Técnicas de programação

Fernando Mayer
2019-08-29 (última atualização 2026-04-23)



Introdução

- "O R não é uma linguagem rápida". Não é por acaso:
  - Foi desenvolvido para análise de dados interativa
  - Não foi pensado em ser o mais eficiente possível
  - Para a maioria das tarefas, é rápido o bastante
- Por isso, é importante:
  - Identificar os "gargalos" no código, que podem ser otimizados
  - Identificar expressões que causam erros, e como lidar com estes erros
- Técnicas gerais de programação:
  - Identificar e arrumar problemas de performance
  - Identificar e arrumar erros (*bugs*)

Introdução

As técnicas que precisamos conhecer são:

Performance de código

- **Benchmarking** é o processo de avaliar a performance de operações específicas repetidamente.
- **Profiling** é o processo de fazer o benchmark para cada instrução de uma função/rotina.

Tratamento de erros

- **Debugging** é o processo de buscar e resolver erros. Técnicas de debugging permitem avaliar a função passo a passo quando chamada.
- **Error handling** (*tratamento de erro*) é o processo de "prever" possíveis erros e fazer alguma ação quando ocorrerem.
- Também conhecido por *exception handling* (tratamento de excessão)

Performance de código

Pensando em performance

Antes de medir ou otimizar, vale internalizar uma ordem natural para atacar problemas de desempenho no R:

1. **Vetorizar** sempre que possível (eliminar loops desnecessários)
2. **Pré-alocar** memória quando o loop for inevitável
3. **Medir** (benchmarking) para comparar alternativas
4. **Localizar** o gargalo (profiling) em código maior
5. (avançado) Traduzir o gargalo para C++, paralelizar, etc.

- Os passos 1 e 2 resolvem a enorme maioria dos problemas de "R lento"
- Só depois faz sentido medir com cuidado e perfilar
- Nesta aula cobrimos 1–4

Vetorização

- Muitos operadores e funções do R atuam diretamente sobre **vetores inteiros**
- Um loop em R "nu" avalia cada iteração no interpretador — lento
- A mesma operação vetorizada roda no código C compilado por trás

Considere calcular x_i^2 para $i = 1, \ldots, n$:

Loop explícito:

```
x <- 1:1e6
out <- numeric(length(x))
for(i in seq_along(x)){
    out[i] <- x[i]^2
}
```

Vetorizado:

```
x <- 1:1e6
out <- x^2
```

- A versão vetorizada é mais **rápida**, mais **curta** e mais **legível**
- Sempre que possível, esta deve ser a **primeira** otimização
- Uma boa prática: ao escrever um loop, pergunte-se *"isto não poderia ser uma única expressão vetorial?"*

Vetorização

Comparando as duas versões:

```
library(microbenchmark)
x <- 1:1e6

sq_loop <- function(x) {
    out <- numeric(length(x))
    for(i in seq_along(x)) out[i] <- x[i]^2
    out
}
sq_vec <- function(x) x^2

microbenchmark(sq_loop(x), sq_vec(x), times = 20)
```

```
# Unit: microseconds
#        expr       min       lq      mean    median        uq       max neval cld
#  sq_loop(x) 29791.944 31597.17 31750.359 31829.729 32149.818 32944.661    20  a
#   sq_vec(x)   861.255   903.45  1228.392  1004.712  1116.245  2315.251    20   b
```

Muitas funções do R já são vetorizadas:

- Aritméticas: `+`, `-`, `*`, `/`, `^`
- Matemáticas: `sqrt`, `log`, `exp`, `sin`, `abs`
- Comparações: `<`, `<=`, `==`, `!=`
- Condicionais: `ifelse`, `pmin`, `pmax`
- Sumários por linha/coluna: `rowSums`, `colMeans`, `rowMeans`

Vetorização — quando o loop é inevitável

Nem tudo vetoriza diretamente: recursões, dependências entre iterações, e alguns algoritmos iterativos precisam de loop. Nesses casos:

- **Mover** para fora do loop tudo que não depende da iteração
- **Pré-alocar** o vetor/lista de saída com o tamanho final
- **Evitar** expressões do tipo `c(x, novo)` ou `rbind(df, linha)` dentro do loop

Ruim — cresce o vetor:

```
out <- c()
for(i in 1:n){
    out <- c(out, f(i))
}
```

Melhor — pré-aloca:

```
out <- numeric(n)
for(i in 1:n){
    out[i] <- f(i)
}
```

Na próxima seção veremos **por que** crescer o vetor é tão lento no R.

Semântica de cópia (*copy-on-modify*)

O R segue uma regra chamada **copy-on-modify**:

> *Se um objeto é modificado e há outro nome apontando para ele, o R faz uma cópia antes de modificar — garantindo que o outro nome continue vendo o objeto original.*

Essa é a **garantia semântica**: você nunca modifica "por acidente" um objeto compartilhado. A implementação é mais esperta do que a regra sugere, como veremos em seguida.

A função base `tracemem()` avisa toda vez que um objeto é efetivamente copiado na memória:

```
x <- 1:5
y <- x               # duas referências ao mesmo objeto
tracemem(x)
# [1] "<0x560c12a34b88>"

y[1] <- 10L          # precisa copiar para preservar x
# tracemem[0x560c12a34b88 -> 0x560c12b01c40]:

untracemem(x)
```

- Cada linha `tracemem[...]` impressa **é uma cópia** do objeto na memória

Semântica de cópia — inspecionando com `lobstr`

O pacote `lobstr` oferece ferramentas mais claras para inspecionar endereços e tamanhos em memória:

```
library(lobstr)
x <- 1:5
obj_addr(x)   # endereço
```

```
# [1] "0x55718fef4620"
```

```
obj_size(x)   # tamanho
```

```
# 680 B
```

```
y <- x        # NÃO copia!
obj_addr(y)   # mesmo endereço
```

```
# [1] "0x55718fef4620"
```

```
y[1] <- 10L   # agora copia
obj_addr(y)   # outro endereço
```

```
# [1] "0x5571a1809bd8"
```

```
obj_addr(x)   # x inalterado
```

```
# [1] "0x55718fef4620"
```

- Atribuição simples (`y <- x`) **não** copia: cria apenas um novo **nome** apontando para o mesmo objeto
- A cópia só acontece quando a modificação **precisa** ocorrer por baixo dos panos para preservar a semântica
- O R mantém uma contagem interna de quantos nomes referenciam cada objeto

Modificação *in place* — a otimização

Aqui aparece o aspecto sutil: se há **apenas uma referência** ao objeto, o R pode modificá-lo *in place* com segurança — a garantia de copy-on-modify continua valendo (ninguém mais vê o objeto), mas a cópia deixa de ser necessária.

```
x <- c(1, 2, 3, 4, 5)  # uma referência
tracemem(x)
x[1] <- 10             # SEM cópia!
x[2] <- 20             # SEM cópia!
## Nenhum tracemem[...] é
## impresso
```

```
x <- c(1, 2, 3, 4, 5)
y <- x                 # duas referências
tracemem(x)
x[1] <- 10
## tracemem[...]: <- copiou
x[2] <- 20
## silêncio: agora só x aponta
## para este objeto novo
```

Resumo: copy-on-modify é a **garantia**; modificação *in place* é a **implementação** quando há uma única referência.

Copy-on-modify e o loop que cresce o vetor

Para o R modificar *in place*, **duas** condições precisam estar presentes:

1. **Uma única referência** ao objeto (para não violar copy-on-modify)
2. **Tamanho do objeto inalterado** (para reaproveitar a memória já alocada)

Quando o vetor **cresce**, a condição (2) é violada: mesmo com uma única referência, é preciso alocar um bloco maior e copiar o conteúdo anterior.

```
x <- 1:1e4
out <- numeric(0)
tracemem(out)
for(i in seq_along(x)){
    out[i] <- x[i]^2   # estende o vetor → realocação frequente
}
tracemem(out)
```

O custo total desse padrão é aproximadamente **quadrático** no tamanho do vetor.

Copy-on-modify e pré-alocação

Pré-alocando, **ambas** as condições são satisfeitas: uma única referência e tamanho fixo desde o início.

```
x <- 1:1e4
out <- numeric(length(x))
tracemem(out)
for(i in seq_along(x)){
    out[i] <- x[i]^2   # modifica in place — sem cópias
}
tracemem(out)
```

- O vetor já nasce com o tamanho final → não precisa crescer
- Apenas `out` referencia o objeto → R pode modificar *in place*
- Resultado: custo **linear** no tamanho do vetor

Regra prática: pré-alocar + evitar criar referências desnecessárias é quase sempre o caminho. Sempre que precisar comparar alternativas, use `bench::mark()` — ele mostra tanto o tempo quanto a memória alocada, expondo diretamente o efeito das cópias.

Benchmarking

Funções úteis:

- `base::system.time()`: apenas uma execução
- `microbenchmark::microbenchmark()`: avalia a expressão várias vezes
- `bench::mark()`: moderna, também mede memória alocada

Usando `system.time()`

```
(df <- data.frame(v = 1:4, name = letters[1:4]))
```

```
#   v name
# 1 1    a
# 2 2    b
# 3 3    c
# 4 4    d
```

- Cada expressão precisa ser executada em uma linha
- Cada expressão é executada uma única vez

```
system.time(df[3, 2])
```

```
#    user  system elapsed
#       0       0       0
```

```
system.time(df[3, "name"])
```

```
#    user  system elapsed
#       0       0       0
```

```
system.time(df$name[3])
```

```
#    user  system elapsed
#       0       0       0
```

Benchmarking

Usando `microbenchmark()`

```
library(microbenchmark)
(mb1 <- microbenchmark(df[3, 2], df[3, "name"], df$name[3]))
```

```
# Unit: nanoseconds
#           expr  min     lq    mean median     uq   max neval cld
#       df[3, 2] 4181 4339.5 4860.40 4457.0 4558.5 30792   100  a
#  df[3, "name"] 4155 4372.0 4577.15 4447.5 4527.0  8650   100  a
#     df$name[3]  372  415.5  494.28  436.5  487.0  2573   100   b
```

Outro exemplo:

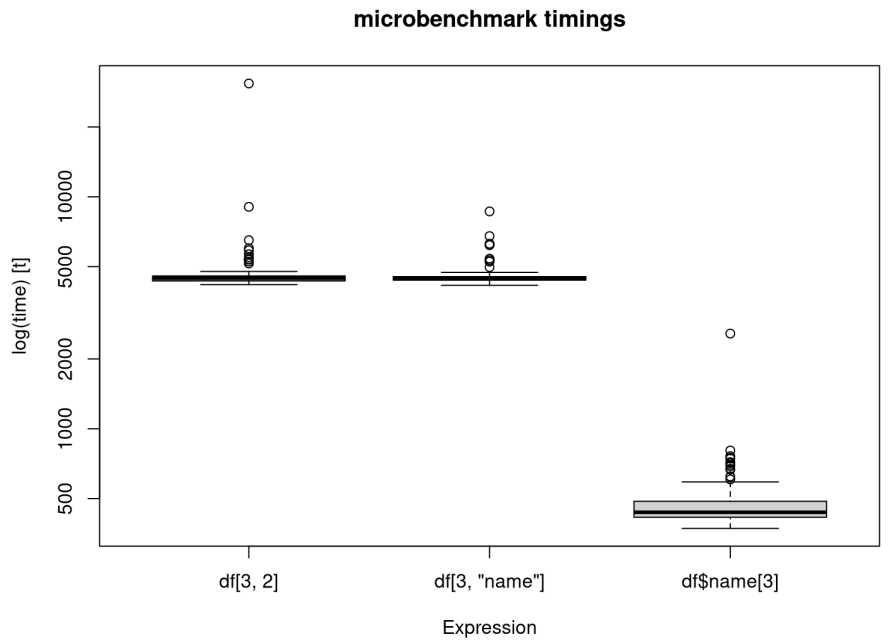
```
x <- runif(1e6)
(mb2 <- microbenchmark(sqrt(x), x^0.5))
```

```
# Unit: milliseconds
#     expr      min       lq     mean   median        uq      max neval cld
#  sqrt(x) 1.499522 1.524966 2.167293 2.274926  2.378688  5.93291   100  a
#    x^0.5 9.168183 9.228896 9.720700 9.998397 10.097791 10.57507   100   b
```

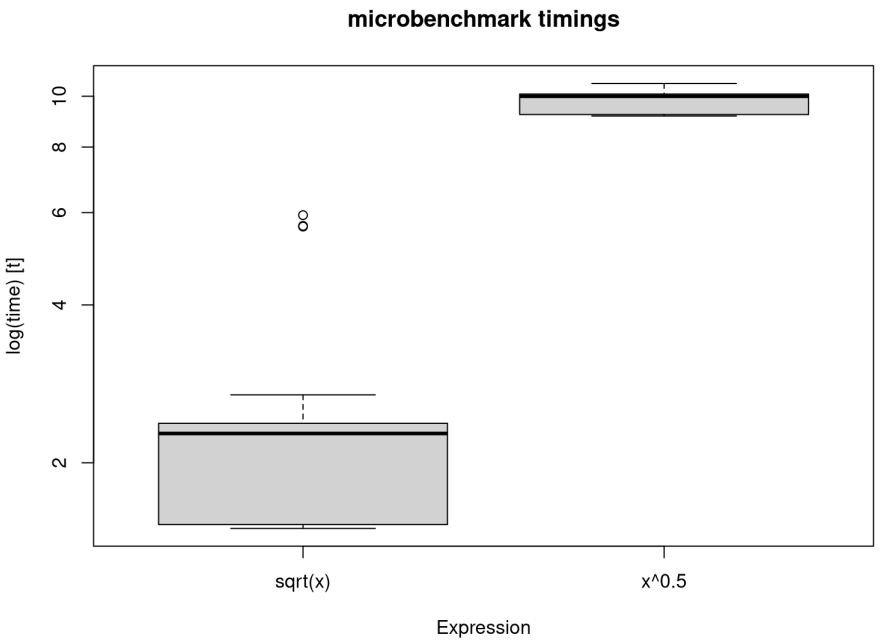
- Várias expressões podem ser passadas na mesma chamada da função
- Cada expressão é avaliada 100 vezes (por padrão), obtendo assim um sumário dos tempos de execução (veja o argumento `times`)

Benchmarking

```
boxplot(mb1)
```




```
boxplot(mb2)
```



Benchmarking

Voltando ao exemplo da aula anterior:

```
## Vetor com uma sequência de 1 a 10.000.000
x <- 1:1e7

## Cria um objeto de armazenamento com o mesmo tamanho do resultado
st1 <- system.time({
    out1 <- numeric(length(x))
    for(i in seq_along(x)){
        out1[i] <- x[i]^2
    }
})
st1
```

```
#    user  system elapsed
#    0.31    0.00    0.31
```

```
## Cria um objeto de tamanho "zero" e vai "crescendo" esse vetor
st2 <- system.time({
    out2 <- numeric(0)
    for(i in seq_along(x)){
        out2[i] <- x[i]^2
    }
})
st2
```

```
#    user  system elapsed
#   1.281   0.082   1.368
```

Benchmarking

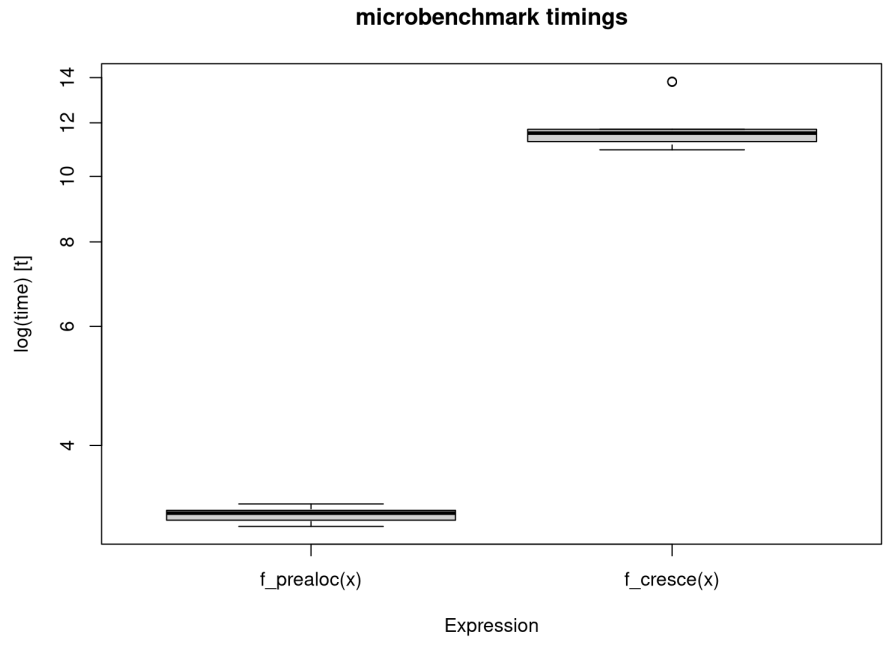
Agora com `microbenchmark()` (reduzindo `n` e `times` para não demorar demais):

```
## Vetor menor e poucas repetições
x <- 1:1e5

## Primeira forma: pré-aloca
f_prealoc <- function(x) {
    out <- numeric(length(x))
    for(i in seq_along(x)){
        out[i] <- x[i]^2
    }
    out
}
## Segunda forma: cresce
f_cresce <- function(x) {
    out <- numeric(0)
    for(i in seq_along(x)){
        out[i] <- x[i]^2
    }
    out
}
## Importante: passar x como argumento!
(mb <- microbenchmark(f_prealoc(x), f_cresce(x),
                      times = 10))
```

```
# Unit: milliseconds
#          expr       min       lq      mean    median        uq       max neval cld
#  f_prealoc(x)  3.036052  3.10086  3.162941  3.176508  3.206914  3.277357    10  a
#   f_cresce(x) 10.946878 11.25780 11.872486 11.588213 11.738630 13.814327    10   b
```

```
boxplot(mb)
```



Benchmarking com `bench::mark()`

O pacote `bench` oferece uma alternativa moderna que também reporta **memória alocada** por expressão:

```
library(bench)
x <- 1:1e5

bm <- bench::mark(
    loop_prealoc = f_prealoc(x),
    loop_cresce  = f_cresce(x),
    vetorizado   = x^2,
    iterations = 20
)
bm
```

```
# # A tibble: 3 × 6
#   expression        min   median `itr/sec` mem_alloc `gc/sec`
#   <bch:expr>   <bch:tm> <bch:tm>     <dbl> <bch:byt>    <dbl>
# 1 loop_prealoc   2.96ms   3.22ms     311.    781.3KB      0
# 2 loop_cresce   11.45ms  11.96ms      83.6   16.55MB     20.9
# 3 vetorizado    88.33µs  89.29µs   10584.     1.15MB      0
```

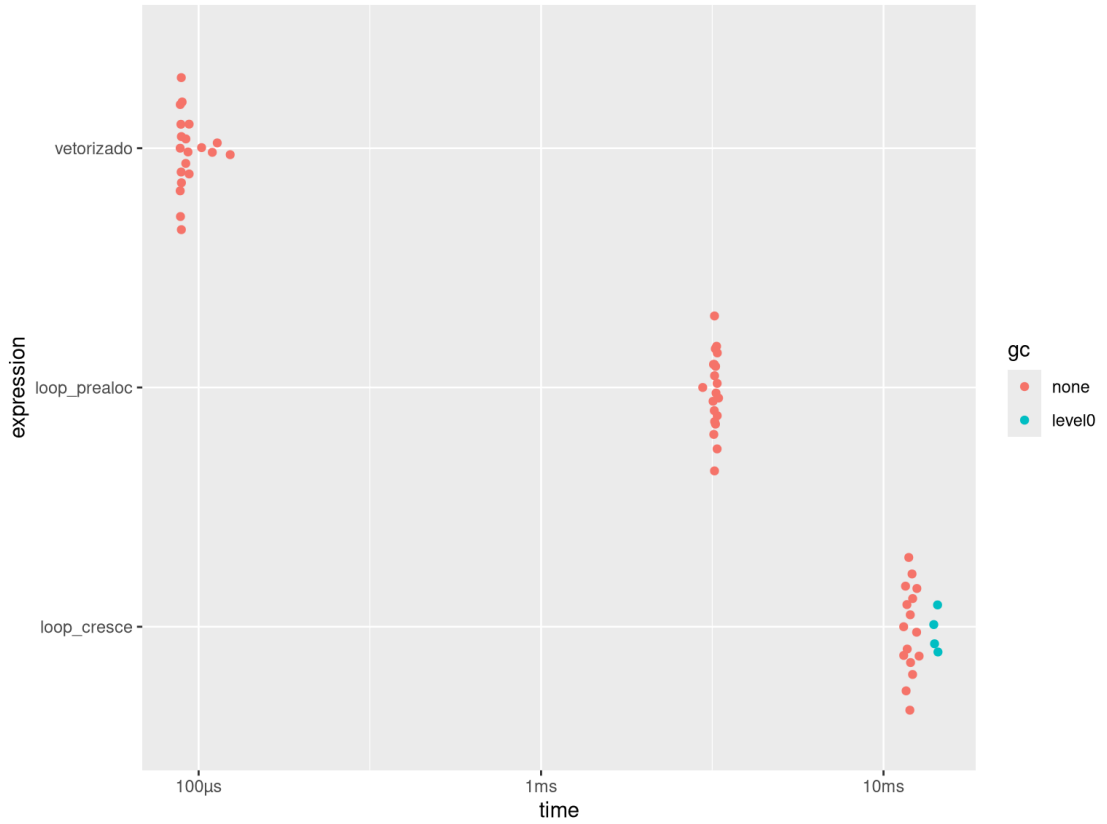
A saída inclui:

- `min`, `median`, `itr/sec`: tempo por iteração
- `mem_alloc`: memória alocada por execução
- `n_gc`: número de coletas de lixo (*garbage collections*)
- Por padrão, **verifica** que todas as expressões retornam o mesmo resultado (desative com `check = FALSE`)

Benchmarking com `bench::mark()`

Visualização:

```
plot(bm)
```



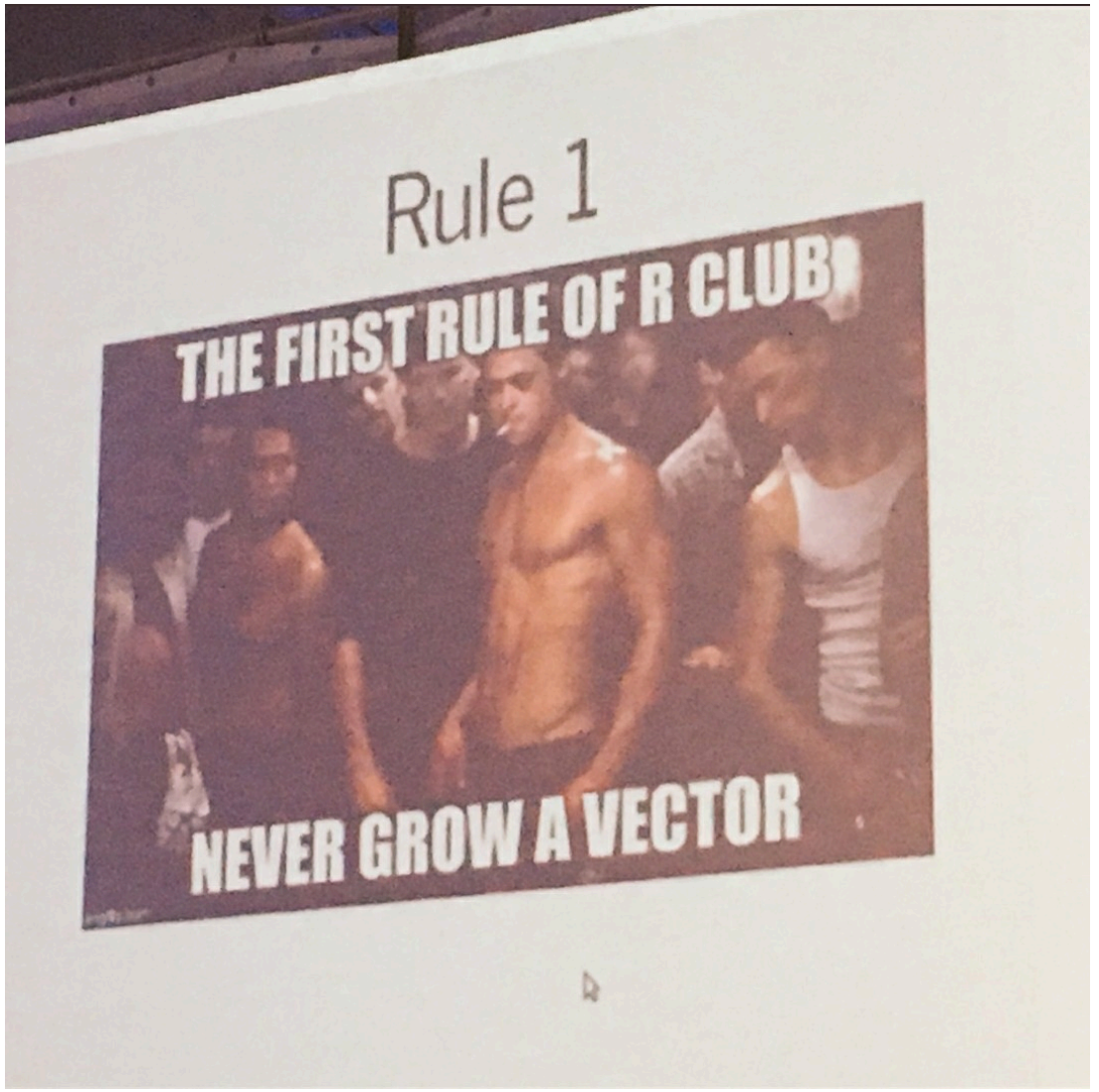
Vantagens sobre `microbenchmark()`:

- Unidade adequada (ns, µs, ms, s) escolhida automaticamente
- Relata **memória alocada**, não apenas tempo — fundamental em simulações grandes
- Checagem de consistência dos resultados das expressões

Em resumo:

- `system.time()` para uma medida pontual
- `microbenchmark()` para comparação detalhada de tempo
- `bench::mark()` para tempo **e** memória

Clube do R



Profiling

Ferramentas:

- `base::Rprof()`: resultados básicos
  - A função `Rprof()` cria um arquivo chamado `Rprof.out`, que armazena cada passo interno da execução das expressões
  - Um resumo pode ser visto com a função `summaryRprof()`
- `profvis::profvis()`: mais detalhes em uma interface

Exemplo com `Rprof()`:

```
## IMPORTANTE: vetor grande o
## bastante para o profiler ver
x <- 1:1e8

## Caso 1: pré-aloca
Rprof() ## Abre a conexão
out1 <- numeric(length(x))
for(i in seq_along(x)){
    out1[i] <- x[i]^2
}
Rprof(NULL) ## Fecha a conexão
summaryRprof()
```

```
## Caso 2: cresce o vetor
x <- 1:1e8

Rprof()
out2 <- numeric(0)
for(i in seq_along(x)){
    out2[i] <- x[i]^2
}
Rprof(NULL)
summaryRprof()
```

Profiling — atenção ao tempo de amostragem

`Rprof()` funciona por **amostragem** da pilha de chamadas a intervalos fixos (20 ms por padrão). Se o código executa em menos tempo que isso, **nenhuma amostra é coletada** e `summaryRprof()` retorna tabelas vazias:

```
$by.self
[1] self.time  self.pct   total.time total.pct
<0 rows> (or 0-length row.names)
$sampling.time
[1] 0
```

O `sampling.time = 0` é o diagnóstico: o profiler não teve tempo de amostrar nada.

Soluções:

Aumentar o trabalho:

```
x <- 1:1e7      # em vez de 1e5
```

Reduzir o intervalo:

```
Rprof(interval = 0.002)
## mínimo prático ~1 ms em Linux
```

Como regra geral: **`Rprof()` só é útil para código que demora pelo menos alguns segundos**. Para comparar implementações rápidas, use `microbenchmark()` ou `bench::mark()`; para explorações visuais em código pequeno, `profvis()`.

Profiling

Exemplo com `profvis()`

- Após a execução da expressão, uma interface gráfica é aberta com informações de tempo de execução e uso de memória *para cada linha*

```
library(profvis)
## Caso 1
profvis({
    out1 <- numeric(length(x))
    for(i in seq_along(x)){
        out1[i] <- x[i]^2
    }
})
```

```
## Caso 2
profvis({
    out2 <- numeric(0)
    for(i in seq_along(x)){
        out2[i] <- x[i]^2
    }
})
```

Exemplo prático

Simula de um modelo linear:

$$Y_i = \beta_0 + \beta_1 X_i + \epsilon_i, \quad i = 1, \ldots, n$$

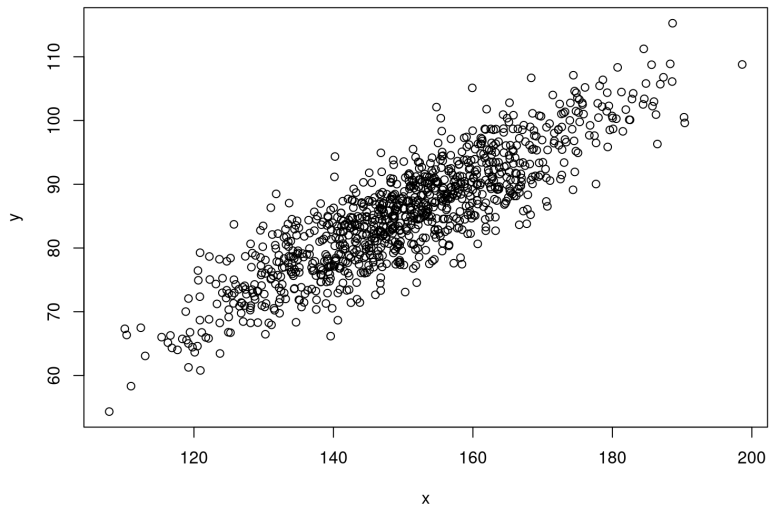
Onde:

$$\begin{aligned} X_i &\sim N(150, 15^2) \\ \epsilon_i &\sim N(0, \sigma^2) \\ \beta_0 &= 10 \\ \beta_1 &= 0.5 \\ \sigma^2 &= 20 \\ n &= 1000 \end{aligned}$$

Use `set.seed(123)`

Simulação dos dados:

```
set.seed(123)
n <- 1000; b0 <- 10; b1 <- 0.5
x <- rnorm(n, mean = 150, sd = 15)
sigma2 <- 20
y <- b0 + b1*x + rnorm(n, mean = 0, sd = sqrt(sigma2))
plot(x, y)
```



Profiling

Um modelo linear ajustado aos dados resulta em:

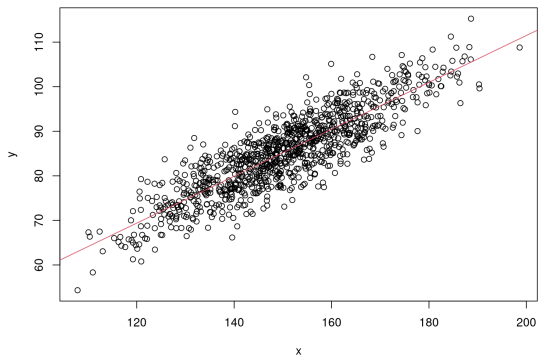
```
## Modelo
m <- lm(y ~ x)
## b0 e b1
coef(m)
```

```
# (Intercept)           x
#   6.2459652   0.5262506
```

```
## sigma^2
summary(m)$sigma^2
```

```
# [1] 20.25663
```

```
plot(x, y); abline(m, col = 2)
```



Uma forma alternativa de se obter estimativas pontuais e variâncias de parâmetros é através da técnica de **bootstrap**:

1. A partir do conjunto de dados coletados, gere uma amostra aleatória (com reposição) de tamanho $m < n$.

2. Ajuste o modelo para esse subconjunto dos dados e obtenha as estimativas pontuais dos parâmetros.

3. Repita esse procedimento r vezes (onde r deve ser algo entre 1000 e 100000, por exemplo).

4. Verifique a distribuição das r estimativas: a média será a estimativa pontual, e a variância será uma estimativa da variância.

Profiling

Implementando o bootstrap e já aplicando a função `profvis()`:

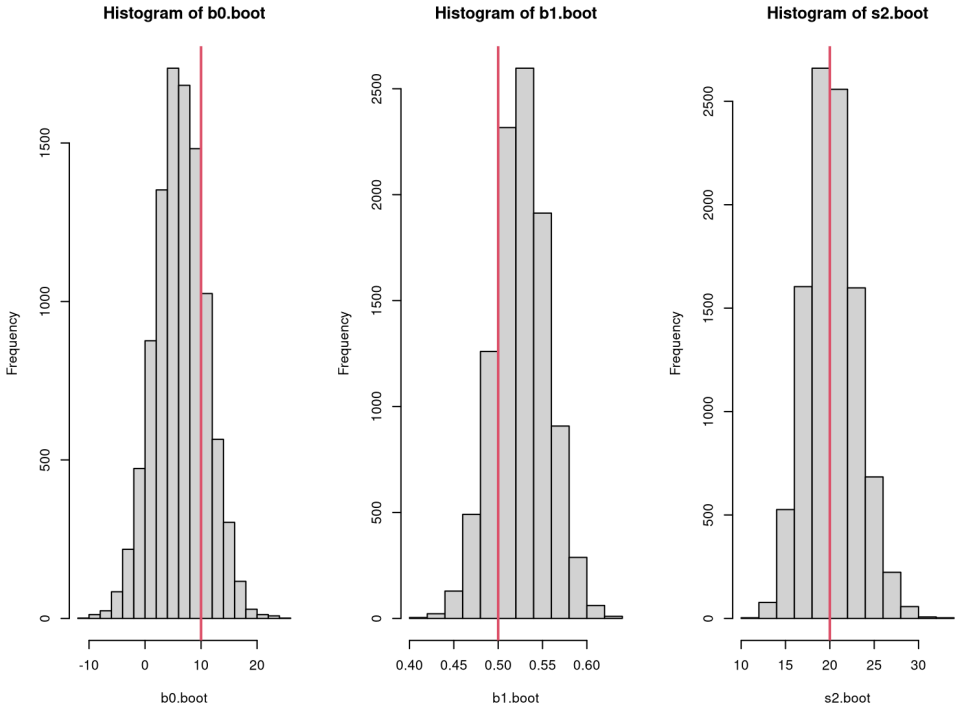
```
## Número de amostras
r <- 1e4
## Número de elementos em cada amostra
m <- 100

## Inicia o profiling
library(profvis)
profvis({
## Vetores para armazenar os resultados
b0.boot <- numeric(r)
b1.boot <- numeric(r)
s2.boot <- numeric(r)
set.seed(123)
for(i in 1:r){
    select <- sample(1:length(y), size = m, replace = TRUE)
    x.boot <- x[select]
    y.boot <- y[select]
    mm <- lm(y.boot ~ x.boot)
    b0.boot[i] <- coef(mm)[1]
    b1.boot[i] <- coef(mm)[2]
    s2.boot[i] <- summary(mm)$sigma^2
}
})
```

Profiling

Resultados do bootstrap

```
par(mfrow = c(1, 3))
hist(b0.boot); abline(v = b0, col = 2, lwd = 2)
hist(b1.boot); abline(v = b1, col = 2, lwd = 2)
hist(s2.boot); abline(v = sigma2, col = 2, lwd = 2)
par(mfrow = c(1, 1))
```




```
lapply(list(b0.boot, b1.boot, s2.boot), summary)
```

```
# [[1]]
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
# -10.651   3.306   6.271   6.310   9.357  24.228
#
# [[2]]
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#  0.4125  0.5057  0.5261  0.5258  0.5458  0.6353
#
# [[3]]
#    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
#   11.71   18.24   20.10   20.23   22.07   33.17
```

Tratamento de erros

Tipos de erros

- **Error**: único capaz de parar a execução da função, e não cria objetos
- **Warning**: avisa que algo pode estar errado, mas executa a função e retorna valores
- **Message**: apenas alguma mensagem que possa ser útil, mas não interfere nos resultados

```
stop("Isso é um erro")
```

```
# Error:
# ! Isso é um erro
```

```
warning("Isso é um warning")
```

```
# Warning: Isso é um warning
```

```
message("Isso é uma mensagem")
```

```
# Isso é uma mensagem
```

Obs.: evite usar `cat()` e `print()` para informar erros ou mensagens em funções.

Validação de argumentos

Antes de **tratar** erros, o melhor caminho é **prevenir** entradas inválidas. A função base `stopifnot()` é ideal para isso.

Uso simples:

```
f <- function(x) {
    stopifnot(is.numeric(x), length(x) == 1)
    x^2
}
f("a")
```

```
# Error in `f()`:
# ! is.numeric(x) is not TRUE
```

```
f(1:3)
```

```
# Error in `f()`:
# ! length(x) == 1 is not TRUE
```

- A mensagem padrão mostra a expressão que falhou
- É mais conciso do que vários `if(...) stop(...)` em sequência

Validação de argumentos

A partir do R 4.0, `stopifnot()` aceita **mensagens nomeadas**: se a condição correspondente falhar, a mensagem é exibida no lugar da expressão.

```
baskara <- function(a, b, c) {
    stopifnot(
        "`a`, `b` e `c` devem ser numericos" =
            is.numeric(a) && is.numeric(b) && is.numeric(c),
        "`a` nao pode ser zero" = a != 0,
        "argumentos devem ter tamanho 1" =
            length(a) == 1 && length(b) == 1 && length(c) == 1
    )
    delta <- b^2 - 4 * a * c
    if(delta < 0) stop("Delta e negativo.")
    denom <- 2 * a
    sqrt_delta <- sqrt(delta)
    c((-b - sqrt_delta)/denom, (-b + sqrt_delta)/denom)
}
baskara(0, 2, 1)
```

```
# Error in `baskara()`:
# ! `a` nao pode ser zero
```

```
baskara("x", 2, 1)
```

```
# Error in `baskara()`:
# ! `a`, `b` e `c` devem ser numericos
```

Encontrando erros

Considere a fórmula de Baskara para encontrar as raízes de uma equação de segundo grau:

$$x = \frac{-\mathrm{b} \pm \sqrt{\mathrm{b}^2 - 4\mathrm{ac}}}{2\mathrm{a}}$$

Uma implementação *naive* ("ingênua") no R seria:

```
baskara <- function(a, b, c) {
    denom <- 2 * a
    delta <- b^2 - 4 * a * c
    sqrt_delta <- sqrt(delta)
    x1 <- (-b - sqrt_delta)/denom
    x2 <- (-b + sqrt_delta)/denom
    return(c(x1, x2))
}
```

- Por quê "ingênua"?
  1. É possível fazer tudo isso em uma linha
  2. Não existem mecanismos para lidar com possíveis erros

Encontrando erros

Alguns resultados:

```
baskara(-3, 2, 1)
```

```
# [1]  1.0000000 -0.3333333
```

```
baskara(0, 2, 1) # 2 * a = 0
```

```
# [1] -Inf  NaN
```

```
baskara(3, 2, 1) # 2^2 - 4 * 3 * 1 = -8
```

```
# Warning in sqrt(delta): NaNs produced
```

```
# [1] NaN NaN
```

Usando `traceback()` para tentar identificar onde está o erro:

```
traceback()
# No traceback available
```

Não é informativo, pois não existe um erro definido formalmente na função.

Encontrando erros

Quais os possíveis problemas (numéricos)?

1. Quando $a = 0$
2. Quando $\sqrt{\Delta} < 0$

Resolvendo o primeiro problema:

```
baskara <- function(a, b, c) {
    if(a == 0) stop("Argumento `a` não pode ser zero.")
    denom <- 2 * a
    delta <- b^2 - 4 * a * c
    sqrt_delta <- sqrt(delta)
    x1 <- (-b - sqrt_delta)/denom
    x2 <- (-b + sqrt_delta)/denom
    return(c(x1, x2))
}
```

```
baskara(0, 2, 1)
```

```
# Error in `baskara()`:
# ! Argumento `a` não pode ser zero.
```

```
traceback()
# 2: stop("Argumento `a` não pode ser zero.") at #2
# 1: baskara(0, 2, 1)
```

Encontrando erros

Resolvendo o segundo problema:

```
baskara <- function(a, b, c) {
    if(a == 0) stop("Argumento `a` não pode ser zero.")
    denom <- 2 * a
    delta <- b^2 - 4 * a * c
    if(delta < 0) stop("Delta é negativo.")
    sqrt_delta <- sqrt(delta)
    x1 <- (-b - sqrt_delta)/denom
    x2 <- (-b + sqrt_delta)/denom
    return(c(x1, x2))
}
```

```
baskara(3, 2, 1)
```

```
# Error in `baskara()`:
# ! Delta é negativo.
```

```
traceback()
# 2: stop("Delta é negativo.") at #5
# 1: baskara(3, 2, 1)
```

- O resultado é lido de baixo para cima.
- O `traceback()` para na função que causou o erro (nesse caso óbvio), e mostra qual a linha correspondente dentro da função.

Encontrando erros

Usando `browser()`

- Permite "entrar" no ambiente (temporário) da função em um ponto específico, depois de chamar a função.

```
baskara <- function(a, b, c) {
    denom <- 2 * a
    delta <- b^2 - 4 * a * c
    browser()
    sqrt_delta <- sqrt(delta)
    x1 <- (-b - sqrt_delta)/denom
    x2 <- (-b + sqrt_delta)/denom
    return(c(x1, x2))
}
```

```
baskara(3, 2, 1)
# Called from: baskara(3, 2, 1)
# Browse[1]> debug at #5: sqrt_delta <- sqrt(delta)
# Browse[2]>
```

Encontrando erros

Note que:

- O prompt do console muda de `>` para `Browse[2]>`
- Você está agora dentro do ambiente da função. Isso significa que:
  - Os objetos que você tem acesso são os argumentos e outras expressões executadas até esse ponto na função. Veja `ls()`.
  - Os objetos criados no seu ambiente global não ficam visíveis.
  - Você pode criar e/ou alterar objetos da função.
- Cada Enter executa uma nova linha, até a última.
- Você pode alterar os argumentos para ver o resultado.
- Para sair digite `Q`.

(Rode o script para ver como funciona).

Encontrando erros

Usando `debug()`:

- Uma forma mais geral para entrar no modelo `browser` da função.
- Não especifica nada de especial dentro da função.
- A função entra em modo `browser()` desde a primeira linha.

Voltando com a função `baskara()` *naive*:

```
baskara <- function(a, b, c) {
    denom <- 2 * a
    delta <- b^2 - 4 * a * c
    sqrt_delta <- sqrt(delta)
    x1 <- (-b - sqrt_delta)/denom
    x2 <- (-b + sqrt_delta)/denom
    return(c(x1, x2))
}
```

```
## Para iniciar o debug, use:
debug(baskara)
## Para entrar em modo de debug, chame a função com argumentos:
baskara(3, 2, 1)
## Para sair do modo debug da função, faça:
undebug(baskara)
## Ou, para entrar em modo de debug apenas uma vez e sair, use:
debugonce(baskara)
```

Debug *post-mortem* com `options(error = recover)`

Uma das ferramentas mais poderosas (e subutilizadas) do R. Ao configurar

```
options(error = recover)
```

sempre que ocorrer um **erro**, o R abre um menu permitindo **entrar em qualquer frame da pilha de chamadas** no momento do erro — com todas as variáveis locais preservadas.

```
options(error = recover)
baskara(0, 2, 1)

# Error in baskara(0, 2, 1) : Argumento `a` não pode ser zero.
#
# Enter a frame number, or 0 to exit
#
# 1: baskara(0, 2, 1)
#
# Selection:
```

Digitando `1` você cai no *browser* dentro da função `baskara()` como ela estava no instante do erro. Para voltar ao normal:

```
options(error = NULL)
```

Debug *post-mortem* — quando vale a pena

Muito útil em **simulações longas** (como o bootstrap de 10^4 iterações): se o erro ocorre só em algumas poucas iterações, você **não** precisa re-executar tudo do zero para investigar.

Opções relacionadas:

```
options(error = recover)    # interativo, post-mortem
options(error = traceback)  # imprime a pilha e volta
options(error = NULL)       # comportamento padrão

options(warn = 0)           # padrão: warnings acumulados
options(warn = 1)           # imprime warnings imediatamente
options(warn = 2)           # converte warnings em errors
```

- `options(warn = 2)` é essencial quando um `warning` importante está sendo ignorado: ele vira `error` e você pode localizá-lo com `traceback()` ou `recover`
- Esse é um par `(warn = 2, error = recover)` que vale lembrar

Ignorando erros

Existem duas funções para ignorar erros:

- `try()`: ignora o erro, mas cria um objeto com a classe `try-error`
- `tryCatch()`: define alguma ação quando ocorrer um erro

Voltando ao exemplo da função `baskara()` com as funções `stop()`:

```
baskara <- function(a, b, c) {
    if(a == 0) stop("Argumento `a` não pode ser zero.")
    denom <- 2 * a
    delta <- b^2 - 4 * a * c
    if(delta < 0) stop("Delta é negativo.")
    sqrt_delta <- sqrt(delta)
    x1 <- (-b - sqrt_delta)/denom
    x2 <- (-b + sqrt_delta)/denom
    return(c(x1, x2))
}
```

Ignorando erros

Usando `try()`:

Sem o `try()`, o objeto não é criado:

```
baskara(0, 2, 1)
```

```
# Error in `baskara()`:
# ! Argumento `a` não pode ser zero.
```

```
er1 <- baskara(0, 2, 1)
```

```
# Error in `baskara()`:
# ! Argumento `a` não pode ser zero.
```

```
er1
```

```
# Error:
# ! object 'er1' not found
```

Com o `try()`, o objeto é criado e possui a classe `try-error`:

```
try(baskara(0, 2, 1))
```

```
# Error in baskara(0, 2, 1) : Argumento `a` não pode ser zero.
```

```
er2 <- try(baskara(0, 2, 1))
```

```
# Error in baskara(0, 2, 1) : Argumento `a` não pode ser zero.
```

```
er2
```

```
# [1] "Error in baskara(0, 2, 1) : Argumento `a` não pode ser zero.\n"
# attr(,"class")
# [1] "try-error"
# attr(,"condition")
# <simpleError in baskara(0, 2, 1): Argumento `a` não pode ser zero.>
```

```
class(er2)
```

```
# [1] "try-error"
```

Ignorando erros

Usando `try()` e `tryCatch()`:

Não existe uma forma pronta de lidar com a classe `try-error`, mas você pode definir o que deve acontecer:

```
fn <- function(...) {
    res <- try(baskara(...))
    ## Idiomático: inherits(), não class() ==
    if(inherits(res, "try-error")){
        warning("Deu erro seu jaguara")
    }
}
er3 <- fn(0, 2, 1)
```

```
# Error in baskara(...) : Argumento `a` não pode ser zero.
```

```
# Warning in fn(0, 2, 1): Deu erro seu jaguara
```

```
er3
```

```
# [1] "Deu erro seu jaguara"
```

- `inherits(res, "try-error")` é preferível a `class(res) == "try-error"` porque objetos podem ter mais de uma classe

Com `tryCatch()` podemos definir previamente o resultado quando ocorrer um erro:

```
tryCatch(baskara(0, 2, 1),
         error = function(cmd) c(NA, NA))
```

```
# [1] NA NA
```

```
tryCatch(baskara(3, 2, 1),
         error = function(cmd) c(NA, NA))
```

```
# [1] NA NA
```

Ignorando erros

Isso é útil quando se deseja que a função não pare durante uma execução

```
baskara2 <- function(a, b, c) {
    if(a == 0) stop("Argumento `a` não pode ser zero.")
    denom <- 2 * a
    delta <- b^2 - 4 * a * c
    ## if(delta < 0) stop("Delta é negativo.")
    sqrt_delta <- tryCatch(sqrt(delta),
                           warning = function(cmd) NA)
    x1 <- (-b - sqrt_delta)/denom
    x2 <- (-b + sqrt_delta)/denom
    return(c(x1, x2))
}
```

```
baskara(0, 2, 1)
```

```
# Error in `baskara()`:
# ! Argumento `a` não pode ser zero.
```

```
baskara2(0, 2, 1)
```

```
# Error in `baskara2()`:
# ! Argumento `a` não pode ser zero.
```

```
baskara(3, 2, 1)
```

```
# Error in `baskara()`:
# ! Delta é negativo.
```

```
baskara2(3, 2, 1)
```

```
# [1] NA NA
```

Ignorando erros — aplicação em simulação

Em simulações longas (por ex. bootstrap), é comum querer que iterações problemáticas retornem `NA` em vez de quebrar tudo:

```
b0.boot <- numeric(r)
b1.boot <- numeric(r)
s2.boot <- numeric(r)
set.seed(123)
for(i in 1:r){
    select <- sample(seq_along(y), size = m, replace = TRUE)
    res <- tryCatch(
        lm(y[select] ~ x[select]),
        error = function(e) NULL
    )
    if(is.null(res)){
        b0.boot[i] <- NA
        b1.boot[i] <- NA
        s2.boot[i] <- NA
    } else {
        b0.boot[i] <- coef(res)[1]
        b1.boot[i] <- coef(res)[2]
        s2.boot[i] <- summary(res)$sigma^2
    }
}
```

- O loop **não para** se uma iteração específica falhar
- As iterações problemáticas aparecem como `NA` nos vetores de saída

Desafio

Desafio

- Analise o resultado da função `profvis()` aplicada ao bootstrap.
- Verifique os pontos críticos da execução e proponha mudanças para otimizar o código.
- Use também as funções `microbenchmark()` e/ou `bench::mark()` para avaliar suas alternativas propostas.
- Dicas:
  - `lm()` faz bem mais do que estimar coeficientes. Existem alternativas mais diretas em `lm.fit()` e no cálculo matricial explícito $\hat{\beta} = (X^\top X)^{-1} X^\top y$
  - A construção da matriz de desenho X pode ser feita **uma vez** fora do loop
  - `summary(mm)$sigma^2` recalcula várias coisas — obtenha $\hat{\sigma}^2$ diretamente dos resíduos