] 16
[1] 25
```

If the intent is to use these values for later, they should be assigned to some variable.

```
> a <- numeric(10)
> print(a)
 [1] 0 0 0 0 0 0 0 0 0 0
> for (i in 1:10) a[i] <- i^2
> print(a)
 [1]   1   4   9  16  25  36  49  64  81 100
```

Type checking. It is often useful to know whether an object is of certain type. There are several functions of the form `is.type` which do this. Note that `is` is not a generic function, even though the naming convention is similar.

```
> airq <- head(airquality, 31) ## Air quality measurements in New York, May 1973
> is.data.frame(airq)
[1] TRUE
> is.list(airq)
[1] TRUE
> is.numeric(airq)
[1] FALSE
> is.function(airq)
[1] FALSE
```

Detecting special values. Some similarly-named functions are used for element-wise checking. The most important of these is `is.na()`, which is needed to identify which elements of a vector are missing.

```
> airq$Ozone
 [1]  41  36  12  18  NA  28  23  19   8  NA   7  16  11  14  18  14  34   6  30
[20]  11   1  11   4  32  NA  NA  NA  23  45 115  37
```

```
> airq$Ozone == NA
 [1] NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA NA
[26] NA NA NA NA NA NA
> is.na(airq$Ozone)
 [1] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE
[13] FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[25]  TRUE  TRUE  TRUE FALSE FALSE FALSE FALSE
> is.na(c(Inf, NaN, NA, 1))
[1] FALSE  TRUE  TRUE FALSE
> is.nan(c(Inf, NaN, NA, 1))
[1] FALSE  TRUE FALSE FALSE
> is.finite(c(Inf, NaN, NA, 1))
[1] FALSE FALSE FALSE  TRUE
```

Coercion methods. There are several functions of the form `as.type` that are used to convert objects of one type to another.

```
> as.numeric(c("1", "2", "2a", "b"))
[1]  1  2 NA NA
> as.numeric(c(TRUE, FALSE, NA))
[1]  1  0 NA
> as.character(c(TRUE, FALSE, NA))
[1] "TRUE"  "FALSE" NA
```

There are some automatic coercion rules that often simplify things.

```
> airq$Ozone > 30
 [1]  TRUE  TRUE FALSE FALSE    NA FALSE FALSE FALSE FALSE    NA FALSE FALSE
[13] FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE  TRUE
[25]    NA    NA    NA FALSE  TRUE  TRUE  TRUE
> sum(airq$Ozone > 30)
[1] NA
> sum(airq$Ozone > 30, na.rm = TRUE)
[1] 7
```

But sometimes they can produce surprising results.

```
> 1 == TRUE
[1] TRUE
> 1 == "1"
[1] TRUE
> "1" == TRUE
[1] FALSE
```

Automatic printing. Although we have not made it explicit, it should be clear by now that objects are usually printed automatically when they are evaluated on the command prompt without assigning them to a variable. This is (almost) equivalent to calling `print()` on the object. This does not always happen; R has a mechanism for suppressing this printing, which is used in some cases. The automatic printing *never* happens inside a function. So remember to deal with whatever objects you evaluate inside your functions. The suppression of automatic printing can sometimes lead to unexpected behaviour. To avoid surprises, use calls to `print()` explictly.

Implicit loops. We often need to apply one particular function to all elements in a vector or a list. Generally, this would be done by looping through all those elements. R has a few functions to do this elegantly. `lapply()` Returns the results as a list. `sapply()` tries to simplify the results and make it a vector or matrix. Other useful functions in the same family are `apply()` and `tapply()`.

```
> lapply(airq, mean)
$Ozone
[1] NA

$Solar.R
[1] NA

$Wind
[1] 11.62258

$Temp
[1] 65.54839

$Month
[1] 5

$Day
[1] 16
> sapply(airq, mean)
   Ozone  Solar.R     Wind     Temp    Month      Day
      NA       NA 11.62258 65.54839  5.00000 16.00000
> sapply(airq, mean, na.rm = TRUE)
   Ozone   Solar.R      Wind      Temp     Month       Day
23.61538 181.29630  11.62258  65.54839   5.00000  16.00000
```

Note the need for `na.rm = TRUE` because there are some missing observations. Unless otherwise specified, all calculations involving an `NA` usually produce an `NA`.

Exercise 5. *Using the full `airquality` dataset, convert the `Month` column to a factor with suitable levels (you may use the predefined variable `month.name`). Obtain the per-month mean Ozone concentration using*

    (1) *a combination of `split()` and `sapply()`*
    (2) *`tapply()`.*

Exercise 6. *Create a 100×10 matrix of $U(0,1)$ random variables. We can think of these as 100 replications of 10 $U(0,1)$ random variables. Use `apply()` to compute a vector of per-replicate (per-row) means. Can you perform the same operation without a loop using a single matrix operation?*

Exercise 7. *Loops (whether implicit or explicit) are generally slower than vectorized operations in R. Compare the speed of the two approaches in the last exercise using the `system.time()` function. You may need several repititions of the procedure to detect a difference; use `replicate()`. How do these two approaches compare in terms of speed to `rowMeans()`?*

Exercise 8. *Find a way to insert a value at a given position in a vector. Implement this using a function. For example, the function call might look like this:*

```
> insert(x, where, what)
> ## x: initial vector
> ## where: which position to insert in
> ## what: what to insert
```

A test case using this function:

```
> x <- 1:10
> x <- insert(x, 5, 0)
> x
 [1]  1  2  3  4  0  5  6  7  8  9 10
```

How would you extend this idea to insert rows in a matrix?

Exercise 9. *How would you check if two numeric vectors (possibly containing NA-s) are the same (without using* `identical()`*)?*

The next exercise involves *hypotrochoids*, which are geometric curves traced out by a point within a circle that is rolling along "inside" another fixed circle (a spirograph is a popular toy that produces hypotrochoids). Our goal is to design a framework where we can easily plot hypotrochoids. Hypotrochoids can be parameterized by the equations

$$
\begin{aligned}
x(t) &= (R - r)\cos t + d\cos\frac{R - r}{r}t \\
y(t) &= (R - r)\sin t - d\sin\frac{R - r}{r}t
\end{aligned}
$$

where R is the radius of the fixed circle, r the radius of the moving circle, and d is the distance of the point being traced from the center of the latter. Fixing $R = 1$, we can plot a hypotrochoid with $r = 0.7$ and $d = 0.16$ as follows.

```
> r <- 0.7
> d <- 0.16
> hypox <- function(t) (1 - r) * cos(t) + d * cos((1-r) * t / r)
> hypoy <- function(t) (1 - r) * sin(t) - d * sin((1-r) * t / r)
> tt <- seq(0, 20 * pi, by = 0.1)
> par(pty = "s")
> plot(x = hypox(tt), y = hypoy(tt), type = "l", axes = FALSE)
```
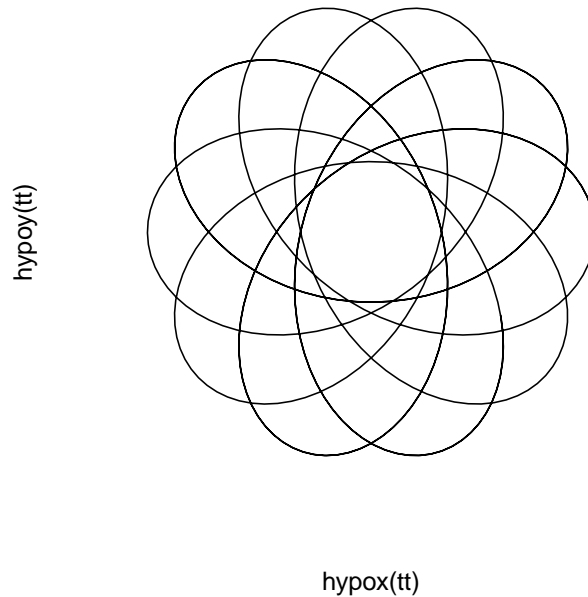
This is sufficient as a quick proof-of-concept, but if we wish to work with hypotrochoids more systematically, we would want a cleaner implementation. The typical R approach is to write a *constructor* function that collects the information required to define a hypotrochoid into an object of a new class. Further manipulation is done using methods.

```
> hypotrochoid <- function(r = runif(1), d = r * runif(1))
  {
      ans <- list(r = r, d = d)
      class(ans) <- "hypotrochoid"
      ans
  }
> print.hypotrochoid <- function(x, ...)
  {
      cat("A hypotrochoid with r =", x$r, "and d =", x$d, fill = TRUE)
  }
> h1 <- hypotrochoid(r = 0.7, d = 0.16)
> h2 <- hypotrochoid() # random choices
> h1
A hypotrochoid with r = 0.7 and d = 0.16
> h2
A hypotrochoid with r = 0.03314217 and d = 0.01703217
>
```

Exercise 10. *Write a suitable* `plot.hypotrochoid()` *method and use it to plot* `h1` *and* `h2`. *Your method should allow the* `to` *and* `by` *arguments for* `seq()` *to be specified.*

Distributions and random variables. One of the frequent uses of R is to genrate pseudo-random numbers from various probablity distributions. R has functions to generate observations from several distributions, as well as functions giving their density and the cumulative distribution functions.

For example, the functions related to the Normal distribution are described in the `?pnorm` help page. Among other common distributions are Lognormal (`?plnorm`), Exponential (`?pexp`), Uniform (`?punif`), Binomial (`?pbinom`) and Poisson (`?ppois`).

Calculating probabilities. Consider a random variable X that is Normally distributed with mean 100 and standard deviation 10.

Exercise 11. *Use* `pnorm()` *to calculate the following probabilities:*
- $P(X < 80)$
- $P(X > 95.5)$
- $P(90 < X < 115)$

An alternative method of calculating these probabilities is to integrate the Normal density function, which unfortunately cannot be integrated analytically. R can perform approximate numerical integration using the function `integrate()`. Probabilities calculated by `pnorm()` are nothing but integrals (areas) of the Normal density computed by `dnorm()`. Thus, an alternative way to calculate $P(90 < X < 115)$ is as follows (this is another example where a function is an argument of another function).

```
> integrate(dnorm, 90, 115, mean = 100, sd = 10)
0.7745375 with absolute error < 8.6e-15
```

How does this compare with what you get using `pnorm()`? Note that it is generally NOT a good idea to use `integrate()` instead of `pnorm()` or other similar functions. Although they solve the same problem, `integrate()` is a general-purpose function while `pnorm()` etc. are tailored to work with specific distributions.

Calculating quantiles. While `pnorm()` calculates probabilities of a random variable being less than or greater than a certain number, `qnorm()` solves the inverse problem: Given a probability p, find a location q such that $P(X < q) = p$. q is called the lower p-th quantile, we can similarly define upper quantiles that satisfy $P(X > q) = p$.

Exercise 12. *For the same random variable X, calculate the lower and upper quantiles corresponding to the probabilities 0.2, 0.4, 0.5, 0.6, and 0.8. Notice that the upper and lower quantiles are symmetric about the mean (100). This does not hold for non-symmetric distributions. Calculate the same quantiles for the Standard Normal distribution $Z \sim N(0,1)$. Verify that the Z-quantiles can be obtained from the X-quantiles by subtracting the mean (100) and dividing by the standard deviation (10).*

Probability as Relative Frequency. Probability theory tells us that if we repeat an experiment a large number of times, the relative frequency of an event A (the number of times the event A occurs as a fraction of the total number times we do the experiment) converges to the probability of A. The goal of the following exercises is to verify this in two examples.

Exercise 13. *If we have a finite number of independent observations $X_1, \ldots, X_n$ from $N(100, 10)$, then the relative frequency of observations that fall between 90 and 115 should converge to $P(90 < X < 115)$ as n increases. Use* `rnorm()` *to obtain a vector of 500 $N(100, 10)$ observations, and plot the cumulative proportions of observations between 90 and 115.*

Hint: use the function `cumsum()`, which works for logical vectors as well as numeric vectors. Remember that you will need the cumulative relative frequencies, i.e., you would need to divide the number of "successes" up to a point by the number of experiments up to that point.

Your plot should look roughly like the following.
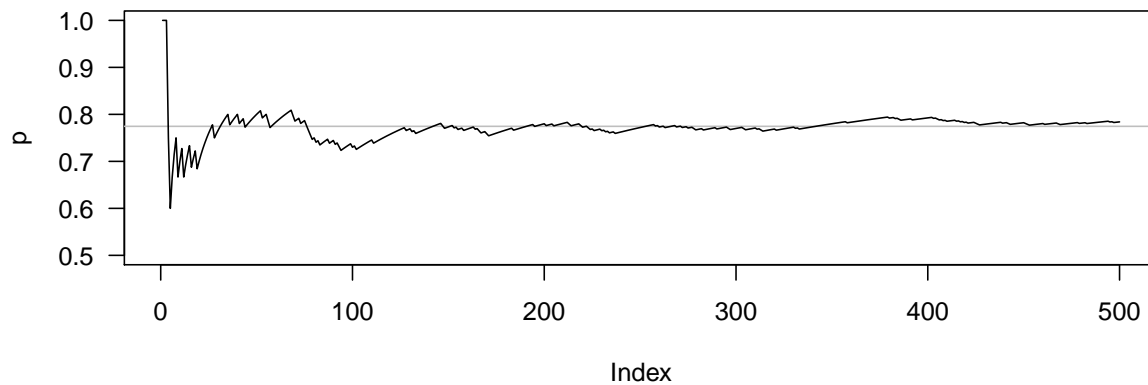
For the next example, consider a regular deck of 52 cards. Suppose our experiment consists of drawing a single card from the full deck, and we are interested in the event

$$A = \{\text{the card drawn is an Ace or from the Clubs suite}\}$$

Exercise 14. *Compute $P(A)$.*

As before, we want to fix the number N of experiments we will do (say $N = 5000$), and simulate the experiment N times to generate a logical (`TRUE`/`FALSE`) vector indicating whether the event A happened for each of the experiments. For $i = 1, 2, \ldots, N$, we then want to calculate (and plot) the relative frequency of A in the first i experiments. The only new step is to actually simulate the experiment N times. The first thing we need is the sample space (the 52 cards). A nice way to do this in R is:

```
> cards <- expand.grid(value = c("A", 2:10, "J", "Q", "K"),
                       suite = c("Clubs", "Diamonds", "Hearts", "Spades"),
                       KEEP.OUT.ATTRS = FALSE)
> head(cards)
  value suite
1     A Clubs
2     2 Clubs
3     3 Clubs
4     4 Clubs
5     5 Clubs
6     6 Clubs
> tail(cards)
   value  suite
47     8 Spades
48     9 Spades
49    10 Spades
```

```
50      J Spades
51      Q Spades
52      K Spades
```

Our sample space thus consists of each row of the data frame `cards`. Simple random sampling with and without replacement from a finite population can be performed by the `sample()` function. We can sample from the row indices using

```
> sample(1:52, N, replace = TRUE)
```

and use that as an index to get a new data frame with N rows, each representing one card drawn at random from the deck.

```
> N <- 5000
> cards.sample <- cards[sample(1:52, N, rep = TRUE), ]
> str(cards.sample)
```
```
'data.frame':        5000 obs. of  2 variables:
 $ value: Factor w/ 13 levels "A","2","3","4",..: 12 1 10 4 12 10 3 9 4 13 ...
 $ suite: Factor w/ 4 levels "Clubs","Diamonds",..: 1 4 2 3 3 2 2 3 1 4 ...
```

The event that the selected card is a Clubs, for instance, can now be obtained (in vectorized form) as `cards.sample$suite == "Clubs"`.

Exercise 15. *Figure out how to determine whether the event A happened for each draw, and from that obtain and plot the cumulative relative frequencies.*