

# MAD-CB



# Programação em R - Conceitos de Controle de Programa

*Tirar algum do misterio sobre como entrar código para construir e executar os modelos estatísticos*

# Já Aprenderemos Alguns Aspectos Importantes de Programação

- Como carregar dados dos arquivos externos em R
  - ▶ com as funções do pacote readr como `read_csv()`
  - ▶ com `load` do R base
- Como manusear dados com as funções de dplyr

- Conceitos e técnicas de fluxo de controle nas programas
- Como fazer testes lógicos



# Conceito Básico de Um Programa

- INPUT – Entrar alguns dados de interesse na memória do computador
- OUTPUT – Mostrar na tela, num arquivo ou impresso o resultado das computações
- ALGORITMO – O código que torna input em output; a análise

# Programas Andam da Primeira Linha até o Final

- Começam na primeira linha
- Segue sequencialmente
- Até a última linha
- Em Ordem



- Métodos que o programador pode usar para modificar este dinâmico básico linear
  - ▶ Repete comandos num loop
  - ▶ Mande controle para outra comando condicionalmente

- Este faz parte da função `lm` que controle a análise de regressão linear

```
if (!is.null(w) && !is.numeric(w))  
  stop("'weights' must be a numeric vector")
```

# for Loops (*aka* Laços)

- Loops permitem que um program fazer tarefas repetitivamente
- Existem vários tipos de loops
- Pouco precisa usar outra forma que o `for` loop
- Vamos aprender este tipo

# Elementos de um Loop

- A palavra `for`
- uma variável iteradora que assumirá valores sucessivos dos elementos de um objeto
- código a ser executado dentro das chaves (`{}`)

# O Que Um Loop Faz

- Iterações com a variável iteradora
  - ▶ Assumindo todos os valores indicados
- Executar o bloco de código entre os chaves

# Uso dos Loops para Calcular Valores

- Um dos usos principais para um loop é o cálculo dos valores de um
  - ▶ `data.frame`
  - ▶ `tibble`
  - ▶ vetor
  - ▶ outra estrutura de dados

# Primeiro Exemplo

- Um exemplo trivial mas fácil a entender
- Queremos preencher os valores de um variável  $x$ 
  - ▶ Colocar os valores 1 até 6 em  $x$

```
x <- numeric(length = 6) # inicializar o vetor e reservar a memória necessária
for (i in seq(1, 6)) {
  x[i] <- i
}
x
```

```
## [1] 1 2 3 4 5 6
```

- $i$  é a variável iteradora
  - ▶ Vai receber em sequência os valores 1 até 6
  - ▶ Comando `seq` (*sequência*)
- Corpo do loop colocará valor de  $i$  em cada elemento da variável  $x$  pelo assignment.



## x Será Preenchida Assim

- Depois de inicialização de x
  - ▶  $x = [0 \ 0 \ 0 \ 0 \ 0 \ 0]$
- 1ª vez no loop:  $i = 1$  e  $x[i]$  terá o valor 1 pelo assignment
  - ▶  $x = [1 \ 0 \ 0 \ 0 \ 0 \ 0]$
- 2ª vez no loop:  $i = 2$  e  $x[i]$  terá o valor 2 pelo assignment
  - ▶  $x = [1 \ 2 \ 0 \ 0 \ 0 \ 0]$
- 3ª vez no loop:  $i = 3$  e  $x[i]$  terá o valor 3 pelo assignment
  - ▶  $x = [1 \ 2 \ 3 \ 0 \ 0 \ 0]$
- O padrão continua até R completa a sexta rodada do loop

## Exemplo 2 – Calcular Raiz Quadrado de Números Aleatórios

- Criar vetor dos valores aleatórios
- Calcular o raiz quadrado desses números num loop
- Começar com um número aleatório dos valores
  - ▶ Ou seja, não sabemos quantos números o program vai calcular
  - ▶ Ou quantas vezes o loop vai rodar
- Para calcular o número de rodadas necessários
  - ▶ Função `seq_along`
  - ▶ Função mede o tamanho de um vetor
- Números aleatórios baseados na distribuição “Uniforme”
  - ▶ Todos os números entre os limites têm chance igual de ser selecionados

- Não precisa a parte “1:” da especificação do vetor porque faz si mesmo a sequência inteira dos elementos

```
seq_along(1:10)
```

```
## [1] 1 2 3 4 5 6 7 8 9 10
```

- Retorna os 2 valores limites (1 e 10)

```
set.seed = 1946
trials <- floor(runif(n = 1, min = 1, max = 50))
numeros <- runif(trials, min = 1, max = 100)
quads <- numeric(length = length(numeros))
for (i in seq_along(numeros)) {
  quads[i] <- sqrt(numeros[i])
}
quads[1:10] # Mostre primeiro 10 raizes quadrados
```

```
## [1] 6.898557 2.262942 5.313424 7.837934 7.899328 6.345453 5.323219
## [8] 9.443110 3.058559 9.084133
```

## Exemplo 3 – Criar Nova Variável num Tibble

- Amostra das cargas virais de HIV num tibble
- Queremos criar nova variável com o logaritmo de carga viral
- Vem do arquivo `counts_demo.csv`
- Precisa carregar os dados antes de criar a variável

# Calculo de log10cv

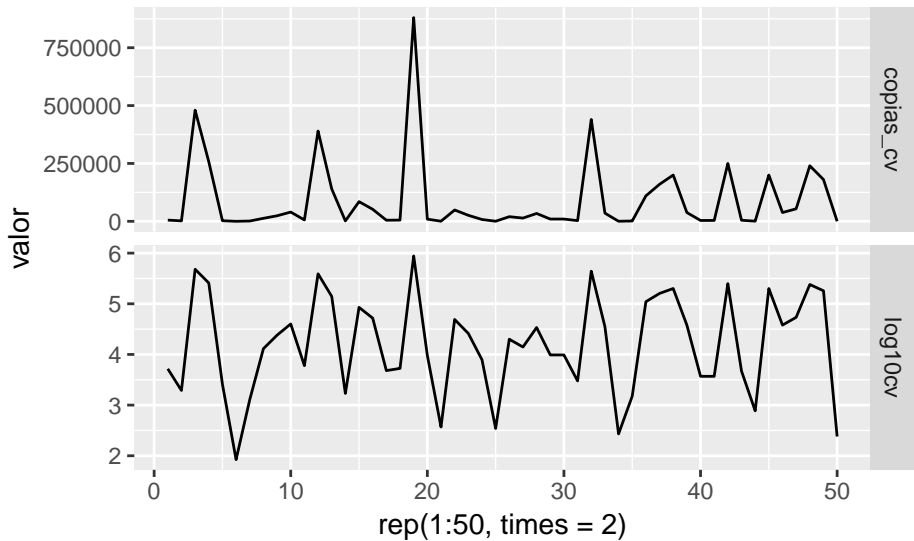
```
cvdados <- read.csv("counts_demo.csv")
cvdados$log10cv <- rep(0, nrow(cvdados)) # inicializar nova variável
for (i in 1:nrow(cvdados)) {
  cvdados$log10cv[i] <- log10(cvdados$copias_cv[i])
}
```

```
kable(cvdados[1:10, c(1,4)])
```

copias_cv	log10cv
5200	3.716003
1947	3.289366
480000	5.681241
257313	5.410462
2585	3.412460
84	1.924279
1286	3.109241
13000	4.113943
24000	4.380211
40000	4.602060

```
grcv <- cvdados %>%  
  select(copias_cv, log10cv) %>%  
  gather(tipo, valor) %>%  
  ggplot(aes(x = rep(1:50, times = 2),  
             y = valor)) + geom_line() +  
    facet_grid(tipo ~ ., scales = "free")
```





# Loops Aninhados

- Pode colocar loops um dentro do outro
- Uso típico quando tem estruturas de dados hierárquicas
  - ▶ Matrizes
  - ▶ Listas dentro das listas
- **Recomendação:** Não fazer mais de 2 ou 3 loops aninhados
  - ▶ Seria difícil entender o código e rastrear o uso das variáveis iteradoras
- Pode dividir a operação em operações separadas
  - ▶ Ou criar funções para fazer os loops interiores

# Exemplo de 2 Loops Aninhados

```
X <- data.frame(a = c("a", "b", "c"), b = c("d", "e", "f"),  
               stringsAsFactors = FALSE)
```

```
X
```

```
##   a b  
## 1 a d  
## 2 b e  
## 3 c f
```

```
# loop  
for (i in seq_len(ncol(X))) {  
  for (j in seq_len(nrow(X))) {  
    print(X[j, i])  
  }  
}
```

```
## [1] "a"  
## [1] "b"  
## [1] "c"  
## [1] "d"  
## [1] "e"  
## [1] "f"
```

- R é uma linguagem *vetorizada*
  - ▶ Muitas operações podem ser feitas em paralelo
- Podemos especificar os valores de um vetor diretamente
  - ▶ Em vez de fazer um loop

# Comparar Primeiro Exemplo com a Versão Vetorizada

- O método vetorizado

```
x <- 1:6  
x
```

```
## [1] 1 2 3 4 5 6
```

- Uma linha de código invés de 3

# Operações Aritméticas nos Vetores sem Loops

- Pode também fazer operações aritméticas diretamente sem um loop

```
x <- 1:4  
y <- 4:1 # sequências podem ir para baixo também  
z <- x + y  
x; y; z
```

```
## [1] 1 2 3 4
```

```
## [1] 4 3 2 1
```

```
## [1] 5 5 5 5
```

# Tipos de Operações Vetorizadas

- Aritméticas
  - ▶ Adição, subtração, multiplicação e divisão
- Comparações lógicas
- Operações com matrizes
  - ▶ Operações baseados em elementos do matriz
  - ▶ Verdadeiro operações matriciais
  - ▶ Aqueles que usam `%*%` e outros símbolos para as matrizes

*Functions are used to **encapsulate** a sequences of expressions that are executed together to achieve a specific goal. (Peng, Kross & Anderson, Mastering Software Development in R, 2017, p. 99.)*

*Funções são usadas para **encapsular** uma sequência das expressões que são executados juntos para alcançar um objetivo específico.*



# R – Uma Linguagem de Funções

- Todos os comandos em R são funções
- As vezes, temos uma tarefa específica e repetitiva que precisamos executar
  - ▶ E os autores dos pacotes não já prepararam
- Colocamos eles num função
  - ▶ Quem usa a programa pode chamar ela a vontade
- Evita que precisamos re-escrever o código cada vez que queremos usar ela

# Funções Têm Argumentos

- Geralmente, funções têm argumentos que o usuário pode modificar
- Ex: função de `invlogit` na aula de regressão logística
  - ▶ Manipula um valor numérico que o usuário especifica (`x`)
  - ▶ Função retorna o `invlogit` deste número

```
invlogit <- function(x) {  
  1/(1 + exp(-x))  
}  
invlogit(10)
```

# Família apply de Funções

- Grupo de funções que ajuda com a aplicação dos comandos aos vetores
  - ▶ Podem ser aplicadas a outros tipos de dados
- Todos tem o nome `xapply()`
  - ▶ Pode trocar `x` para um "l", "s", "t" ou "v"
- Apresentarei aqui a versão *simples* – `sapply()`

# sapply()

- Aplica uma função a cada elemento de um vetor
- Retorna um vetor ou uma matriz
- replicate faz parte da mesma família
- Sintaxe
  - ▶ Especifica um vetor (ou lista)
  - ▶ Especifica uma função a ser aplicada

## 2 Exemplos

```
## Aplicada a um vetor  
sapply(1:10, sqrt)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751  
## [8] 2.828427 3.000000 3.162278
```

```
## Aplicada a um data frame  
sapply(cvdados, summary)
```

```
##      copias_cv  contagem_cd4  contagem_cd8  log10cv  
## Min.           84           66.0           393.0    1.924  
## 1st Qu.        3700          250.8           652.0    3.568  
## Median        17000          472.5           827.5    4.224  
## Mean          90690          513.9           994.3    4.191  
## 3rd Qu.       103800          676.0          1230.0    5.013  
## Max.          880000         1603.0          2000.0    5.944
```

# Funções Vetorizadas dentro de dplyr e sapply()

- `mutate`, `transmute` e `summarize` conseguem o mesmo resultado que `sapply`
  - ▶ para data frames ou tibbles
- Pode usar `sapply` dentro de um fluxo de *tidyverse*
- Para fins exploratórios, frequentemente funciona mais rápido
  - ▶ e fica mais fácil para escrever
- No exemplo, as funções de `dplyr` seleciona os primeiros 10 casos da variável `log10cv`
- Função `sapply` calcula o valor original para esses elementos da variável

```
cvdados %>%  
  select(log10cv) %>%  
  slice(1:10) %>%  
  sapply(function(x){10^x})
```

```
##      log10cv  
## [1,]    5200  
## [2,]    1947  
## [3,]  480000  
## [4,] 257313  
## [5,]    2585  
## [6,]     84  
## [7,]    1286  
## [8,]   13000  
## [9,]   24000  
## [10,]  40000
```

# Funções map no Pacote purrr

- purrr faz parte de *tidyverse*
- Tem família de funções chamada map
- Segue modelo de aplicar uma função a uma estrutura dos dados
- map bem integrada com os outros integrantes de *tidyverse*



# Sintaxe da Família `map`

- `map(.x, .f, ...)`
- Para cada elemento de `.x`, faça `.f`
- `.x` pode ser
  - ▶ Data frame (tibble)
  - ▶ Lista
  - ▶ Vetor
- `.x` e `.f` são marcadores para objetos e funções que vai chamar

# Tipos de map

- `map()` – retorna uma lista ou a mesma classe de dados que ela recebe
- `'map_lgl()` – vetores lógicos
- `'map_chr()` – vetores de caracteres
- `'map_dbl()` – vetores de números (“dbl” quer dizer dupla precisão)
- `'map_int()` – vetores de números inteiros

*Ao início, recomendo que vocês evitem a `map` que produz listas porque vocês não têm muito prática com listas. Invés pode usar calmamente os outros quatro tipos.*

## 2 Exemplos

```
## Mesmo sintaxe que o exemplo de sapply  
map_dbl(1:10, sqrt)
```

```
## [1] 1.000000 1.414214 1.732051 2.000000 2.236068 2.449490 2.645751  
## [8] 2.828427 3.000000 3.162278
```

```
## Aqui a função é uma função criado para este comando  
map_lgl(c(1, 10, 4, 25, 100, 2000, 3, 5000), function(x) {  
  x < 50  
})
```

```
## [1] TRUE TRUE TRUE TRUE FALSE FALSE TRUE FALSE
```

# Testes Lógicos – if ... then ... else

- Existem em qualquer linguagem de computação
- Permite teste de uma condição
- Se for verdade, fazer uma coisa
- Se for falso, fazer outra coisa

- Versão mais simples
- Fazer um teste da condição if
- Se for verdade, executar o código que você colocou no bloco
- Senão, não fazer nada e o programa vai diretamente para a próxima linha

```
if (condição) {fazer algo}  
## continuar com o programa
```

# Exemplo de if

```
## Caso verdadeiro
x <- 4
y <- "FALSO"
if(x == 4){
  y <- "VERDADEIRO"
}
paste("Caso:", y)
```

```
## [1] "Caso: VERDADEIRO"
```

```
## Caso falso
x <- 4
y <- "FALSO"
if(x == 5){ ## Resultado falso
  y <- "VERDADEIRO"
}
paste("Caso:", y)
```

```
## [1] "Caso: FALSO"
```

*y reteve o valor original que demos a ela porque o programa não fez nada para alterar o valor*



# if ... else

- Próximo nível de complexidade
- Introduzir uma ação para condição negativa
  - ▶ Uso do parâmetro else
- Sintaxe

```
if(condição) {  
    ### fazer algo  
} else {  
    ### fazer outra coisa  
}
```

# Mesmo Exemplo neste Formato

```
## Caso verdadeiro
x <- 4
if(x == 4) {
  y = "VERDADEIRO"
} else {
  y <- "FALSO"
}
paste("Caso:", y)
```

```
## [1] "Caso: VERDADEIRO"
```

```
## Caso falso
x <- 4
if(x == 5){ ## Resultado falso
  y = "VERDADEIRO"
} else {
  y <- "FALSO"
}
paste("Caso:", y)
```

```
## [1] "Caso: FALSO"
```

# Função ifelse()

- Função que permite que você faz todo o teste lógico dentro de um comando
- Sintaxe: `ifelse(teste, sim, não)`
- 3 Elementos do Comando
  - ▶ teste – teste lógico
  - ▶ sim – resultado para resposta VERDADEIRO
  - ▶ não – resultado para resposta FALSA
- ifelse muito compacto em código
- Execução rápido

# Mesmo Exemplo – De Novo

```
x = 4

## Caso Verdadeiro
y <- ifelse(x == 4, y <- "VERDADEIRO", "FALSO")
paste("Caso:", y)
```

```
## [1] "Caso: VERDADEIRO"
```

```
## Caso Falso
y = ifelse(x == 5, "VERDADEIRO", "FALSO")
paste("Caso:", y)
```

```
## [1] "Caso: FALSO"
```