

# Efektivní R

—  
Statistický workshop Mariánská (podzim 2019)

Lubomír Štěpánek<sup>1, 2</sup>



<sup>1</sup>Oddělení biomedicínské statistiky  
Ústav biofyziky a informatiky  
1. lékařská fakulta  
Univerzita Karlova v Praze



<sup>2</sup>Katedra biomedicínské informatiky  
Fakulta biomedicínského inženýrství  
České vysoké učení technické v Praze

9. listopadu 2019

(2019) Lubomír Štěpánek, CC BY-NC-ND 3.0 (CZ)



Dílo lze dále svobodně šířit, ovšem s uvedením původního autora a s uvedením původní licence. Dílo není možné šířit komerčně ani s ním jakkoliv jinak nakládat pro účely komerčního zisku. Dílo nesmí být jakkoliv upravováno. Autor neručí za správnost informací uvedených kdekoli v předložené práci, přesto vynaložil nezanedbatelné úsilí, aby byla uvedená fakta správná a aktuální, a práci sepsal podle svého nejlepšího vědomí a svých „nejlepších“ znalostí problematiky.

# Obsah

- 1 Úvod
- 2 Efektivní kód
- 3 Input/Output (I/O)
- 4 Paralelizace
- 5 C++ v R
- 6 Dynamizace
- 7 Tvorba balíčku
- 8 Reference





# Intermezzo – hexbiny

- pravidelné šestiúhelníkové samolepky fixních definovaných rozměrů
- typické (nejen) pro R komunitu
- více na

<http://hexb.in/>



laptop

Maxwella Ogdena

## Některá evangelia efektivní syntaxe a sémantiky

- 1 Miluj syntax svou!
- 2 Budeš vektorizovat!
- 3 Vektory Tobě svěřené nenecháš iterativně růst!
- 4 Budeš ctít funkce rodiny `apply()` Tobě představené!
- 5 Ne-`for()`-smilníš!
- 6 Paralelizuj, jsi-li hoden.
- 7 Mocné umění C++ v R dobře skrývej před nepřítelem!

# Efektivní syntaxe

- smart code má svá pravidla, i když se tím rychlost jeho provedení nezvýší
  - na jeden řádek patří maximálně 80 znaků
  - indentace kódu je vhodná, alespoň dvě mezery (lépe čtyři) na každou úroveň
  - názvy proměnných a funkcí je vhodné sjednotit, např. proměnné pomocí `snake_notace` a funkce pomocí `camelCaps` notace



# Efektivní syntaxe

- je vhodné dodržovat mezery
- např.

1 ||  $x < -5$

může znamenat

```
1 || x <- 5 # at se x rovná pěti
```

ale i

```
1 || x < -5 # je x menší než -5?
```

pak záleží na kontextu (zda je někdy předtím definováno  $x$ )

# Efektivní syntaxe

- je výhodné se vyhnout adresaci dolarem typu `$var` a nahradit ji adresací typu `[, var]`
- důvodem je umožnění dynamického iterování v případě adresace typu `[, var]`

```
1   for(my_variable in colnames(mtcars)){
2       cat(
3           mean(mtcars$my_variable)
4       )
5       cat("\n")
6   }   # nebude fungovat
7
8   for(my_variable in colnames(mtcars)){
9       cat(
10          mean(mtcars[, my_variable])
11      )
12      cat("\n")
13  }   # funguje
```

# Prealokace

- objekty je vhodné v kódu „prealokovat“, tj. přiřadit jim iniciálně jejich datovou strukturu, byť jsou hodnotami populovány až později

```
1  bezAlokace <- function(n){  
2      my_output <- NULL  
3  
4      for(i in 1:n){my_output <- c(my_output, i)}  
5  
6      return(my_output)  
7  }  
8  
9  sAlokaci <- function(n){  
10     my_output <- rep(0, n)  
11  
12     for(i in 1:n){my_output[i] <- i}  
13  
14     return(my_output)  
15 }
```

# Prealokace

- porovnání rychlosti exekuce obou funkcí

```
1 library(microbenchmark)
2
3 n <- 5000
4
5 microbenchmark(
6   times = 100,
7   unit = "ns",
8   bezAlokace(n),
9   sAlokaci(n)
10 )
11 # Unit: nanoseconds
12 # bezAlokace(n) mean 36479684.57 median 35515340
13 # sAlokaci(n) mean 401711.53 median 387873
```

- prealokované objekty jsou exekuvány obecně rychleji

# Vektorizace

- čím méně používá R-ková funkce nebo procedura R-kových funkcionalit, než se „dostane“ k původním C++ procedurám, tím bude obecně rychleji provedena
- proto je výhodnější pracovat s vektorem než se skalárem, nad kterým je třeba provést ještě další operace

```
1  bezVektorizace <- function(n){  
2      x <- NULL  
3      for(i in 1:n){x <- c(x, runif(1) + 1)}  
4      return(x)  
5  }  
6  
7  sVektorizaci <- function(n){  
8      y <- runif(n) + 1  
9      return(y)  
10 }
```

# Vektorizace

- porovnání rychlosti exekuce obou funkcí

```
1 library(microbenchmark)
2
3 n <- 5000
4
5 microbenchmark(
6   times = 100,
7   unit = "ns",
8   bezVektorizace(n),
9   sVektorizaci(n)
10 )
11 # Unit: nanoseconds
12 # bezVektorizace(n) mean 50419159.4 median 48017275
13 # sVektorizaci(n) mean 197811.4 median 167637
```

- přístup s vektorizací je exekuvován obecně rychleji

# Intermezzo

- Najděme pomocí Monte-Carlo integrace velikost určitého integrálu

$$\int_0^1 x^2 dx.$$

# Intermezzo

- Najděme pomocí Monte-Carlo integrace velikost určitého integrálu

$$\int_0^1 x^2 dx.$$

- zřejmě je

$$\int_0^1 x^2 dx = \left[ \frac{x^3}{3} \right]_0^1 = \frac{1}{3} - 0 = \frac{1}{3}.$$



- 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99

$$\int_0^1 x^2 \mathrm{d}x.$$

```
1      # možné řešení
2
3      n_of_hits <- 0
4      for(i in 1:100000){
5          if(runif(1) < runif(1) ^ 2){
6              n_of_hits <- n_of_hits + 1
7          }
8      }
9      n_of_hits / 100000
10     # 0.333
```

# Intermezzo

- Najděme pomocí Monte-Carlo integrace velikost určitého integrálu

$$\int_0^1 x^2 dx.$$

```
1 | # lépe však
2 |
3 | mean(runif(100000) <= runif(100000) ^ 2)
4 | # 0.333
```

# Intermezzo

- Přepište následující kód pomocí vektorizace.

```
1  c <- 0
2
3  for(i in 1:10000){
4
5      x <- runif(1)
6      y <- runif(1)
7      if(x <= y){c <- c + 1}
8
9  }
10
11 c / 10000
```

# Intermezzo

- Z intervalu  $\langle 0, 1 \rangle$  náhodně vybereme dvě čísla  $x$  a  $y$ . Určeme, s jakou pravděpodobností platí

$$\frac{\max\{x^2, y^2\}}{\min\{x, y\}} \geq 2.$$

# Prodlužování vektorů

- prodlužování vektoru během iterace pomocí klauzule `x <- c(x, ...)` je obecně pomalé a je lepší se mu vyhnout
- vytvořme vektor třetích mocnin čísel 1 až 1000

# Prodlužování vektorů

- prodlužování vektoru během iterace pomocí klauzule `x <- c(x, ...)` je obecně pomalé a je lepší se mu vyhnout
- vytvořme vektor třetích mocnin čísel 1 až 1000

```
1 | # prodlužování vektoru
2 | x <- NULL
3 | for(i in 1:1000){x <- c(x, i ^ 3)}
4 |
5 | # prealokace
6 | x <- rep(0, 1000)
7 | for(i in 1:1000){x[i] <- i ^ 3}
8 |
9 | # vektorizace
10 | x <- c(1:1000) ^ 3
```

# Prodlužování vektorů

- prodlužování vektoru během iterace pomocí klauzule `x <- c(x, ...)` je obecně pomalé a je lepší se mu vyhnout
- vytvořme vektor třetích mocnin čísel 1 až 1000

```
1 | my_start <- Sys.time()
2 | x <- NULL # prodlužování vektoru
3 | for(i in 1:1000){x <- c(x, i ^ 3)}
4 | my_stop <- Sys.time(); my_stop - my_start # 0.050s
5 |
6 | my_start <- Sys.time()
7 | x <- rep(0, 1000) # prealokace
8 | for(i in 1:1000){x[i] <- i ^ 3}
9 | my_stop <- Sys.time(); my_stop - my_start # 0.037s
10 |
11 | my_start <- Sys.time()
12 | x <- c(1:1000) ^ 3 # vektorizace
13 | my_stop <- Sys.time(); my_stop - my_start # 0.015s
```

# Intermezzo

- 1 Určeme pomocí vektorizace počet všech čísel menších než 1000, která po dělení 7 vracejí zbytek 2.
- 2 Najdeme pomocí vektorizace všechny kladné celočíselné dělitele čísla 7278548.
- 3 Najdeme pomocí vektorizace všechna čísla menší než 1000, která jsou dělitelná 5, 7 a 8.
- 4 Najdeme pomocí vektorizace největší společný dělitel a nejmenší společný násobek čísel 22375 a 63366.
- 5 Určeme pomocí vektorizace, zda je číslo 4732363 prvočíslem.



# Caching proměnných

- jev *caching* známe z webového prohlížeče – statické struktury typu obrázky se tzv. cachují, tj. dočasně se stahují desktopově, aby se při opětovném nahrání stránky ihned znovu nahrály
- např. chceme každý člen matice vydělit součtem celé matice

```
1 | # méně vhodně
2 | set.seed(1)
3 | my_matrix <- matrix(runif(10000), nrow = 100)
4 |
5 | apply(
6 |   my_matrix,
7 |   2,
8 |   function(i){i / sum(my_matrix)}
9 | )
```

# Caching proměnných

- např. chceme každý člen matice vydělit součtem celé matice

```
1 | # lepší řešení
2 |
3 | set.seed(1)
4 | my_matrix <- matrix(runif(10000), nrow = 100)
5 | sum_of_my_matrix <- sum(my_matrix)
6 |
7 | apply(
8 |   my_matrix,
9 |   2,
10 |   function(i){i / sum_of_my_matrix}
11 | )
```

# Rodina funkcí \*apply()

- jde o funkce dobře optimalizované tak, že v rámci svého vnitřního kódu „co nejdříve“ volají C++ ekvivalenty R-kové funkce
- díky tomu jsou exekučně rychlé
- nejužitečnější je `apply()` a `lapply()`

# Funkce `apply()`

- vrací vektor výsledků funkce FUN nad maticí či datovou tabulkou X, kterou čte po řádcích (`MARGIN = 1`), nebo sloupcích (`MARGIN = 2`)
- syntaxe je `apply(X, MARGIN, FUN, ...)`

```
1 | apply(mtcars, 2, mean)
2 |
3 | my_start <- Sys.time()
4 | x <- apply(mtcars, 2, mean)
5 | my_stop <- Sys.time(); my_stop - my_start # 0.019s
6 |
7 | my_start <- Sys.time()
8 | x <- NULL
9 | for(i in 1:dim(mtcars)[2]){
10 |   x <- c(x, mean(mtcars[, i]))
11 |   names(x)[length(x)] <- colnames(mtcars)[i]
12 | }
13 | my_stop <- Sys.time(); my_stop - my_start # 0.039s
```

# Funkce `lapply()`

- vrací list výsledků funkce FUN nad vektore či listem X
- syntaxe je `lapply(X, FUN, ...)`
- vhodná i pro adresaci v listu

```
1      set.seed(1)
2      my_long_list <- lapply(
3          sample(c(80:120), 100, TRUE),
4          function(x) sample(
5              c(50:150), x, replace = TRUE
6          )
7      )      # list vektorů náhodné délky
8             # generovaných z náhodných čísel
9
10     lapply(my_long_list, "[", 14)
11           # z každého prvku listu (vektoru)
12           # vybírám jen jeho 14. prvek
```

# Náhrada for cyklu funkcí lapply()

- lapply se hodí pro přepis for() cyklu do vektorizované podoby
- obě procedury jsou ekvivalentní stran výstupu, lapply() je významně rychlejší

```
1      # for cyklus
2      x <- NULL
3      for(i in 1:N){
4          x <- c(x, FUN)
5      }
6
7      # lapply
8      x <- unlist(
9          lapply(
10             1:N,
11             FUN
12          )
13      )
```

# Náhrada for cyklu funkcí lapply()

- příklad s odhadem času exekuce kódu

```
1  # for cyklus
2  my_start <- Sys.time()
3
4  for_x <- NULL
5  for(i in 1:100000){for_x <- c(for_x, i ^ 5)}
6
7  my_stop <- Sys.time(); my_stop - my_start # 18.45s
8
9  # lapply
10 my_start <- Sys.time()
11
12 lapply_x <- unlist(lapply(
13   1:100000,
14   function(i) i ^ 5    # koncept anonymní funkce
15 ))
16
17 my_stop <- Sys.time(); my_stop - my_start # 0.10s
```

# Real-time výpis do konzole při iterování

- při iterativních procesech typu `for` cyklus trvajících delší uživatelský čas je vhodné mít představu, v jaké fázi se proces kdy nachází<sup>1</sup>
- klíčem je použití příkazu `flush.console()` před iterací

```
1 | x <- NULL
2 |
3 | flush.console()
4 | for(i in 1:100000){
5 |     x <- c(x, i ^ 5)
6 |     cat(
7 |         paste(
8 |             "Proces hotov z ",
9 |             i / 100000 * 100,
10 |             " %.\n", sep = " "
11 |         )
12 |     )
13 | }
```

<sup>1</sup>zvlášť je-li lineární





# Dynamické iterování

- dynamické iterování umožňuje aplikovat jednu proceduru nebo funkci opakovaně na mnoho podobných objektů pomocí krátkého kódu
- chtějme například vytvořit 26 vektorů  $a, b, \dots, z$  tak, že  $j$ -tý vektor je tvořen právě  $j$  jedničkami, kde  $j$  odpovídá indexu názvu vektoru v rámci anglické abecedy, tedy např. pro  $a$  je  $j = 1$ , pro  $c$  je  $j = 3$  apod.

```
1      # naivní řešení
2
3      a <- c(1)
4      b <- c(1, 1)
5      c <- c(1, 1, 1)
6      # ...
```

# Dynamické iterování

- chtějme například vytvořit 26 vektorů  $a, b, \dots, z$  tak, že  $j$ -tý vektor je tvořen právě  $j$  jedničkami, kde  $j$  odpovídá indexu názvu vektoru v rámci anglické abecedy, tedy např. pro  $a$  je  $j = 1$ , pro  $c$  je  $j = 3$  apod.

```
1      # o něco lepší řešení
2
3      a <- rep(1, which(letters == "a"))
4      b <- rep(1, which(letters == "b"))
5      c <- rep(1, which(letters == "c"))
6      # ...
```

# Dynamické iterování

- chtějme například vytvořit 26 vektorů  $a, b, \dots, z$  tak, že  $j$ -tý vektor je tvořen právě  $j$  jedničkami, kde  $j$  odpovídá indexu názvu vektoru v rámci anglické abecedy, tedy např. pro  $a$  je  $j = 1$ , pro  $c$  je  $j = 3$  apod.

```
1 | # řešení s dynamickým iterováním
2 |
3 | for(my_letter in letters){
4 |
5 |     assign(
6 |         my_letter,
7 |         rep(1, which(letters == my_letter))
8 |     )
9 |
10 | }
```

# Dynamické iterování

- nyní chceme všechny vektory  $a$ ,  $b$ ,  $\dots$ ,  $z$  vynásobit číslem 2 a přičíst ke každé jeho složce vždy náhodnou hodnotu bílého šumu  $\mathcal{N}(0, 1^2)$

```
1 | # naivní řešení
2 |
3 | a <- 2 * a + rnorm(1)
4 | b <- 2 * b + rnorm(1)
5 | c <- 2 * c + rnorm(1)
6 | # ...
```

# Dynamické iterování

- nyní chceme všechny vektory  $a, b, \dots, z$  vynásobit číslem 2 a přičíst ke každé jeho složce vždy náhodnou hodnotu bílého šumu  $\mathcal{N}(0, 1^2)$

```
1      # řešení s dynamickým iterováním
2
3      for(my_letter in letters){
4
5          my_vector <- get(my_letter)
6
7          assign(
8              my_letter,
9              2 * my_vector + rnorm(1)
10         )
11
12     }
```

## Funkce do.call()

- máme-li list argumentů, pro který chceme volat danou funkci, lze využít příkaz `do.call()`

```
1 my_list <- list(1:10, trim = 0, na.rm = TRUE)
2
3 do.call(what = "mean", args = my_list) # 5.5
4
5 # odpovídá příkazu
6 # mean(1:10, trim = 0, na.rm = TRUE)
```

# Funkce `do.call()`

- funkce `do.call()` umožňuje iterovat nad funkcemi
- předpokládejme, že chceme pro vektory hodnot  $x$  a  $(y)$  zjistit postupně průměr, minimum, maximum, medián, směrodatnou odchylku, rozptyl a vždy poslední cifru čísla

```
1  set.seed(1)
2  x <- floor(runif(100) * 100)
3  set.seed(2)
4  y <- floor(runif(100) * 100)
5
6  # možné řešení
7  mean(x)
8  min(x)
9  # ...
10 lapply(x, function(i) i %% 10)
11 mean(y)
12 # ...
13 lapply(y, function(i) i %% 10)
```



# Funkce do.call()

- lépe však

```
1  set.seed(1)
2  x <- floor(runif(100) * 100)
3  set.seed(2)
4  y <- floor(runif(100) * 100)
5
6  for(my_vector_name in c("x", "y")){
7    my_vector <- get(my_vector_name)
8    for(my_function in c(
9      "mean", "min",
10     "max", "median",
11     "sd", "var",
12     function(i) i %% 10
13   )){
14     cat(do.call(my_function, list(my_vector)))
15     cat("\n")
16   }
17 }
```













# Literatura

-  Hadley Wickham. *Advanced R*. Boca Raton, FL: CRC Press, 2015. ISBN: 978-1466586963.
-  Colin Gillespie. *Efficient R programming : a practical guide to smarter programming*. Sebastopol, CA: O'Reilly Media, 2016. ISBN: 978-1491950784.

Děkuji za pozornost!

lubomir.stepanek@vse.cz

lubomir.stepanek@lf1.cuni.cz

lubomir.stepanek@fbmi.cvut.cz

► GitHub

[https://github.com/LStepanek/marianska\\_podzim\\_2019](https://github.com/LStepanek/marianska_podzim_2019)