Working with R

A University of Queensland Advanced Workshop

Session 14: Vulgar Fractions in R

Bill Venables, CSIRO/Data61, Dutton Park

Rhetta Chappell, Griffith University

2–5 February, 2021

Contents

1 Why would anyone want to do that?

To see patterns more clearly in numbers, particularly in matrices, for one example. What is the pattern here?

```
Hmat(7)

        [,1]    [,2]    [,3]    [,4]    [,5]    [,6]    [,7]
[1,] 0.50000 0.33333 0.25000 0.20000 0.16667 0.14286 0.12500
[2,] 0.25000 0.20000 0.16667 0.14286 0.12500 0.11111 0.10000
[3,] 0.16667 0.14286 0.12500 0.11111 0.10000 0.09091 0.08333
[4,] 0.12500 0.11111 0.10000 0.09091 0.08333 0.07692 0.07143
[5,] 0.10000 0.09091 0.08333 0.07692 0.07143 0.06667 0.06250
[6,] 0.08333 0.07692 0.07143 0.06667 0.06250 0.05882 0.05556
[7,] 0.07143 0.06667 0.06250 0.05882 0.05556 0.05263 0.05000
```

I'm glad you asked.

```
library(fractional)
Hmat(7) %>% fractional

     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]  1/2  1/3  1/4  1/5  1/6  1/7  1/8
[2,]  1/4  1/5  1/6  1/7  1/8  1/9 1/10
[3,]  1/6  1/7  1/8  1/9 1/10 1/11 1/12
[4,]  1/8  1/9 1/10 1/11 1/12 1/13 1/14
[5,] 1/10 1/11 1/12 1/13 1/14 1/15 1/16
[6,] 1/12 1/13 1/14 1/15 1/16 1/17 1/18
[7,] 1/14 1/15 1/16 1/17 1/18 1/19 1/20

detach("package:fractional", unload = TRUE)
```

So how do we do it? We need:

- A good algorithm for finding vulgar fractions (well enough),
- A coding strategy that does not involve too much work.

2 An algorithm

The idea is old, and well described in old books such as Khovanskii (1963). See the vignettes in Venables (2016).

A continued fraction is a development of the form:

$$b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cdots}}}$$

Stopping gives the *convergents*:

$$\frac{P_n}{Q_n} = b_0 + \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cfrac{a_3}{b_3 + \cdots \qquad \\ \qquad b_{n-1} + \cfrac{a_n}{b_n}}}}$$

Calculate by recurrence. With

$$\frac{P_{-1}}{Q_{-1}} = \frac{1}{0}, \frac{P_0}{Q_0} = \frac{b_0}{1}, \frac{P_1}{Q_1}, \frac{P_2}{Q_2}, \ldots$$

It can be shown that

$$\left.\begin{array}{rcl} P_{n+1} & = & b_{n+1}P_n + a_{n+1}P_{n-1} \\ Q_{n+1} & = & b_{n+1}Q_n + a_{n+1}Q_{n-1} \end{array}\right\} \quad n = 0, 1, 2, \ldots$$

Getting the a_n and b_n.

1. Let x be a real number for which the approximation is wanted.
2. Write $b_0 = \lfloor x \rfloor$ and put $x = b_0 + (x - \lfloor x \rfloor) = b_0 + r_0$.
3. $0 \leq r_0 < 1$, by definition. There are two cases:
   - If $r_0 = 0$ the process is complete. The rational approximation is exact.
   - If $r_0 > 0$, note that $1/r_0 > 1$. Write $1/r_0 = \lfloor 1/r_0 \rfloor + (1/r_0 - \lfloor 1/r_0 \rfloor) = b_1 + r_1$, with $b_1 \geq 1$ and $0 \leq r_1 < 1$. Then:

$$x = b_0 + \frac{1}{1/r_0} = b_0 + \frac{1}{b_1 + r_1}$$

4. Continuing in this way we produce a continued fraction expansion for

the real number of the form:

$$x = b_0 + \cfrac{1}{b_1 + \cfrac{1}{b_2 + \cfrac{1}{b_3 + \cdots}}}$$

In English: The a_i are all 1. Easy. Let x be the number.

- b_0 is the largest whole number less than x. If there is any remainder, r_0:
- b_1 is the largest whole number less than $1/r_0$. If there is any remainder, r_1:
- b_2 is the largest whole number less than $1/r_1$. If there is any remainder, r_2: (you get the picture). Stop when good enough.

3 An R coding

```
#' @describeIn ratAppr Workhorse function for a single value
#' @export
.ratAppr <- function(x, eps = 1.0e-6, maxConv = 20) {
  PQ1 <- c(1, 0)
  PQ2 <- c(floor(x), 1)
  r <- x - PQ2[1]
  i <- 0
  while((i <- i+1) < maxConv && abs(x - PQ2[1]/PQ2[2]) > eps) {
    b <- floor(1/r)
    r <- 1/r - b
    PQ0 <- PQ1
    PQ1 <- PQ2
    PQ2 <- b*PQ1 + PQ0
  }
  return(c(PQ2, i-1))
}
```

3.1 Vectorization

```
ratAppr <- function(x, eps = 1.0e-6, maxConv = 20) {
  vapply(x, FUN = .ratAppr,
         FUN.VALUE = c(Pn = 0, Qn = 0, n = 0),
         eps = eps, maxConv = maxConv)
}
```

Always check as you go:

```
.ratAppr(base::pi)

[1] 355 113   3

ratAppr(c(1:10/7, pi, sqrt(2)))

   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
Pn    1    2    3    4    5    6    1    8    9    10   355  1393
Qn    7    7    7    7    7    7    1    7    7     7   113   985
n     1    2    3    3    4    3    0    1    2     2     3     8
```

Looks OK.

3.2 The main function

```r
vulgar <- function(x, eps = 1.0e-6, maxConv = 20) {
  structure(x, eps = eps, maxConv = maxConv,
            class = c("vulgar", class(x)))
}
```

I.e. do *NOTHING*, but take note of what has been requested and pack it into the object.

The yang to this ying is just as simple:

```r
unvulgar <- function(x) {
  x <- unclass(x)
  attr(x, "eps") <- attr(x, "maxConv") <- NULL
  x
}
```

3.3 Arithmetic

In *S3* land, this is done by giving a method for a *group generic function*, *Ops*. First a couple of helpers:

```
getAttr <- function(x)
  UseMethod("getAttr")
getAttr.default <- function(x)      ## not "vulgar" make it up
  list(eps = 1.0e-6, maxConv = 20)
getAttr.vulgar <- function(x)       ## is "vulgar", get it
  attributes(x)[c("eps", "maxConv")]
```

Now the group generic method. "Just do what you always do":

```
Ops.vulgar <- function (e1, e2) {
  ax <- getAttr(e1)
  e1 <- unclass(e1)
  if (!missing(e2)) {  ## not unary minus or plus
    ax2 <- getAttr(e2)
    ax <- list(eps = min(ax$eps, ax2$eps),
               maxConv = max(ax$maxConv, ax2$maxConv))
    e2 <- unclass(e2)
  }
  res <- NextMethod(.Generic)
  if(typeof(res) == "logical") { ## it was a logical operator
    res
  } else {
    with(ax, vulgar(res, eps = eps, maxConv = maxConv))
  }
}
```

Check as you go:

```
Hmat <- function(k) {
  M <- matrix(0, k, k)
  1/(2 * row(M) + col(M) - 1)
}
vulgar(Hmat(3)) + 1
```

```
      [,1]  [,2]  [,3]
[1,] 1.500 1.333 1.250
[2,] 1.250 1.200 1.167
[3,] 1.167 1.143 1.125
attr(,"eps")
[1] 1e-06
attr(,"maxConv")
[1] 20
attr(,"class")
[1] "vulgar" "matrix" "array"
```

OK, but some work still to do.

3.4 Coercion to character

We do this indirectly. `base::as.character` is a `.Primitive` generic function.

```
as.character.vulgar <- function (x, eps = attr(x, "eps"),
                                 maxConv = attr(x, "maxConv"), ...) {
  x <- unclass(x)
  ax <- attributes(x)
  rx <- ratAppr(as.vector(x), eps = eps, maxConv = maxConv)
  fractions <- sub("/1$", "", paste(rx["Pn", ], rx["Qn", ], sep = "/"))
  ax$maxConv <- ax$eps <- NULL
  attributes(fractions) <- ax
  class(fractions) <- "vulgarCharacter"
  fractions
}
```

The main work comes when we actually *look* at a `vulgar` object.

The function `base::print` is an *S3* generic, so more methods!

3.5 Printing

Zeros as *displayed* as dots, ., to reduce clutter.

```
print.vulgarCharacter <- function(x, ...) {
  y <- x
  x <- gsub("^0$", ".", unclass(x))
  NextMethod("print", quote = FALSE, ...)
  invisible(y)
}
print.vulgar <- function (x, ...) {
  x0 <- x
  y <- gsub("^0$", ".", as.character.vulgar(x))
  y <- format(y, justify = "right")
  ax <- attributes(x)
  ax$class <- ax$eps <- ax$maxConv <- NULL
  x <- do.call("structure", c(list(y), ax))
  NextMethod("print", quote = FALSE, ...)
  invisible(x0)
}
```

Fingers crossed, let's check it:

```r
(vulgar(Hmat(3)) + 1) %>% as.character
```

```
     [,1] [,2] [,3]
[1,] 3/2  4/3  5/4 
[2,] 5/4  6/5  7/6 
[3,] 7/6  8/7  9/8 
```

```r
cbind(diag(1/1:3), 0) - vulgar(cbind(0, diag(1/3:1)))
```

```
     [,1] [,2] [,3] [,4]
[1,]    1 -1/3    .    .
[2,]    .  1/2 -1/2    .
[3,]    .    .  1/3   -1
```

```r
contr.helmert(4) %>% cbind(Ave = 1, .) %>% solve %>% vulgar
```

```
    1      2      3      4     
Ave   1/4    1/4    1/4    1/4
     -1/2    1/2      .      .
     -1/6   -1/6    1/3      .
    -1/12  -1/12  -1/12    1/4
```

4 Moving to compiled code

Stepping it up a notch: going to **C++**.

```cpp
#include <Rcpp.h>
using namespace Rcpp;

IntegerVector ratApp_one(double x, double eps, int maxConv) {
  int p0, p1 = 1, p2 = (int) floor(x),
      q0, q1 = 0, q2 = 1, b, i = 0;
  double z = x - (double) p2;
  while(++i < maxConv) {
    if(fabs(x - (double) p2 / (double) q2) < eps) break;
    z = 1/z; b = (int) floor(z); z = z - b;
    p0 = p1; p1 = p2; p2 = b*p1 + p0;
    q0 = q1; q1 = q2; q2 = b*q1 + q0;
  }
  return IntegerVector::create(p2, q2, i-1);
}

//' @describeIn ratAppr C++ version of the same function
//' @export
```

```
//' @import Rcpp
//' @useDynLib vulgar
// [[Rcpp::export]]
IntegerMatrix ratApp(NumericVector x, double eps = 1.0e-6, int maxConv =
20) {
  int nx = x.length();
  IntegerMatrix PQC(3, nx);
  PQC.attr("dimnames") =
    List::create(CharacterVector::create("Pn", "Qn", "n"),
                 R_NilValue);
  for(int i = 0; i < nx; i++) {
    PQC(_, i) = ratApp_one(x[i], eps, maxConv);
  }
  return PQC;
}
```

Check that it works, too. Using *Rcpp::sourceCpp* creates a function of the same name with an odd appearance

```
ratApp

function (x, eps = 1e-06, maxConv = 20L)
.Call(<pointer: 0x7fe104693e50>, x, eps, maxConv)
```

It should give the same results as the function *ratAppr* written in pure **R**, but faster.

```
u <- runif(11); rbind(R = ratAppr(u), Cpp = ratApp(u))

   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
Pn  470  701  109  376  618  163 1091  168   68   581  1398
Qn  699 1070  235  947 1241  973 1672  179  849   857  3709
n     5    6    5    6    6    6    7    6    5     7     8
Pn  470  701  109  376  618  163 1091  168   68   581  1398
Qn  699 1070  235  947 1241  973 1672  179  849   857  3709
n     5    6    5    6    6    6    7    6    5     7     8

all.equal(ratAppr(u), ratApp(u))

[1] TRUE
```

4.1 Timings

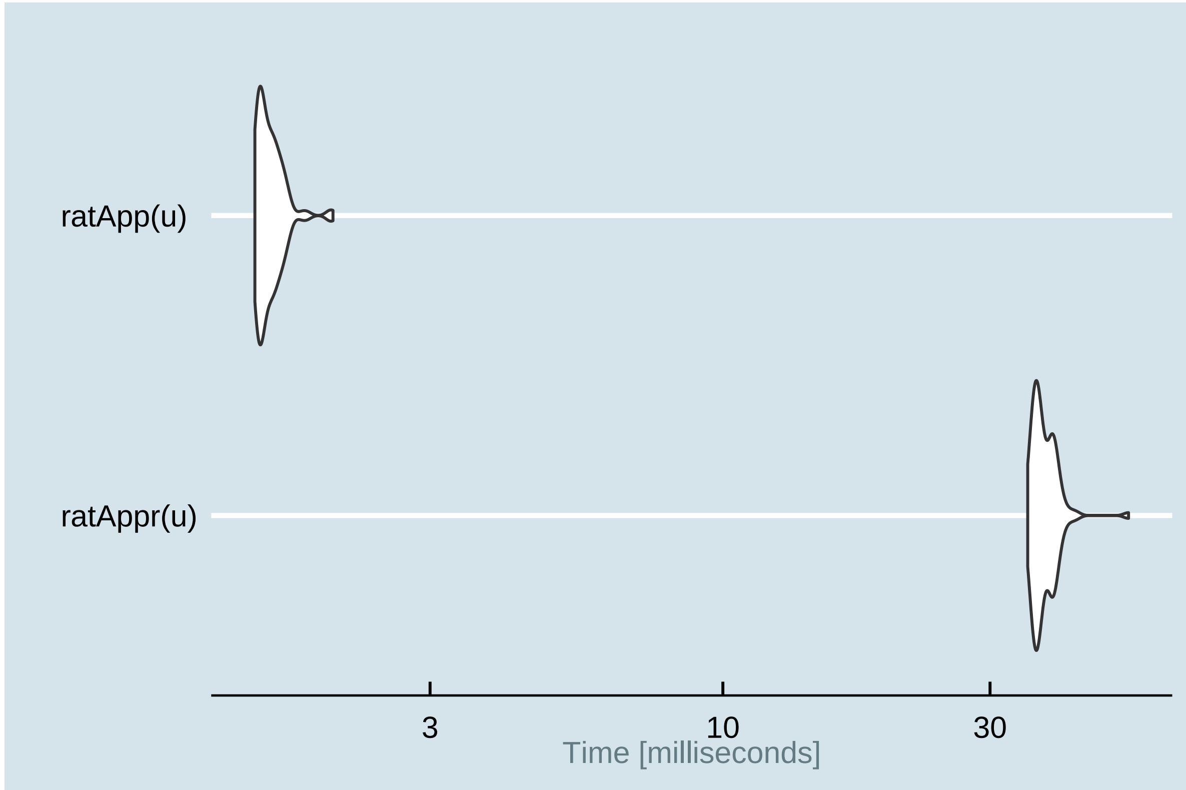
To get a feeling for speed, consider random inputs

```
die <- Sys.time() %>% as.POSIXlt %>% unclass %>%
  (function(x) x$min*x$sec) %>% round
set.seed(die)  ## The die is cast!

library(microbenchmark)
u <- runif(10000)
(bm <- microbenchmark(ratAppr(u), ratApp(u)))
```

```
Unit: milliseconds
       expr   min     lq   mean median     uq    max neval cld
 ratAppr(u) 35.03 36.074 37.538 36.879 38.789 53.025   100   b
  ratApp(u)  1.46  1.491  1.554  1.531  1.593  2.013   100  a
```

```
suppressMessages(autoplot(bm)) + ggthemes::theme_economist()
```



Reality check: The possibility of anyone wanting to look at 10000 numbers in the form of vulgar fractions, even to check for patterns, is low, and if they did, they are probably willing to wait 130 milliseconds or so for them to be calculated. Until further applications arise, this is just a *programming* exercise.

Make the change, and *check it still works*:

```
as.character.vulgar <- function (x, eps = attr(x, "eps"),
                                 maxConv = attr(x, "maxConv"), ...) {
  x <- unclass(x)
  ax <- attributes(x)
  ## the only chenged line is the one below
  rx <- ratApp(as.vector(x), eps = eps, maxConv = maxConv)
  fractions <- sub("/1$", "", paste(rx["Pn", ], rx["Qn", ], sep = "/"))
  ax$maxConv <- ax$eps <- NULL
  attributes(fractions) <- ax
  class(fractions) <- "vulgarCharacter"
  fractions
}
cbind(Ave = 1, contr.helmert(4)) %>% solve %>% vulgar
```

```
    1     2     3     4
Ave   1/4   1/4   1/4   1/4
     -1/2   1/2     .     .
     -1/6  -1/6   1/3     .
    -1/12 -1/12 -1/12   1/4
```

5 Creating a package

The procedure for creating a package from these materials inside **RStudio** is relatively simple.

- Save the **R** functions *in text form* in a file, or files with the file extension `.R`.
- If you want to use the **C++** version, save the text version in a file with file extension `.cpp`. (The upside is that it is fast; the downside is that you must have the tools to build it, of course, and the source package is only portable to others with those tools as well.)
- Open the `New Project` menu in **RStudio**.
  - Specify that you want a `Package` project, preferably in a *new* rather than an existing directory.
  - Add the file names of the files you have just created with the software, both **R** and **C++**, to the box in the menu (but more files can be added later, if need be).

- Click the `Create Project` box. This will re-start the **R** session with your newly created project as the working directory. Your main directory will have several files including templates of the `DESCRIPTION` and `NAMESPACE` files, and sub-directories `R`, `man`, possibly `inst` and `vignettes` if you include `.Rmd` files, and `src` if you are using **C++**. There are the **RStudio** project files as well, which are excluded when the package is built. *Explore* the file structure that has been created.

- At this stage you should be able to click on the `Build` tab and build a very rough first draft of the package. Test it.

- Fill out the `DESCRIOTION` file in the top directory. *Do not change* the `NAMESPACE` file, though.

- Fill out the *roxygen2* comments above all functions you want to have exported, at least. **This is the only fairly big job, and does need to be done carefully.** You will need to refer frequently to the online documentation, which fortunately is fairly comprehensive and clear.[a]

- With the documentation comments completed, Go to the `Build` tab and select `More` and on to `Configure build tools`

  Check the box that asks if you want your `NAMESPACE` file to be constructed from your documentation comments—you do—and any other boxes you thing might be relevant.

- Think about adding a small `.Rmd` vignette to the package. To do, create a `vignettes` (plural) sub-directory, and place your markdown file(s) into it. Then check the box that asks if you want the vignettes re-built. Also add a line

  ```
  VignetteBuilder: knitr
  ```

  to the `DESCRIPTION` file.

- To initialize a vignette, it often pays to build a first cut using

[a] The clear advantage of using inline documentation is that the computer does all of the really tedious editing and correcting, and ensuring consistency, rather than you. It is well worth the trouble of learning how to use it. But it is something that has to be learned.

```
devtools::build_vignettes()
```

but once the process it started, it should automatically update as things change.

If you do simply open an `.Rmd` file while in a package project, much of the detail should be handled automatically, but check it!

- Re-build the package and hope for the best! It sometimes pays to use

  ```
  devtools::document()
  ```

  beforehand, and check if the help information (in the `man` sub-directory) looks OK, as well as some modifications elsewhere, e.g. to `DESCRIPTION` again.

- Using the `Build` tools, check the package. You may want to configure the build tools to add "`--as-cran`" to the checking options, if you wish ultimately to publish it on `CRAN` (but there is already at least one `vulgar` package on CRAN, namely `fractional`).

- Build a source package if you want to pass it on to colleagues.

References

Khovanskii, A. N. (1963). *The Application of Continued Fractions and Their Generalizations to Problems in Approximation Theory*. P. Noordhoff N. V. Translated by Peter Wynn.

Venables, W. N. (2016). *fractional: Vulgar Fractions in R*. CSIRO, Australia. R package version 0.1.3.

Session information

Date: 2021-01-29

- R version 4.0.3 (2020-10-10), `x86_64-pc-linux-gnu`
- Locale: `LC_CTYPE=en_AU.UTF-8`, `LC_NUMERIC=C`, `LC_TIME=en_AU.UTF-8`, `LC_COLLATE=en_AU.UTF-8`, `LC_MONETARY=en_AU.UTF-8`, `LC_MESSAGES=en_AU.UTF-8`, `LC_PAPER=en_AU.UTF-8`, `LC_NAME=C`, `LC_ADDRESS=C`, `LC_TELEPHONE=C`, `LC_MEASUREMENT=en_AU.UTF-8`, `LC_IDENTIFICATION=C`
- Running under: `Ubuntu 20.04.1 LTS`
- Matrix products: default
- BLAS: `/usr/lib/x86_64-linux-gnu/blas/libblas.so.3.9.0`
- LAPACK: `/usr/lib/x86_64-linux-gnu/lapack/liblapack.so.3.9.0`
- Base packages: base, datasets, graphics, grDevices, methods,

parallel, stats, utils

- Other packages: doParallel 1.0.16, dplyr 1.0.3, english 1.2-5, forcats 0.5.1, foreach 1.5.1, GGally 2.1.0, ggplot2 3.3.3, ggthemes 4.2.4, gridExtra 2.3, haven 2.3.1, iterators 1.0.13, knitr 1.31, lattice 0.20-41, lme4 1.1-26, Matrix 1.3-2, mboost 2.9-4, mgcv 1.8-33, microbenchmark 1.4-7, nlme 3.1-151, patchwork 1.1.1, purrr 0.3.4, randomForest 4.6-14, rbenchmark 1.0.0, Rcpp 1.0.6, readr 1.4.0, rpart 4.1-15, scales 1.1.1, SOAR 0.99-11, stabs 0.6-3, stringr 1.4.0, tibble 3.0.5, tidyr 1.1.2, tidyverse 1.3.0, visreg 2.7.0, WWRCourse 0.2.3, WWRData 0.1.0, WWRGraphics 0.1.2, WWRUtilities 0.1.2, xtable 1.8-4

- Loaded via a namespace (and not attached): assertthat 0.2.1, backports 1.2.1, boot 1.3-26, broom 0.7.3, cellranger 1.1.0, cli 2.2.0, codetools 0.2-18, colorspace 2.0-0, compiler 4.0.3, crayon 1.3.4, DBI 1.1.1, dbplyr 2.0.0, digest 0.6.27, ellipsis 0.3.1, evaluate 0.14, fansi 0.4.2, farver 2.0.3, Formula 1.2-4, fractional 0.1.3, fs 1.5.0, generics 0.1.0, glue 1.4.2, grid 4.0.3, gtable 0.3.0, highr 0.8, hms 1.0.0, httr 1.4.2, inum 1.0-1, jsonlite 1.7.2, lazyData 1.1.0, libcoin 1.0-7, lifecycle 0.2.0, lubridate 1.7.9.2, magrittr 2.0.1, MASS 7.3-53, minqa 1.2.4, modelr 0.1.8, multcomp 1.4-15, munsell 0.5.0, mvtnorm 1.1-1, nloptr 1.2.2.2, nnls 1.4, partykit 1.2-11, PBSmapping 2.73.0, pillar 1.4.7, pkgconfig 2.0.3, plyr 1.8.6, quadprog 1.5-8, R6 2.5.0, RColorBrewer 1.1-2, readxl 1.3.1, reprex 1.0.0, reshape 0.8.8, rlang 0.4.10, rstudioapi 0.13, rvest 0.3.6, sandwich 3.0-0, splines 4.0.3, statmod 1.4.35, stringi 1.5.3, survival 3.2-7, TH.data 1.0-10, tidyselect 1.1.0, tools 4.0.3, vctrs 0.3.6, withr 2.4.1, xfun 0.20, xml2 1.3.2, zoo 1.8-8