

# Ecologia de Paisagens com R

Darren Norris: [darren.norris@unifap.br](mailto:darren.norris@unifap.br)

18 dezembro, 2023

## Sumário

<b>I Apresentação</b>	<b>5</b>
<b>Bem-vindos</b>	<b>5</b>
Agradecimentos . . . . .	5
<b>Prefácio à primeira edição</b>	<b>6</b>
<b>Introdução</b>	<b>7</b>
O que você vai aprender . . . . .	7
Como este livro está organizado . . . . .	7
O que você não vai aprender . . . . .	7
Prerequisites . . . . .	7
R . . . . .	7
RStudio . . . . .	8
O universo arrumado - tidyverse . . . . .	8
Outros pacotes . . . . .	8
Código no livro . . . . .	8
Organização do código no livro . . . . .	10
<b>II Escala e padrões</b>	<b>11</b>
<b>1 Escala</b>	<b>12</b>
1.1 Apresentação . . . . .	12
1.2 Escala: breve definição . . . . .	12
1.3 Pacotes e dados . . . . .	13
1.3.1 Pacotes . . . . .	13
1.3.2 Dados . . . . .	14
1.4 Alterando a resolução . . . . .	17
1.5 Escala espacial e desenho amostral . . . . .	21
1.5.1 Obter e carregar dados (vectores) . . . . .	21
1.5.2 Visualizar os arquivos (camadas vector) . . . . .	22
1.5.3 Obter e carregar dados (raster) . . . . .	23
1.5.4 Visualizar os arquivos (camadas raster e vector) . . . . .	23
1.5.5 Reclassificação . . . . .	25
1.6 Comparação multiescala . . . . .	27
1.6.1 Pergunta 4 . . . . .	28
1.6.2 Pergunta 5 . . . . .	30
1.6.3 Pergunta 6 . . . . .	30
1.7 Próximos passos: repetindo para muitas amostras. . . . .	30

<b>2 Métricas da paisagem</b>	<b>32</b>
2.1 Apresentação . . . . .	32
2.2 Métricas da paisagem e pacote “landscapemetrics” . . . . .	33
2.2.1 Pacotes . . . . .	33
2.3 Dados . . . . .	35
2.3.1 Exibir dados raster e sobreposição com locais de amostragem . . . . .	36
2.4 Calculo de métricas . . . . .	39
2.4.1 Ponto único, raio único, métrica única . . . . .	39
2.4.2 Ponto único, distâncias variados, métrica única . . . . .	41
2.4.3 Ponto único, distâncias variados, métricas variadas . . . . .	48
<b>III Exemplos de caso</b>	<b>53</b>
<b>3 Garimpo do Lourenço</b>	<b>54</b>
3.1 Apresentação . . . . .	54
3.2 Pacotes necessarios: . . . . .	55
3.3 Área de estudo . . . . .	55
3.4 Dados . . . . .	55
3.4.1 Ponto de referência (EPSG: 4326) . . . . .	55
3.4.2 Ponto de referência (EPSG: 31976) . . . . .	56
3.4.3 Verificar com mapa de base (OpenStreetMap) . . . . .	56
3.4.4 Dados: MapBiomas cobertura da terra . . . . .	57
3.5 Calculo de métricas . . . . .	58
3.5.1 Métricas para a paisagem . . . . .	59
3.5.2 Métricas para as classes . . . . .	59
3.5.3 Métricas para as manchas . . . . .	60
3.5.4 Quais métricas devo escolher? . . . . .	61
3.5.5 Métricas por classe de mineração . . . . .	61
3.5.6 Exportar as métricas . . . . .	63
3.6 Preparando os resultados . . . . .	63
3.7 Uma tabela versatil . . . . .	66
3.8 Reorganização . . . . .	66
3.9 Uma figura elegante . . . . .	67
3.9.1 Gráfico de barra . . . . .	67
3.9.2 Gráfico de boxplot . . . . .	68
3.10 Comparação entre anos . . . . .	69
3.11 Conclusões e próximos passos . . . . .	73
<b>IV R e RStudio: Código com certeza</b>	<b>74</b>
<b>4 Pré-requisitos</b>	<b>75</b>
4.1 Introdução . . . . .	75
4.2 Instalação do R . . . . .	75
4.3 Instalação do RStudio . . . . .	76
4.4 Versão do R . . . . .	76
4.5 Pacotes . . . . .	76
4.6 Dados . . . . .	77
<b>5 Introdução ao R</b>	<b>78</b>
Pré-requisitos do capítulo . . . . .	78
5.1 Contextualização . . . . .	78
5.2 R e RStudio . . . . .	78
5.3 Funcionamento da linguagem R . . . . .	80

5.3.1	Console . . . . .	81
5.3.2	Scripts . . . . .	81
5.3.3	Operadores . . . . .	82
5.3.4	Objetos . . . . .	85
5.3.5	Funções . . . . .	86
5.3.6	Pacotes . . . . .	87
5.3.7	Ajuda ( <i>Help</i> ) . . . . .	91
5.3.8	Ambiente ( <i>Environment</i> ) . . . . .	92
5.3.9	Citações . . . . .	93
5.3.10	Principais erros de iniciantes . . . . .	94
5.4	Estrutura e manipulação de objetos . . . . .	96
5.4.1	Atributo dos objetos . . . . .	97
5.4.2	Manipulação de objetos unidimensionais . . . . .	105
5.4.3	Manipulação de objetos multidimensionais . . . . .	108
5.4.4	Valores faltantes e especiais . . . . .	112
5.4.5	Diretório de trabalho . . . . .	113
5.4.6	Importar dados . . . . .	114
5.4.7	Conferência dos dados importados . . . . .	115
5.4.8	Exportar dados . . . . .	116
5.5	Para se aprofundar . . . . .	117
5.5.1	Livros . . . . .	117
5.5.2	Links . . . . .	117
5.6	Exercícios . . . . .	117
<b>6</b>	<b>Tidyverse</b>	<b>119</b>
	Pré-requisitos do capítulo . . . . .	119
6.1	Contextualização . . . . .	119
6.2	<i>tidyverse</i> . . . . .	120
6.3	readr, readxl e writexl . . . . .	122
6.4	tibble . . . . .	122
6.5	pipe: base ( >) e magrittr (%>%)	123
6.6	tidy . . . . .	124
6.6.1	palmerpenguins . . . . .	126
6.6.2	glimpse() . . . . .	126
6.6.3	unite() . . . . .	127
6.6.4	separate() . . . . .	127
6.6.5	drop_na() e replace_na() . . . . .	128
6.6.6	pivot_longer() e pivot_wider() . . . . .	129
6.7	dplyr . . . . .	131
6.7.1	Gramática . . . . .	131
6.7.2	Sintaxe . . . . .	131
6.7.3	palmerpenguins . . . . .	132
6.7.4	relocate() . . . . .	132
6.7.5	rename() . . . . .	133
6.7.6	select() . . . . .	133
6.7.7	pull() . . . . .	134
6.7.8	mutate() . . . . .	135
6.7.9	arrange() . . . . .	135
6.7.10	filter() . . . . .	137
6.7.11	slice() . . . . .	139
6.7.12	distinct() . . . . .	140
6.7.13	count() . . . . .	141
6.7.14	group_by() . . . . .	141
6.7.15	summarise() . . . . .	142

6.7.16 bind_rows() e bind_cols() . . . . .	143
6.7.17 *_join() . . . . .	144
6.7.18 Operações de conjuntos e comparação de dados . . . . .	145
6.8 stringr . . . . .	145
6.9 forcats . . . . .	147
6.10 lubridate . . . . .	149
6.11 purrr . . . . .	156
6.12 Para se aprofundar . . . . .	160
6.12.1 Livros . . . . .	160
6.12.2 Links . . . . .	160
6.13 Exercícios . . . . .	160
<b>7 Primeiros passos com uma raster</b>	<b>161</b>
7.0.1 Obter e carregar dados (raster) . . . . .	161
7.0.2 Reclassificação . . . . .	163
<b>8 Primeiros passos com vector</b>	<b>165</b>
8.1 Obter e carregar dados (vectores) . . . . .	165
8.2 Visualizar os arquivos (camadas vector) . . . . .	168
<b>V Encerramento</b>	<b>169</b>
<b>Bibliografia</b>	<b>170</b>
<b>A Annexo 1</b>	<b>171</b>
A.1 Download mapbiomas cover . . . . .	171
A.2 Crop mapbiomas cover . . . . .	172
A.3 Plot with legend . . . . .	173
<b>B Soluções de exercícios</b>	<b>174</b>

## Part I

# Apresentação

### Bem-vindos

Este é um trabalho em andamento do 1<sup>a</sup> edição: “Ecologia de Paisagens com R”.

Versões:

- web: <https://darrenorris.github.io/epr/>
- pdf: <https://github.com/darrenorris/epr/blob/main/docs/epr.pdf>

Este é um material introdutório destinado principalmente a estudantes de graduação e cursos de pós-graduação em ecologia e áreas correlatas.

O objetivo é de apresentar os capacidades e opções para desenvolver e integrar pesquisas na Ecologia da Paisagem no ambiente estatística de R.

Esperamos que ele seja utilizado tanto por quem quer se aprofundar em análises comumente utilizadas em Ecologia da Paisagem, mesmo por quem tem poucas habilidades quantitativas.

O conteúdo e organização dos capítulos estão separados em partes.

- O primeiro parte é “Apresentação”.  
o objetivo dessa parte é incluir os aspectos mais gerais sobre os componentes e da estrutura do livro e seus objetivos.
- O segundo é “Escala e Padrões”.  
Aqui, os capítulos incluem uma breve introdução usando R para explorar alguns componentes-chaves da Ecologia da Paisagem.
- A terceira é “Exemplos de caso”.  
Aqui, os capítulos apresentam a aplicação de análises específicas e atualmente utilizadas em Ecologia da Paisagem.
- A quarta parte do livro é chamada de “R e RStudio”.  
Esta parte trata o funcionamento da linguagem R. Aqui, nós aprendemos desde como instalar o R e o RStudio até a manipulação e visualização de dados geoespaciais no R.
- A parte final é “Encerramento”.  
Aqui os capítulos apresentam a bibliografia, alguns apêndices (exemplos de código) e soluções dos exercícios dos capítulos anteriores.

### Agradecimentos

Este livro não é apenas o resultado dos autores. Mas é o resultado de muitas pessoas na comunidade R e Ecologia da Paisagem no Brasil. Muito obrigado!

## Prefácio à primeira edição

Há quem diga que a velocidade com que a tecnologia e a ciência avançam tende a tornar livros e manuais sobre métodos rapidamente obsoletos. A evolução dos computadores pessoais e a ampliação do acesso a estes e à Internet têm transformado o jeito como aprendemos e ensinamos. As pessoas estão cada vez mais familiarizados com a tecnologia e esperam que o ensino seja dinâmico e interativo.

Portanto, um livro aberto e disponível livremente, que as pessoas possam compartilhar e contribuir torna-se uma opção cada vez mais relevante. A intenção é para o livro seja utilizado como material de referência (fornecendo informações atualizadas); Como ferramenta de aprendizagem (apresentando diferentes ideias e abordagens) e como projeto de colaboração (permitindo que pessoas trabalhem juntas para criar um recurso educacional valioso).

# Introdução

Ecologia da Paisagem é uma disciplina empolgante que permite transformar dados brutos em compreensão, insight e conhecimento.

## O que você vai aprender

Ecologia da Paisagem é um campo vasto e não há como dominar tudo lendo um único livro. Este livro visa fornecer uma base sólida nas ferramentas mais importantes e conhecimento suficiente para encontrar os recursos para aprender mais quando necessário. Um modelo das etapas de um projeto típico de Ecologia da Paisagem se parece com.....

Six key steps for functional landscape analyses of habitat change <https://doi.org/10.1007/s10980-020-01048-y>  
- Acknowledge ecological theory and conceptual paradigms - Evaluate the fit of available data - Assess the three facets of the scale concept - Recognize different sampling designs - Use proper conceptual models - Measure meaningful raster characteristics

## Como este livro está organizado

A descrição anterior das ferramentas da Ecologia da Paisagem é organizada aproximadamente de acordo com a ordem em que você as usa em uma análise (embora, é claro, você as itere várias vezes). Em nossa experiência, no entanto, aprender a importar e organizar os dados primeiro não é o ideal, porque 80% do tempo é rotineiro e chato e, nos outros 20% do tempo, é estranho e frustrante. Esse é um péssimo lugar para começar a aprender um novo assunto! Em vez disso, começaremos com a visualização e transformação dos dados que já foram importados e organizados. Dessa forma, quando você ingerir e organizar seus próprios dados, sua motivação permanecerá alta porque você sabe que a dor vale o esforço.

Dentro de cada capítulo, tentamos aderir a um padrão consistente: comece com alguns exemplos motivadores para que você possa ver o quadro geral e depois mergulhe nos detalhes. Cada seção do livro é combinada com exercícios para ajudá-lo a praticar o que aprendeu. Embora possa ser tentador pular os exercícios, não há melhor maneira de aprender do que praticar em problemas reais.

## O que você não vai aprender

Base teórica pra trás os conceitos e cálculos. O foco é sobre a aplicação, e isso não preciso avanços teóricos. Incluímos referências para que é possível obter maior detalhes sobre os conceitos e teorias ecológicas e cálculos usados na Ecologia da Paisagem.

## Prerequisites

Fizemos algumas suposições sobre o que você já sabe para aproveitar ao máximo este livro. Você deve ser geralmente alfabetizado numericamente, e com conhecimento prévia de ecologia, geoprocessamento e uso de sistemas de informação geográfica.

Você precisa de quatro coisas para executar o código deste livro: R, RStudio, uma coleção de pacotes R chamada **tidyverse** e um punhado de outros pacotes. Os pacotes são as unidades fundamentais do código R reproduzível. Eles incluem funções reutilizáveis, documentação que descreve como usá-los e dados de amostra.

## R

Para fazer o download do R, acesse CRAN, a **comprehensive R archive network**, <https://cloud.r-project.org>. Uma nova versão principal do R é lançada uma vez por ano e há 2 a 3 versões secundárias a cada ano. É uma boa ideia atualizar regularmente. A atualização pode ser um pouco complicada, especialmente para as versões principais que exigem a reinstalação de todos os seus pacotes, mas adiar só piora as coisas. Recomendamos R 4.2.0 ou posterior para este livro.

## RStudio

RStudio é um ambiente de desenvolvimento integrado, ou IDE, para programação R, que você pode baixar em <https://posit.co/download/rstudio-desktop/>. O RStudio é atualizado algumas vezes por ano e avisa automaticamente quando uma nova versão é lançada, para que não haja necessidade de verificar novamente. É uma boa ideia atualizar regularmente para aproveitar os melhores e mais recentes recursos. Para este livro, certifique-se de ter pelo menos o RStudio 2022.02.0.

## O universo arrumado - tidyverse

Você também precisará instalar alguns pacotes do R. Um **pacote** do R é uma coleção de funções, dados e documentação que estende os recursos do R base. O uso de pacotes é a chave para o uso bem-sucedido do R. A maioria dos pacotes que você aprenderá neste livro faz parte do chamado tidyverse. Todos os pacotes no tidyverse compartilham uma filosofia comum de programação de dados e R e são projetados para trabalhar juntos.

Você pode instalar o tidyverse completo com uma única linha de código:

```
install.packages("tidyverse")
```

No seu computador, digite essa linha de código no console e pressione enter para executá-lo. R irá baixar os pacotes do CRAN e instalá-los em seu computador.

Você não poderá usar as funções, objetos ou arquivos de ajuda em um pacote até carregá-lo com `library()`. Depois de instalar um pacote, você pode carregá-lo usando a função `library()`:

```
library(tidyverse)
```

Isso diz a você que o tidyverse carrega nove pacotes: dplyr,forcats, ggplot2, lubridate, purrr, readr, stringr, tibble, alignr. Eles são considerados o **núcleo** do tidyverse porque você os usará em quase todas as análises.

Os pacotes no tidyverse mudam com bastante frequência. Você pode ver se há atualizações disponíveis executando `tidyverse_update()`.

## Outros pacotes

Existem muitos outros pacotes excelentes que não fazem parte do tidyverse porque resolvem problemas em um domínio diferente ou são projetados com um conjunto diferente de princípios subjacentes. Isso não os torna melhores ou piores, apenas diferentes. Em outras palavras, o complemento do tidyverse não é o universo bagunçado, mas muitos outros universos de pacotes inter-relacionados. Ao lidar com mais projetos de Ecologia da Paisagem com R, você aprenderá novos pacotes e novas formas de pensar sobre os dados.

Usaremos outras pacotes de fora do tidyverse neste livro. Por exemplo, usaremos os seguintes pacotes porque eles fornecem conjuntos de funções e dados interessantes para trabalharmos no processo de aprendizado de R:

```
install.packages(c("sp", "sf", "raster", "mapview", "tmap",
                   "terra", "kableExtra", "landscapetrics"))
```

## Código no livro

- Objetivo não é de apresentar detalhes sobre os cálculos/métodos estatísticas ou os funções no R. Existem diversos exemplos disponíveis “[Ciência de Dados com R–Introdução.....](#)”: e com google “r cran introdução tutorial”..... Alem disso, existem grupos de ajuda, como por exemplo: [R Brasil](#) e [Stack Overflow em Português](#)
- O objetivo é de apresentar um livro mostrando os capacidades e opções para desenvolver e integrar pesquisas na ecologia da paisagem no ambiente estatística de R

Obviamente, todas as tarefas de geoprocessamento podem ser desenvolvidas anteriormente em um SIG ([QGIS](#)). Então porque use R? R tem a capacidade (baseada em código) para alternar entre tarefas de processamento,

modelagem e visualização de dados geográficos e não geográficos. Além disso, como é possível importar, modificar, analisar e visualizar dados espaciais no mesmo ambiente com script/código, o R permite fluxos de trabalho transparentes e reproduzíveis ([A Ciência Aberta](#)).

Aliás, atualmente a grande maioria dos artigos científicos publicados na revista [Landscape Ecology](#) incluir análises usando R.

## Organização do código no livro

O capítulo está organizado em etapas de processamento, com blocos de código em caixas cinzas:  
código de R para executar

Para seguir os passos, os blocos de código precisam ser executados em sequência. Se você pular uma etapa, ou rodar fora de sequência o próximo bloco de código provavelmente não funcionará.

As linhas de código de R dentro de cada caixa também precisam ser executados em sequência. O símbolo `#` é usado para incluir comentários sobre os passos no código (ou seja, linhas começando com `#` não são código de executar).

```
# Passo 1  
código de R passo 1 # texto e números têm cores diferentes  
# Passo 2  
código de R passo 2  
# Passo 3  
código de R passo 3
```

Aém disso, os símbolos `#>` e/ou `[1]` no início de uma linha indicam o resultado que você verá no console de R.

```
# Passo 1  
1+1
```

```
[1] 2
```

```
# Passo 2  
x <- 1 + 1  
# Passo 3  
x
```

```
[1] 2
```

```
# Passo 4  
x + 1
```

```
[1] 3
```

## Part II

# Escala e padrões

# 1 Escala

## 1.1 Apresentação

Nesta capítulo vamos entender a importância de escala na ecologia da paisagem através cálculos com a proporção de floresta. Durante o capítulo você aprenderá a

1. Alterar escala (resolução e extensão espacial),
2. Calcular a área de uma classe de habitat,
3. Desenvolve uma comparação multiescala.

É muito importante ficar claro para você o que é escala (e o que não é!), e qual a importância desse conceito na elaboração do desenho amostral, na coleta de dados, nas análises e na tomada de decisão. Nesse tutorial usaremos conteúdo baseado no Capítulo 2 do livro [Spatial Ecology and Conservation Modeling](#) (Fletcher and Fortin 2018) e “[Tutorial Escala](#)” do Dr. Alexandre Martensen.

## 1.2 Escala: breve definição

Todos os processos e padrões ecológicos têm uma dimensão temporal e espacial. Assim sendo, o conceito de escala não somente representar essas dimensões, mas também, ajudar nos apresentá-los de uma forma que facilite o entendimento sobre os processos e padrões sendo estudados.

Na ecologia o termo escala refere-se à dimensão ou domínio espaço-temporal de um processo ou padrão. Na ecologia da paisagem, a escala é frequentemente descrita por sua componentes: resolução e extensão.

- **Resolução:** menor unidade espacial de medida para um padrão ou processo.
- **Extensão:** descreve o comprimento ou tamanho de área sob investigação.

Resolução e extensão tendem a covariar – estudos com maior extensão tendem a ter resolução maiores também. Parte dessa covariância é prática: é difícil trabalhar em grandes extensões com dados coletados em tamanhos de resolução finos. No entanto, parte dessa covariância também é conceitual: muitas vezes em grandes extensões, podemos esperar que processos operando em resolução muito fina forneçam somente “ruído” e não dados/informações relevantes sobre os sistemas. Como os desafios computacionais diminuíram e a disponibilidade de dados de alta resolução aumentou, a covariância entre resolução e extensão nas investigações diminuiu.

Lembrando, na primeira aula, vimos que a escala espacial pode ser interpretada com base em três dimensões:

- no fenômeno de interesse;
- na amostragem que ocorre;

e/ou

- na análise

Para que a Ecologia da Paisagem gere evidências científicas robustas e úteis, a escala nas três dimensões deve ser consistente e apropriada para o estudo. Aqui nos concentramos na dimensão “análise”, e aprendemos como a escala espacial pode ser alterada e representada em modelos ecológicos.

## 1.3 Pacotes e dados

Em geral é necessário baixar alguns pacotes para que possamos fazer as nossas análises. Neste caso precisamos os seguintes pacotes, que deve estar instalado antes:

- [tidyverse](#),
- [sf](#),
- [terra](#),
- [mapview](#),
- [tmap](#).

### 1.3.1 Pacotes

No R, carregar os pacotes necessários com o código:

```
library(tidyverse)
library(sf)
library(terra)
library(mapview)
library(tmap)
library(eprdados)
```

Caso os pacotes não tenham sido instalados, o R vai avisar através de uma mensagem tipo: [Error in library\(nomepacote\) there is no package called nomepacote](#). Neste caso, para instalá-los consulte o capítulo aqui [Capítulo 4 instalação de pacotes](#) e aqui [Capítulo 4 pacotes](#).

### 1.3.2 Dados

Vamos olhar um exemplo do mundo real. Uma pequena amostra do Rio Araguari, perto de Porto Grande. O ponto central é de longitude: -51.406312 latitude: 0.726236. Para visualizar o ponto no Google Earth: <https://earthengine.google.com/timelapse#v=0.72154,-51.41543,11.8,latLng&t=2.24&ps=25&bt=19840101&et=20201231&startDwell=0&endDwell=0>.

Vamos trabalhar com os dados de [MapBiomas](#), que produz mapeamento anual da cobertura e uso da terra no Brasil desde 1985. Os dados de MapBiomas vem no formato de raster, que tem uma classificação da terra feito a partir da classificação pixel a pixel de imagens das satélites Landsat. Todo processo é feito com algoritmos de aprendizagem de máquina (machine learning) através da plataforma Google Earth Engine, que oferece imensa capacidade de processamento na nuvem. Mais detalhes sobre a metodologia aqui: [Metodologia MapBiomas](#).

Para carregar um arquivo raster trabalhamos com o pacote [terra](#). O pacote tem vários funções para a análise e modelagem de dados geográficos. Nós podemos ler os dados de cobertura da terra no arquivo “.tif” com a função [rast](#).

```
# arquivo no pacote "eprdados"
arquivo <- system.file("raster/amostra_mapbiomas_2020.tif",
                      package = "eprdados")
# carregar
ramostra <- rast(arquivo)
```

Plotar para verificar.

```
plot(ramostra)
```

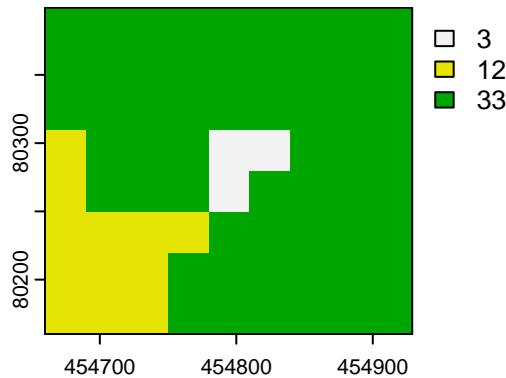


Figura 1.1: Mapbiomas 2020. Uma pequena amostra do Rio Araguari, perto de Porto Grande.

Podemos também verificar informações sobre o raster (metadados) - simplesmente rodar o nome do objeto: `ramostra`

```
## class      : SpatRaster
## dimensions : 8, 9, 1  (nrow, ncol, nlyr)
## resolution : 29.89281, 29.89281  (x, y)
## extent     : 454659.8, 454928.9, 80160.06, 80399.2  (xmin, xmax, ymin, ymax)
## coord. ref. : SIRGAS 2000 / UTM zone 22N (EPSG:31976)
## source     : amostra_mapbiomas_2020.tif
## name       : mapbiomas_2020
## min value  : 3
```

```
## max value : 33
```

Isso nos mostra informações sobre escala espacial (resolução e extensão) e o sistema de coordenadas (SIRGAS 2000 / UTM zone 22N , [EPSG:31976](#)). Além disso é possível obter informações específicas através de funções específicas.

```
# Obter informações sobre escala espacial
# resolução, comprimento e largura do pixel em metros
res(ramostra)
# numero de colunas
ncol(ramostra)
# numero de linhas
nrow(ramostra)
```

**1.3.2.1 Pergunta 1 Sobre o objeto ramostra. Com base nos resultados obtidos, qual o área do pixel em metros quadrados? Qual o área total da paisagem em hectares e quilometros quadrados?**

---

Olhando a mapa (Figura 1.1), existem três classes com valores de 3, 12 e 33. O objetivo principal não é de fazer mapas, mas, a visualização dos dados é um passo importante para verificar e entender os padrões. Portanto, segue exemplo mostrando uma forma de visualizar o arquivo de raster como mapa.

Para entender o que os valores (3, 12, 33) representam no mundo real precisamos de uma referência (legenda). Para a MapBiomas Coleção 6, arquivo: [Cod\\_Class\\_legenda\\_Col6\\_MapBiomas\\_BR.pdf](#). Existe também arquivos para fazer as mapas com cores corretas em [QGIS](#) ou [ArcGIS](#).

Olhando a legenda ([Cod\\_Class\\_legenda\\_Col6\\_MapBiomas\\_BR.pdf](#)), sabemos que “3”, “12” e “33” representem cobertura de “Formação Florestal”, “Formação Campestre”, e “Rio, Lago e Oceano”. Então podemos fazer um mapa mostrando tais informações.

Daqui pra frente vamos aproveitar uma forma mais elegante de apresentar mapas e gráficos. Isso seria através funções em 2 pacotes:

- [tmap](#) é utilizado para gerar mapas temáticos
- [ggplot2](#), que faz parte do “tidyverse”, e é utilizado para produção de gráficos, e pode representar dados geoespaciais.

Exemplos : [Pacotes ggplot2 e tmap](#)

Mais exemplos sobre o uso de [ggplot2](#) no R cookbook : <http://www.cookbook-r.com/Graphs/> .

E com mais exemplos de mapas e dados espaciais no R: [sf](#) e [ggplot2](#) : <https://www.r-spatial.org/r/2018/10/25/ggplot2-sf.html>

[Capítulo 9](#) no livro [Geocomputation with R](#) : <https://geocompr.robinlovelace.net/adv-map.html>

Primeiramente precisamos incluir as informações relevantes da legenda. Ou seja, incluir os nomes para cada valor de classe.

```
# legenda e cores na sequencia correta
classe_valor <- c(3, 12, 33)
classe_legenda <- c("Formação Florestal",
                     "Formação Campestre", "Rio, Lago e Oceano")
classe_cores <- c("#006400", "#B8AF4F", "#0000FF")
```

Agora podemos fazer o mapa com as classes e os cores seguindo o padrão recomendado pela MapBiomas para Coleção 6.

```
tm_shape(ramostra) +
  tm_raster(style = "cat",
             palette = c("3" = "#006400", "12" ="#B8AF4F",
                         "33"= "#0000FF"), legend.show = FALSE) +
  tm_grid(labels.format = list(big.mark = ""))
  tm_add_legend(type = "fill", labels = classe_legenda,
                 col = classe_cores, title = "Classe") +
  tm_compass(position = c("right", "bottom")) +
  tm_scale_bar(breaks = c(0, 0.05, 0.1), text.size = 1,
                text.color = "white", position=c("right", "bottom")) +
  tm_layout(legend.position = c("right","top"),legend.bg.color = "white")
```

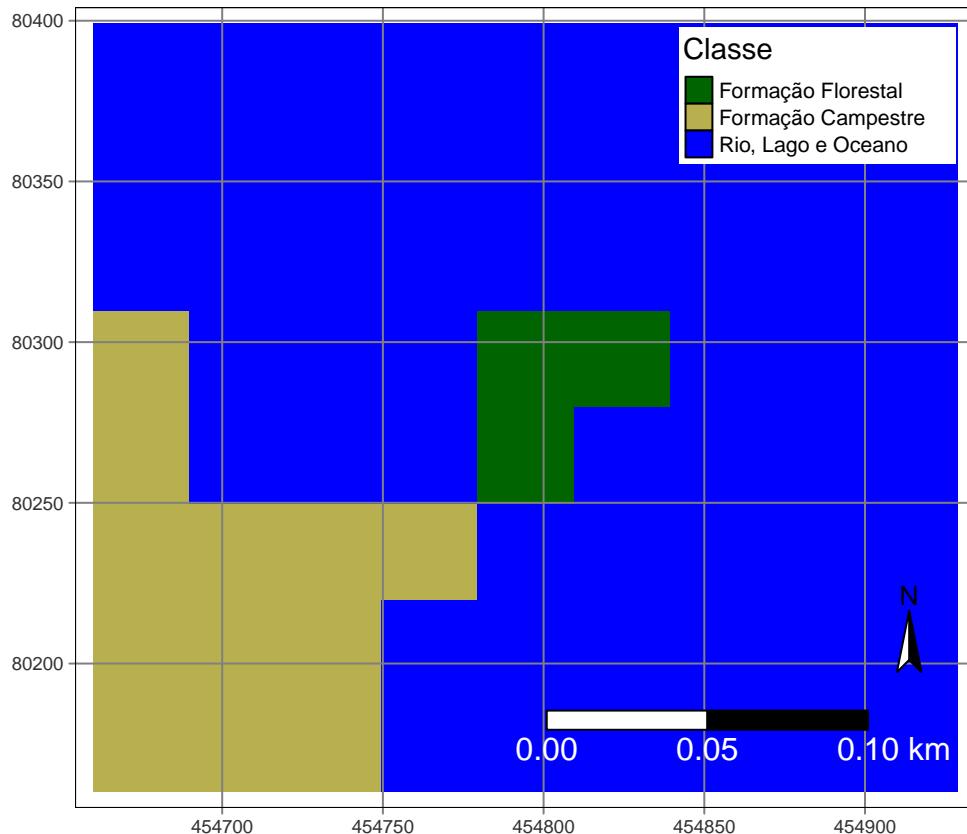


Figura 1.2: Paisagem com valores e classes de cobertura da terra. Mapbiomas 2020. Uma pequena amostra do Rio Araguari, perto de Porto Grande.

## 1.4 Alterando a resolução

Alterando a resolução serve como exemplo mostrando como os passos/etapas/cálculos mude dependendo o tipo de dados. Ou seja, é preciso adotar metodologias diferentes para dados categóricos (por exemplo classificação de cobertura da terra) e dados contínuos (por exemplo distância até rio).

Alterando a resolução às vezes seria necessário, por exemplo, quando preciso padronizar dados/imagens oriundos de fontes diferentes com resoluções diferentes e/ou para reduzir a complexidade da modelagem. Lembrando - em cada nível de resolução, são observáveis processos e padrões que não podem necessariamente ser inferidos daqueles abaixo ou acima.

Agora iremos degradar a resolução desses dados, ou seja, iremos alterar o tamanho dos pixels. Como exemplo, iremos juntar (agregar) 3 pixels em um único pixel. Como você acha que podemos fazer isso? Quais valores esse pixel que vai substituir os 3 originais deve ter? Existem diversas maneiras de se fazer isso, como por exemplo através a média ou valor modal (valor mais comum). O valor mais comum da área, é particularmente adequado quando temos um mapa categórico, como por exemplo a classificação do MapBiomas. Segue exemplo de código para agregar com a média e o valor mais frequente (modal).

```
# Média
ramostra_media <- aggregate(ramostra, fact=3, fun="mean")
ramostra_media <- resample(ramostra, ramostra_media)

# Modal
ramostra_modal <- aggregate(ramostra, fact=3, fun="modal")
```

```
ramostra_modal <- resample(ramostra, ramostra_modal, method="near")
```

1.4.0.1 Pergunta 2 Utilizando as funções disponíveis no pacote tmap, crie mapas temáticos dos objetos ramostra\_media e ramostra\_modal. Inclua cópias do seu código e mapas na sua resposta. Você pode usar o printscreen para mostrar o RStudio com seu código e mapas.

---

Visualizar os resultados apresentados em Figura 1.4. Os valores calculados pela média não fazem sentido para uma classificação categórica. Os valores calculados pela modal são consistentes com o original e fazem sentido.

Em cada nível de resolução, são observáveis processos e padrões que não podem necessariamente ser inferidos daqueles abaixo ou acima. Aqui por exemplo, mudamos a proporção de cobertura florestal em nossa pequeno paisagem quando juntamos 3 pixels em um único: a proporção de floresta moudou de 4% (3/72) para 11% (1/9). Ou seja, com cada passo mudamos a representação do mundo.

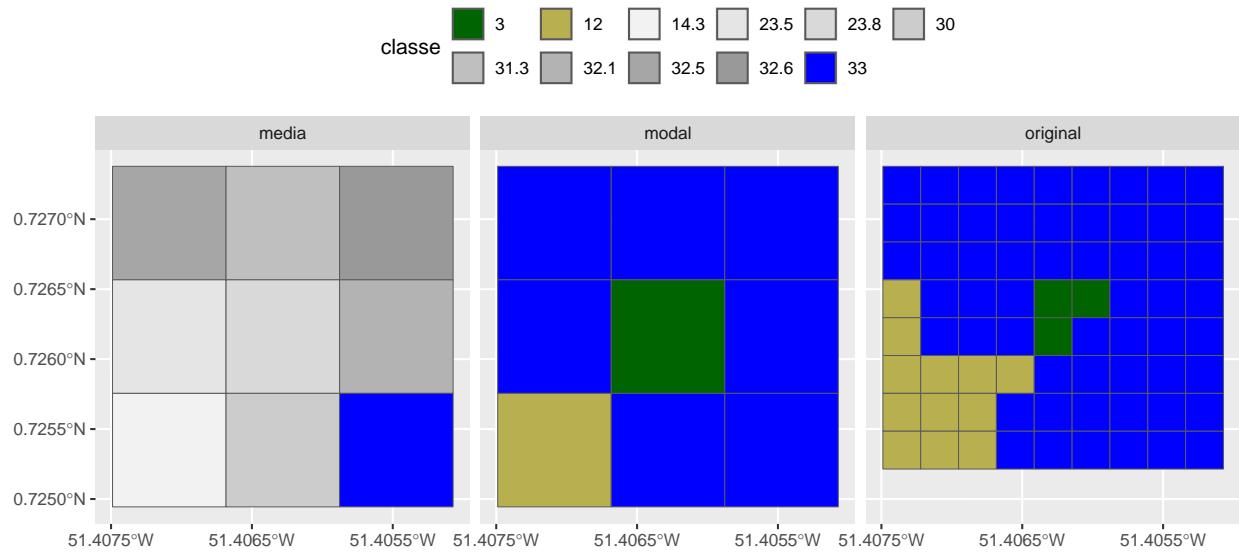


Figura 1.3: Mudanças causadas pela agregação.

**1.4.0.2 Pergunta 3** Confira o código e os resultados obtidos anteriormente, quando mudamos a resolução da raster ramostra (por exemplo figura 1.4). Explique o que aconteceu. Como e porque moudou os valores em cada caso (média e modal)?

---

## 1.5 Escala espacial e desenho amostral

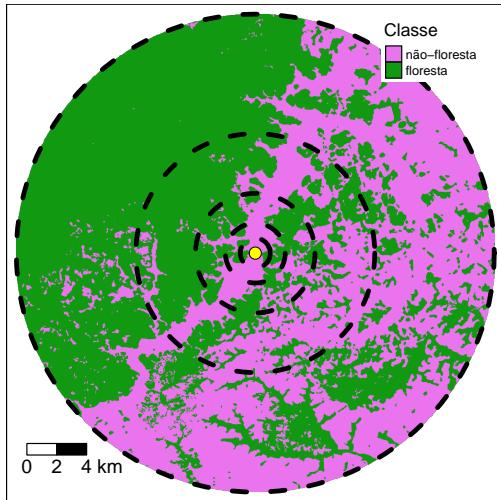


Figura 1.4: A cobertura florestal ao redor de um ponto de amostragem pode variar em escalas diferentes. Para entender essas variações, podemos criar buffers circulares de diferentes extensões ao redor dos pontos de amostragem. Esses buffers representam áreas de diferentes tamanhos ao redor de cada ponto. Quantificando a quantidade de floresta que ocorre em cada buffer, podemos obter uma visão geral da escala em que a cobertura florestal muda ao redor dos pontos de amostragem. Por exemplo, podemos descobrir que a cobertura florestal é mais alta dentro de um buffer de 5 km do que em um buffer de 10 km.

Dado o papel que a escala pode desempenhar em nossa compreensão dos padrões e processos ecológicos, como escala deve ser considerada no desenho do estudo? Claramente, a resposta a esta pergunta irá variar dependendo dos fenômenos de interesse, mas ecologistas e estatísticos têm fornecido algumas orientações importantes. As questões-chave incluem o tamanho da unidade de amostragem (resolução), o tipo de unidade de amostra e localizações da unidade de amostra, incluindo o espaçamento entre as amostras (distância entre as amostras) e o tamanho da área de estudo.

Com a disponibilidade de imagens de satélite é possível responder questões importantes relacionadas ao desenho do estudo antes de qualquer trabalho de campo. Uma técnica de geoprocessamento (bordas - **Buffers**) é um dos mais frequentemente adotados para quantificar escala espacial na ecologia da paisagem.

O objetivo é criar buffers circulares de diferentes extensões ao redor dos sitios de amostragem (pontos, pixels, manchas, transets lineares etc). Aqui, vamos entender a escala em que a cobertura de floresta muda ao redor dos rios. Para isso, quantificamos a quantidade de floresta que ocorre em várias distâncias em pontos ao longo dos rios a montante das hidrelétricas no Rio Araguari. Para ilustrar esta abordagem geral, usamos o banco de dados MapBiomas Coleção 6 de 2020, e vincule esses dados de cobertura da terra aos pontos de amostragem em rios.

### 1.5.1 Obter e carregar dados (vectores)

Antes de quantificar a quantidade de floresta, precisamos carregar os dados de rios e pontos de amostragem. O formato de vector é diferente de “tif” (raster), portanto o processo de importação é diferente. Aqui, nós só precisamos de duas dessas camadas, ambos do pacote ‘eprdados’: “rio\_linhacentral” e “rio\_pontos”. A primeira camada de dados contém o eixo central de 260 km de rios a montante da Barragem Cachoeira Caldeirão. Os dados foram obtidos a partir de registros de trajetória GPS durante levantamentos de barco. Os rios foram divididos em 52 seções, cada uma com aproximadamente 5 km de extensão. A segunda camada de dados, “rio\_pontos”, contém 52 pontos espaçados regularmente ao longo dos rios. Cada ponto está localizado a aproximadamente 5 km de distância do ponto anterior.

### 1.5.2 Visualizar os arquivos (camadas vector)

Visualizar para verificar. Mapa com ambos a linha central e pontos de rios em trechos de 5km.

```
ggplot(rio_linhacentral) +  
  geom_sf(aes(color=rio)) +  
  geom_sf(data = rio_pontos, shape=21, aes(fill=zone))
```

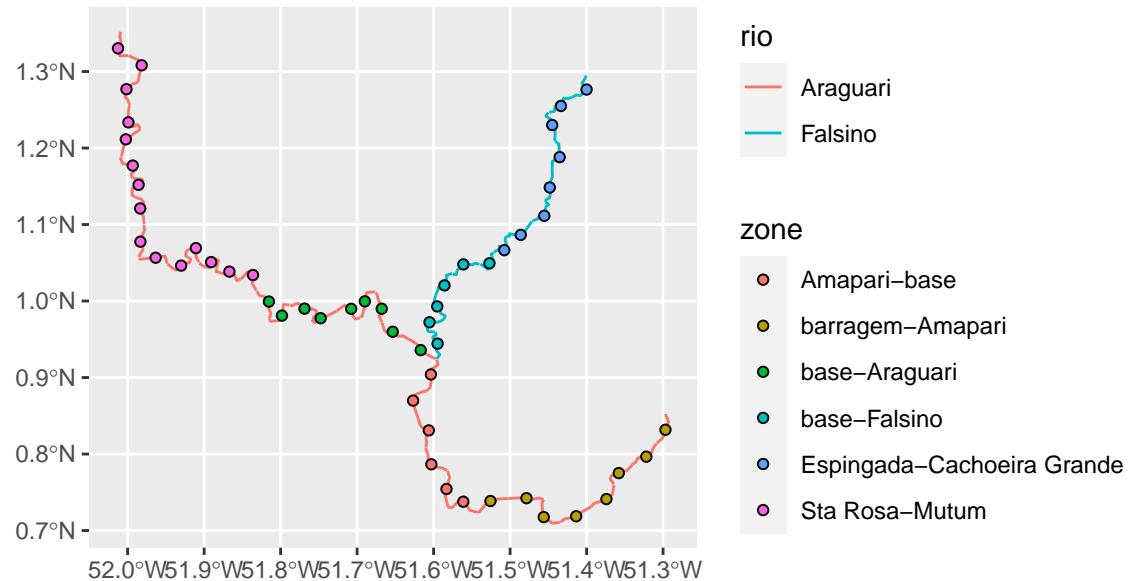


Figura 1.5: Pontos ao longo dos rios a montante das hidrelétricas no Rio Araguari.

Mapa interativo (funcione somente com internet) Mostrando agora com fundo de mapas “base” (OpenStreetMap/ESRI etc)

```
#  
mapview(rio_linhacentral, zcol = "rio")
```

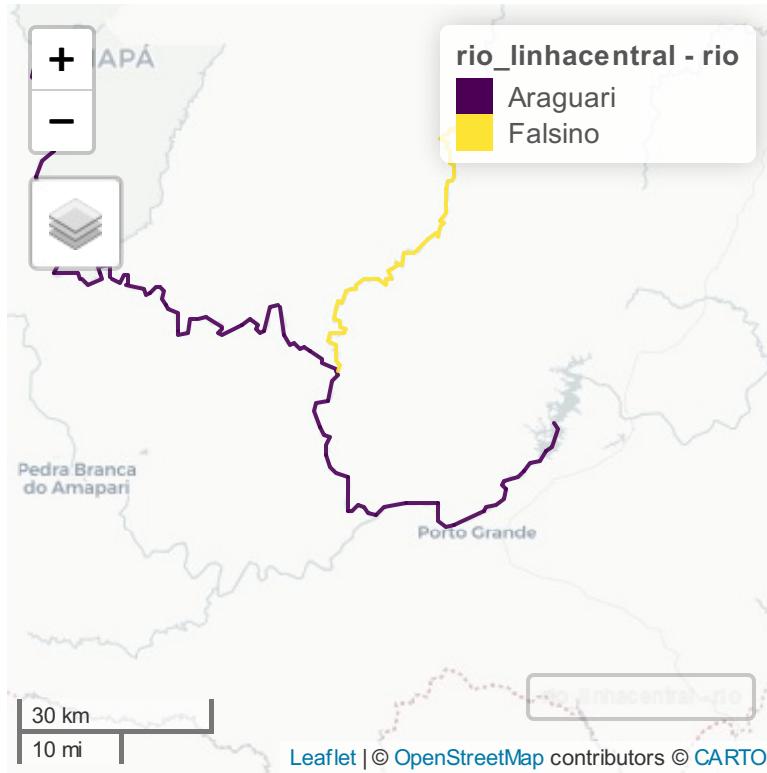


Figura 1.6: Linhas dos rios a montante das hidrelétricas no Rio Araguari.

### 1.5.3 Obter e carregar dados (raster)

Mais uma vez vamos aproveitar os dados de MapBiomass. Agora um arquivo raster com cobertura de terra no entorno dos rios em 2020, (formato “.tif”, tamanho 1.3 MB). O código abaixo vai carregar os dados e criar o objeto “mapbiomas\_2020”:

```
meuSIGr <- system.file("raster/utm_cover_AP_rio_2020.tif",
                       package = "eprdados")
# carregar
mapbiomas_2020 <- rast(meuSIGr)
```

### 1.5.4 Visualizar os arquivos (camadas raster e vector)

Visualizar para verificar. É possível de visualizar varios camadas de raster e vetor juntos com funções no pacote tmap (<https://r-tmap.github.io/tmap-book/index.html>).

```
# Passo necessário para agilizar o processamento
mapbiomas_2020_modal <- aggregate(mapbiomas_2020,
                                      fact = 10,
                                      fun = "modal")

# Plot
tm_shape(mapbiomas_2020_modal) +
  tm_raster(title = "Classe", style = "cat", palette = "Set3") +
  tm_shape(rio_linhacentral) +
  tm_lines(col="blue") +
  tm_shape(rioPontos) +
  tm_dots(size = 0.2, col = "yellow") +
  tm_compass(position=c("left", "top"))
```

```
tm_scale_bar(breaks = c(0, 25, 50), text.size = 1,  
             position=c("left", "bottom")) +  
tm_layout(legend.position = c("right", "top"), legend.bg.color="white")
```

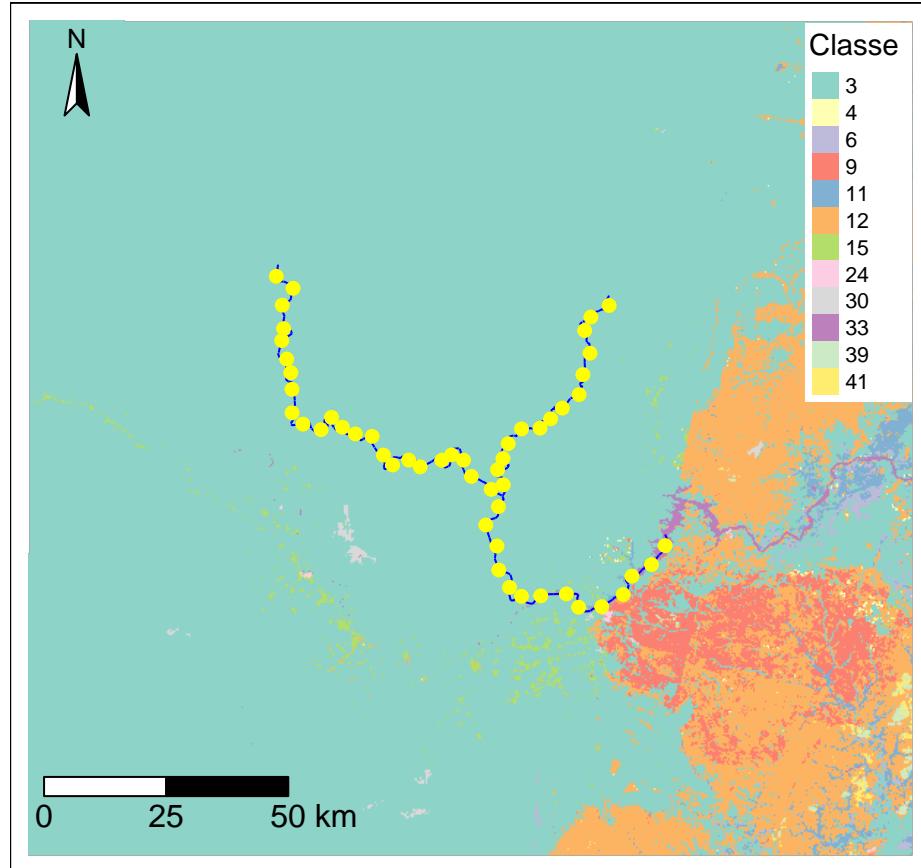


Figura 1.7: Cobertura da terra ao redor do Rio Araguari em 2020. Mostrando os pontos de amostragem (pontos amarelas) cada 5 quilômetros ao longo do rio.

### 1.5.5 Reclassificação

Para simplificar nossa avaliação de escala, reclassificamos a camada mapbiomas\_2020 em uma camada binária de floresta/não-floresta. Essa tarefa de geoprocessamento pode ser realizada anteriormente usando SIG ([QGIS](#)). Aqui vamos reclassificar as categorias de cobertura da terra (agrupando diferentes áreas de cobertura florestal tipos) usando alguns comandos genéricos do R para criar uma nova camada com a cobertura de floresta em toda a região de estudo. Para isso, criamos um mapa do mesmo resolução e extensão, e então podemos redefinir os valores do mapa. Neste caso, queremos agrupar a cobertura da terra categorias 3 e 4 (Formação Florestal e Formação Savânica, respectivamente).

```
# criar uma nova camada de floresta
floresta_2020 <- mapbiomas_2020
# Com valor de 0
values(floresta_2020) <- 0
# Atualizar categorias florestais agrupados com valor de 1
floresta_2020[mapbiomas_2020 == 3 | mapbiomas_2020 == 4] <- 1
```

Vizualizar para verificar.

```
# Passo necessário para agilizar o processamento
floresta_2020_modal <- aggregate(floresta_2020,
                                    fact=10,
                                    fun="modal")

# Plot
tm_shape(floresta_2020_modal) +
  tm_raster(style = "cat",
             palette = c("0" = "#E974ED", "1" ="#129912"), legend.show = FALSE) +
  tm_add_legend(type = "fill", labels = c("não-floresta", "floresta"),
                col = c("#E974ED", "#129912"), title = "Classe") +
  tm_shape(rio_linhacentral) +
  tm_lines(col="blue") +
  tm_shape(rioPontos) +
  tm_dots(size = 0.2, col = "yellow") +
  tm_scale_bar(breaks = c(0, 25, 50), text.size = 1,
               text.color = "white", position=c("left", "bottom")) +
  tm_layout(legend.position = c("right","top"),legend.bg.color = "white")
```

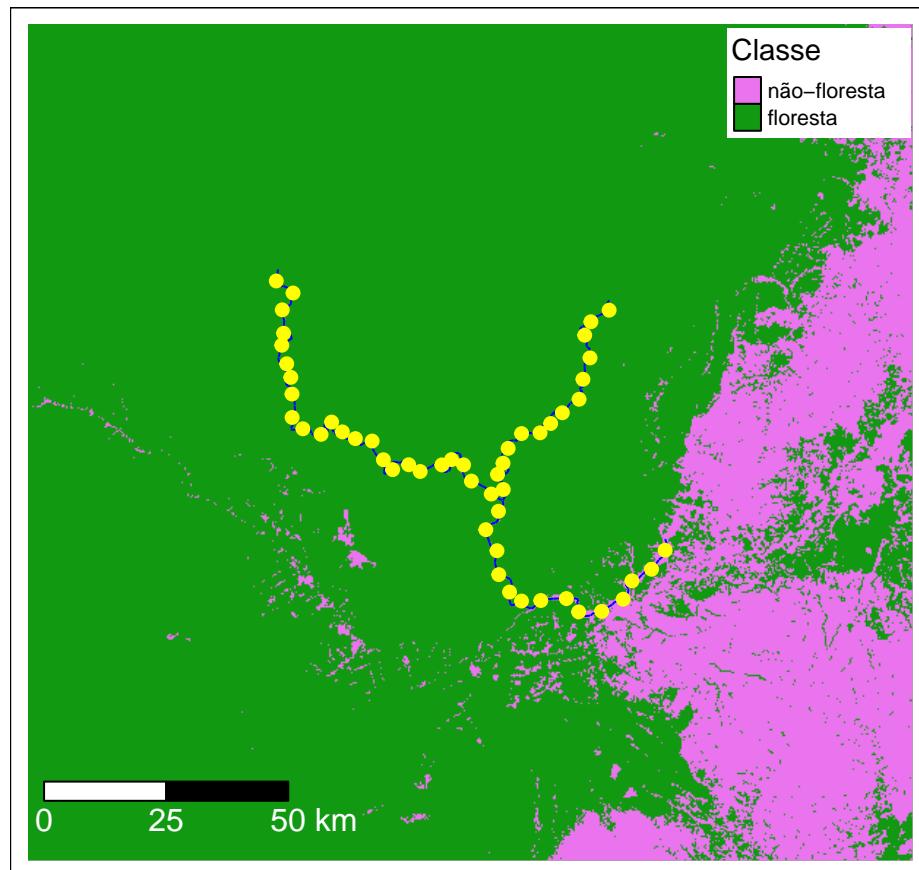


Figura 1.8: Floresta ao redor do Rio Araguari. MapBiomas 2020 reclassificado em floresta e não-floresta. Mostrando os pontos de amostragem (pontos amarelos) cada 5 quilômetros ao longo do rio.

## 1.6 Comparação multiescala

Em seguida, com as coordenadas dos pontos de amostragem, podemos calcular a quantidade de floresta que circunda cada local de amostragem em diferentes extensões. Primeiramente, vamos fazer só para um ponto, assim para entender o processo e os passos melhor.

```
rio_pontos_31976 <- st_transform(rio_pontos, 31976)
# Buffer
rio_pontos_31976_b1000 <- st_buffer(rio_pontos_31976[1, ], dist = 1000)

# Recorte com buffer de 1000 metros (mudando a extensão).
buffer.forest1.1km <- crop(floresta_2020, snap="out", rio_pontos_31976_b1000)
# Máscara para que os pixels fora do polígono sejam nulos.
buffer.forest1.1km <- mask(buffer.forest1.1km, rio_pontos_31976_b1000, touches=TRUE)
names(buffer.forest1.1km) <- "forest_2020_1km"
```

Vizualizar para verificar.

```
# Plot
tm_shape(buffer.forest1.1km) +
  tm_raster(style = "cat",
             palette = c("0" = "#E974ED",
                         "1" ="#129912"), legend.show = FALSE) +
tm_shape(rioPontos_31976[1, ]) +
  tm_symbols(shape = 21, col = "yellow",
             border.col = "black", border.lwd = 0.2, size=0.5) +
tm_shape(rioPontos_31976_b1000) +
  tm_borders(col = "black", lwd = 4, lty = "dashed") +
tm_add_legend(type = "fill", labels = c("não-floresta", "floresta"),
              col = c("#E974ED", "#129912"), title = "Classe") +
tm_compass(position=c("left", "top")) +
tm_scale_bar(breaks = c(0, 0.5, 1), text.size = 1,
             position=c("left", "bottom")) +
tm_layout(legend.position = c("right", "top"), legend.bg.color = "white")
```

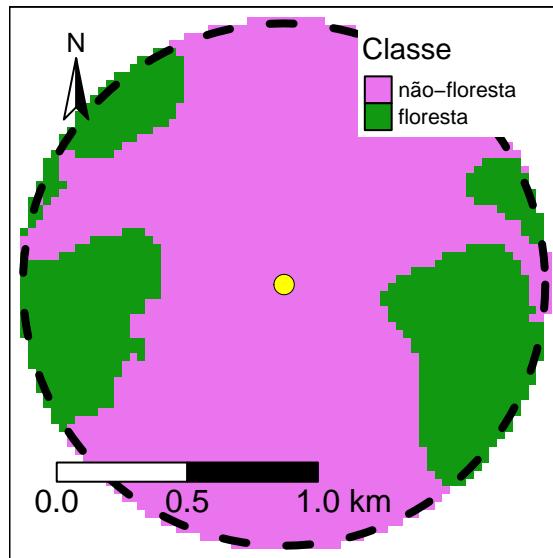


Figura 1.9: Ilustração da determinação da quantidade de habitat ao redor de um ponto. Para um determinada extensão, o habitat de interesse é isolado. Um buffer (linha tracejada) é colocado ao redor de um ponto (amarela) e o número de células (pixels) que contém o habitat é somado e multiplicado pela área de cada pixel.

### 1.6.1 Pergunta 4

Qual é a extensão em número de pixels desse recorte (buffer.forest1.1km)?

---

Temos valores de 0 (não-floresta) e 1 (floresta). Então, para saber a área de floresta podemos somar o número de células (pixels) que contém o habitat e multiplica pela área de cada pixel conforme o código:

```
# 1) Somatório.  
# No caso igual o numero de pixels de floresta.  
# Para todo a paisagem, somatorio "global".  
# Não deve incluir pixels nulos, então use "na.rm = TRUE".  
soma_floresta <- global(buffer.forest1.1km, "sum", na.rm = TRUE)  
soma_floresta  
  
## sum  
## forest_2020_1km 939  
  
# 2) Área de cada pixel.  
# Sabemos o sistema de coordenadas (EPSG = 31976).  
# EPSG 31976 é uma sistema projetado com unidade em metros.  
buffer.forest1.1km  
  
## class : SpatRaster  
## dimensions : 68, 68, 1 (nrow, ncol, nlyr)  
## resolution : 29.89281, 29.89281 (x, y)  
## extent : 465959.3, 467992, 90921.47, 92954.18 (xmin, xmax, ymin, ymax)  
## coord. ref. : SIRGAS 2000 / UTM zone 22N (EPSG:31976)  
## source(s) : memory  
## varname : utm_cover_AP_rio_2020  
## name : forest_2020_1km  
## min value : 0  
## max value : 1  
  
# Portanto, o tamanho de cada pixel é igual.  
area_pixel_m2 <- 29.89281 * 29.89281  
area_pixel_m2  
  
## [1] 893.5801  
  
# 3) Calculos de área.  
# Área de floresta m2  
area_floresta_m2 <- soma_floresta * area_pixel_m2  
area_floresta_m2  
  
## sum  
## forest_2020_1km 839071.7  
  
# Área de floresta hectares  
area_floresta_ha <- area_floresta_m2 / 10000  
area_floresta_ha  
  
## sum  
## forest_2020_1km 83.90717
```

Para uma comparação multiescala, vamos repetir o mesmo processo, mas agora com distâncias de 250, 500, 1000, 2000 e 4000 metros, dobrando a escala (extensão) em cada passo.

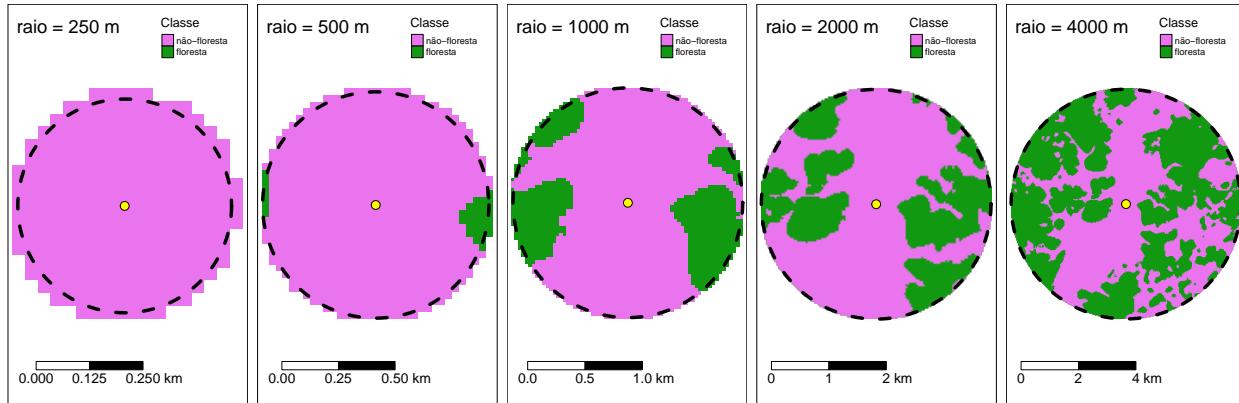


Figura 1.10: Cobertura florestal em extensões diferentes ao redor de um local de amostragem.

Aspectos quantitativos das paisagens mudam fundamentalmente com a escala. Por exemplo, nesse caso, parece que a proporção de floresta aumenta à medida que a extensão aumenta de 500 para 4000 metros. Esta percepção visual é confirmada pelos valores calculados, onde as áreas são:

- raio 250 m = 0 hectares de floresta
- raio 500 m = 6,3 hectares de floresta
- raio 1000 m = 84,3 hectares de floresta
- raio 2000 m = 502,6 hectares de floresta
- raio 4000 m = 3351,0 hectares de floresta

### 1.6.2 Pergunta 5

Usando os valores listadas acima de raio e área de floresta para os diferentes buffers circulares, calcule a proporção de floresta em cada uma das diferentes extensões de buffer. Apresente 1) os resultados incluindo cálculos. 2) um gráfico com valores de extensão no eixo x e proporção da floresta no eixo y. 3) Em menos de 200 palavras apresente a sua interpretação do gráfico.

---

### 1.6.3 Pergunta 6

A modelagem multiescala quantifica as condições do ambiente em múltiplas escalas alterando o resolução ou a extensão da análise e, em seguida, avaliando qual das escalas consideradas explica melhor um padrão ou processo. Escolha 1 espécie aquática e 1 espécie terrestre que ocorram na região a montante das hidrelétricas no Rio Araguari. Com base nas diferenças entre extensões (indicados no exemplo anterior) e as características funcionais das espécies (por exemplo área de vida), escolher as extensões mais adequadas para um estudo multiescala de cada espécie.

---

## 1.7 Próximos passos: repetindo para muitas amostras.

Neste exemplo compararmos a área de floresta em torno de um único ponto de amostragem. Para calcular o mesmo para todos os 52 pontos, seriam necessárias varias repetições (52 pontos x 5 extensões = 260 repetições).

Poderíamos escrever código para executar esse processo automaticamente. Felizmente, alguém já escreveu funções para fazer isso e muito mais. O próximo tutorial sobre métricas de paisagem mostrará exemplos usando o pacote “landscapemetrics” (<https://r-spatialecology.github.io/landscapemetrics/>).

## 2 Métricas da paisagem

### 2.1 Apresentação

As métricas da paisagem nos ajudam a entender as mudanças na paisagem de diferentes perspectivas (visual, ecológica, cultural).

Assim sendo, análises com métricas de paisagem é um atividade fundamental na ecologia da paisagem. Nesta capítulo aprenderemos sobre como analisar a cobertura da terra com métricas de paisagem em R. As tecnicas será ilustrada através cálculos usando a cobertura florestal ao redor do Rio Araguari. Ao longo do caminho, revisaremos modelos lineares e não lineares, aprenderemos sobre manipulação de dados em R e aprenderemos como criar gráficos com o pacote `ggplot2`.

No capitulo você aprenderá a:

- Importar e plotar dados raster em R e mapear locais de amostragem com os pacotes `terra`, `sf` e `tmap`.
- Calcular métricas de paisagem com o pacote `landscapemetrics`.
- Calcular métricas de paisagem em locais de amostragem e dentro de um buffer ao redor deles (comparação multiescala).
- Construir gráficos com o pacote `ggplot2`.
- Comparação de padrões lineares e não-lineares.

## 2.2 Métricas da paisagem e pacote “landscapemetrics”

As métricas de paisagem são a forma que os ecólogos de paisagem usam para descrever os padrões espaciais de paisagens para depois avaliar a influência destes padrões espaciais nos padrões e processos ecológicos.

A paisagem objeto de análise é definida pelo usuário e pode representar qualquer fenômeno espacial. As métricas simplesmente quantifica a heterogeneidade espacial da paisagem representada no mapa categórico; cabe ao usuário estabelecer uma base sólida para definir e dimensionar a paisagem em termos de conteúdo temático e resolução e extensão espacial. É importante ressaltar que o resultado das métricas só é relevante se a paisagem definida for adequado em relação ao fenômeno em consideração.

`landscapemetrics` tem funções para calcular métricas de paisagem em paisagens categóricos (onde tem uma classificação de cobertura de terra/habitat - modelo mancha-corredor-matriz), em um fluxo de trabalho organizado. O pacote pode ser usado como um substituto do FRAGSTATS ([McGarigal et al. 1995, https://doi.org/10.2737/PNW-GTR-351](#)), pois oferece um fluxo de trabalho reproduzível para análise de paisagem em um único ambiente. Pode também obter FRAGSTATS aqui: <https://fragstats.org/>. `landscapemetrics` também permite cálculos de quatro métricas teóricas de complexidade da paisagem: entropia marginal, entropia condicional, entropia conjunta e informação mútua ([Nowosad e Stepinski 2019 https://doi.org/10.1007/s10980-019-00830-x](#)).

### 2.2.1 Pacotes

Além do “`landscapemetrics`”, precisamos carregar alguns pacotes a mais para facilitar a organização e apresentação de dados espaciais (vector e raster) e os resultados.

Carregar pacotes (que deve estar instalado antes):

```
library(tidyverse)
library(sf)
library(raster)
library(terra)
library(tmap)
library(gridExtra)
library(kableExtra)
library(mgcv)
library(eprdados)
```

**2.2.1.1 `landscapemetrics`** Agora, vamos para o pacote principal `landscapemetrics`. Digite o código abaixo e veja o resultado. Leia com atenção e preste particular atenção na organização da página de ajuda.

```
library(landscapemetrics)
# Olhar a pagina de ajuda
?landscapemetrics
```

No final da página você vai encontrar a palavra “Index”. Clique nela e você verá todas as funções do pacote. Desça até as `lsm_...` e clique em algumas delas ali. Explorar! Para listar todas as métricas disponíveis, você pode usar a função `list_lsm()`. A função também permite mostrar métricas filtradas por nível, tipo ou nome da métrica. Para obter mais informações sobre as métricas, consulte os arquivos de ajuda correspondentes ou <https://r-spatialecology.github.io/landscapemetrics>.

Digite o código abaixo e veja o resultados, mostrando exemplos das métricas diferentes disponíveis no pacote.

```
# métricas de agregação, nível de fragmento
landscapemetrics::list_lsm(level = "patch", type = "aggregation metric")
# métricas de agregação, nível de classe
landscapemetrics::list_lsm(level = "class", type = "aggregation metric")
#
landscapemetrics::list_lsm(metric = "area")
```

```
# ajudar com opções da função  
?landscapemetrics::list_lsm
```

Nesse pacote o formato geral para uma função é em três partes “lsm\_nível\_métrica”. A primeira parte é sempre lsm\_ (“landscapemetric”), seguinda do “nível\_” e por fim a “métrica”:

1. Ou seja, todas as funções que calculam métricas começam com lsm\_ .....
2. Daí você deve incluir o nível da análise: “p”, “c” ou “l”.  
Sendo, “p” para patch (ou seja, para a mancha/fragmento), “c” para classe e “l” para landscape ou seja, métricas para a paisagem como um todo.
3. E daí existem inúmeras métricas.

Como por exemplo a `cpland`, que é o percentual de área central - “core area” na paisagem, como vimos na aula teórica. Assim sendo, a função `lsm_c_cpland` vai calcular a métrica porcentagem da área central em cada classe. Lembrando existem metricas que podem se calculados nos três níveis, e metricas que só pode se calculados somente para um nível específico. Ja sabendo o nome da função, podemos buscar ajudar para entender mais detalhes. Digite o código abaixo e veja o resultado, mostrando um exemplo para uma métrica.

```
# ajudar com opções para uma função específica  
?landscapemetrics::lsm_c_cpland
```

**2.2.1.2 Pergunta 1** Descreva brevemente 2 métricas de cada nível (patch, class, landscape) usando ajudar (usando ? e/ou `list_lsm`) e/ou a leitura disponível no Google Classroom (Base teórica 4 Dados, métricas, análises). Incluindo na descrição - o nome, porque serve, unidades de medida, e relevância ecológica.

---

## 2.3 Dados

Vamos continuar as analises que começo no capítulo “Escala”, trabalhando com os mesmos bancos de dados. Então, primeiramente carregar os dados de cobertura da terra com a função rast. E, em seguida implementar uma reclassificação para gerar uma camada binaria de floresta (valor de “1”) e não-floresta (valor de “0”).

```
# Carregar
mapbiomas_2020 <- rast(utm_cover_AP_rio_2020)

# Reclassificação -
# Criar uma nova camada de floresta (novo objeto de raster copiando mapbiomas_2020,
# assim para ter os mesmos coordenados, resolução e extensão)
floresta_2020 <- mapbiomas_2020
# Todos os pixels com valor de 0
values(floresta_2020) <- 0
# Atualizar com valor de 1 quando pixels originais são de floresta (classe 3 e 4)
floresta_2020[mapbiomas_2020==3 | mapbiomas_2020==4] <- 1
```

Plotar para verificar, incluindo nomes e os cores para classes de floresta (valor = 1) e não-floresta (valor = 0).

```
# Passo necessário para agilizar o processamento
floresta_2020_modal <- aggregate(floresta_2020,
                                    fact=10,
                                    fun="modal")

# Plot
tm_shape(floresta_2020_modal) +
  tm_raster(style = "cat",
             palette = c("0" = "#E974ED", "1" ="#129912"), legend.show = FALSE) +
  tm_add_legend(type = "fill", labels = c("não-floresta", "floresta"),
                 col = c("#E974ED", "#129912"), title = "Classe") +
  tm_layout(legend.bg.color = "white")
```

Se estiver todo certo, vocês devem ter uma imagem assim:

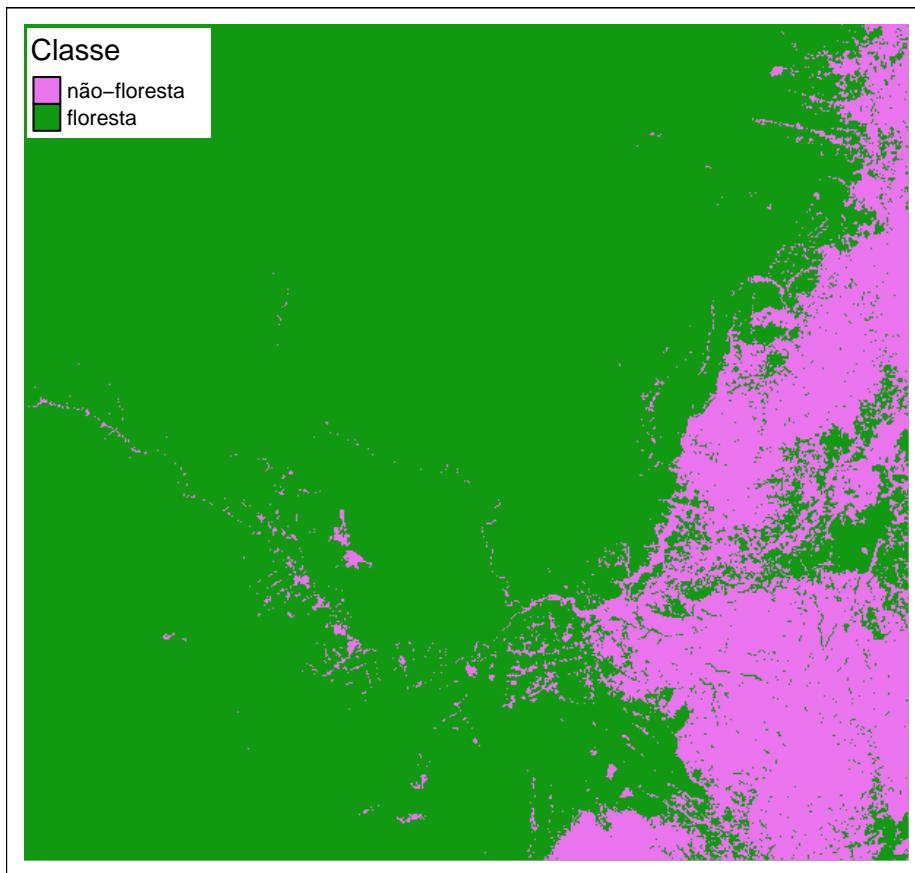


Figura 2.1: Floresta ao redor do Rio Araguari. MapBiomas 2020 reclassificado em floresta e não-floresta.

### 2.3.1 Exibir dados raster e sobreposição com locais de amostragem

Agora temos a paisagem, precisamos também os pontos de amostra. Por isso, precisamos carregar os dados de rios e pontos de amostragem que usamos no tutorial Escala. Vamos carregar as camadas, e no mesmo tempo implementar uma reprojeção, assim para que as sistemas de coordenadas ficam iguais para todas as camadas - tanto de vector quanto raster.

No código abaixo, usamos |>, que estabelece a ligação entre os passos do processo. Ou seja, |> passa o objeto anterior automaticamente para a próxima função como primeiro argumento. Primeiro, carregamos os dados. Em seguida, usamos a função `st_transform` para converter as coordenadas para o mesmo sistema de referência do arquivo raster. As duas etapas ficam ligados através a função |>.

```
# pontos cada 5 km
rioPontos_31976 <- rioPontos |>
  st_transform(31976)
# linha central de rios
rioLinhaCentral_31976 <- rioLinhaCentral |>
  st_transform(31976)
```

Visualizer para verificar.

```
# Passo necessário para agilizar o processamento
floresta_2020_modal <- aggregate(floresta_2020,
                                    fact=10,
                                    fun="modal")

# Mapa
tm_shape(floresta_2020_modal) +
  tm_raster(style = "cat",
             palette = c("0" = "#E974ED", "1" ="#129912"), legend.show = FALSE) +
  tm_add_legend(type = "fill", labels = c("não-floresta", "floresta"),
                col = c("#E974ED", "#129912"), title = "Classe") +
  tm_shape(rio_linhacentral_31976) +
  tm_lines(col="blue") +
  tm_shape(rioPontos_31976) +
  tm_dots(size = 0.2, col = "yellow") +
  tm_layout(legend.bg.color="white")
```

Depois de executar (“run”) o código acima, você deverá ver a figura a seguir.

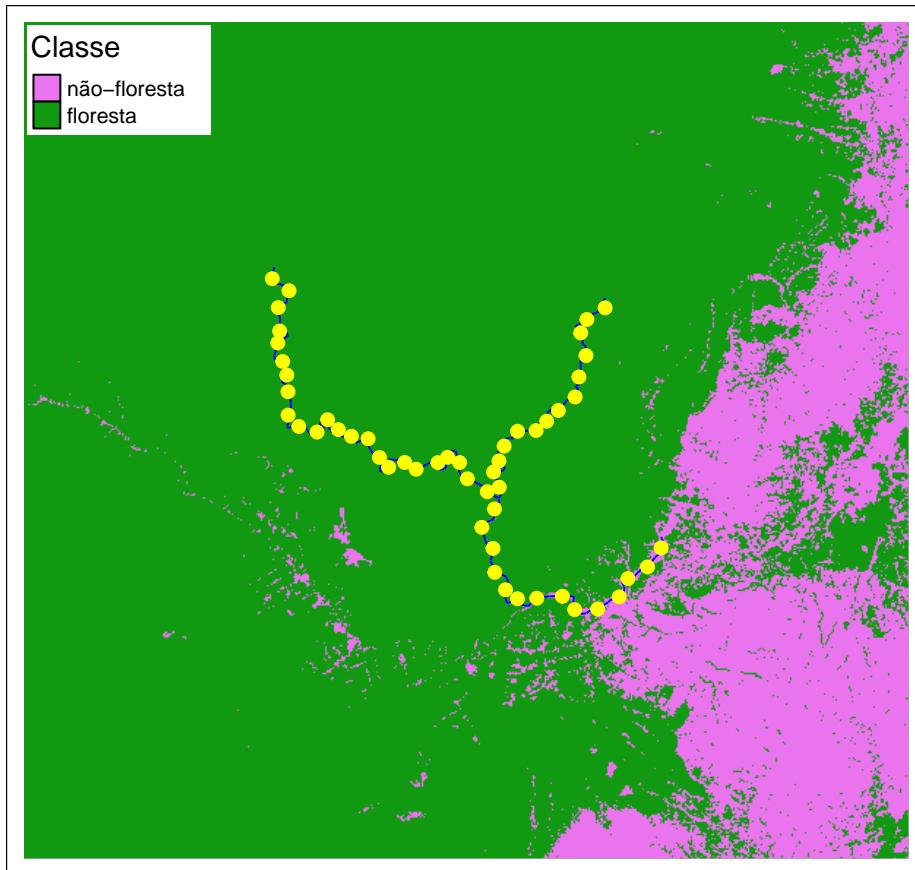


Figura 2.2: Cobertura da terra ao redor do Rio Araguari em 2020. Mostrando os pontos de amostragem (pontos amarelos) cada 5 quilômetros ao longo do rio (linha azul).

## 2.4 Calculo de métricas

Agora com todos as camadas organizados e verificados podemos calcular as métricas de paisagem. Para ilustrar como rodar as funções e cálculos com landscapemetrics, vamos calcular o percentual de área central na paisagem ( `cpland` ). Vamos estudar uma classe (floresta), portanto vamos incluir as métricas para nível de classe. No geral, as métricas de paisagem em nível de classe são mais eficazes na definição de processos ecológicos (Tischendorf, L. Can landscape indices predict ecological processes consistently?. *Landscape Ecology* 16, 235–254 (2001). <https://doi.org/10.1023/A:1011112719782>).

Para calcular as métricas de paisagem dentro de um certo buffer em torno de pontos de amostra, existe a função `sample_lsm()`. Através da função `sample_lsm()` podemos calcular mais de 50 métricas da paisagem, dentro de buffers (raios/distâncias) diferentes.

### 2.4.1 Ponto único, raio único, métrica única

Métricas de área central (“core area”) são consideradas medidas da qualidade de habitat, uma vez que indica quanto existe realmente de área efetiva de um fragmento/classe, após descontar-se o efeito de borda. Vamos calcular a percentual de área central (“core area”) no entorno de um ponto de amostragem. Isso seria, a percentual de áreas centrais (excluídas as bordas de 30 m) de cada classe em relação à área total da paisagem.

Para a função `sample_lsm()` funcionar, precisamos informar (i) a paisagem (arquivo de raster), (ii) ponto (arquivo vector), (iii) raio do buffer desejada, (iv) forma do buffer (círculo ou quadrado) e por final (v) a métrica desejada. Cada opção tem especificações particulares, assim para que a função pode receber dados em formatos diferentes e produzir resultados conforme as necessidades de diversos casos.

Para este primeiro exemplo trabalharemos com apenas um ponto. Fazemos isso selecionando a primeira linha (primeiro ponto) do objeto com os 52 pontos de amostragem (“rio\_pontos\_31976”) usando colchetes: `[1, ]`.

```
# com a paisagem "floresta_2020",
# fazer uma amostragem de um ponto "rio_pontos_31976[1, ]"
# dentro de uma buffer com raio de 1000 metros e forma circular
# e calcular a métrica "cpland"

minha_amostra_1000 <- sample_lsm(landscape = floresta_2020,
                                    y = rio_pontos_31976[1, ],
                                    size = 1000,
                                    shape = "circle",
                                    metric = "cpland",
                                    edge_depth = 1)
```

Depois que executar (“run”), podemos olhar os dados com o código a seguir. Rodando o nome do objeto, como no próximo bloco de código podemos verificar os resultados.

```
minha_amostra_1000
```

Os dados deve ter os valores (coluna “value”) da métrica (coluna “metric”) de cada classe (coluna “class”) conforme a próxima tabela:

layer	level	class	id	metric	value	plot_id	percentage_inside
1	class	0	NA	cpland	66.95102	1	103.4332
1	class	1	NA	cpland	19.59274	1	103.4332

**2.4.1.1 Pergunta 2** O modelo mancha-corredor-matriz é frequentemente adotado na ecologia da paisagem. Com base nas aulas teóricas e usando os valores no objeto `minha_amostra_1000` apresentados na tabela acima, identificar qual classe representar a matriz na paisagem. Há alguma informação faltando que limita a sua capacidade de identificar qual classe representar a

matriz? Se sim, o que precisa ser adicionado? Justifique as suas respostas de forma clara e concisa.

---

#### 2.4.2 Ponto único, distâncias variados, métrica única

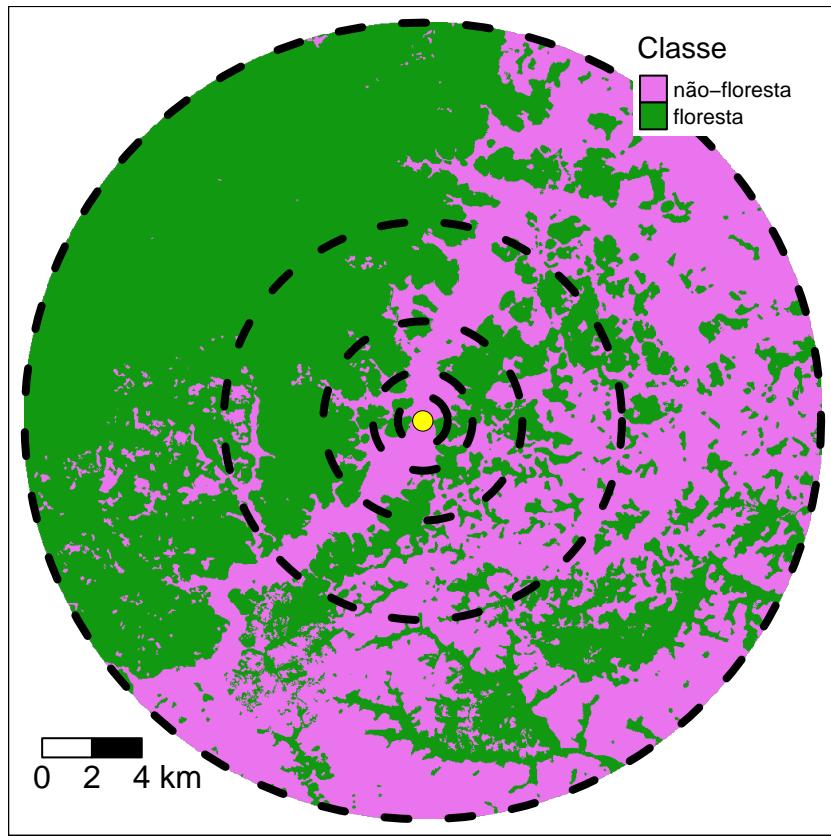


Figura 2.3: Cobertura florestal em extensões diferentes ao redor de um ponto de amostragem.

Para uma comparação multiescala, vamos calcular a mesma métrica, no mesmo ponto, mas agora com extensões diferentes. Continuando o exemplo no tutorial anterior (Escala), vamos repetir o mesmo processo, mas agora com raios de 250, 500, 1000, 2000, 4000, 8000 e 16000 metros, dobrando a escala (extensão) em cada passo.

Para obter resultados com extensões diferentes, precisamos primeiramente repetir o código, ajustando para cada extensão, e depois juntar os resultados. O código a seguir calculará a mesma métrica para as diferentes distâncias. No exemplo, usamos |>, que estabelece a ligação entre os passos do processo. Neste caso, para incluir uma coluna nova (“raio”) para manter o valor das diferentes distâncias.

```
# raio 250 metros
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
            size = 250, shape = "circle",
            metric = "cpLand") |>
  mutate(raio = 250) -> minhaAmostra_250
# raio 500 metros
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
            size = 500, shape = "circle",
            metric = "cpLand") |>
  mutate(raio = 500) -> minhaAmostra_500
# raio 1 km (1000 metros)
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
            size = 1000, shape = "circle",
            metric = "cpLand") |>
  mutate(raio = 1000) -> minhaAmostra_1000
# raio 2 km
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
            size = 2000, shape = "circle",
            metric = "cpLand") |>
  mutate(raio = 2000) -> minhaAmostra_2000
# raio 4 km
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
            size = 4000, shape = "circle",
            metric = "cpLand") |>
  mutate(raio = 4000) -> minhaAmostra_4000
# raio 8 km
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
            size = 8000, shape = "circle",
            metric = "cpLand") |>
  mutate(raio = 8000) -> minhaAmostra_8000
# raio 16 km
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
            size = 16000, shape = "circle",
            metric = "cpLand") |>
  mutate(raio = 16000) -> minhaAmostra_16000
```

E agora, o código a seguir juntará os resultados das diferentes extensões.

```
bind_rows(minha amostra_250,
          minha amostra_500,
          minha amostra_1000,
          minha amostra_2000,
          minha amostra_4000,
          minha amostra_8000,
          minha amostra_16000) -> amostras_metrica
```

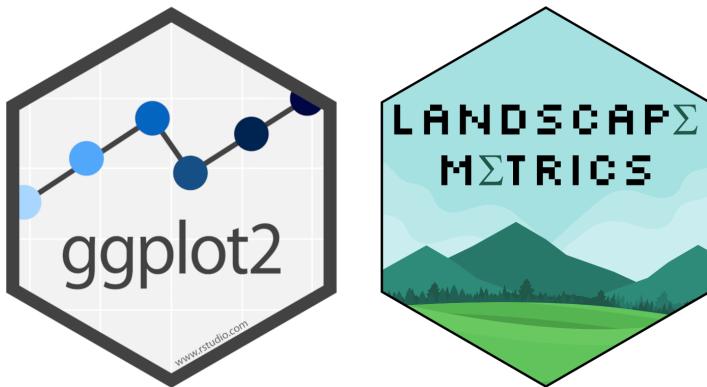
Depois que executar (“run”), podemos olhar os dados “amostras\_metrica” com o código a seguir.

```
amostras_metrica
```

Os dados deve ter os valores (coluna value) da métrica (coluna metric) de cada classe (coluna class) para cada distância (coluna raio):

layer	level	class	id	metric	value	plot_id	percentage_inside	raio
1	class	0	NA	cpland	80.3	1	113	250
1	class	0	NA	cpland	86.4	1	106	500
1	class	1	NA	cpland	0.7	1	106	500
1	class	0	NA	cpland	67.0	1	103	1000
1	class	1	NA	cpland	19.6	1	103	1000
1	class	0	NA	cpland	59.1	1	102	2000
1	class	1	NA	cpland	27.9	1	102	2000
1	class	0	NA	cpland	41.9	1	101	4000
1	class	1	NA	cpland	44.2	1	101	4000
1	class	0	NA	cpland	39.6	1	100	8000
1	class	1	NA	cpland	48.5	1	100	8000
1	class	0	NA	cpland	39.2	1	100	16000
1	class	1	NA	cpland	50.7	1	100	16000

#### 2.4.2.1 Faça um gráfico



Uma imagem vale mais que mil palavras. Portanto, gráficos/figuras/imagens são uma das mais importantes formas de comunicar a ciência. Os dados apresentados em uma tabela podem ser difíceis de entender. Portanto, a primeira pergunta que você deve se fazer é se você pode transformar aquela tabela (chata e feia) em algum tipo de gráfico. Lembrando, sempre pode incluir a tabela como anexo.

Aqui, vamos fazer um grafico com os dados amostras\_metrika, usando o pacote [ggplot2](#).

O ggplot2 faz parte do conjunto de pacotes [tidyverse](#), e é um pacote de visualização de dados. “gg” se refere a uma gramática de gráficos. A ideia principal é criar um gráfico como se fosse uma frase, onde cada elemento do gráfico seria uma palavra, organizados em uma sequencia logica para construir uma frase completa (gráfico final). Você fornece os dados, informa ao ggplot2 como mapear variáveis para estética, quais tipos/formatos gráficas usar e ele cuida dos detalhes.

Isto nos permite construir gráficos tão complexos quanto quisermos. Os gráficos criados com ggplot2 são, em geral, mais elegantes do que os gráficos tradicionais do R. Para mais exemplos e tutoriais com mais detalhes veja os capítulos sobre ggplot2 nos livros:

- [Ciéncia de Dados com R](#)
- [Análises Ecológicas no R](#)
- No livro em inglês [R Graphics Cookbook](#) .
- E sempre pode buscar exemplos no Google, por exemplo digitando: ggplot2 grafico de barra no Google, tem mais de 50 mil resultados com paginas de imagens, código pronto e exemplos no YouTube.

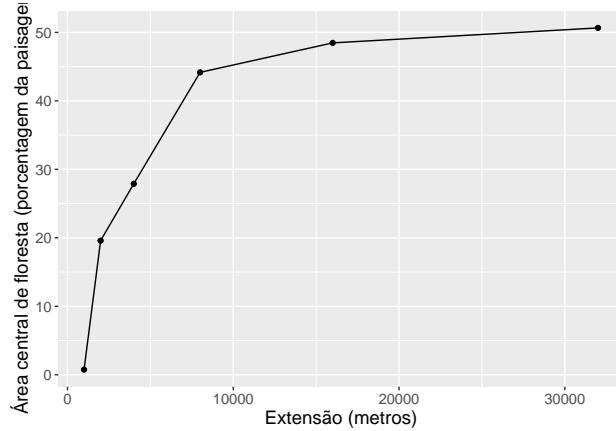
O ggplot2 exige que os dados a serem plotados estejam em um “dataframe” ([tabela de dados](#)). Ou seja, sempre teremos que transformar os dados para dataframe ou construir um dataframe com os dados que possuímos. Dataframe é um formato comum e fácil de trabalhar. Por exemplo, se você importar uma planilha de dados, o resultado seria como dataframe (para mais detalhes veja [Estrutura e manipulação de objetos](#) e [lendo dados](#) ). Além disso, o resultado das funções de [landscapemetrics](#) é sempre um dataframe, ou seja os resultados da função `sample_lsm()` são prontos para um grafico.

O principal função a ser utilizado é `ggplot()`. Para ggplot, precisamos os dados (dataframe), e depois cria o “mapeamento” das variáveis, normalmente usando `aes` (de aesthetics). Ou seja, você especifica quais são as variáveis dos eixos x e y dentro de `aes()`. Através dele vamos definir qual é a variável preditora/explanadora (eixo x) e qual é a variável resposta (eixo y) em nosso conjunto de dados. Depois da função `ggplot()`, na sequencia no codigo nós especificamos qual tipo de grafico com um “geom”. Por exemplo, `geom_point()` para plotar pontos, `geom_boxplot()` para um boxplot, etc. Para a lista completa de geoms e todas as outras opções do pacote, visite a página do projeto ggplot2 <https://ggplot2.tidyverse.org/index.html>.

Aqui vamos fazer um gráfico com valores de extensão no eixo x e proporção da floresta central no eixo y. Assim sendo, com o código a seguir, vamos informar (i) os dados, selecionando classe de floresta através de um filtro e acrescentando uma coluna nova (“ext\_m”) com a extensão em metros, (ii) as colunas para os eixos x e y, (iii) tipo de grafico (grafico de pontos - `geom_point()` e grafico de linha - `geom_line()` ), (iv) nomes para os eixos. No exemplo, usamos `|>`, que estabelece a ligação entre os passos do processo, ligando os dados (`amostras_metrica`) e o grafico `ggplot`. Note que no código a seguir, adicionamos um geom com um “+”. No ggplot2, nós criamos gráficos em camadas, e adicionamos camada a camada com um “+”. Assim, é possível ajustar qualquer elemento do grafico.

```
# arrumar os dados
amostras_metrica |>
  filter(class==1) |> mutate(ext_m = 2*raio) |>
# fazer o grafico
  ggplot(aes(x=ext_m, y=value)) +
  geom_point() + geom_line() +
  labs(x = "Extensão (metros)",
       y = "Área central de floresta (porcentagem da paisagem)")
```

Depois de executar (“run”) o código acima, você deverá ver o grafico a seguir.



**2.4.2.2 Pergunta 3** Em vez de extensão, você preciso incluir o tamanho (área do círculo) correspondente a cada raio. Incluir uma cópia do código ajustado para produzir uma figura com tamanho (área em quilômetros quadrados) no eixo x.

**2.4.2.3 Faça um gráfico elegante** Podemos ajustar qualquer elemento do grafico com ggplot2. Agora, vamos mudar as unidades de metros para quilometros, aumentar o tamanho dos pontos, incluir uma linha reta para ilustrar a tendência geral, colocar o titulo longo do eixo y em duas linhas, e aumentar o tamanho da fonte para o texto ficar mais claro.

```
# arrumar os dados
amostras_metrica |>
  filter(class==1) |>
  mutate(ext_m = 2*raio,
        ext_km = (2*raio)/1000) |>
# fazer o grafico
ggplot(aes(x=ext_km, y=value)) +
  geom_point(size = 4) +
  geom_line() +
  stat_smooth(method = "lm", se = FALSE, color = "green",
             linetype = "dashed") +
  labs(x = "Extensão (quilômetros)",
       y = "Área central de floresta\n(porcentagem da paisagem)") +
  theme(text = element_text(size = 18))

## `geom_smooth()` using formula = 'y ~ x'
```

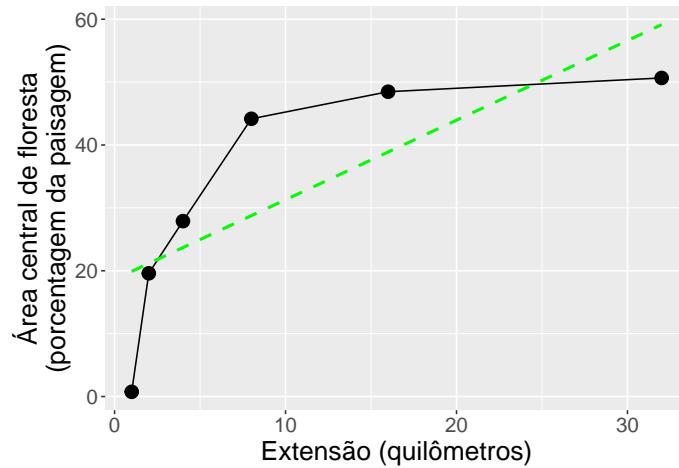


Figura 2.4: Comparaçao da área central de floresta em diferentes extensões.

**2.4.2.4 Pergunta 4 Em menos de 200 palavras apresente a sua interpretação do gráfico em figura 5.2.**

**2.4.2.5 Modelos linear e não linear** Um dos desafios mais frequentes é como melhor representar os dados observados para gerar evidências científicas robustas e informações confiáveis. Nós vimos que as mudanças na métrica porcentagem de área central de floresta não segue uma linha reta em relação de escala (extensão). Para ir além de uma descrição simplista dos padrões observados, na ecologia da paisagem uma variedade de modelos estatísticos são usados. Não vamos rodar modelos (ainda), mas é importante entender algumas das opções disponíveis ao interpretar os gráficos.

Por exemplo, modelos de regressão são amplamente usados em diversas aplicações para descrever a relação entre uma variável resposta Y e uma variável explicativa x. Os modelos lineares são uma generalização dos testes de hipótese clássicos mais simples ([Modelos linear e Modelos lineares](#)). Uma regressão linear, só pode ser aplicada para dados em que tanto a variável preditora quanto a resposta são contínuas, enquanto uma análise de variância é utilizada quando a variável preditora/explícata é categórica. Os modelos lineares generalizados não têm essa limitação, podemos usar variáveis contínuas ou categóricas indistintamente ([Modelos Lineares Generalizados](#)).

Mas, no caso de padrões ecológicos, será que um modelo linear é o melhor modelo para representar a relação que explica “y” em função de “x”? Um número crescente de pesquisadores compartilham o sentimento de que as relações entre variáveis biológicas/ecológicas são melhores descritas por funções não lineares. Processos ecológicos (como por exemplo crescimento, mortalidade, dispersão, e competição) raramente são relacionadas linearmente às variáveis explicativas.

A principal vantagem do modelo não linear sobre o linear é que 1) sua escolha está associada à conhecimento prévio sobre a relação a ser modelada e 2) geralmente apresenta interpretação prática para os parâmetros. Em modelos não-lineares dados observados de uma variável resposta são descritos por uma função de uma ou mais variáveis explicativas que é não linear seus parâmetros. Assim como nos modelos lineares o objetivo é identificar e estabelecer a relação entre variáveis explicativas e resposta. Entretanto, enquanto os modelos lineares definem, em geral, relações empíricas/teóricas, os modelos não-lineares são, em grande parte das vezes, motivados pelo conhecimento do tipo de relação entre as variáveis. Desta forma, as aplicações surgem nas diversas áreas onde relações físicas, biológicas, cinéticas, químicas, fisiológicas, dentre outras, são estabelecidas por funções não lineares que devem ter coeficientes (parâmetros) identificados (estimados) a partir de dados observados, dados experimentais e/ou dados simulados.

Como as mudanças na estrutura da paisagem caracterizam-se por serem não-lineares, para desenvolver análises estatísticas robustas pode (i) aplicar uma transformação (por exemplo, “log”) ou (ii) adotar modelos não-lineares.

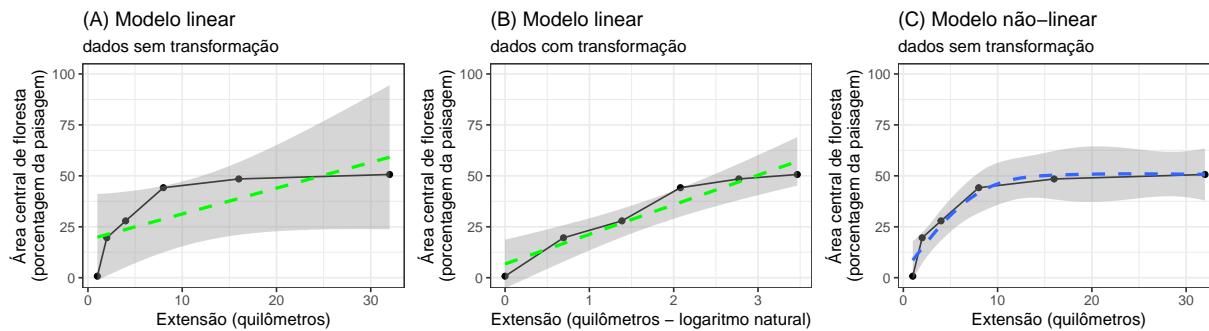


Figura 2.5: Comparação de padrões lineares e não-lineares.

**2.4.2.6 Pergunta 5** [Comparar os resultados apresentados nas figuras com modelos lineares e não-lineares. Como podemos estabelecer qual seria o melhor modelo? Qual modelo seria mais adequado para identificar limiares no padrão de área central de floresta?](#)

#### 2.4.3 Ponto único, distâncias variados, métricas variadas

No exemplo anterior comparamos uma métrica da paisagem em torno de um único ponto de amostragem. Mas sabemos que uma combinação de várias métricas é necessária para entender os padrões na paisagem. Aqui mostraremos como incluir cálculos de diferentes métricas de paisagem ao mesmo tempo.

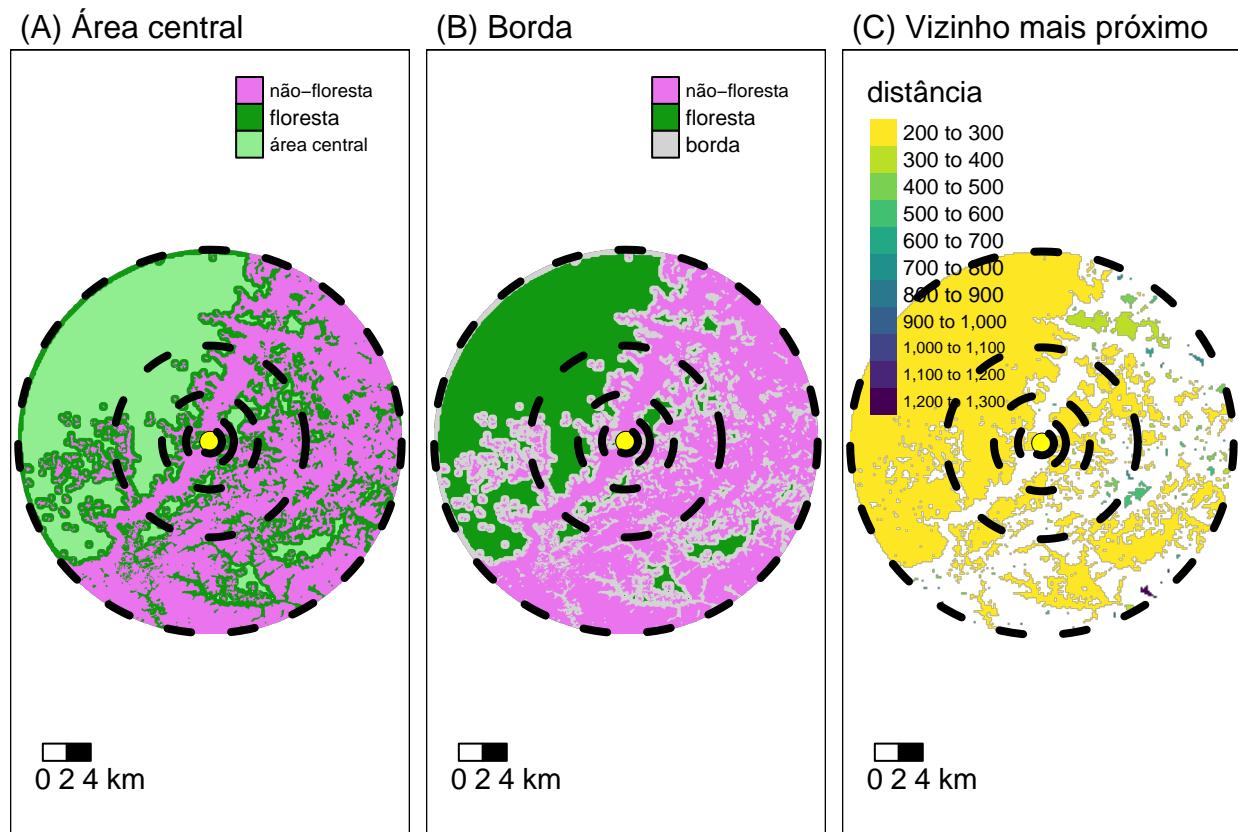


Figura 2.6: Ilustração da determinação de métricas da paisagem diferentes ao redor de um ponto. Exemplo com a estrutura da paisagem representado com três características (A) Área central, (B) Borda e (C) Vizinho mais próximo. O habitat de interesse (classe) é isolado. Um buffer (linha tracejada) é colocado ao redor de um ponto (amarela) e as métricas calculadas. E em seguida o processo é repetido em diferentes extensões.

Não deve calcular todas as métricas disponíveis, mas sim, escolher aquelas que podem ser realmente adequadas para sua pergunta de pesquisa.

Calculando todas as métricas se chama um “tiro no escuro”, algo cujo resultado se desconhece ou é imprevisível. Isso não é recomendado. Para fazer uma escolha melhor (mais robusta), seguindo princípios básicos da ciência, precisamos ler os estudos anteriores (artigos) para obter as métricas mais relevantes para nosso objetivo, pergunta e/ou a hipótese a ser testada. Aqui, como exemplo ilustrativa vamos calcular alguns das métricas mais comuns. Mas, isso não representa necessariamente as métricas mais adequados ou recommendedas.

- Métricas de área e borda (area and edge metrics). Quantificam a composição da paisagem:
  - `pland` = percentage of landscape. Percentagem da paisagem. Porcentagem de cobertura da classe na paisagem.
  - `ed` = edge density . Densidade de borda que é igual à soma dos comprimentos (m) de todos os segmentos de borda que envolvem o fragmento, dividida pela área total da paisagem ( $m^2$ ), sendo posteriormente convertido em hectares.
  - `cpland` = core area percentage of landscape. Percentual de área central (“core”) na paisagem. Percentual de áreas centrais (excluídas as bordas de 30 m) em relação à área total da paisagem. O termo “Core area” foi traduzido como área central ou área núcleo. Aqui vamos adotar área central.
- Métricas de agregação. Quantificam a configuração da paisagem:
  - `enn` = euclidian nearest neighbour distance. Distância euclidiana do vizinho mais próximo.
  - `enn_cv` = Coefficient of variation of euclidean nearest-neighbor distance. Coeficiente de variação da distância euclidiana do vizinho mais próximo. A métrica resume cada classe como o Coeficiente de variação das distâncias euclidianas do vizinho mais próximo entre as manchas pertencentes à classe. O valor de `enn_cv` = 0 se a distância euclidiana do vizinho mais próximo for idêntica para todas as manchas. Aumenta, sem limite, à medida que a variação do ENN aumenta.
  - `enn_sd` = Standard deviation of euclidean nearest-neighbor distance. Desvio padrão da distância euclidiana do vizinho mais próximo.
  - `pd` = Patch density. Densidade das manchas.
  - `cohesion` = Cohesion index. Índice de coesão das manchas.

Para incluir cálculos de diferentes métricas de paisagem ao mesmo tempo, precisamos acrescentar somente uma nova linha de código. Uma nova linha, que cria um objeto com os nomes das funções para as métricas que queremos calcular..... Também precisamos usar a opção “what” na função para aceitar os nomes das funções .

```
# Objeto com os nomes das funções para calcular as métricas desejadas.
# 6 métricas,
# um (enn) com 3 estatísticas (mn = media,
#                               sd = desvio padrão,
#                               cv = coeficiente de variação)
minhas_metricas <- c("lsm_c_pland", "lsm_c_ed", "lsm_c_cpland",
                      "lsm_c_enn_mn", "lsm_c_enn_sd", "lsm_c_enn_cv",
                      "lsm_c_pd", "lsm_c_cohesion")

# 6 Métricas calculadas para cada extensão
# raio 250 metros
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
            size = 250, shape = "circle",
            what = minhas_metricas) |>
  mutate(raio = 250) -> metricasAmostra_250
# raio 500 metros
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
            size = 500, shape = "circle",
            what = minhas_metricas) |>
```

```

    mutate(raio = 500) -> metricas_amostra_500
# raio 1 km (1000 metros)
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
           size = 1000, shape = "circle",
           what = minhas_metricas) |>
  mutate(raio = 1000) -> metricas_amostra_1000
# raio 2 km (2000 metros)
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
           size = 2000, shape = "circle",
           what = minhas_metricas) |>
  mutate(raio = 2000) -> metricas_amostra_2000
# raio 4 km (4000 metros)
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
           size = 4000, shape = "circle",
           what = minhas_metricas) |>
  mutate(raio = 4000) -> metricas_amostra_4000
# raio 8 km (8000 metros)
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
           size = 8000, shape = "circle",
           what = minhas_metricas) |>
  mutate(raio = 8000) -> metricas_amostra_8000
# raio 16 km (16000 metros)
sample_lsm(floresta_2020, y = rioPontos_31976[1, ],
           size = 16000, shape = "circle",
           what = minhas_metricas) |>
  mutate(raio = 16000) -> metricas_amostra_16000

```

E agora, o código a seguir juntará os resultados das diferentes extensões.

```

bind_rows(metricas_amostra_250,
          metricas_amostra_500,
          metricas_amostra_1000,
          metricas_amostra_2000,
          metricas_amostra_4000,
          metricas_amostra_8000,
          metricas_amostra_16000) -> amostras_metricas

```

Depois que executar (“run”), podemos olhar os dados com o código a seguir.

```
amostras_metricas
```

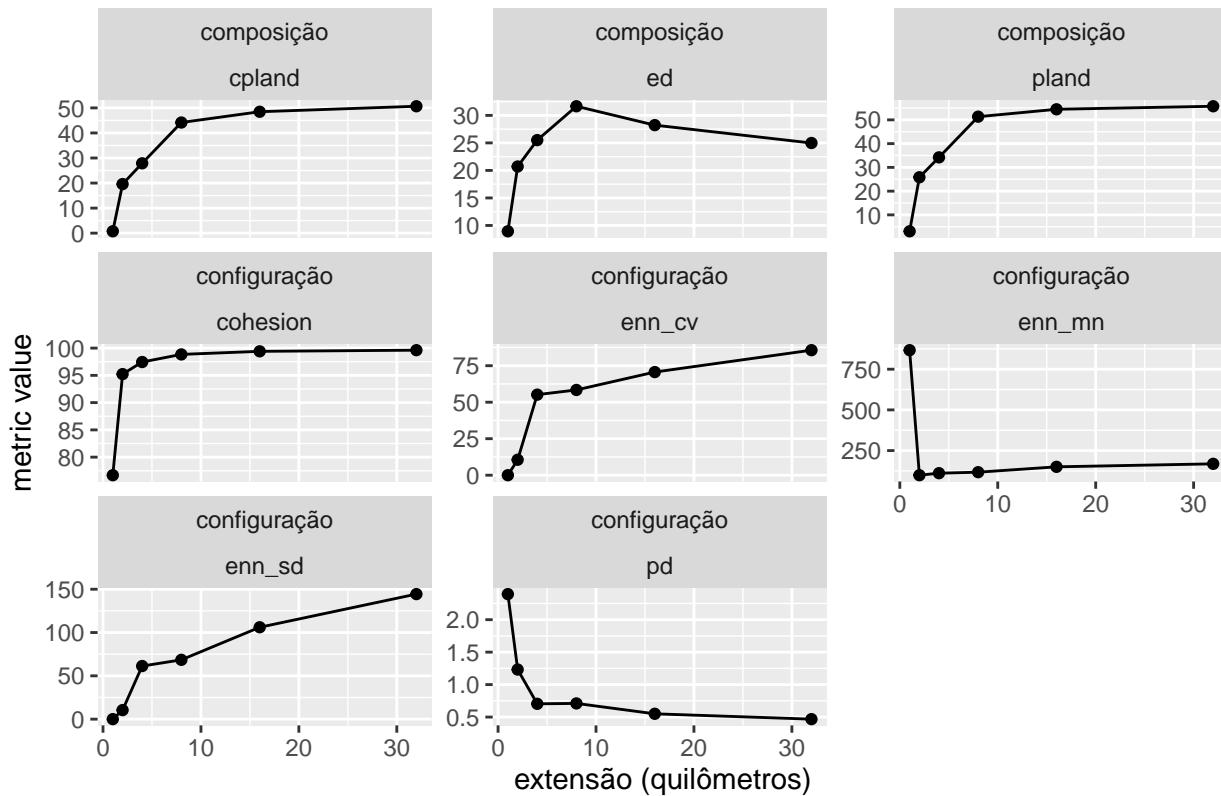
Os dados deve ter os valores (coluna “value”) das métricas (coluna metric) de cada classe (coluna “class”) para cada distância (coluna “raio”):

layer	level	class	id	metric	value	plot_id	percentage_inside	raio
1	class	0	NA	cohesion	100.00	1	113.4	250
1	class	0	NA	cpland	80.32	1	113.4	250
1	class	0	NA	ed	0.00	1	113.4	250
1	class	NA	NA	enn_cv	NA	1	113.4	250
1	class	NA	NA	enn_mn	NA	1	113.4	250
1	class	NA	NA	enn_sd	NA	1	113.4	250
1	class	0	NA	pd	4.49	1	113.4	250
1	class	0	NA	pland	100.00	1	113.4	250
1	class	0	NA	cohesion	99.95	1	106.5	500
1	class	1	NA	cohesion	76.69	1	106.5	500

Agora, vamos fazer um grafico com os dados amostras\_métricas, usando o pacote ggplot2. Para ajudar na visualização incluímos quais métricas são para composição e configuração e nomes que são mais fáceis de entender. A função `mutate` é usado para incluir novas colunas.

```
metricas_composicao <- c("pland", "ed", "cpland")
# arrumar dados
amostras_metricas |>
  filter(class==1) |>
  mutate(ext_km = (2*raio)/1000,
        met_cat = if_else(metric %in% metricas_composicao,
                           "composição", "configuração")) |>
# fazer grafico
ggplot(aes(x=ext_km, y=value)) +
  geom_point() +
  geom_line() +
  facet_wrap(met_cat~metric, scales = "free_y") +
  labs(title = "Comparação multiescala de várias métricas",
       x = "extensão (quilômetros)",
       y = "metric value")
```

Comparação multiescala de várias métricas



**2.4.3.1 Pergunta 6** Com base nos resultados apresentados (figura e tabela) caracterizar as mudanças na paisagem em função de extensões diferentes. Olhando os graficos prever como seria o padrão para extensões maiores (lembrando que valores são dobrados - por exemplo raio de 250 metros gerar uma extensão de 500 metros). Seria relevante repetir incluindo calculos para extensões maiores (por exemplo 64 km e 128 km)? Justifique sua caracterização e previsões de forma clara e concisa, apoie sua escolha com exemplos da literatura científica.

**2.4.3.2 Pergunta 7** Usando como base o conteúdo das aulas, leitura disponível no Google Classroom (Base teórica 4 Dados, métricas, análises), e/ou exemplos apresentados aqui no tutorial, selecione pelo menos oito métricas de nível classe para caracterizar a paisagem de estudo e objectivos da sua projeto. Justifique sua seleção de forma clara e concisa, apoie sua escolha com exemplos da literatura científica.

---

## Part III

# Exemplos de caso

## 3 Garimpo do Lourenço

### 3.1 Apresentação

Mudanças na paisagem ao redor do Garimpo do Lourenço. Changes in the landscape surrounding the Lourenço gold mine.

Código de R e dados para calcular métricas de paisagem associadas com a exploração de recursos minerários.

O objetivo é calcular métricas de paisagem e descrever a composição e a configuração da paisagem no entorno do Garimpo do Lourenço.

As métricas de paisagem são a forma que os ecólogos de paisagem usam para descrever os padrões espaciais de paisagens para depois avaliar a influência destes padrões espaciais nos padrões e processos ecológicos.

Nesta exemplo (<https://rpubs.com/darren75/lourenco>) aprenderemos sobre como analisar a cobertura da terra com métricas de paisagem em R.

Este exemplo tem como base teórica o modelo “mancha-corredor-matriz” - uma representação da paisagem em manchas de habitat (fragmentos).

### 3.2 Pacotes necessários:

```
library(tidyverse)
library(readxl)
library(terra)
library(sf)
library(landscapemetrics)
library(mapview)
library(knitr)
library(gridExtra)
```

### 3.3 Área de estudo

Para alcançar o objetivo de caracterizar a paisagem no entorno do Garimpo do Lourenço, precisamos estabelecer a extensão da área de estudo. Isso seria estabelecida com base nos objetivos e estudos anteriores. Sabemos que atividades associadas com a mineração pode aumentar a perda da floresta até 70 km além dos limites do processo de mineração: Sonter et. al. 2017. Mining drives extensive deforestation in the Brazilian Amazon <https://www.nature.com/articles/s41467-017-00557-w>

Para visualizar um exemplo com a Extração de bauxita na Flona Saracá-Taquera: <https://earthengine.google.com/timelapse/#v=-1.70085,-56.45017,8.939,latLng&t=2.70>

E aqui com o Garimpo do Lourenço: <https://earthengine.google.com/timelapse#v=2.2994,-51.68423,11.382,latLng&t=0.03>

### 3.4 Dados

#### 3.4.1 Ponto de referência (EPSG: 4326)

Aqui vamos incluir um raio de 20 km além do ponto de acesso para o Garimpo do Lourenço em 1985. Isso representa uma área quadrada de 40 x 40 km (1600 km<sup>2</sup>).

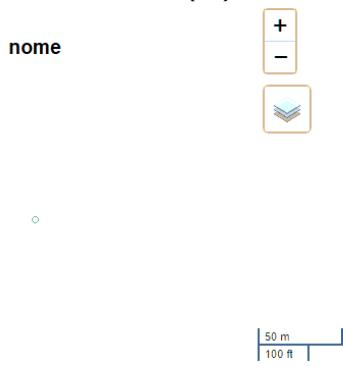
```
# Tabela de dados com coordenados de acesso em 1985.
acesso <- data.frame(nome = "garimpo do Lourenço",
                      coord_x = -51.630871,
                      coord_y = 2.318514)

# Converter para objeto espacial, com sistema de coordenados geográfica.
sf_acesso <- st_as_sf(acesso,
                      coords = c("coord_x", "coord_y"),
                      crs = 4326)
```

Visualizar para verificar.

```
# 
plot(sf_acesso) # teste basica
mapview(sf_acesso) #verificar com mapa de base (OpenStreetMap)
```

(A) basica



(B) com mapa de base



### 3.4.2 Ponto de referência (EPSG: 31976)

As análises da paisagem com o modelo “mancha-corredor-matriz” depende de uma classificação categórica. Portanto, deve optar para uma sistema de coordenados projetados, com pixels de área igual e com unidade em metros. Temos um raio de 20 km, que é um area geográfica onde o retângulo envolvente é menor que um fuso UTM. Assim sendo, vamos adotar a sistema de coordenados projetados de datum SIRGAS 2000, especificamente EPSG:31976 (SIRGAS 2000/UTM zone 22N).

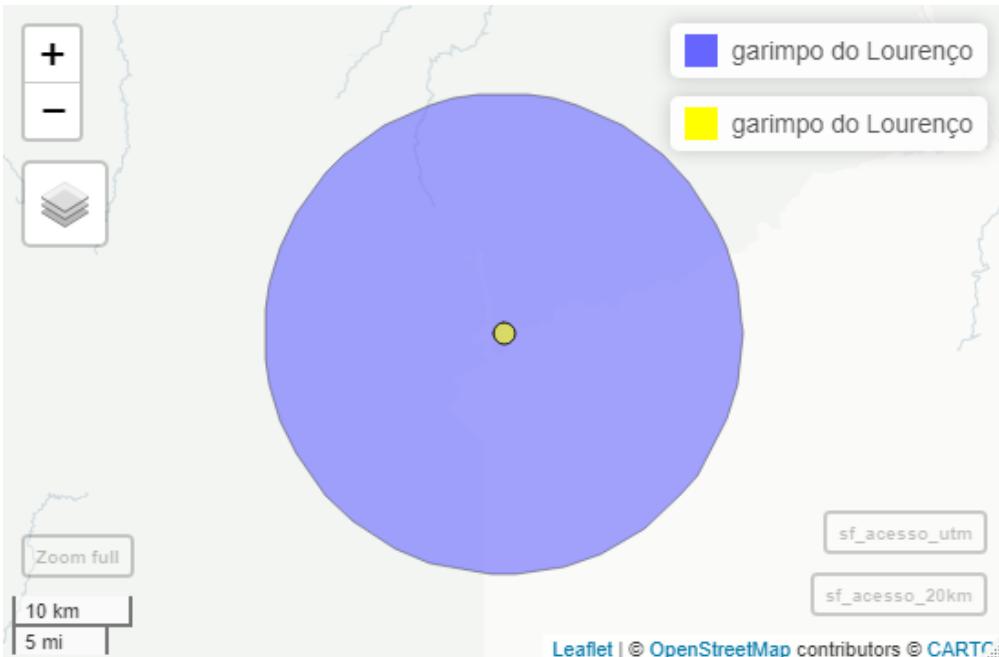
Precisamos então reprojetar o objeto original (em coordenadas geográficas) para a sistema de coordenados projetados. Em seguida, vamos produzir um polígono com raio de 20 km no entorno do ponto.

```
# Reprojetar o ponto.
sf_acesso_utm <- st_transform(sf_acesso, crs = 31976)
# Polígono com raio de 500 metros no entorno do ponto.
sf_acesso_500m <- st_buffer(sf_acesso_utm, dist=500) %>%
  mutate(raio_km = 0.5)
# Polígono com raio de 1 km no entorno do ponto.
sf_acesso_1km <- st_buffer(sf_acesso_utm, dist=1000) %>%
  mutate(raio_km = 1)
# Polígono com raio de 2 km no entorno do ponto.
sf_acesso_2km <- st_buffer(sf_acesso_utm, dist=2000) %>%
  mutate(raio_km = 2)
# Polígono com raio de 4 km no entorno do ponto.
sf_acesso_4km <- st_buffer(sf_acesso_utm, dist=4000) %>%
  mutate(raio_km = 4)
# Polígono com raio de 20 km no entorno do ponto.
sf_acesso_20km <- st_buffer(sf_acesso_utm, dist=20000)

acesso_buffers <- bind_rows(sf_acesso_500m, sf_acesso_1km,
                           sf_acesso_2km, sf_acesso_4km)
```

### 3.4.3 Verificar com mapa de base (OpenStreetMap).

```
# 
mapview(sf_acesso_20km) +
  mapview(sf_acesso_utm, color = "black", col.regions = "yellow")
```



### 3.4.4 Dados: MapBiomas cobertura da terra

Agora vamos olhar cobertura e uso da terra no espaço que preciso (área de estudo). Para isso, vamos utilizar um arquivo de raster do projeto [MapBiomas](#) com cobertura de terra ao redor do Garimpo do Lourenço em 1985. Este arquivo no formato raster, tem apenas valores inteiros, em que cada célula/pixel representa uma área considerada homogênea, como uso do solo ou tipo de vegetação. Arquivo ".tif" disponível aqui: [utm\\_cover\\_AP\\_lorenco\\_1985.tif](#)

Não vamos construir mapas, portanto os cores nas visualizações não corresponde ao mundo real (por exemplo, verde não é floresta). Para visualizar em QGIS preciso baixar um arquivo com a legenda e cores para Coleção<sup>6</sup> (<https://mapbiomas.org/codigos-de-legenda>) e segue tutoriais: <https://www.youtube.com/watch?v=WtyotodHK8E>.

Este vez, a entrada de dados espaciais seria através a importação de um raster (arquivo de .tif). Lembre-se, para facilitar, os arquivos deve ficar no mesmo diretório do seu código (verifique com `getwd()`). Como nós já sabemos a sistema de coordenadas desejadas, o geoprocessamento da raster foi concluído antes de começar com as análises da paisagem.

```
r1985 <- rast("utm_cover_AP_lorenco_1985.tif")
r1985

#class      : SpatRaster
#dimensions : 1341, 1341, 1  (nrow, ncol, nlyr)
#resolution : 29.87713, 29.87713  (x, y)
#extent     : 409829.5, 449894.7, 236241.1, 276306.3  (xmin, xmax, ymin, ymax)
#coord. ref.: SIRGAS 2000 / UTM zone 22N (EPSG:31976)
#source     : utm_cover_AP_lorenco_1985.tif
#name       : classification_1985
#min value  : 
#max value  : 
```

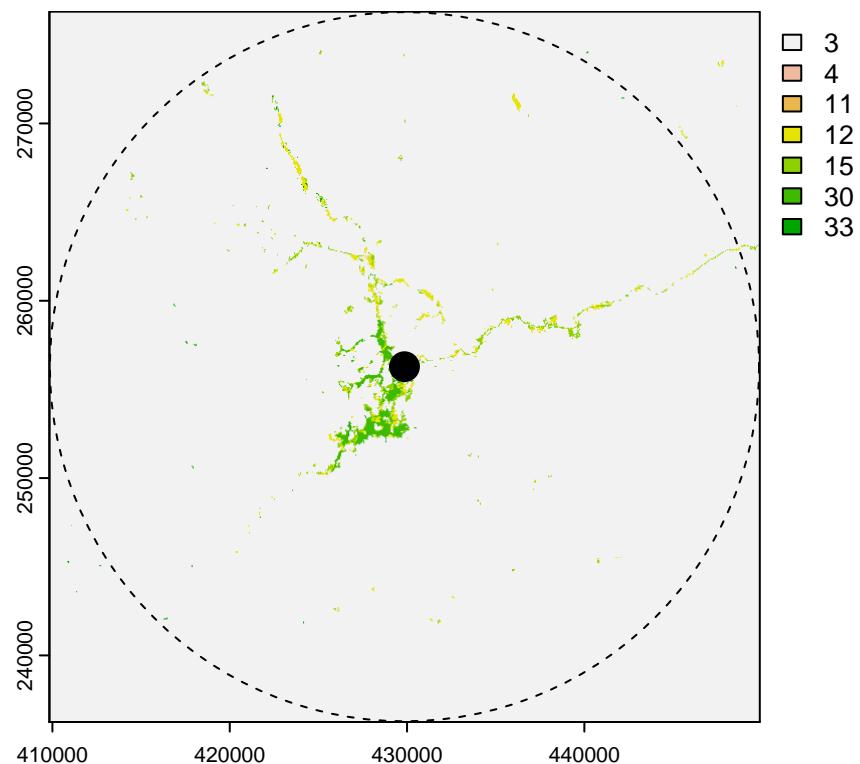
Ou use o função `file.choose()`, que faz a busca para arquivos.

```
r1985 <- rast(file.choose())
r1985
```

```
##           used   (Mb) gc trigger   (Mb) max used   (Mb)
## Ncells  6798698 363.1    13230425 706.6    13230425 706.6
## Vcells 31940250 243.7    96509484 736.4   134390330 1025.4
```

Agora que o arquivo foi importado, podemos visualizá-lo.

```
# Visualizar para verificar
# Gradiente de cores padrão não corresponde
# ao mundo real (por exemplo verde não é floresta)
plot(r1985, type="classes")
plot(sf_acesso_20km, add = TRUE, lty ="dashed", color = "black")
plot(sf_acesso_utm, add = TRUE, cex = 2, pch = 19, color = "black")
```



### 3.5 Calculo de métricas

Vamos olhar alguns exemplos de métricas para cada nível da análise:

- landscape (métricas para a paisagem como um todo).
- class (métricas por classe ou tipo de habitat).
- patch (para a mancha ou fragmento).

Primeiro, precisamos verificar se o raster está no formato correto.

```
check_landscape(r1985)
```

```
##   layer      crs units   class n_classes OK
## 1     1 projected   m integer          7  v
```

```
# layer crs    units   class n_classes OK
# 1 projected  m    integer        7  v
```

Tudo certo (veja a coluna do “OK”)!

### 3.5.1 Métricas para a paisagem

Vamos começar avaliando a área total da paisagem (área) de estudo.

```
area.total <- lsm_l_ta(r1985)
area.total #160264 Hectares
```

```
## # A tibble: 1 x 6
##   layer level      class     id metric   value
##   <int> <chr>     <int> <int> <chr>    <dbl>
## 1     1 landscape NA     NA ta      160264.
```

Agora vamos ver a distância total de borda (te= “total edge”).

```
##           used   (Mb) gc trigger   (Mb) max used   (Mb)
## Ncells  6797919 363.1   13230425 706.6  13230425 706.6
## Vcells 31939283 243.7   96509484 736.4 134390330 1025.4
te <- lsm_l_te(r1985)
te # 547140 metros
```

```
## # A tibble: 1 x 6
##   layer level      class     id metric   value
##   <int> <chr>     <int> <int> <chr>    <dbl>
## 1     1 landscape NA     NA te      547140.
```

Total de borda mede a configuração da paisagem porque uma paisagem altamente fragmentada terá muitas bordas. No entanto, a borda total é uma medida absoluta, dificultando comparações entre paisagens com áreas totais diferentes. Mas pode ser aplicado para comparar a configuração na mesma paisagem em anos diferentes.

Agora vamos ver a densidade de Borda (“Edge Density”). Densidade de Borda mede a configuração da paisagem porque uma paisagem altamente fragmentada terá valores mais altos. “Densidade” é uma medida adequada para comparações de paisagens com áreas totais diferentes.

```
ed <- lsm_l_ed(r1985)
ed #3.41 metros por hectare
```

```
## # A tibble: 1 x 6
##   layer level      class     id metric   value
##   <int> <chr>     <int> <int> <chr>    <dbl>
## 1     1 landscape NA     NA ed      3.41
```

### 3.5.2 Métricas para as classes

Área de cada classe em hectares.

```
lsm_c_ca(r1985)
```

```
## # A tibble: 7 x 6
##   layer level class     id metric   value
##   <int> <chr>  <int> <int> <chr>    <dbl>
## 1     1 class    3     NA ca      158582.
## 2     1 class    4     NA ca       1.70
```

```

## 3      1 class    11     NA ca       1.79
## 4      1 class    12     NA ca      548.
## 5      1 class    15     NA ca      563.
## 6      1 class    30     NA ca      526.
## 7      1 class    33     NA ca      41.4

```

Como tem varios classes é dificil de interpretar os resultados porque os numeros (3, 4, 11....) não tem uma referencia do mundo real. Para entender os resultados, precisamos acrescentar nomes para os valores. Ou seja incluir uma coluna de legenda com os nomes para cada classe. Para isso precisamos outro arquivo com os nomes.

Baixar o arquivo de legenda: [mapbiomas\\_6\\_legend.xlsx](#).

Agora carregar o arquivo com o código a seguir.

```
class_nomes <- read_excel(file.choose())
```

Agora rodar de novo, com os resultados juntos com a legenda de cada classe. Nos resultados acima, os valores na coluna “class” são as mesmas que tem na coluna “aid” no objeto “class\_nomes”, onde também tem os nomes . Assim, podemos repetir, mas agora incluindo os nomes para cada valor de class, com base na ligação (join) entre as colunas.

```

# Área de cada classe em hectares, incluindo os nomes para cada classe
lsm_c_ca(r1985) %>%
  left_join(class_nomes, by = c("class" = "aid"))

# Número de fragmentos (manchas)
lsm_c_np(r1985) %>%
  left_join(class_nomes, by = c("class" = "aid"))

# Maior número de manchas em classes de cobertura classificadas como
# pasto (pasture) e formação campestre (grassland).

#   layer level class    id metric value class_description group_description
# 1 1 class    3     NA np      28 Forest Formation Natural forest
# 2 1 class    4     NA np      2 Savanna Formation Natural forest
# 3 1 class   11     NA np      7 Wetlands           Natural non fore
# 4 1 class   12     NA np     246 Grassland        Natural non fore.
# 5 1 class   15     NA np     262 Pasture          Farming
# 6 1 class   30     NA np      35 Mining            Non vegetated
# 7 1 class   33     NA np     50 River,Lake and Ocean Water

```

### 3.5.3 Métricas para as manchas

Vamos calcular o tamanho de cada mancha agora.

```
mancha_area <- lsm_p_area(r1985) # 630 manchas
mancha_area
```

Agora queremos saber o tamanho da maior mancha em cada class, e portanto o tamanho da maior mancha de mineração.

```

mancha_area %>%
  group_by(class) %>%
  summarise(max_ha = max(value))
# 30.8 hectares (class 15 = mineração)

```

### 3.5.4 Quais métricas devo escolher?

A decisão deve ser tomada com base em uma combinação de fatores. Incluindo tais fatores como: base teórica, considerações estatísticas, relevância para o objetivo/hipótese e a escala e heterogeneidade na paisagem de estudo.

Queremos caracterizar áreas de mineração na paisagem, e aqui vamos olhar somente uma paisagem, em um momento do tempo. Então as métricas para a paisagem como todo não tem relevância.

Estamos olhando uma classe (mineração), portanto vamos incluir as métricas para classes. Além disso, as métricas de paisagem em nível de classe são mais eficazes na definição de processos ecológicos (Tischendorf, L. Can landscape indices predict ecological processes consistently?. *Landscape Ecology* 16, 235–254 (2001). <https://doi.org/10.1023/A:1011112719782.>).

```
# métricas de composição para a paisagem por classes
list_lsm(level = "class", type = "area and edge metric")

# métricas de configuração para a paisagem por classes
list_lsm(level = "class", type = "aggregation metric")
```

### 3.5.5 Métricas por classe de mineração

Aqui vamos calcular todos as métricas por classe (função calculate\_lsm()).

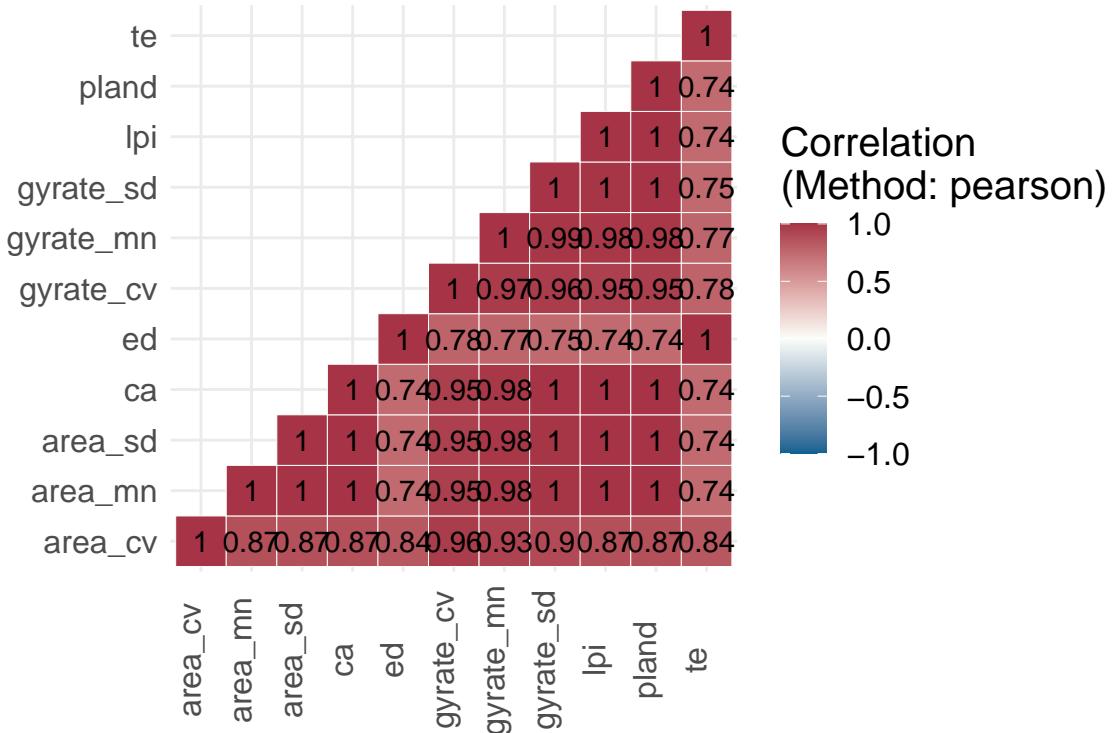
```
# métricas de composição para a paisagem por classes
metrics_comp <- calculate_lsm(r1985, level = "class", type = "area and edge metric")

# métricas de configuração para a paisagem por classes
metrics_config <- calculate_lsm(r1985, level = "class", type = "aggregation metric")
```

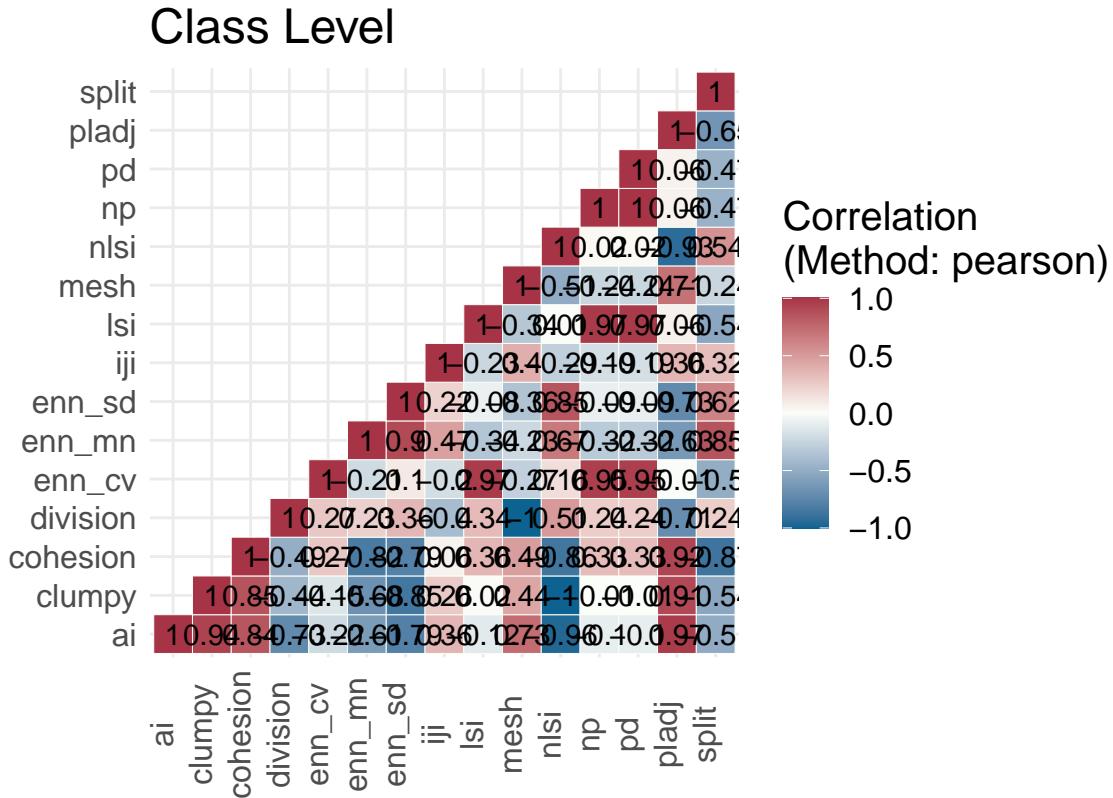
E aqui, calcular correlações entre todos as métricas por classe (função show\_correlation()).

```
show_correlation(data = metrics_comp, method = "pearson", labels = TRUE)
```

## Class Level



```
show_correlation(data = metrics_config, method = "pearson", labels = TRUE)
```



Temos muitos valores e muitas métricas. Este se chama um “tiro no escuro”, algo cujo resultado se desconhece ou é imprevisível. Isso não é recomendado. Para fazer uma escolha melhor (mais robusta), seguindo princípios básicos da ciência, precisamos ler os estudos anteriores (artigos) para obter as métricas mais relevantes para nosso objetivo e a hipótese a ser testada. Com base em os estudos anteriores e os objetivos vamos incluir 8 métricas nos resultados.

### 3.5.6 Exportar as métricas

O próximo passo é comunicar os resultados obtidos. Para isso precisamos resumir e apresentar as métricas selecionadas em tabelas e figuras. Agora já fizemos os cálculos, as tabelas e figuras podem ser feitas no R ([figuras](#)), tanto quanto em aplicativos diferentes (por exemplo tabelas através [“tabelas dinâmicas”] no [Microsoft Excel](#) ou [LibreOffice calc](#)). Mas por isso, primeiramente precisamos exportar os resultados (veja mais exemplos aqui: [Introdução ao R import-export](#)).

O arquivo vai sair no mesmo diretório do seu código (verifique com `getwd()`).

```
bind_rows(metrics_comp, metrics_config) -> metricas_1985
write.csv2(metricas_1985, "metricas_lourenco_1985.csv", row.names=FALSE)
```

## 3.6 Preparando os resultados

A entrada de dados seria com as métricas da paisagem calculados anteriormente.

Vocês devem baixar o arquivo de Excel [metricas\\_lourenco\\_1985.xlsx](#). Lembre-se, para facilitar, os dados deve ficar no mesmo diretório do seu código (verifique com `getwd()`).

No caso de um arquivo de Excel simples, a importação poderia ser feita através menu de “Import Dataset” na janela/panel “Environment” de Rstudio. Ou com linhas de código:

```

metricas_1985 <- read_excel("metricas_lourenco_1985.xlsx")
metricas_1985

# layer level class id      metric      value
# <dbl> <chr> <dbl> <chr> <chr>      <dbl>
#   1 class     3 NA    area_cv  529.
#   1 class     4 NA    area_cv  22.3
#   1 class    11 NA   area_cv  71.2

```

Ou use o função file.choose(), que faz a busca para arquivos.

```

metricas_1985 <- read_excel(file.choose())
metricas_1985

```

```

## # A tibble: 182 x 6
##   layer level class id      metric      value
##   <dbl> <chr> <dbl> <chr> <chr>      <dbl>
## 1 1     class     3 NA    area_cv  529.
## 2 2     class     4 NA    area_cv  22.3
## 3 3     class    11 NA   area_cv  71.2
## 4 4     class    12 NA   area_cv  169.
## 5 5     class    15 NA   area_cv  175.
## 6 6     class    30 NA   area_cv  276.
## 7 7     class    33 NA   area_cv  74.2
## 8 8     class     3 NA    area_mn 5664.
## 9 9     class     4 NA    area_mn  0.848
## 10 10    class    11 NA   area_mn  0.255
## # i 172 more rows

```

Os dados são padronizados (“tidy”), mas ainda não parece adequados para apresentação em tabelas ou figuras. Temos muitos valores e muitas métricas (listadas na coluna “metric”). Com base em os estudos anteriores e os objetivos vamos incluir 8 métricas (4 de composição e 4 de configuração).

Métricas de composição:

- mean patch area (lsm\_c\_area\_mn) Área médio das manchas por classe.
- SD patch area (lsm\_c\_area\_sd) Desvio padrão das áreas das manchas por classe.
- class area percentage of landscape (lsm\_c\_pland) Porcentagem de área na paisagem por classe.
- largest patch index (lsm\_c\_lpi) Índice de maior mancha (proporção da paisagem).

Métricas de configuração:

- aggregation index (lsm\_c\_ai) Índice de agregação.
- patch cohesion index (lsm\_c\_cohesion) Índice de coesão das manchas.
- number of patches (lsm\_c\_np) Número de manchas.
- patch density (lsm\_c\_pd) Densidade de manchas.

Escolheremos (atraves um filtro) as métricas que queremos para obter uma tabela de dados. Mantendo os dados originais, assim sendo para acrescentar mais métricas nos resultados, preciso somente acrescentar mais no código.

```

# Arquivo com os nomes das classes
class_in <- "C:\\\\Users\\\\user\\\\Documents\\\\Articles\\\\gis_layers\\\\gisdata\\\\inst\\\\raster\\\\mapbiomas_cover_1"
class_nomes <- read_excel(class_in)

# Especificar métricas desejados
met_comp <- c("pland", "lpi", "area_mn", "area_sd")
met_conf <- c("ai", "cohesion", "np", "pd")

```

```
met.todos <- c(met_comp, met_conf)

# Escolher métricas desejados do conjunto completo
metricas_1985 %>%
filter(metric %in% met.todos) %>%
left_join(class_nomes, by = c("class" = "aid")) -> metricas_nomes
```

### 3.7 Uma tabela versatil

Mas, ainda não tem uma coluna com os nomes das métricas. Portanto, solução simples é de exportar no formato de .csv e finalizar/editar no Excel / calc.

Outra opção que pode facilitar, particularmente quando pode há mudanças e revisões, é produzir a tabela no R. Aqui vamos repetir no R os passos que vocês conhecem com as ferramentas de Excel (arraste e solte, copiar-colar, filtro, tabela dinâmica).

### 3.8 Reorganização

Escolhendo as colunas desejadas (select), reorganizando para as métricas ficam nas colunas (pivot\_wider) e colocando as colunas novas na sequência desejada (select).

```
metricas_nomes %>%
# Escolher métricas desejados do conjunto completo de métricas.
dplyr::select(c(type_class, classe_descricao, hexadecimal_code,
metric, value)) %>%
# reorganizando
pivot_wider(names_from = metric, values_from = value) -> metricas_tab
```

### 3.9 Uma figura elegante

Uma imagem vale mais que mil palavras. Portanto, gráficos/figuras/imagens são uma das mais importantes formas de comunicar a ciência.

Como exemplo ilustrativo, aqui vamos produzir gráficos comparando métricas de composição e configuração da paisagem ao redor do Garimpo do Lourenço.

É uma boa ideia gastar bastante tempo para tornar figuras científicas as mais informativas e atraentes possíveis. Escusado será dizer que a precisão empírica é primordial. E por isso, o que fica excluído/omitido é tão importante quanto o que foi incluído. Para ajudar, você deve se perguntar o seguinte ao criar uma figura: eu apresentaria essa figura em uma apresentação para um grupo de colegas? Eu o apresentaria a um público de não especialistas? Eu gostaria que essa figura aparecesse em um artigo de notícias sobre meu trabalho? É claro que todos esses locais exigem diferentes graus de precisão, complexidade e estética, mas uma boa figura deve servir para educar simultaneamente públicos muito diferentes.

Tabelas versus gráficos — A primeira pergunta que você deve se fazer é se você pode transformar aquela tabela (chata e feia) em algum tipo de gráfico. Você realmente precisa dessa tabela no texto principal? Você não pode simplesmente traduzir as entradas das células em um gráfico de barras/columnas/xy? Se você pode, você deve. Quando uma tabela não pode ser facilmente traduzida em uma figura, na maioria das vezes a provavelmente pertence às Informações Suplementares/Anexos/Apêndices.

#### 3.9.1 Gráfico de barra

Primeiramente, vamos produzir uma gráfico de barra comparando a proporção que cada classe representa na paisagem.

```
# Incluindo cores conforme legenda da Mapbiomas Coleção 6
# Legenda nomes ordem alfabetica
classe_cores <- c("Campo Alagado e Área Pantanosa" = "#45C2A5",
"Formação Campestre" = "#B8AF4F",
"Formação Florestal" = "#006400",
"Formação Savântica" = "#00ff00",
"Mineração" = "#af2a2a",
"Pastagem" = "#FFD966",
"Rio, Lago e Oceano" = "#0000FF")
```

E agora o grafico.....

```
# Grafico de barra basica
metricas_tab %>%
  mutate(class_prop = pland) %>%
  ggplot(aes(x = classe_descricao, y = class_prop)) +
  geom_col()

# Agora com ajustes
# Agrupando por tipo (natural e antropico)
# Com cores conforme legenda da Mapbiomas Coleção 6
# Corrigindo texto dos eixos.
# Mudar posição da legenda para o texto com nomes longas encaixar.
metricas_tab %>%
  mutate(class_prop = pland) %>%
  ggplot(aes(x = type_class, y = class_prop,
  fill = classe_descricao)) +
  scale_fill_manual("classe", values = classe_cores) +
  geom_col(position = position_dodge2(width = 1)) +
```

```

coord_flip() +
labs(title = "MapBiomas cobertura da terra",
subtitle = "Entorno do Garimpo do Lorenço 1985",
y = "Proporção da paisagem (%)",
x = "") +
theme(legend.position="bottom") +
guides(fill = guide_legend(nrow = 4))

```

Uma imagem vale mais que mil palavras:

Mas existe uma separação grande na faixa de valores e ainda é difícil de ver todas as classes. Temos uma distribuição com valores muito mais altos comparada com os outros. extremos. Uma solução seria uma transformação (por exemplo “log”), assim os valores ficarem mais próximos.

### 3.9.2 Gráfico de boxplot

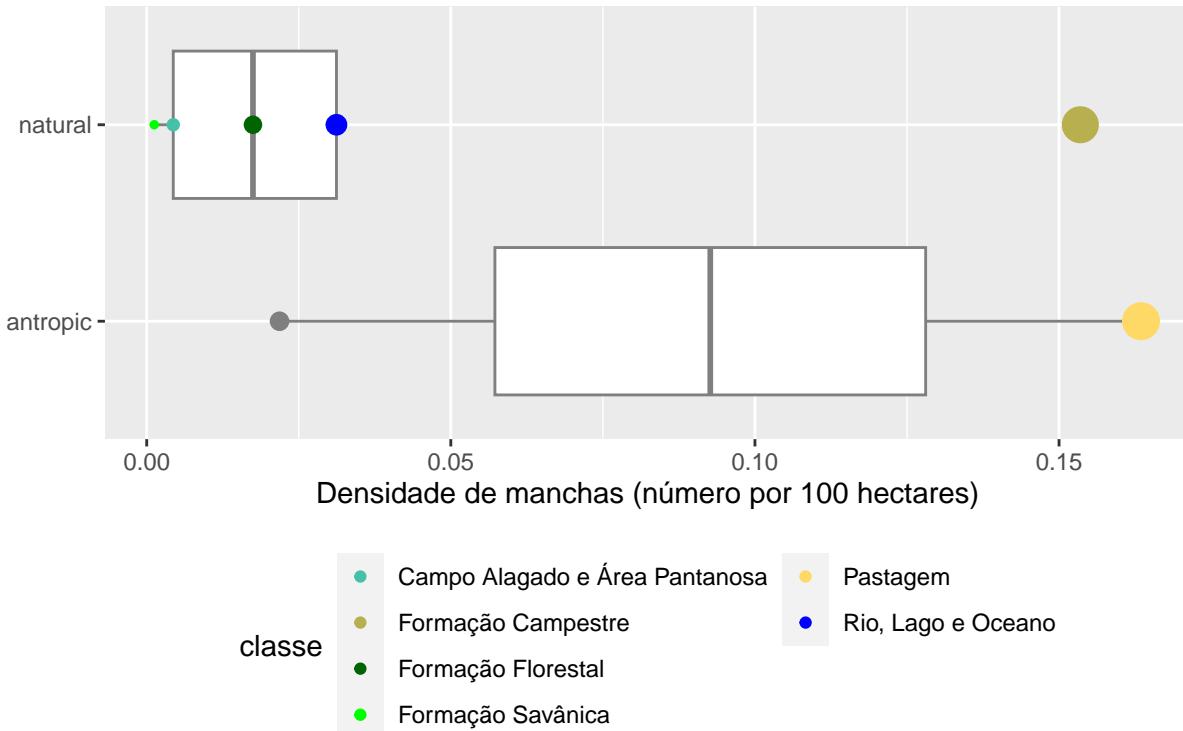
Agora com uma métrica de configuração:

```

# Métrica de configuração: Densidade de manchas (coluna "pd").
# Agrupando por tipo (natural e antrópico)
# Incluindo boxplot indicando tendência central (mediano)
# Com cores conforme legenda da Mapbiomas Coleção 6
# Tamanho dos pontos proporcional o numero de manchas
# Corrigindo texto dos eixos.
# Mudar posição da legenda para o texto com nomes longas encaixar.
metricas_tab %>%
ggplot(aes(x = type_class, y = pd)) +
geom_boxplot(colour = "grey50") +
geom_point(aes(size = np, colour = classe_descricao)) +
scale_color_manual("classe", values = classe_cores) +
scale_size(guide = "none") +
coord_flip() +
labs(title = "MapBiomas cobertura da terra",
subtitle = "Entorno do Garimpo do Lorenço 1985",
y = "Densidade de manchas (número por 100 hectares)",
x = "") +
theme(legend.position="bottom") +
guides(col = guide_legend(nrow = 4))

```

## MapBiomas cobertura da terra Entorno do Garimpo do Lorenço 1985



### 3.10 Comparação entre anos

Calcular as métricas para 3 anos.

```
# metricas desejados
what_metricas <- c("lsm_c_pland", "lsm_c_lpi", "lsm_c_area_mn", "lsm_c_area_sd",
                     "lsm_c_ai", "lsm_c cohesion", "lsm_c_np", "lsm_c_pd")
# rodar
metricas_anos <- sample_lsm(landscape = mapbiomas_85a20,
                             y = acesso_buffers,
                             plot_id = data.frame(acesso_buffers)[, 'raio_km'],
                             what = what_metricas,
                             edge_depth = 1)

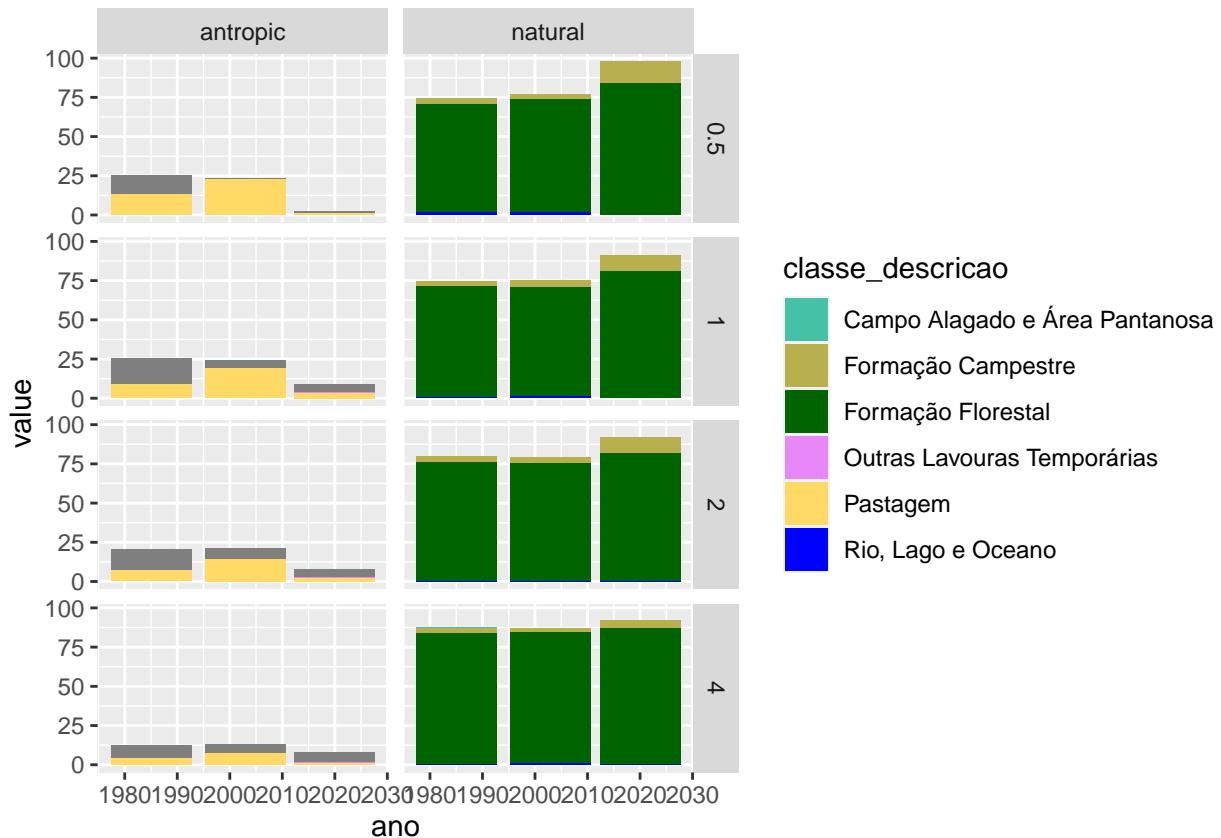
# Organizar dados
# Dados referentes os buffers
resultados_anos <- acesso_buffers %>%
  left_join(metricas_anos %>%
    dplyr::mutate(value = round(value, 2),
    ano = case_when(layer==1 ~1985,
                    layer==2~2003,
                    layer==3~2020)) %>%
    dplyr::select(ano, plot_id, class, metric, value),
    by=c("raio_km"="plot_id"))
# Dados referentes os classes
resultados_anos <- resultados_anos %>%
```

```
left_join(class_nomes, by = c("class" = "aid"))
```

Agora grafico de barra com varios anos

```
# It's recommended to use a named vector
# Legenda nomes ordem alfabetica
classe_cores <- c("Campo Alagado e Área Pantanosa" = "#45C2A5",
"Formação Campestre" = "#B8AF4F",
"Formação Florestal" = "#006400",
"Formação Savânicas" = "#00ff00",
"Mineração" = "#af2a2a",
"Pastagem" = "#FFD966",
"Rio, Lago e Oceano" = "#0000FF",
"Outras Lavouras Temporárias" = "#e787f8")

resultados_anos %>%
  mutate(rcor = paste("#", hexdecimal_code, sep="")) %>%
  filter(metric=="pland") %>%
  ggplot(aes(x=ano, y=value)) +
  geom_col(position="stack", aes(fill=classe_descricao)) +
  scale_fill_manual(values = classe_cores) +
  facet_grid(raio_km~type_class)
```



Agora com “pland” e densidade de manchas juntos.

```
resultados_anos %>%
  mutate(rcor = paste("#", hexdecimal_code, sep="")) %>%
```

```

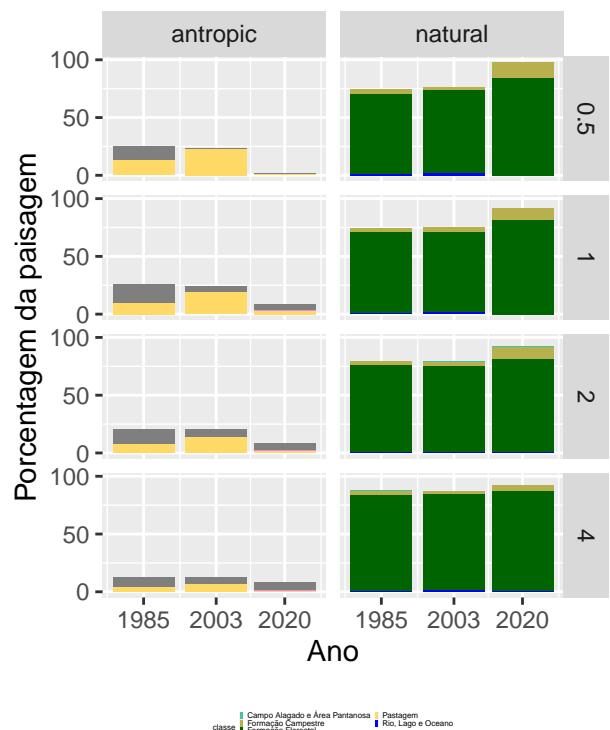
filter(metric=="pland") %>%
ggplot(aes(x=ano, y=value)) +
geom_col(position="stack", aes(fill=classe_descricao)) +
scale_fill_manual("classe", values = classe_cores) +
scale_y_continuous(breaks=c(0,50,100)) +
scale_x_continuous(breaks=c(1985,2003, 2020)) +
facet_grid(raio_km~type_class) +
labs(title = "MapBiomas cobertura da terra",
subtitle = "Entorno do Garimpo do Lorenço 1985-2020",
y = "Porcentagem da paisagem",
x = "Ano") +
theme(legend.position="bottom",
      legend.title = element_text(size = 3),
      legend.text = element_text(size = 3),
      legend.key.size = unit(0.1, "lines")) +
guides(fill = guide_legend(nrow = 4)) -> fig_pland

# Densidade de manchas
resultados_anos %>%
  mutate(rcor = paste("#", hexadecimal_code, sep="")) %>%
  filter(metric=="pd") %>%
ggplot(aes(x = factor(ano), y = value)) +
geom_boxplot(colour = "grey50") +
geom_point(aes(colour = classe_descricao)) +
scale_color_manual("classe", values = classe_cores) +
scale_size(guide = "none") +
facet_grid(raio_km~type_class) +
labs(title = "MapBiomas cobertura da terra",
subtitle = "Entorno do Garimpo do Lorenço 1985-2020",
y = "Densidade de manchas (número por 100 hectares)",
x = "Ano") +
theme(legend.position="bottom",
      legend.title = element_text(size = 3),
      legend.text = element_text(size = 3),
      legend.key.size = unit(0.1, "lines")) +
guides(fill = guide_legend(nrow = 4)) -> fig_pd

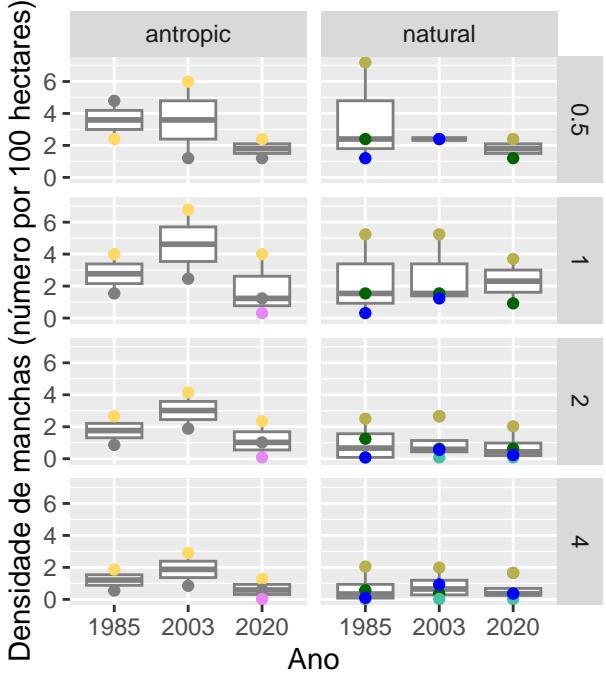
grid.arrange(fig_pland, fig_pd, nrow=1)

```

**MapBiomas cobertura da terra**  
Entorno do Garimpo do Lorenço 1985–2020



**MapBiomas cobertura da terra**  
Entorno do Garimpo do Lorenço 1985–2020



### 3.11 Conclusões e próximos passos

Os resultados apresentados na figura anterior não segam os resultados esperados que a cobertura de classes antrópicos ia aumentar ao longo do tempo. Para entender melhor os padões, precisamos:

- Verificar padrões nas outras metricas calculados
- Verificar padrões com mais anos
- Verificar padrões usando poligonos/pontos dos processos de mineração (dados no SIGMINE <https://www.gov.br/anm/pt-br> e [https://app.anm.gov.br/dadosabertos/SIGMINE/PROCESSOS\\_MINERARIOS/](https://app.anm.gov.br/dadosabertos/SIGMINE/PROCESSOS_MINERARIOS/))

Alem disso, seria relevante buscar complementar os dados de MapBiomass com uma classificação supervisionado usando imagens de Sentinel-2 (exemplo com QGIS Semi-Automatic Classification plugin aqui: <https://fromgistsors.blogspot.com/2016/09/basic-tutorial-2.html>). Assim para aumentar a precisão dos resultados.

Uma forma alternativa para visualização as mudanças entre anos seria um diagrama “Sankey”/“Alluvial”. Como exemplo, veja figura 3 no artigo “Rapid land use conversion in the Cerrado has affected water transparency in a hotspot of ecotourism, Bonito, Brazil” <https://doi.org/10.1177/19400829221127087>.

Prata River Basin LULCC

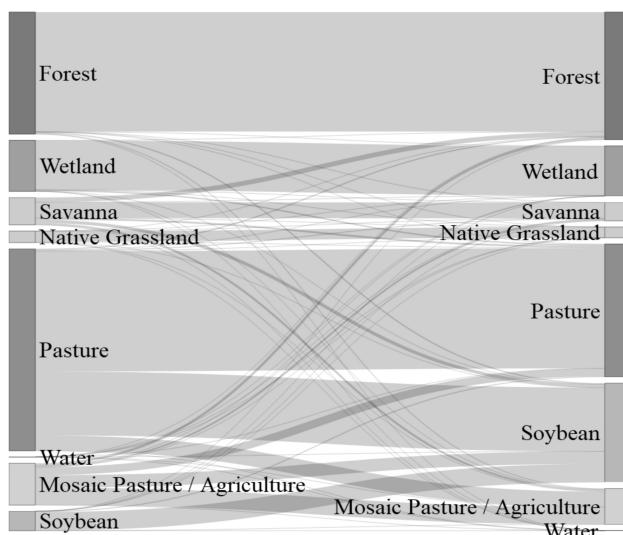


Figura 3.1: Diagrama Sankey mostrando a mudança de uso da terra na bacia do rio Prata entre 2010 (lado esquerdo) e 2020 (lado direito). Fonte: Figura 3. Chiaravalloti et. al. 2022. Tropical Conservation Science. doi:10.1177/19400829221127087

No R pode fazer com o pacote “networkD3” segue tutoriais:

- <https://www.displayr.com/sankey-diagrams-r/>
- <https://epirhandbook.com/en/diagrams-and-charts.html#alluvialsankey-diagrams>
- <https://rpubs.com/droach/CPP526-codethrough>

## Part IV

# R e RStudio: Código com certeza

## 4 Pré-requisitos

Conteúdo copiado, com pequenas alterações e atualizações de [Capítulo 3 Pré-requisitos](#), do livro [Análises Ecológicas no R](#).

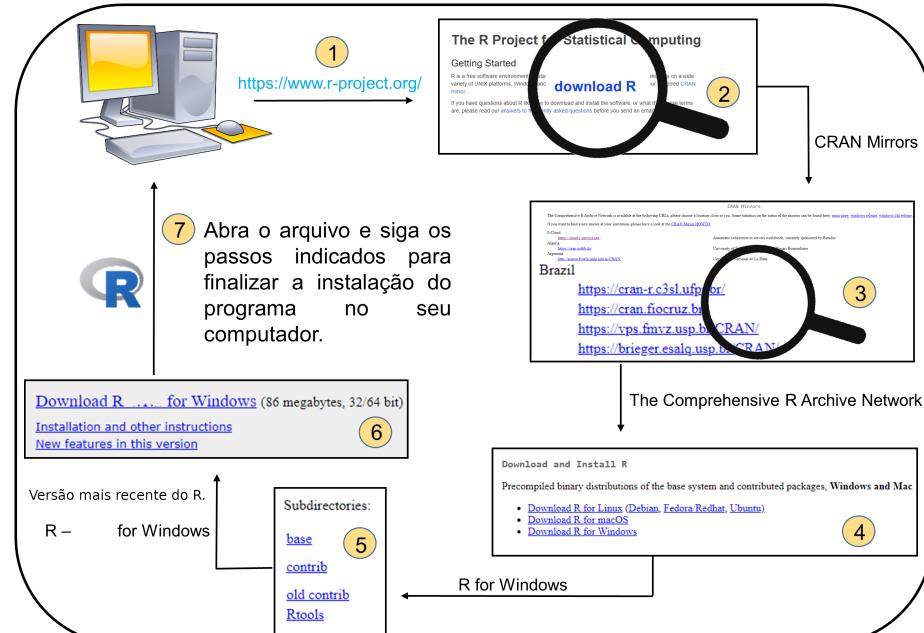
### 4.1 Introdução

O objetivo deste capítulo é informar como fazer a instalação dos Programas R e RStudio, além de descrever os pacotes e dados necessários para reproduzir os exemplos do livro.

### 4.2 Instalação do R

Abaixo descrevemos os sete passos necessários para a instalação do programa R no seu computador. Para Windows, use link <https://cran.r-project.org/bin/windows/base/> e vai para passo 6 :

1. Para começarmos a trabalhar com o R é necessário baixá-lo na página do R Project. Então, acesse esse site <http://www.r-project.org>
2. Clique no link **download R**
3. Na página *CRAN Mirros (Comprehensive R Archive Network)*, escolha uma das páginas espelho do Brasil mais próxima de você para baixar o programa
4. Escolha agora o sistema operacional do seu computador (passos adicionais existem para diferentes distribuições Linux ou MacOS). Aqui faremos o exemplo com o Windows
5. Clique em **base** para finalmente chegar à página de download com a versão mais recente do R
6. Clique no arquivo **Download R (versão mais recente) for Windows** que será instalado no seu computador
7. Abra o arquivo que foi baixado no seu computador e siga os passos indicados para finalizar a instalação do programa R



Fonte das figuras:  
imagem computador [https://pt.wikipedia.org/wiki/Computador\\_pessoal](https://pt.wikipedia.org/wiki/Computador_pessoal)  
imagem da lupa <https://openclipart.org/detail/185356/magnifier>

Figura 4.1: Esquema ilustrativo demonstrando os passos necessários para instalação do programa R no computador.

Importante

Para o Sistema Operacional (SO) Windows, alguns pacotes são dependentes da instalação separada do [Rtools40](#). Da mesma forma, GNU/Linux e MacOS também possuem dependências de outras bibliotecas para pacotes específicos, mas que não abordaremos aqui. Essas informações de dependência geralmente são retornadas como erros e você pode procurar ajuda em fóruns específicos.

### 4.3 Instalação do RStudio

O RStudio possui algumas características que o tornam popular: várias janelas de visualização, marcação e preenchimento automático do script, integração com controle de versão, dentre outras funcionalidades.

Abaixo descrevemos os cinco passos necessários para a instalação do RStudio no seu computador:

1. Para fazer o download do RStudio, acessamos o site <https://posit.co/download/rstudio-desktop/>,
2. Clique em **download...**,
3. Abra o arquivo que foi baixado no seu computador e siga os passos indicados para finalizar a instalação do programa RStudio.



Fonte das figuras: imagem computador [https://pt.wikipedia.org/wiki/Computador\\_pessoal](https://pt.wikipedia.org/wiki/Computador_pessoal)

Figura 4.2: Esquema ilustrativo demonstrando os passos necessários para instalação do programa RStudio no computador.

### 4.4 Versão do R

Todas os códigos, pacotes e análises disponibilizados no livro foram realizados no Programa R versão 4.3.2 (10-12-2023).

### 4.5 Pacotes

Descrevemos no Capítulo 5 o que são e como instalar os pacotes para realizar as análises estatísticas no R.

Importante

Criamos o pacote [eprdados](#) que contém todas as informações e dados utilizados neste livro. Assim, recomendamos que você instale e carregue este pacote no início de cada capítulo para ter acesso aos dados necessários para executar as funções no R.

Para a instalação do pacote `eprdados` no macOS, você precisará ter instalado o programa XCode que pode ser baixado [aqui](#). Este programa é disponibilizado gratuitamente pela Apple e é necessário para compilar quaisquer programas distribuídos em código fonte (ou seja, sem um binário). Após instalar esse programa e o pacote `devtools`, você poderá instalar o `eprdados` utilizando as instruções abaixo.

Abaixo, listamos todos os pacotes utilizados no livro. Você pode instalar os pacotes agora ou esperar para instalá-los quando ler o Capítulo 5 e entender o que são as funções `install.packages()`, `library()` e `install_github()`. Para fazer a instalação, você vai precisar estar conectado à internet.

```
install.packages(c("ade4", "adespatial", "ape", "bbmle", "betapart", "BiodiversityR", "car", "cati", "da
```

Diferente dos pacotes anteriores que são baixados do CRAN, alguns pacotes são baixados do GitHub dos pesquisadores responsáveis pelos pacotes. [GitHub](#) é um repositório remoto de códigos que permite controle de versão, muito utilizado por desenvolvedores e programadores. Nestes casos, precisamos carregar o pacote `devtools` para acessar a função `install_github`. Durante as instalações destes pacotes, algumas vezes o R irá pedir para você digitar um número indicando os pacotes que você deseja fazer update. Neste caso, digite 1 para indicar que ele deve atualizar os pacotes dependentes antes de instalar os pacotes requeridos.

```
library(devtools)
install_github("darrennorris/eprdados")
install_github("ropensci/rnaturalearthhires")
```

## 4.6 Dados

A maioria dos exemplos do livro utilizam dados reais extraídos de artigos científicos que já foram publicados ou dados que foram coletados por um dos autores deste livro. Todos os dados, publicados ou simulados, estão disponíveis no pacote `eprdados`. Além disso, em cada capítulo fazemos uma breve descrição dos dados para facilitar a compreensão sobre como essas variáveis estão relacionadas com as perguntas do exemplo.

## 5 Introdução ao R

Conteúdo copiado, com pequenas alterações e atualizações de [Capítulo 4 Introdução ao R](#), do livro [Análises Ecológicas no R](#).

### Pré-requisitos do capítulo

Pacotes e dados que serão utilizados neste capítulo.

```
## Pacotes
library(palmerpenguins)

## Dados necessários
pinguins <- palmerpenguins::penguins
```

#### 5.1 Contextualização

O objetivo deste capítulo é apresentar os aspectos básicos da linguagem R para a realização dos principais passos para a manipulação, visualização e análise de dados. Abordaremos aqui as questões básicas sobre a linguagem R, como: i) R e RStudio, ii) funcionamento da linguagem, iii) estrutura e manipulação de objetos, iv) exercícios e v) principais livros e material para se aprofundar nos seus estudos.

Todo processo de aprendizagem torna-se mais efetivo quando a teoria é combinada com a prática. Assim, recomendamos fortemente que você, leitor(a) acompanhe os códigos e exercícios deste livro, ao mesmo tempo que os executa em seu computador e não só os leia passivamente. Além disso, se você tiver seus próprios dados é muito importante tentar executar e/ou replicar as análises e/ou gráficos. Por motivos de espaço, não abordaremos todas as questões relacionadas ao uso da linguagem R neste capítulo. Logo, aconselhamos que você consulte o material sugerido no final do capítulo para se aprofundar.

Este capítulo, na maioria das vezes, pode desestimular as pessoas que estão iniciando, uma vez que o mesmo não apresenta os códigos para realizar as análises estatísticas. Contudo, ele é essencial para o entendimento e interpretação do que está sendo informado nas linhas de código, além de facilitar a manipulação dos dados antes de realizar as análises estatísticas. Você perceberá que não usará este capítulo para fazer as análises, mas voltará aqui diversas vezes para relembrar qual é o código ou o que significa determinada expressão ou função usada nos próximos capítulos.

#### 5.2 R e RStudio

Com o R é possível manipular, analisar e visualizar dados, além de escrever desde pequenas linhas de códigos até programas inteiros. O R é a versão em código aberto de uma linguagem de programação chamada de S, criada por John M. Chambers (Stanford University, CA, EUA) nos anos 1980 no Bell Labs. No final dos anos 1990, Robert Gentleman e Ross Ihaka (ambos da Universidade de Auckland, Nova Zelândia), iniciaram o desenvolvimento da versão livre da linguagem S, a linguagem R, com o seguinte histórico: Desenvolvimento (1997-2000), Versão 1 (2000-2004), Versão 2 (2004-2013), Versão 3 (2013-2020) e Versão 4 (2020). Atualmente a linguagem R é mantida por uma rede de colaboradores denominada *R Core Team*. A origem do nome R é desconhecida, mas reza a lenda que ao lançarem o nome da linguagem os autores se valeram da letra que vinha antes do S, uma vez que a linguagem R foi baseada nela e utilizaram a letra “R”. Outra história conta que pelo fato do nome dos dois autores iniciarem por “R”, batizaram a linguagem com essa letra, vai saber.

Um aspecto digno de nota é que a linguagem R é uma linguagem de programação interpretada, assim como o [Python](#). Isso a faz ser mais fácil de ser utilizada, pois processa linhas de código e as transforma em linguagem de máquina (código binário que o computador efetivamente lê), apesar desse fato diminuir a velocidade de processamento.

Para começarmos a trabalhar com o R é necessário baixá-lo na página do **R Project**. Os detalhes de instalação são apresentados no Capítulo 4. Reserve um tempo para explorar esta página do R-Project

(<https://www.r-project.org/>). Existem vários livros dedicados a diversos assuntos baseados no R. Além disso, estão disponíveis manuais em diversas línguas para serem baixados gratuitamente.

Como o R é um software livre, não existe a possibilidade de o usuário entrar em contato com um serviço de suporte de usuários. Ao invés disso, existem várias listas de e-mails que fornecem suporte à [comunidade de usuários](#). Nós, particularmente, recomendamos o ingresso nas seguintes listas: R-help, R-sig-ecolog, [R-br](#) e [discourse.curso-r](#). Os dois últimos grupos reúnem pessoas usuárias brasileiras do programa R.

Apesar de podermos utilizar o R com o IDE (Ambiente de Desenvolvimento Integrado - *Integrated Development Environment*) RGui que vem com a instalação da linguagem R para usuários Windows ou no próprio terminal para usuários Linux e MacOS, existem alguns IDEs específicos para facilitar nosso uso dessa linguagem.

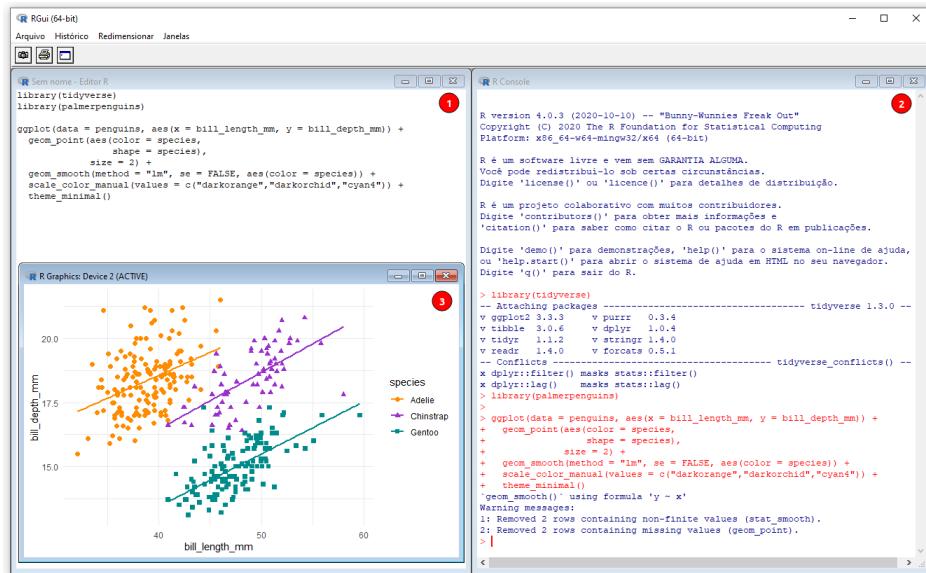


Figura 5.1: Interface do RGui. Os números indicam: (1) R Script, (2) R Console, e (3) R Graphics.

Dessa forma, nós utilizamos o IDE RStudio, e assumimos que você que está lendo fará o mesmo.

O RStudio permite diversas personalizações, grande parte delas contidas em **Tools > Global options**. Incentivamos as leitoras e leitores a “fuçar” com certa dose de cuidado, nas opções para personalização. Dentre essas mudanças, destacamos três:

1. **Tools > Global options > Appearance > Editor theme:** para escolher um tema para seu RStudio
2. **Tools > Global options > Code > [X] Soft-wrap R source files:** com essa opção habilitada, quando escrevemos comentários longos ou mudamos a largura da janela que estamos trabalhando, todo o texto e o código se ajustam a janela automaticamente
3. **Tools > Global options > Code > Display > [X] Show Margins e Margin column (80):** com essa opção habilitada e para esse valor (80), uma linha vertical irá aparecer no script marcando 80 caracteres, um comprimento máximo recomendado para padronização dos scripts

Importante

Para evitar possíveis erros é importante instalar primeiro o software da linguagem R e depois o IDE RStudio.

O RStudio permite também trabalhar com projetos. **Projeto do RStudio** é uma forma de organizar os arquivos de scripts e dados dentro de um diretório, facilitando o compartilhamento de fluxo de análises de dados e aumentando assim a reprodutibilidade. Podemos criar um Projeto do RStudio indo em **File > New Project** ou no ícone de cubo azul escuro que possui um R dentro com um círculo verde com um sinal de + na parte superior esquerda ou ainda no canto superior direito que possui cubo azul escrito **Project** que serve para gerenciar os projetos e depois em **New Project**. Depois de escolher uma dessas opções, uma janela se abrirá onde escolhemos uma das três opções: i) New Directory (para criar um diretório novo com

diversas opções), ii) Existing Directory (para escolher um diretório já existente) e iii) Version Control (para criar um projeto que será versionado pelo git ou Subversion).

### 5.3 Funcionamento da linguagem R

Nesta seção, veremos os principais conceitos para entender como a linguagem R funciona ou como geralmente utilizamos o IDE RStudio no dia a dia, para executar nossas rotinas utilizando a linguagem R. Veremos então: i) console, ii) script, iii) operadores, iv) objetos, v) funções, vi) pacotes, vii) ajuda (*help*), viii) ambiente (*environment/workspace*), ix) citações e x) principais erros.

Antes de iniciarmos o uso do R pelo RStudio é fundamental entendermos alguns pontos sobre as janelas e o funcionamento delas no RStudio.

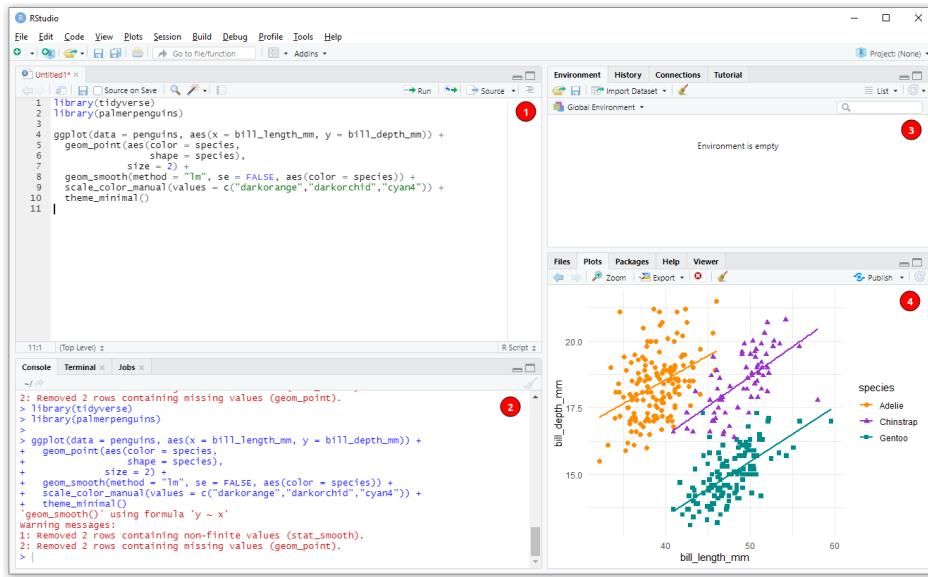


Figura 5.2: Interface do RStudio. Os números indicam: (1) janela com abas de Script, R Markdown, dentre outras; (2) janela com abas de Console, Terminal e Jobs; (3) janela com abas de Environment, History, Conections e Tutorial; e (4) janela com abas de Files, Plots, Packages, Help e Viewer.

Detalhando algumas dessas janelas e abas, temos:

- **Console:** painel onde os códigos são rodados e vemos as saídas
- **Editor/Script:** painel onde escrevemos nossos códigos em R, R Markdown ou outro formato
- **Environment:** painel com todos os objetos criados na sessão
- **History:** painel com o histórico dos códigos rodados
- **Files:** painel que mostra os arquivos no diretório de trabalho
- **Plots:** painel onde os gráficos são apresentados
- **Packages:** painel que lista os pacotes
- **Help:** painel onde a documentação das funções é exibida

No RStudio, alguns atalhos são fundamentais para aumentar nossa produtividade:

- **F1:** abre o painel de *Help* quando digitado em cima do nome de uma função
- **Ctrl + Enter:** roda a linha de código selecionada no script
- **Ctrl + Shift + N:** abre um novo script
- **Ctrl + S:** salva um script
- **Ctrl + Z:** desfaz uma operação
- **Ctrl + Shift + Z:** refaz uma operação
- **Alt + -:** insere um sinal de atribuição (<-)

- **Ctrl + Shift + M**: insere um operador pipe (%>%)
- **Ctrl + Shift + C**: comenta uma linha no script - insere um (#)
- **Ctrl + I**: indenta (recuo inicial das linhas) as linhas
- **Ctrl + Shift + A**: reformata o código
- **Ctrl + Shift + R**: insere uma sessão (# -----)
- **Ctrl + Shift + H**: abre uma janela para selecionar o diretório de trabalho
- **Ctrl + Shift + F10**: reinicia o console
- **Ctrl + L**: limpa os códigos do console
- **Alt + Shift + K**: abre uma janela com todos os atalhos disponíveis

### 5.3.1 Console

O console é onde a versão da linguagem R instalada é carregada para executar os códigos da linguagem R (Figura 5.2 janela 2). Na janela do console aparecerá o símbolo >, seguido de uma barra vertical | que fica piscando (cursor), onde digitamos ou enviamos nossos códigos do script. Podemos fazer um pequeno exercício: vamos digitar 10 + 2, seguido da tecla Enter para que essa operação seja executada.

```
10 + 2
```

```
## [1] 12
```

O resultado retorna o valor 12, precedido de um valor entre colchetes. Esses colchetes demonstram a posição do elemento numa sequência de valores. Se fizermos essa outra operação 1:42, o R vai criar uma sequência unitária de valores de 1 a 42. A depender da largura da janela do console, vai aparecer um número diferente entre colchetes indicando sua posição na sequência: antes do número 1 vai aparecer o [1], depois quando a sequência for quebrada, vai aparecer o número correspondente da posição do elemento, por exemplo, [37].

```
1:42
```

```
## [1] 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] 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42
```

Podemos ver o histórico dos códigos executados no console na aba **History** (Figura 5.2 janela 3).

### 5.3.2 Scripts

Scripts são arquivos de texto simples, criados com a extensão (terminação) .R (Figura 5.2 janela 1). Para criar um script, basta ir em **File > New File > R Script**, ou clicar no ícone com uma folha branca e um círculo verde com um sinal de +, logo abaixo de **File**, ou ainda usando o atalho **Ctrl + Shift + N**.

Uma vez escrito os códigos no script podemos rodar esses códigos de duas formas: i) todo o script de uma vez, clicando em *Source* (que fina no canto superior direito da aba script) ou usando o atalho **Ctrl + Shift + Enter**; ou ii) apenas a linha onde o cursor estiver posicionado, independentemente de sua posição naquela linha, clicando em **Run** ou usando o atalho **Ctrl + Enter**.

Devemos sempre salvar nossos scripts, tomando por via de regra: primeiro criar o arquivo e depois ir salvando nesse mesmo arquivo a cada passo de desenvolvimento das análises (não é raro o RStudio fechar sozinho e você perder algum tempo de trabalho). Há diversos motivos para se criar um script: continuar o desenvolvimento desse script em outro momento ou em outro computador, preservar trabalhos passados, ou ainda compartilhar seus códigos com outras pessoas. Para criar ou salvar um script basta ir em **File > Save**, escolher um diretório e nome para o script e salvá-lo. Podemos ainda utilizar o atalho **Ctrl + S**.

Em relação aos scripts, há ainda os comentários, representados pelos símbolos # (hash), #' (hash-linha) e #> (hash-maior). A diferença entre eles é que para o segundo e terceiro, quando pressionamos a tecla **Enter** o comentário #' e #> são inseridos automaticamente na linha seguinte. Linhas de códigos do script contendo comentários em seu início não são lidos pelo console do R. Se o comentário estiver no final da linha, essa linha de código ainda será lida. Os comentários são utilizados geralmente para: i) descrever informações sobre dados ou funções e/ou ii) suprimir linhas de código.

É interessante ter no início de cada script um cabeçalho identificando o objetivo ou análise, autor e data para facilitar o compartilhamento e reproduzibilidade. Os comentários podem ser inseridos ou retirados das linhas com o atalho **Ctrl + Shift + C**.

```
#' ---
#' Título: Capítulo 04 - Introdução ao R
#' Autor: Maurício Vancine
#' Data: 11-11-2021
#' ---
```

Além disso, podemos usar comentários para adicionar informações sobre os códigos.

```
## Comentários
# O R não lê a linha do código depois do # (hash).
42 # Essas palavras não são executadas, apenas o 42, a resposta para questão fundamental da vida, o uni
## [1] 42
```

Por fim, outro ponto fundamental é ter boas práticas de estilo de código. Quanto mais organizado e padronizado estiver seus scripts, mais fácil de entendê-los e de procurar possíveis erros. Existem dois guias de boas práticas para adequar seus scripts: [Hadley Wickham](#) e [Google](#). Para simplificar a vida temos o pacote `styler` (<https://styler.r-lib.org/>), que serve para adequar o código.

Ainda em relação aos scripts, temos os *Code Snippets* (Fragments de código), que são macros de texto usadas para inserir rapidamente fragmentos comuns de código. Por exemplo, o snippet `fun` insere uma definição de função R. Para mais detalhes, ler o artigo do RStudio [Code Snippets](#).

```
# fun {snippet}
fun
name <- function(variables) {

}
```

Uma aplicação bem interessante dos *Code Snippets* no script é o `ts`. Basta digitar esse código e em seguida pressionar a tecla `Tab` para inserir rapidamente a data e horário atuais no script em forma de comentário.

```
# ts {snippet}
# Thu Nov 11 18:19:26 2021 -----
```

### 5.3.3 Operadores

No R, podemos agrupar os operadores em cinco tipos: aritméticos, relacionais, lógicos, atribuição e diversos. Grande parte deles são descritos na Tabela “Principais operadores no R.”.

Principais operadores no R.

Operador

Tipo

Descrição

- Aritmético

Adição

- Aritmético

Subtração

- \* Aritmético

Multiplicação

/  
Aritmético  
Divisão  
%%  
Aritmético  
Resto da divisão  
%/%  
Aritmético  
Divisão inteira  
^ ou \*\*  
Aritmético  
Expoente  
>  
Relacional  
Maior  
<  
Relacional  
Menor  
>=  
Relacional  
Maior ou igual  
<=  
Relacional  
Menor ou igual  
==  
Relacional  
Igualdade  
!=  
Relacional  
Diferença  
!  
Lógico  
Lógico NÃO  
&  
Lógico  
Lógico elementar E

&&  
Lógico  
Lógico E  
||  
Lógico  
Lógico OU  
<- ou =  
Atribuição  
Atribuição à esquerda  
«-  
Atribuição  
Super atribuição à esquerda  
->  
Atribuição  
Atribuição à direita  
-»  
Atribuição  
Super atribuição à direita  
:  
Diversos  
Sequência unitária  
%in%  
Diversos  
Elementos que pertencem a um vetor  
%\*%  
Diversos  
Multiplicar matriz com sua transposta  
%>%  
Diversos  
Pipe (pacote magrittr)  
|>  
Diversos  
Pipe (R base nativo)  
%-%  
Diversos  
Intervalo de datas (pacote lubridate)

Como exemplo, podemos fazer operações simples usando os operadores aritméticos.

```
## Operações aritméticas  
10 + 2 # adição
```

```
## [1] 12  
10 * 2 # multiplicação
```

```
## [1] 20
```

Precisamos ficar atentos à prioridade dos operadores aritméticos:

PRIORITÁRIO () > ^ > \* ou / > + ou - NÃO PRIORITÁRIO

Veja no exemplo abaixo como o uso dos parênteses muda o resultado.

```
## Sem especificar a ordem  
# Segue a ordem dos operadores.  
1 * 2 + 2 / 2 ^ 2
```

```
## [1] 2.5  
## Especificando a ordem  
# Segue a ordem dos parenteses.  
((1 * 2) + (2 / 2)) ^ 2
```

```
## [1] 9
```

#### 5.3.4 Objetos

Objetos são palavras às quais são atribuídos dados. A atribuição possibilita a manipulação de dados ou armazenamento dos resultados de análises. Utilizaremos os símbolos < (menor), seguido de - (menos), sem espaço, dessa forma <- . Também podemos utilizar o símbolo de igual (=), mas não recomendamos, por não fazer parte das boas práticas de escrita de códigos em R. Podemos inserir essa combinação de símbolos com o atalho Alt + -. Para demonstrar, vamos atribuir o valor 10 à palavra obj\_10 , e chamar esse objeto novamente para verificar seu conteúdo.

```
## Atribuição - símbolo (<-)  
obj_10 <- 10  
obj_10
```

```
## [1] 10
```

Importante

Recomendamos sempre verificar o conteúdo dos objetos chamando-os novamente para confirmar se a atribuição foi realizada corretamente e se o conteúdo corresponde à operação realizada.

Todos os objetos criados numa sessão do R ficam listados na aba Environment . Além disso, o RStudio possui a função autocomplete , ou seja, podemos digitar as primeiras letras de um objeto (ou função) e em seguida apertar Tab para que o RStudio liste tudo que começar com essas letras.

Dois pontos importantes sobre atribuições: primeiro, o R sobrescreve os valores dos objetos com o mesmo nome, deixando o objeto com o valor da última atribuição.

```
## Sobrescreve o valor dos objetos  
obj <- 100  
obj
```

```
## [1] 100
```

```
## O objeto 'obj' agora vale 2
obj <- 2
obj
```

```
## [1] 2
```

Segundo, o R tem limitações ao nomear objetos:

- nome de objetos só podem começar por letras (a-z ou A-Z) ou pontos (.)
- nome de objetos só podem conter letras (a-z ou A-Z), números (0-9), underscores (\_) ou pontos (.)
- R é *case-sensitive*, i.e., ele reconhece letras maiúsculas como diferentes de letras minúsculas. Assim, um objeto chamado “resposta” é diferente do objeto “RESPOSTA”
- devemos evitar acentos ou cedilha (ç) para facilitar a memorização dos objetos e também para evitar erros de codificação (*encoding*) de caracteres
- nomes de objetos não podem ser iguais a nomes especiais, reservados para programação (**break**, **else**, **FALSE**, **for**, **function**, **if**, **Inf**, **NaN**, **next**, **repeat**, **return**, **TRUE**, **while**)

Podemos ainda utilizar objetos para fazer operações e criar objetos. Isso pode parecer um pouco confuso para os iniciantes, mas é fundamental aprender essa lógica para passar para os próximos passos.

```
## Definir dois objetos
```

```
val1 <- 10
```

```
val2 <- 2
```

```
## Operações com objetos e atribuição
```

```
adi <- val1 + val2
```

```
adi
```

```
## [1] 12
```

### 5.3.5 Funções

Funções são códigos preparados para realizar uma tarefa específica de modo simples. Outra forma de entender uma função é: códigos que realizam operações em argumentos. Devemos retomar ao conceito do ensino médio de funções: os dados de entrada são argumentos e a função realizará alguma operação para modificar esses dados de entrada. A estrutura de uma função é muito similar à sintaxe usada em planilhas eletrônicas, sendo composta por:

```
nome_da_função(argumento1, argumento2, ...)
```

1. **Nome da função**: remete ao que ela faz
2. **Parênteses**: limitam a função
3. **Argumentos**: valores, parâmetros ou expressões onde a função atuará
4. **Vírgulas**: separam os argumentos

Os argumentos de uma função podem ser de dois tipos:

1. **Valores ou objetos**: a função alterará os valores em si ou os valores atribuídos aos objetos
2. **Parâmetros**: valores fixos que informam um método ou a realização de uma operação. Informa-se o nome desse argumento, seguido de “=” e um número, texto ou TRUE ou FALSE

Alguns exemplos de argumentos como valores ou objetos.

```
## Funções - argumentos como valores
sum(10, 2)
```

```
## [1] 12
```

```
## Funções - argumentos como objetos
sum(va1, va2)
```

```
## [1] 12
```

Vamos ver agora alguns exemplos de argumentos usados como parâmetros. Note que apesar do valor do argumento ser o mesmo (10), seu efeito no resultado da função `rep()` muda drasticamente. Aqui também é importante destacar um ponto: i) podemos informar os argumentos sequencialmente, sem explicitar seus nomes, ou ii) independente da ordem, mas explicitando seus nomes. Entretanto, como no exemplo abaixo, devemos informar o nome do argumento (i.e., parâmetro), para que seu efeito seja o que desejamos.

```
## Funções - argumentos como parâmetros
```

```
## Repetição - repete todos os elementos
```

```
rep(x = 1:5, times = 10)
```

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

```
## [38] 3 4 5 1 2 3 4 5 1 2 3 4 5
```

```
## Repetição - repete cada um dos elementos
```

```
rep(x = 1:5, each = 10)
```

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

```
## [38] 4 4 4 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
```

Um ponto fundamental e que deve ser entendido é o fluxo de atribuições do resultado da operação de funções a novos objetos. No desenvolvimento de qualquer script na linguagem R, grande parte da estrutura do mesmo será dessa forma: atribuição de dados a objetos > operações com funções > atribuição dos resultados a novos objetos > operações com funções desses novos objetos > atribuição dos resultados a novos objetos. Ao entender esse funcionamento, começamos a entender como devemos pensar na organização do nosso script para montar as análises que precisamos.

```
## Atribuição dos resultados
```

```
## Repetição
```

```
rep_times <- rep(1:5, times = 10)
```

```
rep_times
```

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

```
## [38] 3 4 5 1 2 3 4 5 1 2 3 4 5
```

```
## Somar e atribuir
```

```
rep_times_soma <- sum(rep_times)
```

```
rep_times_soma
```

```
## [1] 150
```

```
## Raiz e atribuir
```

```
rep_times_soma_raiz <- sqrt(rep_times_soma)
```

```
rep_times_soma_raiz
```

```
## [1] 12.24745
```

Por fim, é fundamental também entender a origem das funções que usamos no R. Todas as funções são advindas de pacotes. Esses pacotes possuem duas origens.

1. pacotes já instalados por padrão e que são carregados quando abrimos o R (*R Base*)
2. pacotes que instalamos e carregamos com funções

### 5.3.6 Pacotes

Pacotes são conjuntos extras de funções para executar tarefas específicas, além dos pacotes instalados no *R Base*. Existe literalmente milhares de pacotes (~19,000 enquanto estamos escrevendo esse livro) para as mais diversas tarefas: estatística, ecologia, geografia, sensoriamento remoto, econometria, ciências sociais, gráficos, *machine learning*, etc. Podemos verificar este vasto conjunto de pacotes pelo [link](#) que lista por

nome os pacotes oficiais, ou seja, que passaram pelo crivo do **CRAN**. Existem ainda muito mais pacotes em desenvolvimento, geralmente disponibilizados em repositórios do **GitHub** ou **GitLab**.

Podemos listar esses pacotes disponíveis no **CRAN** com esse código.

```
## Número atual de pacotes no CRAN: 20192 em 15/12/2023
nrow(available.packages())
```

Primeiramente, com uma sessão do R sem carregar nenhum pacote extra, podemos verificar pacotes carregados pelo *R Base* utilizando a função `search()`.

```
## Verificar pacotes carregados
search()
```

Podemos ainda verificar todos pacotes instalados em nosso computador com a função `library()`.

```
## Verificar pacotes instalados
library()
```

No R, quando tratamos de pacotes, devemos destacar a diferença de dois conceitos: instalar um pacote e carregar um pacote. A instalação de pacotes possui algumas características:

- Instala-se um pacote apenas uma vez
- Precisamos estar conectados à internet
- O nome do pacote precisa estar entre aspas na função de instalação
- Função (CRAN): `install.packages()`

Vamos instalar o pacote `vegan` diretamente do CRAN, que possui funções para realizar uma série de análise em ecologia. Para isso, podemos ir em **Tools > Install Packages...**, ou ir na aba **Packages**, procurar o pacote e simplesmente clicar em “Install”. Podemos ainda utilizar a função `install.packages()`.

```
## Instalar pacotes
install.packages("vegan")
```

Podemos conferir em que diretórios um pacote será instalado com a função `.libPaths()`.

```
## Diretórios de instalação dos pacotes
.libPaths()
```

```
## [1] "C:/Users/user/AppData/Local/R/win-library/4.3"
## [2] "C:/Program Files/R/R-4.3.2/library"
```

Importante

Uma vez instalado um pacote, não há necessidade de instalá-lo novamente. Entretanto, todas às vezes que iniciarmos uma sessão no R, precisamos carregar os pacotes com as funções que precisamos utilizar.

O carregamento de pacotes possui algumas características:

- Carrega-se o pacote toda vez que se abre uma nova sessão do R
- Não precisamos estar conectados à internet
- O nome do pacote não precisa estar entre aspas na função de carregamento
- Funções: `library()` ou `require()`

Vamos carregar o pacote `vegan` que instalamos anteriormente. Podemos ir na aba **Packages** e assinalar o pacote que queremos carregar ou utilizar a função `library()`.

```
## Carregar pacotes
library(vegan)
```

Como dissemos, alguns pacotes em desenvolvimento encontram-se disponíveis em repositórios como por exemplo: [GitHub](#), [GitLab](#) e [Bioconductor](#). Para instalar pacotes do GitHub, por exemplo, precisamos instalar

e carregar o pacote `devtools`. Para funcionar, deve instalar “RTools” antes (<https://cran.r-project.org/bin/windows/Rtools/>) .

```
## Instalar pacote devtools
install.packages("devtools")

## Carregar pacote devtools
library(devtools)
```

Uma vez instalado e carregado esse pacote, podemos instalar o pacote do GitHub, utilizando a função `devtools::install_github()`. Precisamos atentar para usar essa forma “`nome_usuario/nome_repositorio`”, retirados do link do repositório de interesse. Como exemplo, podemos instalar o pacote `eprdados` do repositório do GitHub `darrennorris/eprdados` e depois utilizar a função `library()` para carregá-lo. Para funcionar, deve instalar “RTools” antes (<https://cran.r-project.org/bin/windows/Rtools/>) .

```
## Instalar pacote do github
devtools::install_github("darrennorris/eprdados")

## Carregar pacote do github
library("eprdados")
```

Podemos ver a descrição de um pacote com a função `packageDescription()`.

```
## Descrição de um pacote
packageDescription("vegan")

## Package: vegan
## Title: Community Ecology Package
## Version: 2.6-4
## Authors@R: c(person("Jari", "Oksanen", role=c("aut","cre"),
##                 email="jhoksane@gmail.com"), person("Gavin L.", "Simpson",
##                 role="aut", email="ucfagls@gmail.com"), person("F.
##                 Guillaume", "Blanchet", role="aut"), person("Roeland",
##                 "Kindt", role="aut"), person("Pierre", "Legendre",
##                 role="aut"), person("Peter R.", "Minchin", role="aut"),
##                 person("R.B.", "O'Hara", role="aut"), person("Peter",
##                 "Solymos", role="aut"), person("M. Henry H.", "Stevens",
##                 role="aut"), person("Eduard", "Szoecs", role="aut"),
##                 person("Helene", "Wagner", role="aut"), person("Matt",
##                 "Barbour", role="aut"), person("Michael", "Bedward",
##                 role="aut"), person("Ben", "Bolker", role="aut"),
##                 person("Daniel", "Borcard", role="aut"), person("Gustavo",
##                 "Carvalho", role="aut"), person("Michael", "Chirico",
##                 role="aut"), person("Miquel", "De Caceres", role="aut"),
##                 person("Sebastien", "Durand", role="aut"), person("Heloisa
##                 Beatriz Antoniazi", "Evangelista", role="aut"),
##                 person("Rich", "FitzJohn", role="aut"), person("Michael",
##                 "Friendly", role="aut"), person("Brendan", "Furneaux",
##                 role="aut"), person("Geoffrey", "Hannigan", role="aut"),
##                 person("Mark O.", "Hill", role="aut"), person("Leo", "Lahti",
##                 role="aut"), person("Dan", "McGlinn", role="aut"),
##                 person("Marie-Helene", "Ouellette", role="aut"),
##                 person("Eduardo", "Ribeiro Cunha", role="aut"),
##                 person("Tyler", "Smith", role="aut"), person("Adrian",
##                 "Stier", role="aut"), person("Cajo J.F.", "Ter Braak",
##                 role="aut"), person("James", "Weedon", role="aut"))
```

```

## Depends: permute (>= 0.9-0), lattice, R (>= 3.4.0)
## Suggests: parallel, tcltk, knitr, markdown
## Imports: MASS, cluster, mgcv
## VignetteBuilder: utils, knitr
## Description: Ordination methods, diversity analysis and other
##               functions for community and vegetation ecologists.
## License: GPL-2
## BugReports: https://github.com/vegandevs/vegan/issues
## URL: https://github.com/vegandevs/vegan
## NeedsCompilation: yes
## Packaged: 2022-10-11 08:36:07 UTC; jaricksa
## Author: Jari Oksanen [aut, cre], Gavin L. Simpson [aut], F. Guillaume
##        Blanchet [aut], Roeland Kindt [aut], Pierre Legendre [aut],
##        Peter R. Minchin [aut], R.B. O'Hara [aut], Peter Solymos
##        [aut], M. Henry H. Stevens [aut], Eduard Szoeecs [aut], Helene
##        Wagner [aut], Matt Barbour [aut], Michael Bedward [aut], Ben
##        Bolker [aut], Daniel Borcard [aut], Gustavo Carvalho [aut],
##        Michael Chirico [aut], Miquel De Caceres [aut], Sebastien
##        Durand [aut], Heloisa Beatriz Antoniazi Evangelista [aut],
##        Rich FitzJohn [aut], Michael Friendly [aut], Brendan Furneaux
##        [aut], Geoffrey Hannigan [aut], Mark O. Hill [aut], Leo Lahti
##        [aut], Dan McGlinn [aut], Marie-Helene Ouellette [aut],
##        Eduardo Ribeiro Cunha [aut], Tyler Smith [aut], Adrian Stier
##        [aut], Cajo J.F. Ter Braak [aut], James Weedon [aut]
## Maintainer: Jari Oksanen <jhoksane@gmail.com>
## Repository: CRAN
## Date/Publication: 2022-10-11 12:40:02 UTC
## Built: R 4.3.2; x86_64-w64-mingw32; 2023-11-02 03:07:22 UTC; windows
## Archs: x64
##
## -- File: C:/Users/user/AppData/Local/R/win-library/4.3/vegan/Meta/package.rds

```

A maioria dos pacotes possui conjuntos de dados que podem ser acessados pela função `data()`. Esses conjuntos de dados podem ser usados para testar as funções do pacote. Se estiver com dúvida na maneira como você deve preparar a planilha para realizar uma análise específica, entre na Ajuda (*Help*) da função e veja os conjuntos de dados que estão no exemplo desta função. Como exemplo, vamos carregar os dados `dune` do pacote `vegan`, que são dados de observações de 30 espécies vegetais em 20 locais.

```

## Carregar dados de um pacote
library(vegan)
data(dune)
dune[1:6, 1:6]

##   Achimill Agrostol Airaprae Alopogeni Anthodor Bellpere
## 1      1       0       0       0       0       0
## 2      3       0       0       2       0       3
## 3      0       4       0       7       0       2
## 4      0       8       0       2       0       2
## 5      2       0       0       0       4       2
## 6      2       0       0       0       3       0

```

E um último ponto fundamental sobre pacotes, diz respeito à atualização dos mesmos. Os pacotes são atualizados com frequência, e infelizmente (ou felizmente, pois as atualizações podem oferecer algumas quebras entre pacotes), não se atualizam sozinhos. Muitas vezes, a instalação de um pacote pode depender da versão dos pacotes dependentes, e geralmente uma janela com diversas opções numéricas se abre perguntando se você quer que todos os pacotes dependentes sejam atualizados. Podemos ir na aba **Packages** e clicar em “Update”

ou usar a função `update.packages(checkBuilt = TRUE, ask = FALSE)` para atualizá-los, entretanto, essa é uma função que costuma demorar muito para terminar de ser executada.

```
## Atualização dos pacotes
update.packages(checkBuilt = TRUE, ask = FALSE)
```

Destacamos e incentivamos ainda uma prática que achamos interessante para aumentar a reprodutibilidade de nossos códigos e scripts: a de chamar as funções de pacotes carregados dessa forma `pacote::função()`. Com o uso dessa prática, deixamos claro o pacote em que a função está implementada. Esta prática é importante por que com frequência pacotes diferentes criam funções com mesmo nome, mas com características internas (argumentos) diferentes. Assim, não expressar o pacote de interesse pode gerar erros na execução de suas análises. Destacamos aqui o exemplo de como instalar pacotes do GitHub do pacote `devtools`.

```
## Pacote seguido da função implementada daquele pacote
devtools::install_github()
```

### 5.3.7 Ajuda (*Help*)

Um importante passo para melhorar a usabilidade e ter mais familiaridade com a linguagem R é aprender a usar a ajuda (*help*) de cada função. Para tanto, podemos utilizar a função `help()` ou o operador `?`, depois de ter carregado o pacote. O arquivo de ajuda do R possui tópicos, que nos auxiliam muito no entendimento dos dados de entrada, argumentos e que operações estão sendo realizadas. Abaixo descrevemos esses tópicos:

- **Description:** resumo da função
- **Usage:** como utilizar a função e quais os seus argumentos
- **Arguments:** detalha os argumentos e como os mesmos devem ser especificados
- **Details:** detalhes importantes para se usar a função
- **Value:** mostra como interpretar a saída (*output*) da função (os resultados)
- **Note:** notas gerais sobre a função
- **Authors:** autores da função
- **References:** referências bibliográficas para os métodos usados para construção da função
- **See also:** funções relacionadas
- **Examples:** exemplos do uso da função. Às vezes pode ser útil copiar esse trecho e colar no R para ver como funciona e como usar a função.

Vamos realizar um exemplo, buscando o `help` da função `aov()`, que realiza uma análise de variância.

```
## Ajuda
help(aov)
?aov
```

Além das funções, podemos buscar detalhes de um pacote específico, para uma página simples do `help` utilizando a função `help()` ou o operador `?`. Entretanto, para uma opção que ofereça uma descrição detalhada e um índice de todas as funções do pacote, podemos utilizar a função `library()`, mas agora utilizando o argumento `help`, indicando o pacote de interesse entre aspas.

```
## Ajuda do pacote
help(vegan)
?vegan

## Help detalhado
library(help = "vegan")
```

Podemos ainda procurar o nome de uma função para realizar uma análise específica utilizando a função `help.search()` com o termo que queremos em inglês e entre aspas.

```
## Procurar por funções que realizam modelos lineares
help.search("linear models")
```

Outra ferramenta de busca é a página [rseek](#), na qual é possível buscar por um termo não só nos pacotes do R, mas também em listas de emails, manuais, páginas na internet e livros sobre o programa.

### 5.3.8 Ambiente (*Environment*)

O ambiente (*environment*), como vimos, é onde os objetos criados são armazenados. É fundamental entender que um objeto é uma alocação de um pequeno espaço na memória RAM do nosso computador, onde o R armazenará um valor ou o resultado de uma função, utilizando o nome dos objetos que definimos na atribuição. Sendo assim, se fizermos a atribuição de um objeto maior que o tamanho da memória RAM do nosso computador, esse objeto não será alocado, e a atribuição não funcionará, retornando um erro. Existem opções para contornar esse tipo de limitação, mas não a abordaremos aqui. Entretanto, podemos utilizar a função `object.size()` para saber quanto espaço nosso objeto criado está alocando de memória RAM.

```
## Tamanho de um objeto
```

```
object.size(adi)
```

```
## 56 bytes
```

Podemos listar todos os objetos criados com a função `ls()` ou `objects()`.

```
## Listar todos os objetos
```

```
ls()
```

Podemos ainda remover todos os objetos criados com a função `rm()` ou `remove()`. Ou ainda fazer uma função composta para remover todos os objetos do *Environment*.

```
## Remover um objeto
```

```
rm(adi)
```

```
## Remover todos os objetos criados
```

```
rm(list = ls())
```

Quando usamos a função `ls()` agora, nenhum objeto é listado.

```
## Listar todos os objetos
```

```
ls()
```

```
## character(0)
```

Toda a vez que fechamos o R os objetos criados são apagados do **Environment**. Dessa forma, em algumas ocasiões, por exemplo, análises estatísticas que demoram um grande tempo para serem realizadas, pode ser interessante exportar alguns ou todos os objetos criados.

Para salvar todos os objetos, ou seja, todo o *Workspace*, podemos ir em **Session** → **Save Workspace As...** e escolher o nome do arquivo do *Workspace*, por exemplo, “meu\_workspace.RData”. Podemos ainda utilizar funções para essas tarefas. A função `save.image()` salva todo *Workspace* com a extensão `.RData`.

```
## Salvar todo o workspace
```

```
save.image(file = "meu_workspace.RData")
```

Depois disso, podemos fechar o RStudio tranquilamente e quando formos trabalhar novamente, podemos carregar os objetos criados indo em **Session** → **Load Workspace...** ou utilizando a função `load()`.

```
## Carregar todo o workspace
```

```
load("meu_workspace.RData")
```

Entretanto, em algumas ocasiões, não precisamos salvar todos os objetos. Dessa forma, podemos salvar apenas alguns objetos específicos usando a função `save()`, também com a extensão `.RData`.

```
## Salvar apenas um objeto
```

```
save(obj1, file = "meu_obj.RData")
```

```

## Salvar apenas um objeto
save(obj1, obj2, file = "meus_objs.RData")

## Carregar os objetos
load("meus_objs.RData")

```

Ou ainda, podemos salvar apenas um objeto com a extensão .rds. Para isso, usamos as funções `saveRDS()` e `readRDS()`, para exportar e importar esses dados, respectivamente. É importante ressaltar que nesse formato .rds, apenas um objeto é salvo por arquivo criado e que para que o objeto seja criado no *Workspace* do R, ele precisa ser lido e atribuído à um objeto.

```

## Salvar um objeto para um arquivo
saveRDS(obj, file = "meu_obj.rds")

## Carregar esse objeto
obj <- readRDS(file = "meu_obj.rds")

```

### 5.3.9 Citações

Ao utilizar o R para realizar alguma análise em nossos estudos, é fundamental a citação do mesmo. Para saber como citar o R em artigos, existe uma função denominada `citation()`, que provê um formato genérico de citação e um BibTeX para arquivos LaTeX e R Markdown.

```

## Citação do R
citation()

## To cite R in publications use:
##
##   R Core Team (2023). _R: A Language and Environment for Statistical
##   Computing_. R Foundation for Statistical Computing, Vienna,
##   Austria. <https://www.R-project.org/>.
##
## A BibTeX entry for LaTeX users is
##
##   @Manual{,
##     title = {R: A Language and Environment for Statistical Computing},
##     author = {{R Core Team}},
##     organization = {R Foundation for Statistical Computing},
##     address = {Vienna, Austria},
##     year = {2023},
##     url = {https://www.R-project.org/},
##   }
##
## We have invested a lot of time and effort in creating R, please cite
## it when using it for data analysis. See also 'citation("pkgname")'
## for citing R packages.

```

No resultado dessa função, há uma mensagem muito interessante: “See also ‘`citation("pkgname")`’ for citing R packages.”. Dessa forma, aconselhamos, sempre que possível, claro, citar também os pacotes utilizados nas análises para dar os devidos créditos aos desenvolvedores e desenvolvedoras das funções implementadas nos pacotes. Como exemplo, vamos ver como fica a citação do pacote `vegan`.

```

## Citação do pacote vegan
citation("vegan")

## To cite package 'vegan' in publications use:

```

```

## 
##   Oksanen J, Simpson G, Blanchet F, Kindt R, Legendre P, Minchin P,
##   O'Hara R, Solymos P, Stevens M, Szoecs E, Wagner H, Barbour M,
##   Bedward M, Bolker B, Borcard D, Carvalho G, Chirico M, De Caceres
##   M, Durand S, Evangelista H, FitzJohn R, Friendly M, Furneaux B,
##   Hannigan G, Hill M, Lahti L, McGlinn D, Ouellette M, Ribeiro Cunha
##   E, Smith T, Stier A, Ter Braak C, Weedon J (2022). _vegan:
##   Community Ecology Package_. R package version 2.6-4,
##   <https://CRAN.R-project.org/package=vegan>.
## 
## A BibTeX entry for LaTeX users is
##
## @Manual{,
##   title = {vegan: Community Ecology Package},
##   author = {Jari Oksanen and Gavin L. Simpson and F. Guillaume Blanchet and Roeland Kindt and Pier
##   year = {2022},
##   note = {R package version 2.6-4},
##   url = {https://CRAN.R-project.org/package=vegan},
## }

```

Podemos ainda utilizar a função `write_bib()` do pacote `knitr` para exportar a citação do pacote no formato `.bib`.

```

## Exportar uma citação em formato .bib
knitr:::write_bib("vegan", file = "vegan_ex.bib")

```

### 5.3.10 Principais erros de iniciantes



Errar quando se está começando a usar o R é muito comum e faz parte do aprendizado. Entretanto, os erros nunca devem ser encarados como uma forma de desestímulo, mas sim como um desafio para continuar tentando. Todos nós, autores deste livro inclusive, e provavelmente usuários mais ou menos experientes, já passaram por um momento em que se quer desistir de tudo. Jovem aprendiz de R, a única diferença entre você que está iniciando agora e nós que usamos o R há mais tempo são as horas a mais de uso (e ódio). O que temos a mais é experiência para olhar o erro, lê-lo e conseguir interpretar o que está errado e saber buscar ajuda.

Dessa forma, o ponto mais importante de quem está iniciando é ter paciência, calma, bom humor, ler e entender as mensagens de erros. Recomendamos uma prática que pode ajudar: caso não esteja conseguindo resolver alguma parte do seu código, deixe ele de lado um tempo, descanse, faça uma caminhada, tome um banho, converse com seus animais de estimação ou plantas, tenha um pato de borracha ou outro objeto inanimado (um dos autores tem um sapinho de madeira), explique esse código para esse pato (processo conhecido como [Debug com Pato de Borracha](#)), logo a solução deve aparecer.

Listaremos aqui o que consideramos os principais erros dos iniciantes no R.

#### 1. Esquecer de completar uma função ou bloco de códigos

Esquecer de completar uma função ou bloco de códigos é algo bem comum. Geralmente esquecemos de fechar aspas "" ou parênteses (), mas geralmente o R nos informa isso, indicando um símbolo de + no console. Se você cometeu esse erro, lembre-se de apertar a tecla esc do seu computador clicando antes com o cursor do mouse no console do R.

```
sum(1, 2
+
## Error: <text>:3:0: unexpected end of input
## 1: sum(1, 2
## 2:   +
##   ^
```

## 2. Esquecer de vírgulas dentro de funções

Outro erro bastante comum é esquecer de acrescentar a vírgula , para separar argumentos dentro de uma função, principalmente se estamos compondo várias funções acopladas, i.e., uma função dentro da outra.

```
sum(1 2
##
## Error: <text>:1:7: unexpected numeric constant
## 1: sum(1 2
##   ^
##
```

## 3. Chamar um objeto pelo nome errado

Pode parecer simples, mas esse é de longe o erro mais comum que pessoas iniciantes comentem. Quando temos um script longo, é de se esperar que tenhamos atribuído diversos objetos e em algum momento atribuímos um nome do qual não lembramos. Dessa forma, quando chamamos o objeto ele não existe e o console informa um erro. Entretanto, esse tipo de erro pode ser facilmente identificado, como o exemplo abaixo.

```
obj <- 10
OBJ
##
## Error in eval(expr, envir, enclos): object 'OBJ' not found
```

## 4. Esquecer de carregar um pacote

Esse também é um erro recorrente, mesmo para usuários mais experientes. Em scripts de análises complexas, que requerem vários pacotes, geralmente esquecemos de um ou outro pacote. A melhor forma de evitar esse tipo de erro é listar os pacotes que vamos precisar usar logo no início do script.

```
## Carregar dados
data(dune)
##
## Warning in data(dune): data set 'dune' not found
## Função do pacote vegan
decostand(dune[1:6, 1:6], "hell")
```

```
## Error in decostand(dune[1:6, 1:6], "hell"): could not find function "decostand"
```

Geralmente a mensagem de erro será de que a função não foi encontrada ou algo nesse sentido. Carregando o pacote, esse erro é contornado.

```
## Carregar o pacote
library(vegan)
##
## This is vegan 2.6-4
## Carregar dados
data(dune)
```

```

## Função do pacote vegan
decostand(dune[1:6, 1:6], "hell")

## Achimill Agrostol Airaprae Alopogeni Anthodor Bellpere
## 1 1.0000000 0.0000000 0 0.0000000 0.0000000 0.0000000
## 2 0.6123724 0.0000000 0 0.5000000 0.0000000 0.6123724
## 3 0.0000000 0.5547002 0 0.7337994 0.0000000 0.3922323
## 4 0.0000000 0.8164966 0 0.4082483 0.0000000 0.4082483
## 5 0.5000000 0.0000000 0 0.0000000 0.7071068 0.5000000
## 6 0.6324555 0.0000000 0 0.0000000 0.7745967 0.0000000

```

## 5. Usar o nome da função de forma errônea

Esse erro não é tão comum, mas pode ser incômodo às vezes. Algumas funções possuem nomes no padrão “Camel Case”, i.e., com letras maiúsculas no meio do nome da função. Isso às vezes pode confundir, ou ainda, as funções podem ou não ser separadas com ., como `row.names()` e `rownames()`.

```

## Soma das colunas
colsums(dune)

## Error in colsums(dune): could not find function "colsums"
## Soma das colunas
colSums(dune)

```

```

## Achimill Agrostol Airaprae Alopogeni Anthodor Bellpere Bromhord Chenalbu
##      16     48      5    36     21     13     15      1
## Cirsarve Comapalu Eleopalu Elymrepe Empenigr Hyporadi Juncarti Juncbufo
##      2       4    25     26      2      9     18     13
## Lolipere Planlanc Poaprat Poatriv Ranuflam Rumeacet Sagiproc Salirepe
##      58     26     48     63     14     18     20     11
## Scorautu Trifprat Trifrepe Vicilath Bracruta Callcusp
##      54      9     47      4     49     10

```

## 6. Atentar para o diretório correto

Muitas vezes o erro é simplesmente porque o usuário(a) não definiu o diretório correto onde está o arquivo a ser importado ou exportado. Por isso é fundamental sempre verificar se o diretório foi definido corretamente, geralmente usando as funções `dir()` ou `list.files()` para listar no console a lista de arquivos no diretório. Podemos ainda usar o argumento `pattern` para listar arquivos por um padrão textual.

```

## Listar os arquivos do diretório definido
dir()
list.files()

## Listar os arquivos do diretório definido por um padrão
dir(pattern = ".csv")

```

Além disso, é fundamental ressaltar a importância de verificar se o nome do arquivo que importaremos foi digitado corretamente, atentando-se também para a extensão: `.csv`, `.txt`, `.xlsx`, etc.

## 5.4 Estrutura e manipulação de objetos

O conhecimento sobre a estrutura e manipulação de objetos é fundamental para ter domínio e entendimento do funcionamento da linguagem R. Nesta seção, trataremos da estrutura e manipulação de dados no R, no que ficou conhecido como modo *R Base*, em contrapartida ao *tidyverse*, tópico tratado no Capítulo 6. Abordaremos aqui temas chaves, como: i) atributos de objetos, ii) manipulação de objetos unidimensionais e multidimensionais, iii) valores faltantes e especiais, iv) diretório de trabalho e v) importar, conferir e exportar dados tabulares.

#### 5.4.1 Atributo dos objetos

Quando fazemos atribuições de dados no R (`<-`), os objetos gerados possuem três características.

1. **Nome**: palavra que o R reconhece os dados atribuídos
2. **Conteúdo**: dados em si
3. **Atributos**: modos (*natureza*) e estruturas (*organização*) dos elementos

Vamos explorar mais a fundo os **modos** e **estruturas** dos objetos. Vale ressaltar que isso é uma simplificação, pois há muitas classes de objetos, como funções e saídas de funções que possuem outros atributos.

Podemos verificar os atributos dos objetos com a função `attributes()`.

```
## Atributos  
attributes(dune)
```

```
## $names  
## [1] "Achimill" "Agrostol" "Airaprae" "Alopogeni" "Anthodor" "Bellpere"  
## [7] "Bromhord" "Chenalbu" "Cirsarve" "Comapalu" "Eleopalu" "Elymrepe"  
## [13] "Empenigr" "Hyporadi" "Juncarti" "Juncbufo" "Lolipere" "Planlanc"  
## [19] "Poaprat" "Poatriv" "Ranuflam" "Rumeacet" "Sagiproc" "Salirepe"  
## [25] "Scorautu" "Trifprat" "Trifrepe" "Vicilath" "Bracruta" "Callcusp"  
##  
## $row.names  
## [1] "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12" "13" "14"  
## [15] "15" "16" "17" "18" "19" "20"  
##  
## $class  
## [1] "data.frame"
```

#### Modo dos objetos

A depender da natureza dos elementos que compõem os dados e que foram atribuídos aos objetos, esses objetos podem ser, de forma simples um dos cinco modos: numérico do tipo inteiro (*integer*), numérico do tipo flutuante (*double*), texto (*character*), lógico (*logical*) ou complexo (*complex*).

A atribuição de números no R pode gerar dois tipos de modos: **integer** para números inteiros e **double** para números flutuantes ou com decimais.

```
## Numérico double  
obj_numerico_double <- 1  
  
## Modo  
mode(obj_numerico_double)  
  
## [1] "numeric"  
  
## Tipo  
typeof(obj_numerico_double)  
  
## [1] "double"
```

A título de praticidade, ambos são incorporados como o modo numeric, com o tipo double, a menos que especifiquemos que seja inteiro com a letra L depois do número, representando a palavra *Larger*, geralmente usando para armazenar números muito grandes.

```
## Numérico integer  
obj_numerico_inteiro <- 1L  
  
## Modo  
mode(obj_numerico_inteiro)
```

```

## [1] "numeric"
## Tipo
typeof(obj_numericointeiro)

```

```
## [1] "integer"
```

Além de números, podemos atribuir textos, utilizando para isso aspas "".

```

## Caracter ou string
obj_caracter <- "a" # atenção para as aspas

## Modo
mode(obj_caracter)

```

```
## [1] "character"
```

Em algumas situações, precisamos indicar a ocorrência ou não de um evento ou uma operação. Para isso, utilizamos as palavras reservadas (TRUE e FALSE), chamadas de variáveis booleanas, pois assumem apenas duas possibilidades: falso (0) ou verdadeiro (1). Devemos nos ater para o fato dessas palavras serem escritas com letras maiúsculas e sem aspas.

```

## Lógico
obj_logico <- TRUE # maiusculas e sem aspas

## Modo
mode(obj_logico)

```

```
## [1] "logical"
```

Por fim, existe um modo pouco utilizado que cria números complexos (raiz de números negativos).

```

## Complexo
obj_complexo <- 1+1i

## Modo
mode(obj_complexo)

```

```
## [1] "complex"
```

Podemos verificar o modo dos objetos ou fazer a conversão entre esses modos com diversas funções.

```

## Verificar o modo dos objetos
is.numeric()
is.integer()
is.character()
is.logical()
is.complex()

```

```

## Conversões entre modos
as.numeric()
as.integer()
as.character()
as.logical()
as.complex()

```

```

## Exemplo
num <- 1:5
num
mode(num)

```

```

cha <- as.character(num)
cha
mode(cha)

```

### Estrutura dos objetos

Uma vez entendido a natureza dos modos dos elementos dos objetos no R, podemos passar para o passo seguinte e entender como esses elementos são estruturados dentro dos objetos.

Essa estruturação irá nos contar sobre a organização dos elementos, com relação aos modos e dimensionalidade da disposição desses elementos. De modo bem simples, os elementos podem ser estruturados em cinco tipos:

1. **Vetores e fatores**: homogêneo (*um modo*) e unidimensional (*uma dimensão*). Um tipo especial de vetor são os fatores, usados para designar variáveis categóricas
2. **Matrizes**: homogêneo (*um modo*) e bidimensional (*duas dimensões*)
3. **Arrays**: homogêneo (*um modo*) e multidimensional (*mais de duas dimensões*)
4. **Data frames**: heterogêneo (*mais de um modo*) e bidimensional (*duas dimensões*)
5. **Listas**: heterogêneo (*mais de um modo*) e unidimensional (*uma dimensão*)

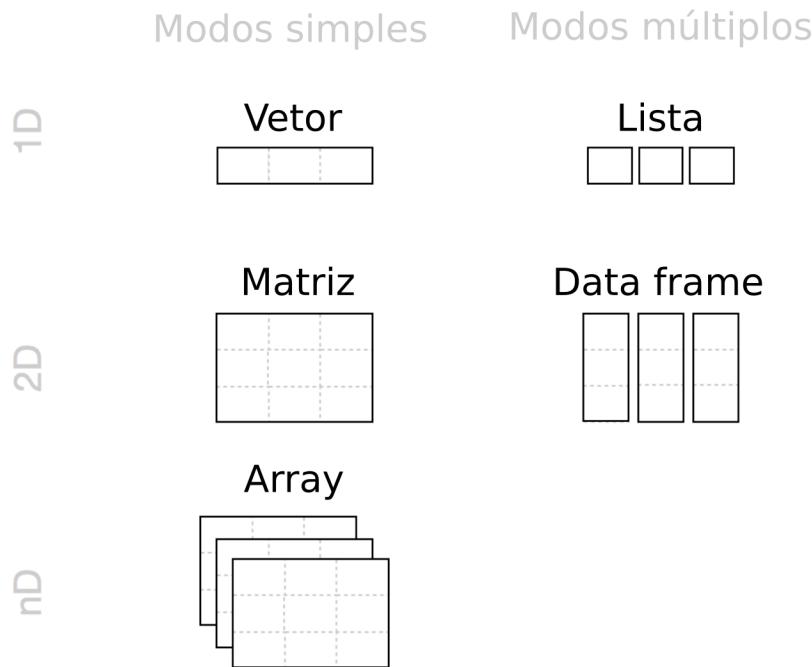


Figura 5.3: Estruturas de dados mais comuns no R: vetores, matrizes, arrays, listas e data frames. Adaptado de: R for Data Science (2e) .

### Vetor

Vetores representam o encadeamento de elementos numa sequência unidimensional. Dessa forma, no R, essa estrutura de dados pode ser traduzida como medidas de uma variável numérica (discretas ou contínuas), variável binária (booleana - TRUE e FALSE) ou descrição (informações em texto).

Há diversas formas de se criar um vetor no R:

1. Concatenando elementos com a função `c()`
2. Criando sequências unitárias : ou com a função `seq()`
3. Criando repetições com a função `rep()`
4. “Colar” palavras com uma sequência numérica com a função `paste()` ou `paste0()`

```

5. Amostrando aleatoriamente elementos com a função sample()

## Concatenar elementos numéricos
concatenar <- c(15, 18, 20, 22, 18)
concatenar

## [1] 15 18 20 22 18

## Sequência unitária (x1:x2)
sequencia <- 1:10
sequencia

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

## Sequência com diferentes espaçamentos
sequencia_esp <- seq(from = 0, to = 100, by = 10)
sequencia_esp

## [1] 0 10 20 30 40 50 60 70 80 90 100

## Repetição
repeticao <- rep(x = c(TRUE, FALSE), times = 5)
repeticao

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

## Cola palavra e sequência numérica
colar <- paste("amostra", 1:5)
colar

## [1] "amostra 1" "amostra 2" "amostra 3" "amostra 4" "amostra 5"

```

Como os vetores são homogêneos, i.e., só comportam um modo, quando combinamos mais de um modo no mesmo objeto ocorre uma dominância de modos. Existe, dessa forma, uma **coerção** dos elementos combinados para que todos fiquem iguais. Essa dominância segue essa ordem:

**DOMINANTE** character > double > integer > logical **RECESSIVO**

Além disso, podemos utilizar as conversões listadas anteriormente para alterar os modos. Vamos exemplificar combinando os vetores criados anteriormente e convertendo-os.

```

## Coerção
c(colar, amostragem)

## [1] "amostra 1" "amostra 2" "amostra 3" "amostra 4" "amostra 5" "54"
## [7] "84"         "55"          "100"         "58"          "45"          "37"
## [13] "43"        "79"          "85"

## Conversão
as.numeric(repeticao)

## [1] 1 0 1 0 1 0 1 0 1 0

```

## Fator

O fator representa medidas de uma variável categórica, podendo ser nominal ou ordinal. É fundamental destacar que fatores no R devem ser entendidos como um vetor de **integer**, i.e., ele é composto por números

inteiros representando os níveis da variável categórica.

Para criar um fator no R usamos uma função específica `factor()`, na qual podemos especificar os **níveis** com o argumento `level`, ou fazemos uma conversão usando a função `as.factor()`. Trabalhar com fatores no *R Base* não é das tarefas mais agradáveis, sendo assim, no Capítulo 6 usamos a versão *tidyverse* usando o pacote `forcats`. Destacamos ainda a existência de fatores nominais para variáveis categóricas nominais e fatores ordinais para variáveis categóricas ordinais, quando há ordenamento entre os níveis, como dias da semana ou classes de altura.

```
## Fator nominal
fator_nominal <- factor(x = sample(x = c("floresta", "pastagem", "cerrado"),
                                      size = 20, replace = TRUE),
                           levels = c("floresta", "pastagem", "cerrado"))
fator_nominal

## [1] pastagem floresta pastagem floresta pastagem floresta cerrado pastagem
## [9] cerrado pastagem pastagem pastagem cerrado floresta cerrado floresta
## [17] cerrado pastagem floresta pastagem
## Levels: floresta pastagem cerrado

## Fator ordinal
fator_ordinal <- factor(x = sample(x = c("baixa", "media", "alta"),
                                       size = 20, replace = TRUE),
                           levels = c("baixa", "media", "alta"), ordered = TRUE)
fator_ordinal

## [1] media media baixa alta media media baixa alta alta media alta baixa
## [13] alta alta baixa media media baixa media alta
## Levels: baixa < media < alta

## Conversão
fator <- as.factor(x = sample(x = c("floresta", "pastagem", "cerrado"),
                                 size = 20, replace = TRUE))
fator

## [1] floresta pastagem cerrado floresta floresta pastagem cerrado floresta
## [9] pastagem pastagem cerrado cerrado pastagem floresta cerrado pastagem
## [17] floresta floresta floresta pastagem
## Levels: cerrado floresta pastagem
```

## Matriz

A matriz representa dados no formato de tabela, com linhas e colunas. As linhas geralmente representam unidades amostrais (locais, transectos, parcelas) e as colunas representam variáveis numéricas (discretas ou contínuas), variáveis binárias (TRUE ou FALSE) ou descrições (informações em texto).

Podemos criar matrizes no R de duas formas. A primeira delas dispondo elementos de um vetor em um certo número de linhas e colunas com a função `matrix()`, podendo preencher essa matriz com os elementos do vetor por linhas ou por colunas alterando o argumento `byrow`.

```
## Vetor
ve <- 1:12

## Matrix - preenchimento por linhas - horizontal
ma_row <- matrix(data = ve, nrow = 4, ncol = 3, byrow = TRUE)
ma_row

##      [,1] [,2] [,3]
## [1,]    1    2    3
```

```

## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
## Matrix - preenchimento por colunas - vertical
ma_col <- matrix(data = ve, nrow = 4, ncol = 3, byrow = FALSE)
ma_col
```

```

##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10
## [3,]    3    7   11
## [4,]    4    8   12
```

A segunda forma, podemos combinar vetores, utilizando a função `rbind()` para combinar vetores por linha, i.e., um vetor embaixo do outro, e `cbind()` para combinar vetores por coluna, i.e., um vetor ao lado do outro.

```

## Criar dois vetores
vec_1 <- c(1, 2, 3)
vec_2 <- c(4, 5, 6)

## Combinar por linhas - vertical - um embaixo do outro
ma_rbind <- rbind(vec_1, vec_2)
ma_rbind
```

```

##      [,1] [,2] [,3]
## vec_1    1    2    3
## vec_2    4    5    6

## Combinar por colunas - horizontal - um ao lado do outro
ma_cbind <- cbind(vec_1, vec_2)
ma_cbind
```

```

##      vec_1 vec_2
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

## Array

O array representa combinação de tabelas, com linhas, colunas e dimensões. Essa combinação pode ser feita em múltiplas dimensões, mas apesar disso, geralmente é mais comum o uso em Ecologia para três dimensões, por exemplo: linhas (unidades amostrais), colunas (espécies) e dimensão (tempo). Isso gera um “cubo mágico” ou “cartas de um baralho”, onde podemos comparar, nesse caso, comunidades ao longo do tempo. Além disso, arrays também são muito comuns em morfometria geométrica ou sensoriamento remoto.

Podemos criar arrays no R dispondo elementos de um vetor em um certo número de linhas, colunas e dimensões com a função `array()`. Em nosso exemplo, vamos compor cinco comunidades de cinco espécies ao longo de três períodos.

```

## Array
ar <- array(data = sample(x = c(0, 1), size = 75, rep = TRUE),
            dim = c(5, 5, 3))
ar

## , , 1
##
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0    0    1    0    1
```

```

## [2,]    1    1    1    1    0
## [3,]    0    1    1    0    0
## [4,]    1    1    0    0    1
## [5,]    0    0    1    1    0
##
## , , 2
##
## [,1] [,2] [,3] [,4] [,5]
## [1,]    1    0    1    1    0
## [2,]    1    0    1    1    0
## [3,]    1    1    0    1    0
## [4,]    1    0    0    1    1
## [5,]    1    0    1    0    0
##
## , , 3
##
## [,1] [,2] [,3] [,4] [,5]
## [1,]    0    1    0    0    1
## [2,]    1    1    0    1    0
## [3,]    1    0    0    0    0
## [4,]    0    1    1    1    1
## [5,]    0    1    0    1    0

```

### Data frame

O data frame também representa dados no formato de tabela, com linhas e colunas, muito semelhante à matriz. Mas diferentemente das matrizes, os data frames comportam mais de um modo em suas colunas. Dessa forma, as linhas do data frame ainda representam unidades amostrais (locais, transectos, parcelas), mas as colunas agora podem representar descrições (informações em texto), variáveis numéricas (discretas ou contínuas), variáveis binárias (TRUE ou FALSE) e variáveis categóricas (nominais ou ordinais).

A forma mais simples de se criar data frames no R é através da combinação de vetores. Essa combinação é feita com a função `data.frame()` e ocorre de forma horizontal, semelhante à função `cbind()`. Sendo assim, todos os vetores precisam ter o mesmo número de elementos, ou seja, o mesmo comprimento. Podemos ainda nomear as colunas de cada vetor. Outra forma, seria converter uma matriz em um data frame, utilizando a função `as.data.frame()`.

```

## Criar três vetores
vec_ch <- c("sp1", "sp2", "sp3")
vec_nu <- c(4, 5, 6)
vec_fa <- factor(c("campo", "floresta", "floresta"))

## Data frame - combinar por colunas - horizontal - um ao lado do outro
df <- data.frame(vec_ch, vec_nu, vec_fa)

df

##   vec_ch vec_nu   vec_fa
## 1     sp1      4     campo
## 2     sp2      5   floresta
## 3     sp3      6   floresta

## Data frame - nomear as colunas
df <- data.frame(especies = vec_ch,
                  abundancia = vec_nu,
                  vegetacao = vec_fa)
df

```

```

##   especies abundancia vegetacao
## 1      sp1          4     campo
## 2      sp2          5 floresta
## 3      sp3          6 floresta
## Data frame - converter uma matriz
ma <- matrix(data = ve, nrow = 4, ncol = 3, byrow = TRUE)
ma

## [,1] [,2] [,3]
## [1,]    1    2    3
## [2,]    4    5    6
## [3,]    7    8    9
## [4,]   10   11   12
df_ma <- as.data.frame(ma)
df_ma

##   V1 V2 V3
## 1  1  2  3
## 2  4  5  6
## 3  7  8  9
## 4 10 11 12

```

## Lista

A lista é um tipo especial de vetor que aceita objetos como elementos. Ela é a estrutura de dados utilizada para agrupar objetos, e é geralmente a saída de muitas funções.

Podemos criar listas através da função `list()`. Essa função funciona de forma semelhante à função `c()` para a criação de vetores, mas agora estamos concatenando objetos. Podemos ainda nomear os elementos (objetos) que estamos combinando.

Um ponto interessante para entender data frames, é que eles são listas, em que todos os elementos (colunas) possuem o mesmo número de elementos, ou seja, mesmo comprimento.

```

## Lista
lista <- list(rep(1, 20), # vector
              factor(1, 1), # factor
              cbind(c(1, 2), c(1, 2))) # matrix
lista

## [[1]]
## [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## 
## [[2]]
## [1] 1
## Levels: 1
## 
## [[3]]
## [,1] [,2]
## [1,]    1    1
## [2,]    2    2
## 
## Lista - nomear os elementos
lista_nome <- list(vector = rep(1, 20), # vector
                     factor = factor(1, 1), # factor
                     matrix = cbind(c(1, 2), c(1, 2))) # matrix
lista_nome

```

```

## $vector
## [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##
## $factor
## [1] 1
## Levels: 1
##
## $matrix
##      [,1] [,2]
## [1,]    1    1
## [2,]    2    2

```

## Funções

Uma última estrutura de objetos criados no R são as funções. Elas são objetos criados pelo usuário e reutilizados para fazer operações específicas. A criação de funções geralmente é um tópico tratado num segundo momento, quando o usuário de R adquire certo conhecimento da linguagem. Aqui abordaremos apenas seu funcionamento básico, diferenciando sua estrutura para entendimento e sua diferenciação das demais estruturas.

Vamos criar uma função simples que retorna a multiplicação de dois termos. Criaremos a função com o nome `multi`, à qual será atribuída uma função com o nome `function()`, com dois argumentos `x` e `y`. Depois disso abrimos chaves {}, que é onde iremos incluir nosso bloco de código. Nossa blocos de código é composto por duas linhas, a primeira contendo a operação de multiplicação dos argumento com a atribuição ao objeto `mu` e a segunda contendo a função `return()` para retornar o valor da multiplicação.

*## Criar uma função*

```

multi <- function(x, y){

  mu <- (x * y)
  return(mu)

}
multi

```

```

## function(x, y){
##
##   mu <- (x * y)
##   return(mu)
##
## }

```

*## Uso da função*

```

multi(42, 23)

```

```

## [1] 966

```

### 5.4.2 Manipulação de objetos unidimensionais

Vamos agora explorar formas de manipular elementos de objetos unidimensionais, ou seja, vetores, fatores e listas.

A primeira forma de manipulação é através da **indexação**, utilizando os operadores `[]`. Com a indexação podemos acessar elementos de vetores e fatores por sua posição. Utilizaremos números, sequência de números ou operações booleanas para retornar partes dos vetores ou fatores. Podemos ainda retirar elementos dessas estruturas com o operador aritmético `-`.

No exemplo a seguir, iremos fixar o ponto de partida da amostragem da função `sample()`, utilizando a função `set.seed(42)` (usamos 42 porque é a resposta para a vida, o universo e tudo mais - O Guia do Mochileiro

das Galáxias, mas poderia ser outro número qualquer). Isso permite que o resultado da amostragem aleatória seja igual em diferentes computadores.

```
## Fixar a amostragem
set.seed(42)

## Amostrar 10 elementos de uma sequência
ve <- sample(x = seq(0, 2, .05), size = 10)
ve

## [1] 1.80 0.00 1.20 0.45 1.75 0.85 1.15 0.30 1.90 0.20

## Seleciona o quinto elemento
ve[5]

## [1] 1.75

## Seleciona os elementos de 1 a 5
ve[1:5]

## [1] 1.80 0.00 1.20 0.45 1.75

## Retira o décimo elemento
ve[-10]

## [1] 1.80 0.00 1.20 0.45 1.75 0.85 1.15 0.30 1.90

## Retira os elementos 2 a 9
ve[-(2:9)]

## [1] 1.8 0.2
```

Podemos ainda fazer uma seleção condicional do vetor. Ao utilizarmos operadores relacionais, teremos como resposta um vetor lógico. Esse vetor lógico pode ser utilizado dentro da indexação para seleção de elementos.

```
## Quais valores são maiores que 1?
ve > 1

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

## Selecionar os valores acima de 1 no vetor ve
ve[ve > 1]

## [1] 1.80 1.20 1.75 1.15 1.90
```

Além da indexação, temos algumas funções que nos auxiliam em algumas operações com objetos unidimensionais.

Table 1: Funções para verificação e resumo de dados unidimensionais.

Função	Descrição
<code>max()</code>	Valor máximo
<code>min()</code>	Valor mínimo
<code>range()</code>	Amplitude
<code>length()</code>	Comprimento
<code>sum()</code>	Soma
<code>cumsum()</code>	Soma cumulativa
<code>prod()</code>	Produto
<code>sqrt()</code>	Raiz quadrada
<code>abs()</code>	Valor absoluto

Função	Descrição
<code>exp()</code>	Expoente
<code>log()</code>	Logaritmo natural
<code>log1p()</code>	Logaritmo natural mais 1 $\log(x + 1)$
<code>log2()</code>	Logaritmo base 2
<code>log10()</code>	Logaritmo base 10
<code>mean()</code>	Média
<code>mean.weighted()</code>	Média ponderada
<code>var()</code>	Variância
<code>sd()</code>	Desvio Padrão
<code>median()</code>	Mediana
<code>quantile()</code>	Quantil
<code>quarters()</code>	Quartil
<code>IQR()</code>	Amplitude interquartil
<code>round()</code>	Arredondamento
<code>sort()</code>	Ordenação
<code>order()</code>	Posição ordenada
<code>rev()</code>	Reverso
<code>unique()</code>	Únicos
<code>summary()</code>	Resumo estatístico
<code>cut()</code>	Divide variável contínua em fator
<code>pretty()</code>	Divide variável contínua em intervalos
<code>scale()</code>	Padronização e centralização
<code>sub()</code>	Substitui caracteres
<code>grep()</code>	Posição de caracteres
<code>any()</code>	Algum valor?
<code>all()</code>	Todos os valores?
<code>which()</code>	Quais valores?
<code>subset()</code>	Subconjunto
<code>ifelse()</code>	Operação condicional

Para listas, também podemos usar a indexação [] para acessar ou retirar elementos.

```
## Lista
li <- list(elem1 = 1, elem2 = 2, elem3 = 3)

## Acessar o primeiro elemento
li[1]

## $elem1
## [1] 1

## Retirar o primeiro elemento
li[-1]

## $elem2
## [1] 2
##
## $elem3
## [1] 3
```

Podemos ainda usar a indexação dupla [[]] para acessar os valores desses elementos.

```
## Acessar o valor do primeiro elemento
li[[1]]
```

```

## [1] 1
## Acessar o valor do segundo elemento
li[[2]]

## [1] 2

```

Para listas nomeadas, podemos ainda utilizar o operador `$` para acessar elementos pelo seu nome.

```

## Acessar o primeiro elemento
li$elem1

```

```

## [1] 1

```

E ainda podemos utilizar funções para medir o comprimento dessa lista, listar os nomes dos elementos ou ainda renomear os elementos: `length()` e `names()`.

```

## Comprimento
length(li)

```

```

## [1] 3

```

```

## Nomes
names(li)

```

```

## [1] "elem1" "elem2" "elem3"
## Renomear
names(li) <- paste0("elemento0", 1:3)
li

```

```

## $elemento01
## [1] 1
##
## $elemento02
## [1] 2
##
## $elemento03
## [1] 3

```

#### 5.4.3 Manipulação de objetos multidimensionais

Da mesma forma que para objetos unidimensionais, podemos manipular elementos de objetos multidimensionais, ou seja, matrizes, data frames e arrays.

Novamente, a primeira forma de manipulação é através da indexação, utilizando os operadores `[]`. Com a indexação podemos acessar elementos de matrizes, data frames e arrays por sua posição. Podemos ainda retirar elementos dessas estruturas com o operador aritmético `-`.

Entretanto, agora temos mais de uma dimensão na estruturação dos elementos dentro dos objetos. Assim, utilizamos números, sequência de números ou operação booleanas para retornar partes desses objetos, mas as dimensões têm de ser explicitadas e separadas por **vírgulas** para acessar linhas e colunas. Essa indexação funciona para matrizes e data frames. Para arrays, especificamos também as dimensões, também separadas por vírgulas para acessar essas dimensões.

```

## Matriz
ma <- matrix(1:12, 4, 3)
ma

```

```

##      [,1] [,2] [,3]
## [1,]    1    5    9
## [2,]    2    6   10

```

```

## [3,]    3    7   11
## [4,]    4    8   12
## Indexação
ma[3, ] # linha 3

## [1] 3 7 11
ma[, 2] # coluna 2

## [1] 5 6 7 8
ma[1, 2] # elemento da linha 1 e coluna 2

## [1] 5
ma[1, 1:2] # elementos da linha 1 e coluna 1 e 2

## [1] 1 5
ma[1, c(1, 3)] # elementos da linha 1 e coluna 1 e 3

## [1] 1 9
ma[-1, ] # retirar a linha 1

##      [,1] [,2] [,3]
## [1,]    2    6   10
## [2,]    3    7   11
## [3,]    4    8   12
ma[, -3] # retirar a coluna 3

##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8

```

Para data frames, além de utilizar números e/ou sequências de números dentro do operador [] simples, podemos utilizar o operador [[]] duplo para retornar apenas os valores de uma linha ou uma coluna. Se as colunas estiverem nomeadas, podemos utilizar o nome da coluna de interesse entre aspas dentro dos operadores [] (retornar coluna) e [[]] (retornar apenas os valores), assim como ainda podemos utilizar o operador \$ para data frames. Essas últimas operações retornam um vetor, para o qual podemos fazer operações de vetores ou ainda atualizar o valor dessa coluna selecionada ou adicionar outra coluna.

```

## Criar três vetores
sp <- paste("sp", 1:10, sep = "")
abu <- 1:10
flo <- factor(rep(c("campo", "floresta"), each = 5))

## data frame
df <- data.frame(sp, abu, flo)
df

##      sp abu     flo
## 1  sp1    1     campo
## 2  sp2    2     campo
## 3  sp3    3     campo
## 4  sp4    4     campo
## 5  sp5    5     campo

```

```

## 6   sp6   6 floresta
## 7   sp7   7 floresta
## 8   sp8   8 floresta
## 9   sp9   9 floresta
## 10  sp10  10 floresta
## [] - números
df[, 1]

## [1] "sp1"  "sp2"  "sp3"  "sp4"  "sp5"  "sp6"  "sp7"  "sp8"  "sp9"  "sp10"
## [] - nome das colunas - retorna coluna
df["flo"]

##      flo
## 1     campo
## 2     campo
## 3     campo
## 4     campo
## 5     campo
## 6   floresta
## 7   floresta
## 8   floresta
## 9   floresta
## 10  floresta
## [[]] - nome das colunas - retorna apenas os valores
df[["flo"]]

## [1] campo     campo     campo     campo     campo     floresta floresta floresta
## [9] floresta floresta
## Levels: campo floresta
## $ funciona apenas para data frame
df$sp

## [1] "sp1"  "sp2"  "sp3"  "sp4"  "sp5"  "sp6"  "sp7"  "sp8"  "sp9"  "sp10"
## Operação de vetores
length(df$abu)

## [1] 10
## Converter colunas
df$abu <- as.character(df$abu)
mode(df$abu)

## [1] "character"
## Adicionar ou mudar colunas
set.seed(42)
df$abu2 <- sample(x = 0:1, size = nrow(df), rep = TRUE)
df

##      sp abu      flo abu2
## 1   sp1  1     campo    0
## 2   sp2  2     campo    0
## 3   sp3  3     campo    0
## 4   sp4  4     campo    0
## 5   sp5  5     campo    1

```

```

## 6   sp6   6 floresta   1
## 7   sp7   7 floresta   1
## 8   sp8   8 floresta   1
## 9   sp9   9 floresta   0
## 10  sp10 10 floresta   1

```

Podemos ainda fazer seleções condicionais para retornar linhas com valores que temos interesse, semelhante ao uso de filtro de uma planilha eletrônica.

```

## Selecionar linhas de uma matriz ou data frame
df[df$abu > 4, ]

```

```

##      sp abu      flo abu2
## 5  sp5   5     campo    1
## 6  sp6   6 floresta   1
## 7  sp7   7 floresta   1
## 8  sp8   8 floresta   1
## 9  sp9   9 floresta   0
df[df$flo == "floresta", ]

```

```

##      sp abu      flo abu2
## 6  sp6   6 floresta   1
## 7  sp7   7 floresta   1
## 8  sp8   8 floresta   1
## 9  sp9   9 floresta   0
## 10 sp10 10 floresta   1

```

Além disso, há uma série de funções para conferência e manipulação de dados que listamos na Tabela seguinte:

Funções para verificação e resumo de dados multidimensionais.

Função

Descrição

`head()`

Mostra as primeiras 6 linhas

`tail()`

Mostra as últimas 6 linhas

`nrow()`

Mostra o número de linhas

`ncol()`

Mostra o número de colunas

`dim()`

Mostra o número de linhas e de colunas

`rownames()`

Mostra os nomes das linhas (locais)

`colnames()`

Mostra os nomes das colunas (variáveis)

`str()`

Mostra as classes de cada coluna (estrutura)  
`summary()`

Mostra um resumo dos valores de cada coluna  
`rowSums()`

Calcula a soma das linhas (horizontal)  
`colSums()`

Calcula a soma das colunas (vertical)  
`rowMeans()`

Calcula a média das linhas (horizontal)  
`colMeans()`

Calcula a média das colunas (vertical)  
`table()`

Tabulação cruzada  
`t()`

Matriz ou data frame transposto

#### 5.4.4 Valores faltantes e especiais

Valores faltantes e especiais são valores reservados que representam dados faltantes, indefinições matemáticas, infinitos e objetos nulos.

1. **NA (Not Available)**: significa dado faltante ou indisponível
2. **NaN (Not a Number)**: representa indefinições matemáticas
3. **Inf (Infinito)**: é um número muito grande ou um limite matemático
4. **NULL (Nulo)**: representa um objeto nulo, sendo útil para preenchimento em aplicações de programação

```
## Data frame com elemento NA
df <- data.frame(var1 = c(1, 4, 2, NA), var2 = c(1, 4, 5, 2))
df
```

```
##   var1 var2
## 1     1     1
## 2     4     4
## 3     2     5
## 4    NA     2
```

```
## Resposta booleana para elementos NA
is.na(df)
```

```
##           var1  var2
## [1,] FALSE FALSE
## [2,] FALSE FALSE
## [3,] FALSE FALSE
## [4,]  TRUE FALSE
```

```
## Algum elemento é NA?
any(is.na(df))
```

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

```

## Remover as linhas com NAs
df_sem_na <- na.omit(df)
df_sem_na

##   var1 var2
## 1     1     1
## 2     4     4
## 3     2     5
## Substituir NAs por 0
df[is.na(df)] <- 0
df

##   var1 var2
## 1     1     1
## 2     4     4
## 3     2     5
## 4     0     2
## Desconsiderar os NAs em funções com o argumento rm.na = TRUE
sum(1, 2, 3, 4, NA, na.rm = TRUE)

## [1] 10
## NaN - not a number
0/0

## [1] NaN
log(-1)

## Warning in log(-1): NaNs produced
## [1] NaN
## Limite matemático
1/0

## [1] Inf
## Número grande
10^310

## [1] Inf
## Objeto nulo
nulo <- NULL
nulo

## NULL

```

#### 5.4.5 Diretório de trabalho

O diretório de trabalho é o endereço da pasta (ou diretório) de onde o R importará ou exportar nossos dados.

Podemos utilizar o próprio RStudio para tal tarefa, indo em **Session > Set Work Directory > Choose Directory...** ou simplesmente utilizar o atalho **Ctrl + Shift + H**.

Podemos ainda utilizar funções do R para definir o diretório. Para tanto, podemos navegar com o aplicativo de gerenciador de arquivos (e.g., Windows Explorer) até nosso diretório de interesse e copiar o endereço na barra superior. Voltamos para o R e colamos esse endereço entre aspas como argumento da função **setwd()**. É fundamental destacar que no Windows é necessário inverter as barras (\ por / ou duplicar elas \\).

Aconselhamos ainda utilizar as funções `getwd()` para retornar o diretório definido na sessão do R, assim como as funções `dir()` ou `list.files()` para listagem dos arquivos no diretório, ambas medidas de conferência do diretório correto.

```
## Definir o diretório de trabalho
setwd("/home/mude/data/github/livro_aer/dados")

## Verificar o diretório
getwd()

## Listar os arquivos no diretório
dir()
list.files()
```

#### 5.4.6 Importar dados

Uma das operações mais corriqueiras do R, antes de realizar alguma análise ou plotar um gráfico, é a de importar dados que foram tabulados numa planilha eletrônica e salvos no formato .csv, .txt ou .xlsx. Ao importar esse tipo de dado para o R, o formato que o mesmo assume, se nenhum parâmetro for especificado, é o da classe `data frame`, prevendo que a planilha de dados possua colunas com diferentes modos.

Existem diversas formas de importar dados para o R. Podemos importar utilizando o RStudio, indo na janela **Environment** e clicar em “Importar Dataset”.

Entretanto, aconselhamos o uso de funções que fiquem salvas em um script para aumentar a reprodutibilidade do mesmo. Dessa forma, as três principais funções para importar os arquivos nos três principais extensões (.csv, .txt ou .xlsx) são, respectivamente: `read.csv()`, `read.table()` e `openxlsx::read.xlsx()`, sendo o último do pacote `openxlsx`.

Para exemplificar como importar dados no R, vamos usar os dados de pinguins no Palmer Station, Antarctica [Palmer Penguins](#). Faremos o download diretamente do site da fonte dos dados.

Vamos antes escolher um diretório de trabalho com a função `setwd()`, e em seguida criar um diretório com a função `dir.create()` chamado “dados”. Em seguida, vamos mudar nosso diretório para essa pasta e criar mais um diretório chamado “tabelas”, e por fim, definir esse diretório para que o conteúdo do download seja armazenado ali.

```
## Escolher um diretório
setwd("/home/mude/data/github/livro_aer")

## Criar um diretório 'dados'
dir.create("dados")

## Escolher diretório 'dados'
setwd("dados")

## Criar um diretório 'tabelas'
dir.create("tabelas")

## Escolher diretório 'tabelas'
setwd("tabelas")
```

Agora podemos fazer o download do arquivo .csv e importar a tabela de dados.

```
## Download
download.file(url = "https://github.com/allisonhorst/palmerpenguins/blob/main/inst/extdata/penguins_raw.csv",
              destfile = "penguins_raw.csv", method = "curl")
```

Agora podemos importar a tabela de dados com a função `read.csv()`, atribuindo ao objeto `penguins_raw`.

```
## Importar a tabela
penguins_raw <- read.csv("dados/tabelas/penguins_raw.csv")
```

Esse arquivo foi criado com separador de decimais sendo `.` e separador de colunas sendo `,`. Caso tivesse sido criado com separador de decimais sendo `,` e separador de colunas sendo `;`, usariammos a função `read.csv2()`.

Para outros formatos, basta usar as outras funções apresentadas, atentando-se para os argumentos específicos de cada função.

Caso o download não funcione ou haja problemas com a importação, os dados estao também no pacote `palmerpenguins`.

```
## Importar os dados pelo pacote palmerpenguins
penguins_raw <- palmerpenguins::penguins_raw
head(penguins_raw)
```

```
## # A tibble: 6 x 17
##   studyName `Sample Number` Species      Region Island Stage `Individual ID`
##   <chr>          <dbl> <chr>        <chr>  <chr> <chr> <chr>
## 1 PAL0708           1 Adelie Penguin~ Anvers Torge~ Adul~ N1A1
## 2 PAL0708           2 Adelie Penguin~ Anvers Torge~ Adul~ N1A2
## 3 PAL0708           3 Adelie Penguin~ Anvers Torge~ Adul~ N2A1
## 4 PAL0708           4 Adelie Penguin~ Anvers Torge~ Adul~ N2A2
## 5 PAL0708           5 Adelie Penguin~ Anvers Torge~ Adul~ N3A1
## 6 PAL0708           6 Adelie Penguin~ Anvers Torge~ Adul~ N3A2
## # i 10 more variables: `Clutch Completion` <chr>, `Date Egg` <date>,
## #   `Culmen Length (mm)` <dbl>, `Culmen Depth (mm)` <dbl>,
## #   `Flipper Length (mm)` <dbl>, `Body Mass (g)` <dbl>, Sex <chr>,
## #   `Delta 15 N (o/oo)` <dbl>, `Delta 13 C (o/oo)` <dbl>, Comments <chr>
```

#### 5.4.7 Conferência dos dados importados

Uma vez importados os dados para o R, geralmente antes de iniciarmos qualquer manipulação, visualização ou análise de dados, fazemos a conferência desses dados. Dentre todas as funções de verificação, destacamos a importância destas funções apresentadas abaixo para saber se as variáveis foram importadas e interpretadas corretamente e reconhecer erros de digitação, por exemplo:

```
penguins <- palmerpenguins::penguins
```

```
## Primeiras linhas
```

```
head(penguins)
```

```
## # A tibble: 6 x 8
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>          <dbl>        <dbl>            <int>       <int>
## 1 Adelie  Torgersen     39.1         18.7            181        3750
## 2 Adelie  Torgersen     39.5         17.4            186        3800
## 3 Adelie  Torgersen     40.3         18              195        3250
## 4 Adelie  Torgersen     NA            NA              NA          NA
## 5 Adelie  Torgersen     36.7         19.3            193        3450
## 6 Adelie  Torgersen     39.3         20.6            190        3650
## # i 2 more variables: sex <fct>, year <int>
```

```
## Últimas linhas
```

```
tail(penguins)
```

```
## # A tibble: 6 x 8
```

```

##   species     island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>      <fct>        <dbl>        <dbl>          <int>        <int>
## 1 Chinstrap Dream         45.7         17            195        3650
## 2 Chinstrap Dream         55.8         19.8           207        4000
## 3 Chinstrap Dream         43.5         18.1           202        3400
## 4 Chinstrap Dream         49.6         18.2           193        3775
## 5 Chinstrap Dream         50.8         19            210        4100
## 6 Chinstrap Dream         50.2         18.7           198        3775
## # i 2 more variables: sex <fct>, year <int>
## Número de linhas e colunas
nrow(pinguins)

## [1] 344
ncol(pinguins)

## [1] 8
dim(pinguins)

## [1] 344 8
## Nome das linhas e colunas
rownames(pinguins)
colnames(pinguins)

## Estrutura dos dados
str(pinguins)

## Resumo dos dados
summary(pinguins)

## Verificar NAs
any(is.na(pinguins))
which(is.na(pinguins))

## Remover as linhas com NAs
pinguins_na <- na.omit(pinguins)

```

Importante

A função `na.omit()` retira a **linha inteira** que possui algum NA, inclusive as colunas que possuem dados que você não tem interesse em excluir. Dessa forma, tenha em mente quais dados você realmente quer remover da sua tabela.

Além das funções apresentadas, recomendamos olhar os seguintes pacotes que ajudam na conferência dos dados importados: [Hmisc](#), [skimr](#) e [inspectDF](#).

#### 5.4.8 Exportar dados

Uma vez realizado as operações de manipulação ou tendo dados que foram analisados e armazenados num objeto no formato de data frame ou matriz, podemos exportar esses dados do R para o diretório que definimos anteriormente.

Para tanto, podemos utilizar funções de escrita de dados, como `write.csv()`, `write.table()` e `openxlsx::write.xlsx()`. Dois pontos são fundamentais: i) o nome do arquivo tem de estar entre aspas e no final dele deve constar a extensão que pretendemos que o arquivo tenha, e ii) é interessante utilizar os argumentos `row.names = FALSE` e `quote=FALSE`, para que o arquivo escrito não tenha o nome das

linhas ou aspas em todas as células, respectivamente.

```
## Exportar dados na extensão .csv
write.csv(pinguins_na, "pinguins_na.csv",
           row.names = FALSE, quote = FALSE)

## Exportar dados na extensão .txt
write.table(pinguins_na, "pinguins_na.txt",
            row.names = FALSE, quote = FALSE)

## Exportar dados na extensão .xlsx
openxlsx::write.xlsx(pinguins_na, "pinguins_na.xlsx",
                      row.names = FALSE, quote = FALSE)
```

## 5.5 Para se aprofundar

Listamos a seguir livros e links com material que recomendamos para seguir com sua aprendizagem em *R Base*.

### 5.5.1 Livros

Recomendamos aos (às) interessados(as) os livros: i) Crawley The R Book, ii) Davies The Book of R: A First Course in Programming and Statistics, iii) Gillespie e Lovelace Efficient R programming, iv) Holmes e Huber Modern Statistics for Modern Biology, v) Irizarry e Love Data Analysis for the Life Sciences with R, vi) James e colaboradores An Introduction to Statistical Learning: with Applications in R, vii) Kabacoff R in Action: Data analysis and graphics with R, viii) Matloff The Art of R Programming: A Tour of Statistical Software Design, ix) Long e Teator R Cookbook e x) Wickham Advanced R.

### 5.5.2 Links

Existem centenas de ferramentas online para aprender e explorar o R. Dentre elas, indicamos os seguintes links (em português e inglês):

#### Introdução ao R

- An Introduction to R - Douglas A, Roos D, Mancini F, Couto A, Lusseau D
- A (very) shortintroduction to R - Paul Torfs & Claudia Brauer
- R for Beginners - Emmanuel Paradis

#### Ciência de dados - Ciência de Dados em R - Curso-R - Data Science for Ecologists and Environmental Scientists - Coding Club

#### Estatística - Estatística Computacional com R - Mayer F. P., Bonat W. H., Zeviani W. M., Krainski E. T., Ribeiro Jr. P. J - Data Analysis and Visualization in R for Ecologists - Data Carpentry

#### Miscelânea

- Materiais sobre R - Beatriz Milz
- R resources (free courses, books, tutorials, & cheat sheets) - Paul van der Laken

## 5.6 Exercícios

**4.1** Use o R para verificar o resultado da operação  $7 + 7 \div 7 + 7 \times 7 - 7$ .

**4.2** Verifique através do R se  $3 \times 2^3$  é maior que  $2 \times 3^2$ .

**4.3** Crie dois objetos (qualquer nome) com os valores 100 e 300. Multiplique esses objetos (função `prod()`) e atribuam ao objeto `mult`. Faça o logaritmo natural (função `log()`) do objeto `mult` e atribuam ao objeto `ln`.

**4.4** Quantos pacotes existem no CRAN nesse momento? Execute essa combinação no Console: `nrow(available.packages(repos = "http://cran.r-project.org"))`.

**4.5** Instale o pacote `tidyverse` do CRAN.

**4.6** Escolha números para jogar na mega-sena usando o R, nomeando o objeto como `mega`. Lembrando: são 6 valores de 1 a 60 e atribuam a um objeto.

**4.7** Crie um fator chamado `tr`, com dois níveis (“cont” e “trat”) para descrever 300 locais de amostragem, 15 de cada tratamento. O fator deve ser dessa forma `cont, cont, cont, ..., cont, trat, trat, ..., trat`.

**4.8** Crie uma matriz chamada `ma`, resultante da disposição de um vetor composto por 130 valores aleatórios entre 0 e 10. A matriz deve conter 100 linhas e ser disposta por colunas.

**4.9** Crie um data frame chamado `df`, resultante da composição desses vetores:

- `id`: 1:30
- `sp`: `sp01, sp02, ..., sp29, sp30`
- `ab`: 30 valores aleatórios entre 0 a 5

**4.10** Crie uma lista com os objetos criados anteriormente: `mega, tr, ma` e `df`.

**4.11** Selecione os elementos ímpares do objeto `tr` e atribua ao objeto `tr_impar`.

**4.12** Selecione as linhas com ids pares do objeto `df` e atribua ao objeto `df_ids_par`.

**4.13** Faça uma amostragem de 10 linhas do objeto `df` e atribua ao objeto `df_amos10`. Use a função `set.seed()` para fixar a amostragem.

**4.14** Amostre 10 linhas do objeto `ma`, mas utilizando as linhas amostradas do `df_amos10` e atribua ao objeto `ma_amos10`.

**4.15** Una as colunas dos objetos `df_amos10` e `ma_amos10` e atribua ao objeto `dados_amos10`.

[Soluções dos exercícios.](#)

## 6 Tidyverse

Conteúdo copiado, com pequenas alterações e atualizações de [Capítulo 5 Tidyverse](#), do livro [Análises Ecológicas no R](#).

### Pré-requisitos do capítulo

Pacotes e dados que serão utilizados neste capítulo.

```
## Pacotes
library(tidyverse)
library(ggplot2)
library(purrr)
library(tibble)
library(dplyr)
library(tidyr)
library(stringr)
library(readr)
library(forcats)
library(palmerpenguins)
library(lubridate)

## Dados
penguins <- palmerpenguins::penguins
penguins_raw <- palmerpenguins::penguins_raw
#tidy_anfibios_locais <- ecodados::tidy_anfibios_locais
```

#### 6.1 Contextualização

Como todo idioma, a linguagem R tem “sotaques” diversos. Ademais, R vem passando por transformações nos últimos anos. Grande parte dessas mudanças estão dentro do paradigma de Ciência de Dados (*Data Science*), uma nova área de conhecimento que vem se moldando a partir do desenvolvimento da sociedade em torno da era digital e da grande quantidade de dados gerados e disponíveis pela internet, de onde advém os pilares das inovações tecnológicas: *Big Data*, *Machine Learning* e *Internet of Things*. A grande necessidade de computação para desenvolver esse novo paradigma colocaram linguagens de programação interpretadas como como R [R](#) como as principais linguagens frente a esses novos desafios. Linguagens de programação de código aberto são uma ótima escolha para diversos propósitos, pois são gratuitas, possuem grandes comunidades de desenvolvedores, e são relativamente fáceis de aprender e aplicar.

As expansões na utilização da linguagem R para a Ciência de Dados começaram a ser implementadas principalmente devido a um pesquisador: [Hadley Wickham](#), que iniciou sua contribuição à comunidade R com o desenvolvimento do já consagrado pacote [ggplot2](#) (Wickham 2016) para a composição de gráficos no R, baseado na gramática de gráficos (Wilkinson and Wills 2005). Depois disso, Wickham dedicou-se ao desenvolvimento do pensamento de uma nova abordagem dentro da manipulação de dados, denominada **Tidy Data** (Dados organizados) (Wickham 2014), na qual focou na limpeza e organização dos mesmos. A ideia postula que dados estão *tidy* quando: i) variáveis estão nas colunas, ii) observações estão nas linhas e iii) valores estão nas células, sendo que para esse último, não deve haver mais de um valor por célula.

A partir dessas ideias, o [tidyverse](#) foi operacionalizado no R como uma coleção de pacotes que atuam no fluxo de trabalho comum da ciência de dados: importação, manipulação, exploração, visualização, análise e comunicação de dados e análises (Wickham et al. 2019). O principal objetivo do *tidyverse* é aproximar a linguagem para melhorar a interação entre ser humano e computador sobre dados, de modo que os pacotes compartilham uma filosofia de design de alto nível e gramática, além da estrutura de dados de baixo nível (Wickham et al. 2019). As principais leituras sobre o tema no R são os artigos “Tidy Data” (Wickham 2014) e “Welcome to the Tidyverse” (Wickham et al. 2019), e o livro “[R for Data Science](#)” (Wickham and Grolemund 2017), além do [Tidyverse](#) que possui muito mais informações.

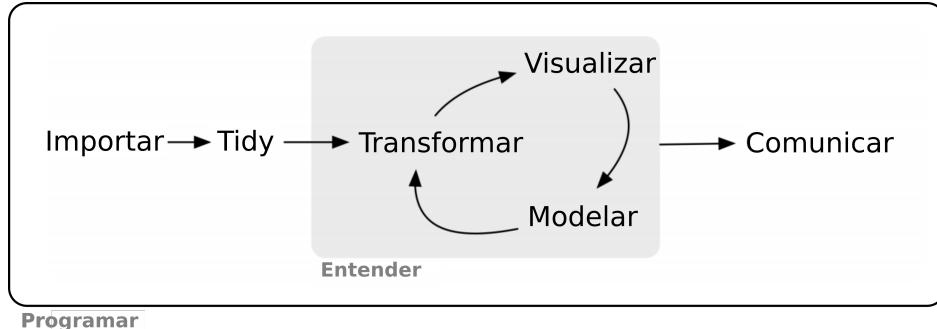


Figura 6.1: Modelo das ferramentas necessárias em um projeto típico de ciência de dados: importar, organizar, entender (transformar, visualizar, modelar) e comunicar, envolto à essas ferramentas está a programação.

## 6.2 *tidyverse*

Uma vez instalado e carregado, o pacote **tidyverse** disponibiliza um conjunto de ferramentas através de vários pacotes. Esses pacotes compartilham uma filosofia de design, gramática e estruturas. Podemos entender o *tidyverse* como um “dialeto novo” para a linguagem R, onde *tidy* quer dizer organizado, arrumado, ordenado, e *verse* é universo. A seguir, listamos os principais pacotes e suas funcionalidades.

- **readr**: importa dados tabulares (e.g. `.csv` e `.txt`)
- **tibble**: implementa a classe `tibble`
- **tidyr**: transformação de dados para `tidy`
- **dplyr**: manipulação de dados
- **stringr**: manipulação de caracteres
- **forcats**: manipulação de fatores
- **ggplot2**: possibilita a visualização de dados
- **purrr**: disponibiliza ferramentas para programação funcional

Além dos pacotes principais, fazemos também menção a outros pacotes que estão dentro dessa abordagem e que trataremos ainda neste capítulo, em outro momento do livro, ou que você leitor(a) deve se familiarizar. Alguns pacotes compõem o *tidyverse* outros são mais gerais, entretanto, todos estão envolvidos de alguma forma com ciência de dados.

- **readxl** e **writexl**: importa e exporta dados tabulares (`.xlsx`)
- **janitor**: examina e limpa dados sujos
- **DBI**: interface de banco de dados R
- **haven**: importa e exporta dados do SPSS, Stata e SAS
- **httr**: ferramentas para trabalhar com URLs e HTTP
- **rvest**: coleta facilmente (raspagem de dados) páginas da web
- **xml2**: trabalha com arquivos XML
- **jsonlite**: um analisador e gerador JSON simples e robusto para R
- **hms**: hora do dia
- **lubridate**: facilita o tratamento de datas
- **magrittr**: provê os operadores pipe (`%>%`, `%$%`, `%<>%`)
- **glue**: facilita a combinação de dados e caracteres
- **rmarkdown**: cria documentos de análise dinâmica que combinam código, saída renderizada (como figuras) e texto
- **knitr**: projetado para ser um mecanismo transparente para geração de relatórios dinâmicos com R
- **shiny**: framework de aplicativo Web para R
- **flexdashboard**: painéis interativos para R
- **here**: facilita a definição de diretórios
- **usethis**: automatiza tarefas durante a configuração e desenvolvimento de projetos (Git, ‘GitHub’ e Projetos RStudio)

- **data.table**: pacote que fornece uma versão de alto desempenho do **data.frame** (importar, manipular e expor)
- **reticulate**: pacote que fornece ferramentas para integrar Python e R
- **sparklyr**: interface R para Apache Spark
- **broom**: converte objetos estatísticos em tibbles organizados
- **modelr**: funções de modelagem que funcionam com o pipe
- **tidymodels**: coleção de pacotes para modelagem e aprendizado de máquina usando os princípios do tidyverse

Destacamos a grande expansão e aplicabilidade dos pacotes **rmarkdown**, **knitr** e **bookdown**, que permitiram a escrita deste livro usando essas ferramentas e linguagem de marcação, chamada **Markdown**.

Para instalar os principais pacotes que integram o *tidyverse* podemos instalar o pacote **tidyverse**.

```
## Instalar o pacote tidyverse
install.packages("tidyverse")
```

Quando carregamos o pacote **tidyverse** podemos notar uma mensagem indicando quais pacotes foram carregados, suas respectivas versões e os conflitos com outros pacotes.

```
## Carregar o pacote tidyverse
library(tidyverse)
```

Podemos ainda listar todos os pacotes do *tidyverse* com a função **tidyverse::tidyverse\_packages()**.

```
## Listar todos os pacotes do tidyverse
tidyverse::tidyverse_packages()
```

```
## [1] "broom"          "conflicted"     "cli"            "dbplyr"
## [5] "dplyr"          "dtplyr"         "forcats"        "ggplot2"
## [9] "googledrive"   "googlesheets4" "haven"          "hms"
## [13] "httr"           "jsonlite"       "lubridate"      "magrittr"
## [17] "modelr"         "pillar"         "purrr"          "ragg"
## [21] "readr"          "readxl"         "reprex"         "rlang"
## [25] "rstudioapi"    "rvest"          "stringr"        "tibble"
## [29] "tidyverse"      "xml2"           "tidyverse"
```

Também podemos verificar se os pacotes estão atualizados, senão, podemos atualizá-los com a função **tidyverse::tidyverse\_update()**.

```
## Verificar e atualizar os pacotes do tidyverse
tidyverse::tidyverse_update(repos = "http://cran.us.r-project.org")
```

Todas as funções dos pacotes *tidyverse* usam **fonte minúscula** e **\_ (underscore)** para separar os nomes internos das funções, seguindo a mesma sintaxe do Python (“Snake Case”). Neste sentido de padronização, é importante destacar ainda que existe um guia próprio para que os scripts sigam a recomendação de padronização, o **The tidyverse style guide**, criado pelo próprio Hadley Wickham. Para pessoas que desenvolvem funções e pacotes existe o **Tidyverse design guide** criado pelo *Tidyverse team*.

```
## Funções no formato snake case
read_csv()
read_tsv()
as_tibble()
left_join()
group_by()
```

Por fim, para evitar possíveis conflitos de funções com o mesmo nome entre pacotes, recomendamos fortemente o hábito de usar as funções precedidas do operador **::** e o respectivo pacote. Assim, garante-se que a função utilizada é referente ao pacote daquela função. Segue um exemplo com as funções apresentadas anteriormente.

```
## Funções seguidas de seus respectivos pacotes
readr::read_csv()
readr::read_tsv()
tibble::as_tibble()
dplyr::left_join()
dplyr::group_by()
```

Seguindo essas ideias do novo paradigma da **Ciência de Dados**, outro conjunto de pacotes foi desenvolvido, chamado de [tidymodels](#) que atuam no fluxo de trabalho da análise de dados em ciência de dados: separação e reamostragem, pré-processamento, ajuste de modelos e métricas de performance de ajustes. Por razões de espaço e especificidade, não entraremos em detalhes desses pacotes.

Um projeto de ciência de dados envolve uma sequência de passos:

1. Importar os dados
2. Organizar os dados
3. Entender os dados (transformar, visualizar, modelar)
4. Comunicar os resultados Nas próximas seções, veremos como cada um desses passos pode ser realizado com funções de pacotes específicos.

### 6.3 readr, readxl e writexl

Dado que possuímos um conjunto de dados e que geralmente esse conjunto de dados estará no formato tabular com umas das extensões: .csv, .txt ou .xlsx, usaremos o pacote **readr** ou **readxl** para importar esses dados para o R. Esses pacotes leem e escrevem grandes arquivos de forma mais rápida, além de fornecerem medidores de progresso de importação e exportação, e imprimir a informação dos modos das colunas no momento da importação. Outro ponto bastante positivo é que também classificam automaticamente o modo dos dados de cada coluna, i.e., se uma coluna possui dados numéricos ou apenas texto, essa informação será considerada para classificar o modo da coluna toda. A classe do objeto atribuído quando lido por esses pacotes é automaticamente um **tibble**, que veremos melhor na seção seguinte. Todas as funções deste pacote são listadas na [página de referência](#) do pacote.

Usamos as funções **readr::read\_csv()** e **readr::write\_csv()** para importar e exportar arquivos .csv do R, respectivamente. Para dados com a extensão .txt, podemos utilizar as funções **readr::read\_tsv()** ou ainda **readr::read\_delim()**. Para arquivos tabulares com a extensão .xlsx, temos de instalar e carregar dois pacotes adicionais: **readxl** e **writexl**, dos quais usaremos as função **readxl::read\_excel()**, para importar dados, atentado para o fato de podermos indicar a aba/planilha com os dados com o argumento **sheet**, e **writexl::write\_xlsx()** para exportar.

Se o arquivo .csv foi criado com separador de decimais sendo . e separador de colunas sendo , usamos as funções listadas acima normalmente. Caso seja criado com separador de decimais sendo , e separador de colunas sendo ;, devemos usar a função **readr::read\_csv2()** para importar e **readr::write\_csv2()** para exportar nesse formato, que é mais comum no Brasil.

Para se aprofundar no tema, recomendamos a leitura do Capítulo 11 [Data import](#) de Wickham & Grolemund (2017).

### 6.4 tibble

O **tibble** (**tbl\_sf**) é uma versão aprimorada do data frame (**data.frame**). Ele é a classe aconselhada para que as funções do tidyverse funcionem melhor sobre conjuntos de dados tabulares importados para o R.

Geralmente, quando utilizamos funções tidyverse para importar dados para o R, é essa classe que os dados adquirem depois de serem importados. Além da importação de dados, podemos criar um tibble no R usando a função **tibble::tibble()**, semelhante ao uso da função **data.frame()**. Podemos ainda converter um **data.frame** para um **tibble** usando a função **tibble::as\_tibble()**. Entretanto, em alguns momentos

precisaremos da classe `data.frame` para algumas funções específicas, e podemos converter um `tibble` para `data.frame` usando a função `tibble::as_data_frame()`.

Existem duas diferenças principais no uso do `tibble` e do `data.frame`: impressão e subconjunto. Objetos da classe `tibbles` possuem um método de impressão que mostra a contagem do número de linhas e colunas, e apenas as primeiras 10 linhas e todas as colunas que couberem na tela no console, além dos modos ou tipos das colunas. Dessa forma, cada coluna ou variável, pode ser do modo numbers (`int` ou `dbl`), character (`chr`), logical (`lgl`), factor (`fctr`), date + time (`dttm`) e date (`date`), além de outras [inúmeras possibilidades](#).

Todas as funções deste pacote são listadas na [página de referência](#) do pacote.

```
## Tibble - impressão
tidy_anfibios_locais
```

Para o subconjunto, como vimos no Capítulo 5, para selecionar colunas e linhas de objetos bidimensionais podemos utilizar os operadores `[]` ou `[[]]`, associado com números separados por vírgulas ou o nome da coluna entre aspas, e o operador `$` para extrair uma coluna pelo seu nome. Comparando um `data.frame` a um `tibble`, o último é mais rígido na seleção das colunas: ele nunca faz correspondência parcial e gera um aviso se a coluna que você está tentando acessar não existe.

```
## Tibble - subconjunto
tidy_anfibios_locais$ref
```

Por fim, podemos “espiar” os dados utilizando a função `tibble::glimpse()` para ter uma noção geral de número de linhas, colunas, e conteúdo de todas as colunas. Essa é a função `tidyverse` da função `R Base str()`.

```
## Espiar os dados
tibble::glimpse(tidy_anfibios_locais[, 1:10])
```

Para se aprofundar no tema, recomendamos a leitura do Capítulo 10 [Tibbles](#) de Wickham & Grolemund (2017).

## 6.5 pipe: base (|>) e magrittr (%>%)

O operador pipe permite o encadeamento de várias funções, eliminando a necessidade de criar objetos para armazenar resultados intermediários. Dessa forma, pipes são uma ferramenta poderosa para expressar uma sequência de múltiplas operações.

Aqui usamos a versão nativo (`|>`), que vem na instalação, que é mais “simples”. O operador pipe original `%>%` vem do pacote `magrittr`. R 4.1.0 introduziu um operador de pipe nativo “`|>`”. O comportamento do pipe nativo é em geral o mesmo do pipe `%>%` fornecido pelo pacote `magrittr`. Ambos os operadores (`|>` e `%>%`) permitem “canalizar” um objeto para uma função ou expressão de chamada, permitindo assim expressar uma sequência de operações que transformam um objeto.

Felizmente, não há necessidade de se comprometer inteiramente com um pipe ou outro - você pode usar o pipe nativo na maioria dos casos, e usar o pipe `magrittr` quando realmente precisar de seus recursos especiais. O pipe torna os códigos em R mais simples, pois podemos realizar múltiplas operações em uma única linha. Ele captura o resultado de uma declaração e o torna a primeira entrada da próxima declaração, então podemos pensar como “EM SEGUIDA FAÇA” ao final de cada linha de código.

A principal vantagem do uso dos pipes é facilitar a depuração (*debugging* - achar erros) nos códigos, porque seu uso torna a linguagem R mais próxima do que falamos e pensamos, uma vez que evita o uso de funções dentro de funções (funções compostas, lembra-se do fog e gof do ensino médio? Evitamos eles aqui também).

Para deixar esse tópico menos estranho a quem possa ver essa operação pela primeira vez, vamos fazer alguns exemplos.

```
## R Base - sem pipe
sqrt(sum(1:100))
```

```
## [1] 71.06335
```

```
## com pipe
```

```
1:100 |>
```

```
sum() |>
```

```
sqrt()
```

```
## [1] 71.06335
```

Essas operações ainda estão simples, vamos torná-las mais complexas com várias funções compostas. É nesses casos que a propriedade organizacional do uso do pipe emerge: podemos facilmente ver o encadeamento de operações, onde cada função é disposta numa linha. Apenas um adendo: a função `set.seed()` fixa a amostragem de funções que geram valores aleatórios, como é o caso da função `rpois()`.

```
## Fixar amostragem
```

```
set.seed(42)
```

```
## R Base - sem pipe
```

```
ve <- sum(sqrt(sin(log10(rpois(100, 10)))))
```

```
ve
```

```
## [1] 91.27018
```

```
## Fixar amostragem
```

```
set.seed(42)
```

```
## com pipe
```

```
ve <- rpois(100, 10) |>
```

```
log10() |>
```

```
sin() |>
```

```
sqrt() |>
```

```
sum()
```

```
ve
```

```
## [1] 91.27018
```

O uso do pipe vai se tornar especialmente útil quando seguirmos para os pacotes das próximas duas seções: `tidyR` e `dplyr`. Com esses pacotes faremos operações em linhas e colunas de nossos dados tabulares, então podemos encadear uma série de funções para manipulação, limpeza e análise de dados.

Há ainda três outras variações do pipe que podem ser úteis em alguns momentos, mas que para funcionar precisam que o pacote `magrittr` esteja carregado:

- `%T>%`: retorna o lado esquerdo em vez do lado direito da operação
- `%$%`: “explode” as variáveis em um quadro de dados
- `%<>%`: permite atribuição usando pipes

Para se aprofundar no tema, recomendamos a leitura do Capítulo 18 Pipes de Wickham & Grolemund (2017).

Importante

A partir da versão do R 4.1+ (18/05/2021), o operador pipe `|>` se tornou nativo do R. Podendo ser inserido com o mesmo atalho `Ctrl + Shift + M`, mas necessitando uma mudança de opção em `Tools > Global Options > Code > [x] Use native pipe operator, |> (requires R 4.1+)`, requerendo que o RStudio esteja numa versão igual ou superior a 1.4.17+.

## 6.6 tidyR

Um conjunto de dados `tidy` (organizados) são mais fáceis de manipular, modelar e visualizar. Um conjunto de dados está no formato `tidy` ou não, dependendo de como linhas, colunas e células são combinadas com

observações, variáveis e valores. Nos dados tidy, as variáveis estão nas colunas, observações estão nas linhas e valores estão nas células, sendo que para esse último, não deve haver mais de um valor por célula (Figura 6.2).

1. Cada variável em uma coluna
2. Cada observação em uma linha
3. Cada valor como uma célula

country	year	cases	population
Afghanistan	1999	745	1937071
Afghanistan	2000	2666	20595360
Brazil	1999	3737	17206362
Brazil	2000	80488	17404898
China	1999	212258	127215272
China	2000	21166	128028583

Variáveis

Figura 6.2: As três regras que tornam um conjunto de dados \*tidy\*.

Todas as funções deste pacote são listadas na [página de referência](#) do pacote.

Para realizar diversas transformações nos dados, a fim de ajustá-los ao formato `tidy` existe uma série de funções para: unir colunas, separar colunas, lidar com valores faltantes (`NA`), transformar a base de dados de formato longo para largo (ou vice-e-versa), além de outras [funções específicas](#).

- `unite()`: junta dados de múltiplas colunas em uma coluna
- `separate()`: separa caracteres em múltiplas colunas

- `separate_rows()`: separa caracteres em múltiplas colunas e linhas
- `drop_na()`: retira linhas com NA do conjunto de dados
- `replace_na()`: substitui NA do conjunto de dados
- `pivot_wider()`: transforma um conjunto de dados longo (*long*) para largo (*wide*)
- `pivot_longer()`: transforma um conjunto de dados largo (*wide*) para longo (*long*)

### 6.6.1 palmerpenguins

Para exemplificar o funcionamento dessas funções, usaremos os dados de medidas de pinguins chamados `palmerpenguins`, disponíveis no pacote `palmerpenguins`.

```
## Instalar o pacote
install.packages("palmerpenguins")
```

Esses dados foram coletados e disponibilizados pela Dra. Kristen Gorman e pela Palmer Station, Antarctica LTER, membro da Long Term Ecological Research Network.

O pacote `palmerpenguins` contém dois conjuntos de dados. Um é chamado de `penguins` e é uma versão simplificada dos dados brutos. O segundo conjunto de dados é `penguins_raw` e contém todas as variáveis e nomes originais. Ambos os conjuntos de dados contêm dados para 344 pinguins, de três espécies diferentes, coletados em três ilhas no arquipélago de Palmer, na Antártica. Destacamos também a versão traduzida desses dados para o português, disponível no pacote `dados`.

Vamos utilizar principalmente o conjunto de dados `penguins_raw`, que é a versão dos dados brutos.

```
## Carregar o pacote palmerpenguins
library(palmerpenguins)
```

Podemos ainda verificar os dados, pedindo uma ajuda de cada um dos objetos.

```
## Ajuda dos dados
?penguins
?penguins_raw
```

### 6.6.2 glimpse()

Primeiramente, vamos observar os dados e utilizar a função `tibble::glimpse()` para ter uma noção geral dos dados.

```
## Visualizar os dados
penguins_raw
```

```
## # A tibble: 344 x 17
##   studyName `Sample Number` Species      Region Island Stage `Individual ID`
##   <chr>        <dbl> <chr>       <chr> <chr> <chr> <chr>
## 1 PAL0708          1 Adelie Penguin~ Anvers Torge~ Adul~ N1A1
## 2 PAL0708          2 Adelie Penguin~ Anvers Torge~ Adul~ N1A2
## 3 PAL0708          3 Adelie Penguin~ Anvers Torge~ Adul~ N2A1
## 4 PAL0708          4 Adelie Penguin~ Anvers Torge~ Adul~ N2A2
## 5 PAL0708          5 Adelie Penguin~ Anvers Torge~ Adul~ N3A1
## 6 PAL0708          6 Adelie Penguin~ Anvers Torge~ Adul~ N3A2
## 7 PAL0708          7 Adelie Penguin~ Anvers Torge~ Adul~ N4A1
## 8 PAL0708          8 Adelie Penguin~ Anvers Torge~ Adul~ N4A2
## 9 PAL0708          9 Adelie Penguin~ Anvers Torge~ Adul~ N5A1
## 10 PAL0708         10 Adelie Penguin~ Anvers Torge~ Adul~ N5A2
## # i 334 more rows
## # i 10 more variables: `Clutch Completion` <chr>, `Date Egg` <date>,
## #   `Culmen Length (mm)` <dbl>, `Culmen Depth (mm)` <dbl>,
```

```

## # `Flipper Length (mm)` <dbl>, `Body Mass (g)` <dbl>, Sex <chr>,
## # `Delta 15 N (o/oo)` <dbl>, `Delta 13 C (o/oo)` <dbl>, Comments <chr>
## Espiar os dados
dplyr::glimpse(penguins_raw)

## Rows: 344
## Columns: 17
## $ studyName          <chr> "PAL0708", "PAL0708", "PAL0708", "PAL0708", "P~  

## $ `Sample Number`    <dbl> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, ~  

## $ Species            <chr> "Adelie Penguin (Pygoscelis adeliae)", "Adelie~  

## $ Region             <chr> "Anvers", "Anvers", "Anvers", "Anver~  

## $ Island              <chr> "Torgersen", "Torgersen", "Torgersen", "Torger~  

## $ Stage               <chr> "Adult, 1 Egg Stage", "Adult, 1 Egg Stage", "A~  

## $ `Individual ID`   <chr> "N1A1", "N1A2", "N2A1", "N2A2", "N3A1", "N3A2"~  

## $ `Clutch Completion` <chr> "Yes", "Yes", "Yes", "Yes", "Yes", "Yes", "No"~  

## $ `Date Egg`         <date> 2007-11-11, 2007-11-11, 2007-11-16, 2007-11-1~  

## $ `Culmen Length (mm)` <dbl> 39.1, 39.5, 40.3, NA, 36.7, 39.3, 38.9, 39.2, ~  

## $ `Culmen Depth (mm)` <dbl> 18.7, 17.4, 18.0, NA, 19.3, 20.6, 17.8, 19.6, ~  

## $ `Flipper Length (mm)` <dbl> 181, 186, 195, NA, 193, 190, 181, 195, 193, 19~  

## $ `Body Mass (g)`     <dbl> 3750, 3800, 3250, NA, 3450, 3650, 3625, 4675, ~  

## $ Sex                 <chr> "MALE", "FEMALE", "FEMALE", NA, "FEMALE", "MAL~  

## $ `Delta 15 N (o/oo)` <dbl> NA, 8.94956, 8.36821, NA, 8.76651, 8.66496, 9.~  

## $ `Delta 13 C (o/oo)` <dbl> NA, -24.69454, -25.33302, NA, -25.32426, -25.2~  

## $ Comments            <chr> "Not enough blood for isotopes.", NA, NA, "Adu~
```

### 6.6.3 unite()

Primeiramente, vamos exemplificar como juntar e separar colunas. Vamos utilizar a função `tidyverse::unite()` para unir as colunas. Há diversos parâmetros para alterar como esta função funciona, entretanto, é importante destacar três deles: `col` nome da coluna que vai receber as colunas unidas, `sep` indicando o caractere separador das colunas unidas, e `remove` para uma resposta lógica se as colunas unidas são removidas ou não. Vamos unir as colunas “Region” e “Island” na nova coluna “region\_island”.

```

## Unir colunas
penguins_raw_unir <- tidyverse::unite(data = penguins_raw,
                                         col = "region_island",
                                         Region:Island,
                                         sep = ",",
                                         remove = FALSE)
head(penguins_raw_unir[, c("Region", "Island", "region_island")])

## # A tibble: 6 x 3
##   Region Island   region_island
##   <chr>  <chr>      <chr>
## 1 Anvers Torgersen Anvers, Torgersen
## 2 Anvers Torgersen Anvers, Torgersen
## 3 Anvers Torgersen Anvers, Torgersen
## 4 Anvers Torgersen Anvers, Torgersen
## 5 Anvers Torgersen Anvers, Torgersen
## 6 Anvers Torgersen Anvers, Torgersen
```

### 6.6.4 separate()

De forma contrária, podemos utilizar as funções `tidyverse::separate()` e `tidyverse::separate_rows()` para separar elementos de uma coluna em mais colunas. Respectivamente, a primeira função separa uma coluna

em novas colunas conforme a separação, e a segunda função separa uma coluna, distribuindo os elementos nas linhas. Novamente, há diversos parâmetros para mudar o comportamento dessas funções, mas destacaremos aqui quatro deles: `col` coluna a ser separada, `into` os nomes das novas colunas, `sep` indicando o caractere separador das colunas, e `remove` para uma resposta lógica se as colunas separadas são removidas ou não. Vamos separar a coluna “Stage” nas colunas “stage” e “egg\_stage”.

```
## Separar colunas
penguins_raw_separar <- tidyverse::separate(data = penguins_raw,
                                             col = Stage,
                                             into = c("stage", "egg_stage"),
                                             sep = ",",
                                             remove = FALSE)
head(penguins_raw_separar[, c("Stage", "stage", "egg_stage")])

## # A tibble: 6 x 3
##   Stage           stage egg_stage
##   <chr>          <chr> <chr>
## 1 Adult, 1 Egg Stage Adult 1 Egg Stage
## 2 Adult, 1 Egg Stage Adult 1 Egg Stage
## 3 Adult, 1 Egg Stage Adult 1 Egg Stage
## 4 Adult, 1 Egg Stage Adult 1 Egg Stage
## 5 Adult, 1 Egg Stage Adult 1 Egg Stage
## 6 Adult, 1 Egg Stage Adult 1 Egg Stage

## Separar colunas em novas linhas
penguins_raw_separar_linhas <- tidyverse::separate_rows(data = penguins_raw,
                                                       Stage,
                                                       sep = ", ")
head(penguins_raw_separar_linhas[, c("studyName", "Sample Number", "Species",
                                     "Region", "Island", "Stage")])

## # A tibble: 6 x 6
##   studyName `Sample Number` Species           Region Island Stage
##   <chr>          <dbl> <chr>           <chr> <chr> <chr>
## 1 PAL0708          1 Adelie Penguin (Pygoscelis ad~ Anvers Torge~ Adult
## 2 PAL0708          1 Adelie Penguin (Pygoscelis ad~ Anvers Torge~ 1 Eg~
## 3 PAL0708          2 Adelie Penguin (Pygoscelis ad~ Anvers Torge~ Adult
## 4 PAL0708          2 Adelie Penguin (Pygoscelis ad~ Anvers Torge~ 1 Eg~
## 5 PAL0708          3 Adelie Penguin (Pygoscelis ad~ Anvers Torge~ Adult
## 6 PAL0708          3 Adelie Penguin (Pygoscelis ad~ Anvers Torge~ 1 Eg~
```

## 6.6.5 `drop_na()` e `replace_na()`

*Valor faltante* (NA) é um tipo especial de elemento que são discutidos no Capítulo 5 e são relativamente comuns em conjuntos de dados. Em *R Base*, vimos algumas formas de lidar com esse tipo de elemento. No formato `tidyverse`, existem também várias formas de lidar com eles, mas aqui focaremos nas funções `tidyverse::drop_na()` e `tidyverse::replace_na()`, para retirar linhas e substituir esses valores, respectivamente.

```
## Remover todas as linhas com NAs
penguins_raw_todas_na <- tidyverse::drop_na(data = penguins_raw)
head(penguins_raw_todas_na)

## # A tibble: 6 x 17
##   studyName `Sample Number` Species           Region Island Stage `Individual ID`
##   <chr>          <dbl> <chr>           <chr> <chr> <chr> <chr>
## 1 PAL0708          7 Adelie Penguin~ Anvers Torge~ Adul~ N4A1
## 2 PAL0708          8 Adelie Penguin~ Anvers Torge~ Adul~ N4A2
```

```

## 3 PAL0708          29 Adelie Penguin~ Anvers Biscoe Adul~ N18A1
## 4 PAL0708          30 Adelie Penguin~ Anvers Biscoe Adul~ N18A2
## 5 PAL0708          39 Adelie Penguin~ Anvers Dream   Adul~ N25A1
## 6 PAL0809          69 Adelie Penguin~ Anvers Torge~ Adul~ N32A1
## # i 10 more variables: `Clutch Completion` <chr>, `Date Egg` <date>,
## #   `Culmen Length (mm)` <dbl>, `Culmen Depth (mm)` <dbl>,
## #   `Flipper Length (mm)` <dbl>, `Body Mass (g)` <dbl>, Sex <chr>,
## #   `Delta 15 N (o/oo)` <dbl>, `Delta 13 C (o/oo)` <dbl>, Comments <chr>
## Remover linhas de colunas específicas com NAs
penguins_raw_colunas_na <- tidyverse::drop_na(data = penguins_raw,
                                             any_of("Comments"))
head(penguins_raw_colunas_na[, "Comments"])

## # A tibble: 6 x 1
##   Comments
##   <chr>
## 1 Not enough blood for isotopes.
## 2 Adult not sampled.
## 3 Nest never observed with full clutch.
## 4 Nest never observed with full clutch.
## 5 No blood sample obtained.
## 6 No blood sample obtained for sexing.

## Substituir NAs por outro valor
penguins_raw_subs_na <- tidyverse::replace_na(data = penguins_raw,
                                              list(Comments = "Unknown"))
head(penguins_raw_subs_na[, "Comments"])

## # A tibble: 6 x 1
##   Comments
##   <chr>
## 1 Not enough blood for isotopes.
## 2 Unknown
## 3 Unknown
## 4 Adult not sampled.
## 5 Unknown
## 6 Unknown

```

### 6.6.6 pivot\_longer() e pivot\_wider()

Por fim, trataremos da pivotagem ou remodelagem de dados. Veremos como mudar o formato do nosso conjunto de dados de longo (*long*) para largo (*wide*) e vice-versa. Primeiramente, vamos ver como partir de um dado longo (*long*) e criar um dado largo (*wide*). Essa é uma operação semelhante à “Tabela Dinâmica” das planilhas eletrônicas. Consiste em usar uma coluna para distribuir seus valores em outras colunas, de modo que os valores dos elementos são preenchidos corretamente, reduzindo assim o número de linhas e aumentando o número de colunas. Essa operação é bastante comum em Ecologia de Comunidades, quando queremos transformar uma lista de espécies em uma matriz de comunidades, com várias espécies nas colunas. Para realizar essa operação, usamos a função `tidyverse::pivot_wider()`. Dos diversos parâmetros que podem compor essa função, dois deles são fundamentais: `names_from` que indica a coluna de onde os nomes serão usados e `values_from` que indica a coluna com os valores.

```

## Selecionar colunas
penguins_raw_sel_col <- penguins_raw[, c(2, 3, 13)]
head(penguins_raw_sel_col)

## # A tibble: 6 x 3

```

```

## `Sample Number` Species `Body Mass (g)`
## <dbl> <chr> <dbl>
## 1 Adelie Penguin (Pygoscelis adeliae) 3750
## 2 Adelie Penguin (Pygoscelis adeliae) 3800
## 3 Adelie Penguin (Pygoscelis adeliae) 3250
## 4 Adelie Penguin (Pygoscelis adeliae) NA
## 5 Adelie Penguin (Pygoscelis adeliae) 3450
## 6 Adelie Penguin (Pygoscelis adeliae) 3650
## Pivatar para largo
penguins_raw_pivot_wider <- tidyr::pivot_wider(data = penguins_raw_sel_col,
                                                names_from = Species,
                                                values_from = `Body Mass (g)`)

head(penguins_raw_pivot_wider)

## # A tibble: 6 x 4
## `Sample Number` `Adelie Penguin (Pygoscelis adeliae)` `Gentoo penguin (Pygo~1
## <dbl> <dbl> <dbl> <dbl>
## 1 1 3750 4500
## 2 2 3800 5700
## 3 3 3250 4450
## 4 4 NA 5700
## 5 5 3450 5400
## 6 6 3650 4550
## # i abbreviated name: 1: `Gentoo penguin (Pygoscelis papua)`
## # i 1 more variable: `Chinstrap penguin (Pygoscelis antarctica)` <dbl>

```

De modo oposto, podemos partir de um conjunto de dados largo (*wide*), ou seja, com várias colunas, e queremos que essas colunas preencham uma única coluna, e que os valores antes espalhados nessas várias colunas sejam adicionados um embaixo do outro, numa única coluna, no formato longo (*long*). Para essa operação, podemos utilizar a função `tidyr::pivot_longer()`. Novamente, dos diversos parâmetros que podem compor essa função, três deles são fundamentais: `cols` indicando as colunas que serão usadas para serem pivotadas, `names_to` que indica a coluna de onde os nomes serão usados e `values_to` que indica a coluna com os valores.

```

## Selecionar colunas
penguins_raw_sel_col <- penguins_raw[, c(2, 3, 10:13)]
head(penguins_raw_sel_col)

## # A tibble: 6 x 6
## `Sample Number` Species `Culmen Length (mm)` `Culmen Depth (mm)` <dbl>
## <dbl> <chr> <dbl> <dbl> <dbl>
## 1 1 Adelie Penguin (Py~ 39.1 18.7
## 2 2 Adelie Penguin (Py~ 39.5 17.4
## 3 3 Adelie Penguin (Py~ 40.3 18
## 4 4 Adelie Penguin (Py~ NA NA
## 5 5 Adelie Penguin (Py~ 36.7 19.3
## 6 6 Adelie Penguin (Py~ 39.3 20.6
## # i 2 more variables: `Flipper Length (mm)` <dbl>, `Body Mass (g)` <dbl>
## Pivatar para largo
penguins_raw_pivot_longer <- tidyr::pivot_longer(data = penguins_raw_sel_col,
                                                 cols = `Culmen Length (mm)`:`Body Mass (g)`,
                                                 names_to = "medidas",
                                                 values_to = "valores")

head(penguins_raw_pivot_longer)

```

```

## # A tibble: 6 x 4
##   `Sample Number` Species           medidas      valores
##   <dbl> <chr>          <chr>        <dbl>
## 1 1 Adelie Penguin (Pygoscelis adeliae) Culmen Length (~ 39.1
## 2 1 Adelie Penguin (Pygoscelis adeliae) Culmen Depth (mm~ 18.7
## 3 1 Adelie Penguin (Pygoscelis adeliae) Flipper Length ~ 181
## 4 1 Adelie Penguin (Pygoscelis adeliae) Body Mass (g) 3750
## 5 2 Adelie Penguin (Pygoscelis adeliae) Culmen Length (~ 39.5
## 6 2 Adelie Penguin (Pygoscelis adeliae) Culmen Depth (mm~ 17.4

```

Para se aprofundar no tema, recomendamos a leitura do Capítulo 12 [Tidy data](#) de Wickham & Grolemund (2017).

## 6.7 dplyr

O **dplyr** é um pacote que facilita a manipulação de dados, com uma gramática simples e flexível (por exemplo, como filtragem, reordenamento, seleção, entre outras). Ele foi construído com o intuito de obter uma forma mais rápida e expressiva de manipular dados tabulares. O **tibble** é a versão de data frame mais conveniente para se usar com pacote **dplyr**.

Todas as funções deste pacote são listadas na [página de referência](#) do pacote.

### 6.7.1 Gramática

Sua gramática simples contém funções verbais para manipulação de dados, baseada em:

- Verbos: `mutate()`, `select()`, `filter()`, `arrange()`, `summarise()`, `slice()`, `rename()`, etc.
- Replicação: `across()`, `if_any()`, `if_all()`, `where()`, `starts_with()`, `ends_with()`, `contains()`, etc.
- Agrupamento: `group_by()` e `ungroup()`
- Junções: `inner_join()`, `full_join()`, `left_join()`, `right_join()`, etc.
- Combinações: `bind_rows()` e `bind_cols()`
- Resumos, contagem e seleção: `n()`, `n_distinct()`, `first()`, `last()`, `nth()`, etc.

Existe uma série de funções para realizar a manipulação dos dados, com diversas finalidades: manipulação de uma tabela, manipulação de duas tabelas, replicação, agrupamento, funções de vetores, além de muitas outras [funções específicas](#).

- `relocate()`: muda a ordem das colunas
- `rename()`: muda o nome das colunas
- `select()`: seleciona colunas pelo nome ou posição
- `pull()`: seleciona uma coluna como vetor
- `mutate()`: adiciona novas colunas ou resultados em colunas existentes
- `arrange()`: reordena as linhas com base nos valores de colunas
- `filter()`: seleciona linhas com base em valores de colunas
- `slice()`: seleciona linhas de diferentes formas
- `distinct()`: remove linhas com valores repetidos com base nos valores de colunas
- `count()`: conta observações para um grupo com base nos valores de colunas
- `group_by()`: agrupa linhas pelos valores das colunas
- `summarise()`: resume os dados através de funções considerando valores das colunas
- `*_join()`: funções que juntam dados de duas tabelas através de uma coluna chave

### 6.7.2 Sintaxe

As funções do **dplyr** podem seguir uma mesma sintaxe: o **tibble** será sempre o primeiro argumento dessas funções, seguido de um operador pipe (`|>`ou `%>%`) e pelo nome da função que irá fazer a manipulação nesses

dados. Isso permite o encadeamento de várias operações consecutivas mantendo a estrutura do dado original e acrescentando mudanças num encadeamento lógico.

Sendo assim, as funções verbais não precisam modificar necessariamente o tibble original, sendo que as operações de manipulações podem e devem ser atribuídas a um novo objeto.

```
## Sintaxe
tb_dplyr <- tb |>
  funcao_verbal1(argumento1, argumento2, ...)
  funcao_verbal2(argumento1, argumento2, ...)
  funcao_verbal3(argumento1, argumento2, ...)
```

Além de `data.frames` e `tibbles`, a manipulação pelo formato `dplyr` torna o trabalho com outros formatos de classes e dados acessíveis e eficientes como `data.table`, SQL e Apache Spark, para os quais existem pacotes específicos.

- `dtplyr`: manipular conjuntos de dados `data.table`
- `dbplyr`: manipular conjuntos de dados SQL
- `sparklyr`: manipular conjuntos de dados no Apache Spark

### 6.7.3 palmerpenguins

Para nossos exemplos, vamos utilizar novamente os dados de pinguins `palmerpenguins`. Esses dados estão disponíveis no pacote `palmerpenguins`. Vamos utilizar principalmente o conjunto de dados `penguins`, que é a versão simplificada dos dados brutos `penguins_raw`.

```
## Carregar o pacote palmerpenguins
library(palmerpenguins)
```

### 6.7.4 relocate()

Primeiramente, vamos reordenar as colunas com a função `dplyr::relocate()`, onde simplesmente listamos as colunas que queremos mudar de posição e para onde elas devem ir. Para esse último passo há dois argumentos: `.before` que indica a coluna onde a coluna realocada deve se mover antes, e o argumento `.after` indicando onde deve se mover depois. Ambos podem ser informados com os nomes ou posições dessas colunas com números.

```
## Reordenar colunas - nome
penguins_relocate_col <- penguins |>
  dplyr::relocate(sex, year, .after = island)
head(penguins_relocate_col)

## # A tibble: 6 x 8
##   species island   sex   year bill_length_mm bill_depth_mm flipper_length_mm
##   <fct>    <fct> <fct> <int>        <dbl>        <dbl>             <int>
## 1 Adelie   Torgersen male  2007       39.1       18.7            181
## 2 Adelie   Torgersen fema~ 2007       39.5       17.4            186
## 3 Adelie   Torgersen fema~ 2007       40.3        18             195
## 4 Adelie   Torgersen <NA>  2007        NA            NA              NA
## 5 Adelie   Torgersen fema~ 2007       36.7       19.3            193
## 6 Adelie   Torgersen male  2007       39.3       20.6            190
## # i 1 more variable: body_mass_g <int>

## Reordenar colunas - posição
penguins_relocate_ncol <- penguins |>
  dplyr::relocate(sex, year, .after = 2)
head(penguins_relocate_ncol)
```

```

## # A tibble: 6 x 8
##   species island   sex   year bill_length_mm bill_depth_mm flipper_length_mm
##   <fct>   <fct>   <fct> <int>        <dbl>        <dbl>            <int>
## 1 Adelie  Torgersen male  2007      39.1       18.7           181
## 2 Adelie  Torgersen female 2007      39.5       17.4           186
## 3 Adelie  Torgersen female 2007      40.3        18             195
## 4 Adelie  Torgersen <NA>  2007       NA          NA             NA
## 5 Adelie  Torgersen female 2007      36.7       19.3           193
## 6 Adelie  Torgersen male  2007      39.3       20.6           190
## # i 1 more variable: body_mass_g <int>

```

### 6.7.5 rename()

Podemos renomear colunas facilmente com a função `dplyr::rename()`, onde primeiramente informamos o nome que queremos que a coluna tenha, seguido do operador `=` e a coluna do nosso dado (“nova\_coluna = antiga\_coluna”). Também podemos utilizar a função `dplyr::rename_with()`, que faz a mudança do nome em múltiplas colunas, que pode depender ou não de resultados booleanos.

```

## Renomear as colunas
penguins_rename <- penguins |>
  dplyr::rename(bill_length = bill_length_mm,
                bill_depth = bill_depth_mm,
                flipper_length = flipper_length_mm,
                body_mass = body_mass_g)
head(penguins_rename)

## # A tibble: 6 x 8
##   species island   bill_length bill_depth flipper_length body_mass sex   year
##   <fct>   <fct>     <dbl>      <dbl>        <int>    <int> <fct> <int>
## 1 Adelie  Torgers~    39.1       18.7        181     3750 male   2007
## 2 Adelie  Torgers~    39.5       17.4        186     3800 female 2007
## 3 Adelie  Torgers~    40.3        18          195     3250 female 2007
## 4 Adelie  Torgers~     NA          NA          NA      NA <NA>  2007
## 5 Adelie  Torgers~    36.7       19.3        193     3450 female 2007
## 6 Adelie  Torgers~    39.3       20.6        190     3650 male   2007

## mudar o nome de todas as colunas
penguins_rename_with <- penguins |>
  dplyr::rename_with(toupper)
head(penguins_rename_with)

## # A tibble: 6 x 8
##   SPECIES ISLAND   BILL_LENGTH_MM BILL_DEPTH_MM FLIPPER_LENGTH_MM BODY_MASS_G
##   <fct>   <fct>     <dbl>      <dbl>        <int>            <int>
## 1 Adelie  Torgersen    39.1       18.7        181     3750
## 2 Adelie  Torgersen    39.5       17.4        186     3800
## 3 Adelie  Torgersen    40.3        18          195     3250
## 4 Adelie  Torgersen     NA          NA          NA      NA
## 5 Adelie  Torgersen    36.7       19.3        193     3450
## 6 Adelie  Torgersen    39.3       20.6        190     3650
## # i 2 more variables: SEX <fct>, YEAR <int>

```

### 6.7.6 select()

Outra operação bastante usual dentro da manipulação de dados tabulares é a seleção de colunas. Podemos fazer essa operação com a função `dplyr::select()`, que seleciona colunas pelo nome ou pela sua posição.

Aqui há uma série de possibilidades de seleção de colunas, desde utilizar operadores como : para selecionar intervalos de colunas, ! para tomar o complemento (todas menos as listadas), além de funções como `dplyr::starts_with()`, `dplyr::ends_with()`, `dplyr::contains()` para procurar colunas com um padrão de texto do nome da coluna.

```
## Selecionar colunas por posição
penguins_select_position <- penguins |>
  dplyr::select(3:6)
head(penguins_select_position)

## # A tibble: 6 x 4
##   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##       <dbl>          <dbl>           <int>        <int>
## 1      39.1          18.7            181        3750
## 2      39.5          17.4            186        3800
## 3      40.3           18              195        3250
## 4       NA             NA              NA         NA
## 5      36.7          19.3            193        3450
## 6      39.3          20.6            190        3650

## Selecionar colunas por nomes
penguins_select_names <- penguins |>
  dplyr::select(bill_length_mm:body_mass_g)
head(penguins_select_names)

## # A tibble: 6 x 4
##   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##       <dbl>          <dbl>           <int>        <int>
## 1      39.1          18.7            181        3750
## 2      39.5          17.4            186        3800
## 3      40.3           18              195        3250
## 4       NA             NA              NA         NA
## 5      36.7          19.3            193        3450
## 6      39.3          20.6            190        3650

## Selecionar colunas por padrão
penguins_select_contains <- penguins |>
  dplyr::select[contains("mm")]
head(penguins_select_contains)

## # A tibble: 6 x 3
##   bill_length_mm bill_depth_mm flipper_length_mm
##       <dbl>          <dbl>           <int>
## 1      39.1          18.7            181
## 2      39.5          17.4            186
## 3      40.3           18              195
## 4       NA             NA              NA
## 5      36.7          19.3            193
## 6      39.3          20.6            190
```

### 6.7.7 `pull()`

Quando usamos a função `dplyr::select()`, mesmo que para apenas uma coluna, o retorno da função é sempre um **tibble**. Caso precisemos que essa coluna se torne um vetor dentro do encadeamento dos **pipes**, usamos a função `dplyr::pull()`, que extrai uma única coluna como vetor.

```

## Coluna como vetor
penguins_select_pull <- penguins |>
  dplyr::pull(bill_length_mm)
head(penguins_select_pull, 15)

## [1] 39.1 39.5 40.3   NA 36.7 39.3 38.9 39.2 34.1 42.0 37.8 37.8 41.1 38.6
## [15] 34.6

```

### 6.7.8 mutate()

Uma das operações mais úteis dentre as operações para colunas é adicionar ou atualizar os valores de colunas. Para essa operação, usaremos a função `dplyr::mutate()`. Podemos ainda usar os argumentos `.before` e `.after` para indicar onde a nova coluna deve ficar, além do parâmetro `.keep` com diversas possibilidades de manter colunas depois de usar a função `dplyr::mutate()`. Por fim, é fundamental destacar o uso das funções de replicação: `dplyr::across()`, `dplyr::if_any()` e `dplyr::if_all()`, para os quais a função fará alterações em múltiplas colunas de uma vez, dependendo de resultados booleanos.

```

## Adicionar colunas
penguins_mutate <- penguins |>
  dplyr::mutate(body_mass_kg = body_mass_g/1e3, .before = sex)
head(penguins_mutate)

```

```

## # A tibble: 6 x 9
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>        <dbl>        <dbl>          <int>       <int>
## 1 Adelie  Torgersen     39.1        18.7          181        3750
## 2 Adelie  Torgersen     39.5        17.4          186        3800
## 3 Adelie  Torgersen     40.3        18            195        3250
## 4 Adelie  Torgersen      NA          NA             NA         NA
## 5 Adelie  Torgersen     36.7        19.3          193        3450
## 6 Adelie  Torgersen     39.3        20.6          190        3650
## # i 3 more variables: body_mass_kg <dbl>, sex <fct>, year <int>

```

```

## Modificar várias colunas
penguins_mutate_across <- penguins |>
  dplyr::mutate(across(where(is.factor), as.character))
head(penguins_mutate_across)

```

```

## # A tibble: 6 x 8
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <chr>   <chr>        <dbl>        <dbl>          <int>       <int>
## 1 Adelie  Torgersen     39.1        18.7          181        3750
## 2 Adelie  Torgersen     39.5        17.4          186        3800
## 3 Adelie  Torgersen     40.3        18            195        3250
## 4 Adelie  Torgersen      NA          NA             NA         NA
## 5 Adelie  Torgersen     36.7        19.3          193        3450
## 6 Adelie  Torgersen     39.3        20.6          190        3650
## # i 2 more variables: sex <chr>, year <int>

```

### 6.7.9 arrange()

Além de operações em colunas, podemos fazer operações em linhas. Vamos começar com a reordenação das linhas com base nos valores das colunas. Para essa operação, usamos a função `dplyr::arrange()`. Podemos reordenar por uma ou mais colunas de forma crescente ou decrescente usando a função `desc()` ou o operador `-` antes da coluna de interesse. Da mesma forma que na função `dplyr::mutate()`, podemos usar as funções de replicação para ordenar as linhas para várias colunas de uma vez, dependendo de resultados booleanos.

```

## Reordenar linhas - crescente
penguinsArrange <- penguins |>
  dplyr::arrange(body_mass_g)
head(penguinsArrange)

## # A tibble: 6 x 8
##   species   island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>     <fct>        <dbl>        <dbl>          <int>       <int>
## 1 Chinstrap Dream        46.9        16.6           192      2700
## 2 Adelie    Biscoe       36.5        16.6           181      2850
## 3 Adelie    Biscoe       36.4        17.1           184      2850
## 4 Adelie    Biscoe       34.5        18.1           187      2900
## 5 Adelie    Dream        33.1        16.1           178      2900
## 6 Adelie    Torgersen    38.6         17             188      2900
## # i 2 more variables: sex <fct>, year <int>

## Reordenar linhas - decrescente
penguinsArrange_desc <- penguins |>
  dplyr::arrange(desc(body_mass_g))
head(penguinsArrange_desc)

## # A tibble: 6 x 8
##   species   island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>     <fct>        <dbl>        <dbl>          <int>       <int>
## 1 Gentoo   Biscoe       49.2        15.2           221      6300
## 2 Gentoo   Biscoe       59.6         17            230      6050
## 3 Gentoo   Biscoe       51.1        16.3           220      6000
## 4 Gentoo   Biscoe       48.8        16.2           222      6000
## 5 Gentoo   Biscoe       45.2        16.4           223      5950
## 6 Gentoo   Biscoe       49.8        15.9           229      5950
## # i 2 more variables: sex <fct>, year <int>

## Reordenar linhas - decrescente
penguinsArrange_desc_m <- penguins |>
  dplyr::arrange(-body_mass_g)
head(penguinsArrange_desc_m)

## # A tibble: 6 x 8
##   species   island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>     <fct>        <dbl>        <dbl>          <int>       <int>
## 1 Gentoo   Biscoe       49.2        15.2           221      6300
## 2 Gentoo   Biscoe       59.6         17            230      6050
## 3 Gentoo   Biscoe       51.1        16.3           220      6000
## 4 Gentoo   Biscoe       48.8        16.2           222      6000
## 5 Gentoo   Biscoe       45.2        16.4           223      5950
## 6 Gentoo   Biscoe       49.8        15.9           229      5950
## # i 2 more variables: sex <fct>, year <int>

## Reordenar linhas - multiplas colunas
penguinsArrange_across <- penguins |>
  dplyr::arrange(across(where(is.numeric)))
head(penguinsArrange_across)

## # A tibble: 6 x 8
##   species   island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>     <fct>        <dbl>        <dbl>          <int>       <int>

```

```

## 1 Adelie Dream          32.1      15.5      188      3050
## 2 Adelie Dream          33.1      16.1      178      2900
## 3 Adelie Torgersen     33.5       19        190      3600
## 4 Adelie Dream          34        17.1      185      3400
## 5 Adelie Torgersen     34.1      18.1      193      3475
## 6 Adelie Torgersen     34.4      18.4      184      3325
## # i 2 more variables: sex <fct>, year <int>

```

### 6.7.10 filter()

Uma das principais e mais usuais operações que podemos realizar em linhas é a seleção de linhas através do filtro por valores de uma ou mais colunas, utilizando a função `dplyr::filter()`. Para realizar os filtros utilizaremos grande parte dos operadores relacionais e lógicos que listamos no Capítulo 5, especialmente os lógicos para combinações de filtros em mais de uma coluna. Além desses operadores, podemos utilizar a função `is.na()` para filtros em elementos faltantes, e as funções `dplyr::between()` e `dplyr::near()` para filtros entre valores, e para valores próximos com certa tolerância, respectivamente. Por fim, podemos usar as funções de replicação para filtro das linhas para mais de uma coluna, dependendo de resultados booleanos.

```

## Filtrar linhas
penguins_filter <- penguins |>
  dplyr::filter(species == "Adelie")
head(penguins_filter)

## # A tibble: 6 x 8
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>           <dbl>         <dbl>            <int>       <int>
## 1 Adelie  Torgersen     39.1          18.7            181        3750
## 2 Adelie  Torgersen     39.5          17.4            186        3800
## 3 Adelie  Torgersen     40.3          18              195        3250
## 4 Adelie  Torgersen     NA             NA              NA          NA
## 5 Adelie  Torgersen     36.7          19.3            193        3450
## 6 Adelie  Torgersen     39.3          20.6            190        3650
## # i 2 more variables: sex <fct>, year <int>

## Filtrar linhas
penguins_filter_two <- penguins |>
  dplyr::filter(species == "Adelie" & sex == "female")
head(penguins_filter_two)

## # A tibble: 6 x 8
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>           <dbl>         <dbl>            <int>       <int>
## 1 Adelie  Torgersen     39.5          17.4            186        3800
## 2 Adelie  Torgersen     40.3          18              195        3250
## 3 Adelie  Torgersen     36.7          19.3            193        3450
## 4 Adelie  Torgersen     38.9          17.8            181        3625
## 5 Adelie  Torgersen     41.1          17.6            182        3200
## 6 Adelie  Torgersen     36.6          17.8            185        3700
## # i 2 more variables: sex <fct>, year <int>

## Filtrar linhas
penguins_filter_in <- penguins |>
  dplyr::filter(species %in% c("Adelie", "Gentoo"),
                sex == "female")
head(penguins_filter_in)

## # A tibble: 6 x 8

```

```

## # A tibble: 6 x 8
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>           <dbl>        <dbl>          <int>       <int>
## 1 Adelie  Torgersen      39.5         17.4          186        3800
## 2 Adelie  Torgersen      40.3         18            195        3250
## 3 Adelie  Torgersen      36.7         19.3          193        3450
## 4 Adelie  Torgersen      38.9         17.8          181        3625
## 5 Adelie  Torgersen      41.1         17.6          182        3200
## 6 Adelie  Torgersen      36.6         17.8          185        3700
## # i 2 more variables: sex <fct>, year <int>

## Filtrar linhas - NA
penguins_filter_na <- penguins |>
  dplyr::filter(!is.na(sex) == TRUE)
head(penguins_filter_na)

## # A tibble: 6 x 8
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>           <dbl>        <dbl>          <int>       <int>
## 1 Adelie  Torgersen      39.1         18.7          181        3750
## 2 Adelie  Torgersen      39.5         17.4          186        3800
## 3 Adelie  Torgersen      40.3         18            195        3250
## 4 Adelie  Torgersen      36.7         19.3          193        3450
## 5 Adelie  Torgersen      39.3         20.6          190        3650
## 6 Adelie  Torgersen      38.9         17.8          181        3625
## # i 2 more variables: sex <fct>, year <int>

## Filtrar linhas - intervalos
penguins_filter_between <- penguins |>
  dplyr::filter(between(body_mass_g, 3000, 4000))
head(penguins_filter_between)

## # A tibble: 6 x 8
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>           <dbl>        <dbl>          <int>       <int>
## 1 Adelie  Torgersen      39.1         18.7          181        3750
## 2 Adelie  Torgersen      39.5         17.4          186        3800
## 3 Adelie  Torgersen      40.3         18            195        3250
## 4 Adelie  Torgersen      36.7         19.3          193        3450
## 5 Adelie  Torgersen      39.3         20.6          190        3650
## 6 Adelie  Torgersen      38.9         17.8          181        3625
## # i 2 more variables: sex <fct>, year <int>

## Filtrar linhas por várias colunas
penguins_filter_if <- penguins |>
  dplyr::filter(if_all(where(is.integer), ~ . > 200))
head(penguins_filter_if)

## # A tibble: 6 x 8
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>           <dbl>        <dbl>          <int>       <int>
## 1 Adelie  Dream        35.7         18            202        3550
## 2 Adelie  Dream        41.1         18.1          205        4300
## 3 Adelie  Dream        40.8         18.9          208        4300
## 4 Adelie  Biscoe       41            20            203        4725
## 5 Adelie  Torgersen    41.4         18.5          202        3875
## 6 Adelie  Torgersen    44.1         18            210        4000

```

```
## # i 2 more variables: sex <fct>, year <int>
```

### 6.7.11 slice()

Além da seleção de linhas por filtros, podemos fazer a seleção das linhas por intervalos, indicando quais linhas desejamos, usando a função `dplyr::slice()`, e informando o argumento `n` para o número da linha ou intervalo das linhas. Essa função possui variações no sufixo muito interessantes: `dplyr::slice_head()` e `dplyr::slice_tail()` seleciona as primeiras e últimas linhas, `dplyr::slice_min()` e `dplyr::slice_max()` seleciona linhas com os maiores e menores valores de uma coluna, e `dplyr::slice_sample()` seleciona linhas aleatoriamente.

```
## Seleciona linhas 300 até final
```

```
penguins_slice <- penguins |>  
  dplyr::slice(300:n())  
head(penguins_slice)
```

```
## # A tibble: 6 x 8
```

```
##   species   island bill_length_mm bill_depth_mm flipper_length_mm body_mass_g  
##   <fct>     <fct>       <dbl>        <dbl>          <int>        <int>  
## 1 Chinstrap Dream      50.6         19.4          193        3800  
## 2 Chinstrap Dream      46.7         17.9          195        3300  
## 3 Chinstrap Dream      52            19            197        4150  
## 4 Chinstrap Dream      50.5         18.4          200        3400  
## 5 Chinstrap Dream      49.5         19            200        3800  
## 6 Chinstrap Dream      46.4         17.8          191        3700
```

```
## # i 2 more variables: sex <fct>, year <int>
```

```
## Seleciona linhas - head
```

```
penguins_slice_head <- penguins |>  
  dplyr::slice_head(n = 5)  
head(penguins_slice_head)
```

```
## # A tibble: 5 x 8
```

```
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g  
##   <fct>    <fct>       <dbl>        <dbl>          <int>        <int>  
## 1 Adelie   Torgersen     39.1        18.7          181        3750  
## 2 Adelie   Torgersen     39.5        17.4          186        3800  
## 3 Adelie   Torgersen     40.3         18            195        3250  
## 4 Adelie   Torgersen      NA           NA             NA          NA  
## 5 Adelie   Torgersen     36.7        19.3          193        3450
```

```
## # i 2 more variables: sex <fct>, year <int>
```

```
## Seleciona linhas - max
```

```
penguins_slice_max <- penguins |>  
  dplyr::slice_max(body_mass_g, n = 5)  
head(penguins_slice_max)
```

```
## # A tibble: 6 x 8
```

```
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g  
##   <fct>    <fct>       <dbl>        <dbl>          <int>        <int>  
## 1 Gentoo   Biscoe      49.2        15.2          221        6300  
## 2 Gentoo   Biscoe      59.6         17            230        6050  
## 3 Gentoo   Biscoe      51.1        16.3          220        6000  
## 4 Gentoo   Biscoe      48.8        16.2          222        6000  
## 5 Gentoo   Biscoe      45.2        16.4          223        5950  
## 6 Gentoo   Biscoe      49.8        15.9          229        5950
```

```
## # i 2 more variables: sex <fct>, year <int>
```

```

## Seleciona linhas - sample
penguins_slice_sample <- penguins |>
  dplyr::slice_sample(n = 30)
head(penguins_slice_sample)

## # A tibble: 6 x 8
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>        <dbl>        <dbl>        <int>       <int>
## 1 Adelie  Biscoe      41.3         21.1         195       4400
## 2 Gentoo  Biscoe      44.5         15.7         217       4875
## 3 Adelie  Torgersen   41.4         18.5         202       3875
## 4 Adelie  Biscoe      37.6          17          185       3600
## 5 Adelie  Dream        36           17.9         190       3450
## 6 Adelie  Biscoe      35.7         16.9         185       3150
## # i 2 more variables: sex <fct>, year <int>

```

### 6.7.12 distinct()

A última operação que apresentaremos para linhas é a retirada de linhas com valores repetidos com base nos valores de uma ou mais colunas, utilizando a função `dplyr::distinct()`. Essa função por padrão retorna apenas a(s) coluna(s) utilizada(s) para retirar as linhas com valores repetidos, sendo necessário acrescentar o argumento `.keep_all = TRUE` para retornar todas as colunas. Por fim, podemos usar as funções de replicação para retirar linhas com valores repetidos para mais de uma coluna, dependendo de resultados booleanos.

```

## Retirar linhas com valores repetidos
penguins_distinct <- penguins |>
  dplyr::distinct(body_mass_g)
head(penguins_distinct)

## # A tibble: 6 x 1
##   body_mass_g
##   <int>
## 1 3750
## 2 3800
## 3 3250
## 4 NA
## 5 3450
## 6 3650

## Retirar linhas com valores repetidos - manter as outras colunas
penguins_distinct_keep_all <- penguins |>
  dplyr::distinct(body_mass_g, .keep_all = TRUE)
head(penguins_distinct_keep_all)

## # A tibble: 6 x 8
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>        <dbl>        <dbl>        <int>       <int>
## 1 Adelie  Torgersen   39.1         18.7         181       3750
## 2 Adelie  Torgersen   39.5         17.4         186       3800
## 3 Adelie  Torgersen   40.3          18          195       3250
## 4 Adelie  Torgersen   NA            NA            NA          NA
## 5 Adelie  Torgersen   36.7         19.3         193       3450
## 6 Adelie  Torgersen   39.3         20.6         190       3650
## # i 2 more variables: sex <fct>, year <int>

```

```

## Retirar linhas com valores repetidos para várias colunas
penguins_distinct_keep_all_across <- penguins |>
  dplyr::distinct(across(where(is.integer)), .keep_all = TRUE)
head(penguins_distinct_keep_all_across)

## # A tibble: 6 x 8
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>           <dbl>          <dbl>            <int>        <int>
## 1 Adelie   Torgersen      39.1          18.7            181        3750
## 2 Adelie   Torgersen      39.5          17.4            186        3800
## 3 Adelie   Torgersen      40.3           18            195        3250
## 4 Adelie   Torgersen       NA             NA              NA         NA
## 5 Adelie   Torgersen      36.7          19.3            193        3450
## 6 Adelie   Torgersen      39.3          20.6            190        3650
## # i 2 more variables: sex <fct>, year <int>

```

### 6.7.13 count()

Agora entraremos no assunto de resumo das observações. Podemos fazer contagens resumos dos nossos dados, utilizando para isso a função `dplyr::count()`. Essa função contará valores de uma ou mais colunas, geralmente para variáveis categóricas, semelhante à função *R Base* `table()`, mas num contexto *tidyverse*.

```

## Contagens de valores para uma coluna
penguins_count <- penguins |>
  dplyr::count(species)
penguins_count

## # A tibble: 3 x 2
##   species     n
##   <fct>   <int>
## 1 Adelie     152
## 2 Chinstrap   68
## 3 Gentoo     124

## Contagens de valores para mais de uma coluna
penguins_count_two <- penguins |>
  dplyr::count(species, island)
penguins_count_two

## # A tibble: 5 x 3
##   species   island     n
##   <fct>   <fct>   <int>
## 1 Adelie   Biscoe     44
## 2 Adelie   Dream      56
## 3 Adelie   Torgersen  52
## 4 Chinstrap Dream     68
## 5 Gentoo   Biscoe     124

```

### 6.7.14 group\_by()

Uma grande parte das operações feitas nos dados são realizadas em grupos definidos por valores de colunas com dados categóricas. A função `dplyr::group_by()` transforma um `tibble` em um `tibble grouped`, onde as operações são realizadas “por grupo”. Essa função é utilizada geralmente junto com a função `dplyr::summarise()`, que veremos logo em seguida. O agrupamento não altera a aparência dos dados (além de informar como estão agrupados). A função `dplyr::ungroup()` remove o agrupamento. Podemos ainda

usar funções de replicação para fazer os agrupamentos para mais de uma coluna, dependendo de resultados booleanos.

```
## Agrupamento
penguins_group_by <- penguins |>
  dplyr::group_by(species)
head(penguins_group_by)

## # A tibble: 6 x 8
## # Groups:   species [1]
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>        <dbl>        <dbl>        <int>       <int>
## 1 Adelie  Torgersen     39.1        18.7        181        3750
## 2 Adelie  Torgersen     39.5        17.4        186        3800
## 3 Adelie  Torgersen     40.3        18          195        3250
## 4 Adelie  Torgersen      NA          NA          NA          NA
## 5 Adelie  Torgersen     36.7        19.3        193        3450
## 6 Adelie  Torgersen     39.3        20.6        190        3650
## # i 2 more variables: sex <fct>, year <int>

## Agrupamento de várias colunas
penguins_group_by_across <- penguins |>
  dplyr::group_by(across(where(is.factor)))
head(penguins_group_by_across)

## # A tibble: 6 x 8
## # Groups:   species, island, sex [3]
##   species island   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <fct>        <dbl>        <dbl>        <int>       <int>
## 1 Adelie  Torgersen     39.1        18.7        181        3750
## 2 Adelie  Torgersen     39.5        17.4        186        3800
## 3 Adelie  Torgersen     40.3        18          195        3250
## 4 Adelie  Torgersen      NA          NA          NA          NA
## 5 Adelie  Torgersen     36.7        19.3        193        3450
## 6 Adelie  Torgersen     39.3        20.6        190        3650
## # i 2 more variables: sex <fct>, year <int>
```

### 6.7.15 summarise()

Como dissemos, muitas vezes queremos resumir nossos dados, principalmente para ter uma noção geral das variáveis (colunas) ou mesmo começar a análise exploratória resumindo variáveis contínuas por grupos de variáveis categóricas. Dessa forma, ao utilizar a função `dplyr::summarise()` teremos um novo `tibble` com os dados resumidos, que é a agregação ou resumo dos dados através de funções. Da mesma forma que outras funções, podemos usar funções de replicação para resumir valores para mais de uma coluna, dependendo de resultados booleanos.

```
## Resumo
penguins_summarise <- penguins |>
  dplyr::group_by(species) |>
  dplyr::summarize(body_mass_g_mean = mean(body_mass_g, na.rm = TRUE),
                   body_mass_g_sd = sd(body_mass_g, na.rm = TRUE))
penguins_summarise

## # A tibble: 3 x 3
##   species   body_mass_g_mean body_mass_g_sd
##   <fct>        <dbl>        <dbl>
## 1 Adelie      3701.        459.
```

```

## # 2 Chinstrap      3733.      384.
## # 3 Gentoo        5076.      504.

## Resumo para várias colunas
penguins_summarise_across <- penguins |>
  dplyr::group_by(species) |>
  dplyr::summarize(across(where(is.numeric), ~ mean(.x, na.rm = TRUE)))
penguins_summarise_across

## # A tibble: 3 x 6
##   species   bill_length_mm bill_depth_mm flipper_length_mm body_mass_g year
##   <fct>       <dbl>        <dbl>          <dbl>        <dbl> <dbl>
## 1 Adelie     38.8         18.3           190.        3701. 2008.
## 2 Chinstrap   48.8         18.4           196.        3733. 2008.
## 3 Gentoo     47.5         15.0           217.        5076. 2008.

```

### 6.7.16 bind\_rows() e bind\_cols()

Muitas vezes teremos de combinar duas ou mais tabelas de dados. Podemos utilizar as funções *R Base* rbind() e cbind(), como vimos no Capítulo 5. Entretanto, pode ser interessante avançar para as funções dplyr::bind\_rows() e dplyr::bind\_cols() do formato *tidyverse*. A ideia é muito semelhante: a primeira função combina dados por linhas e a segunda por colunas. Entretanto, há algumas vantagens no uso dessas funções, como a identificação das linhas pelo argumento .id para a primeira função, e a conferência do nome das colunas pelo argumento .name\_repair para a segunda função.

```

## Selecionar as linhas para dois tibbles
penguins_01 <- dplyr::slice(penguins, 1:5)
penguins_02 <- dplyr::slice(penguins, 51:55)

## Combinar as linhas
penguins_bind_rows <- dplyr::bind_rows(penguins_01, penguins_02, .id = "id")
head(penguins_bind_rows)

## # A tibble: 6 x 9
##   id   species island   bill_length_mm bill_depth_mm flipper_length_mm
##   <chr> <fct>   <fct>       <dbl>        <dbl>          <dbl>
## 1 1    Adelie  Torgersen  39.1         18.7           181
## 2 1    Adelie  Torgersen  39.5         17.4           186
## 3 1    Adelie  Torgersen  40.3         18             195
## 4 1    Adelie  Torgersen  NA            NA             NA
## 5 1    Adelie  Torgersen  36.7         19.3           193
## 6 2    Adelie  Biscoe     39.6         17.7           186
## # i 3 more variables: body_mass_g <int>, sex <fct>, year <int>

## Combinar as colunas
penguins_bind_cols <- dplyr::bind_cols(penguins_01, penguins_02, .name_repair = "unique")

## New names:
## * `species` -> `species...1`
## * `island` -> `island...2`
## * `bill_length_mm` -> `bill_length_mm...3`
## * `bill_depth_mm` -> `bill_depth_mm...4`
## * `flipper_length_mm` -> `flipper_length_mm...5`
## * `body_mass_g` -> `body_mass_g...6`
## * `sex` -> `sex...7`
## * `year` -> `year...8`
## * `species` -> `species...9`

```

```

## * `island` -> `island...10`
## * `bill_length_mm` -> `bill_length_mm...11`
## * `bill_depth_mm` -> `bill_depth_mm...12`
## * `flipper_length_mm` -> `flipper_length_mm...13`
## * `body_mass_g` -> `body_mass_g...14`
## * `sex` -> `sex...15`
## * `year` -> `year...16`

head(penguins_bind_cols)

## # A tibble: 5 x 16
##   species...1 island...2 bill_length_mm...3 bill_depth_mm...4
##   <fct>     <fct>          <dbl>           <dbl>
## 1 Adelie    Torgersen      39.1            18.7
## 2 Adelie    Torgersen      39.5            17.4
## 3 Adelie    Torgersen      40.3             18
## 4 Adelie    Torgersen      NA              NA
## 5 Adelie    Torgersen      36.7            19.3
## # i 12 more variables: flipper_length_mm...5 <int>, body_mass_g...6 <int>,
## #   sex...7 <fct>, year...8 <int>, species...9 <fct>, island...10 <fct>,
## #   bill_length_mm...11 <dbl>, bill_depth_mm...12 <dbl>,
## #   flipper_length_mm...13 <int>, body_mass_g...14 <int>, sex...15 <fct>,
## #   year...16 <int>

```

### 6.7.17 \*\_join()

Finalmente, veremos o último conjunto de funções do pacote `dplyr`, a junção de tabelas. Nessa operação, fazemos a combinação de pares de conjunto de dados tabulares por uma ou mais colunas chaves. Há dois tipos de junções: junção de modificação e junção de filtragem. A junção de modificação primeiro combina as observações por suas chaves e, em seguida, copia as variáveis (colunas) de uma tabela para a outra. É fundamental destacar a importância da coluna chave, que é indicada pelo argumento `by`. Essa coluna deve conter elementos que sejam comuns às duas tabelas para que haja a combinação dos elementos.

Existem quatro tipos de junções de modificações, que são realizadas pelas funções: `dplyr::inner_join()`, `dplyr::left_join()`, `dplyr::full_join()` e `dplyr::right_join()`, e que podem ser representadas na Figura 6.3.

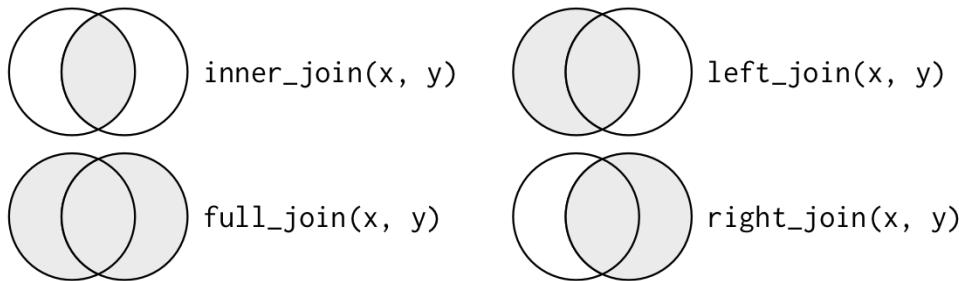


Figura 6.3: Diferentes tipos de joins, representados com um diagrama de Venn.

Considerando a nomenclatura de duas tabelas de dados por `x` e `y`, temos:

- `inner_join(x, y)`: mantém apenas as observações em `x` e em `y`
- `left_join(x, y)`: mantém todas as observações em `x`
- `right_join(x, y)`: mantém todas as observações em `y`
- `full_join(x, y)`: mantém todas as observações em `x` e em `y`

Aqui, vamos demonstrar apenas a função `dplyr::left_join()`, combinando um `tibble` de coordenadas geográficas das ilhas com o conjunto de dados do penguins.

```
## Adicionar uma coluna chave de ids
penguin_islands <- tibble(
  island = c("Torgersen", "Biscoe", "Dream", "Alpha"),
  longitude = c(-64.083333, -63.775636, -64.233333, -63),
  latitude = c(-64.766667, -64.818569, -64.733333, -64.316667))

## Junção - left
penguins_left_join <- dplyr::left_join(penguins, penguin_islands, by = "island")
head(penguins_left_join)

## # A tibble: 6 x 10
##   species island    bill_length_mm bill_depth_mm flipper_length_mm body_mass_g
##   <fct>   <chr>          <dbl>           <dbl>            <int>        <int>
## 1 Adelie  Torgersen      39.1           18.7            181         3750
## 2 Adelie  Torgersen      39.5           17.4            186         3800
## 3 Adelie  Torgersen      40.3            18             195         3250
## 4 Adelie  Torgersen       NA             NA              NA          NA
## 5 Adelie  Torgersen      36.7           19.3            193         3450
## 6 Adelie  Torgersen      39.3           20.6            190         3650
## # i 4 more variables: sex <fct>, year <int>, longitude <dbl>, latitude <dbl>
```

Já o segundo tipo de junção, a junção de filtragem combina as observações da mesma maneira que as junções de modificação, mas afetam as observações (linhas), não as variáveis (colunas). Existem dois tipos.

- `semi_join(x, y)`: mantém todas as observações em `x` que têm uma correspondência em `y`
- `anti_join(x, y)`: elimina todas as observações em `x` que têm uma correspondência em `y`

De forma geral, *semi-joins* são úteis para corresponder tabelas de resumo filtradas de volta às linhas originais, removendo as linhas que não estavam antes do join. Já *anti-joins* são úteis para diagnosticar incompatibilidades de junção, por exemplo, ao verificar os elementos que não combinam entre duas tabelas de dados.

### 6.7.18 Operações de conjuntos e comparação de dados

Temos ainda operações de conjuntos e comparação de dados.

- `union(x, y)`: retorna todas as linhas que aparecem em `x`, `y` ou mais dos conjuntos de dados
- `intersect(x, y)`: retorna apenas as linhas que aparecem em `x` e em `y`
- `setdiff(x, y)`: retorna as linhas que aparecem `x`, mas não em `y`
- `setequal(x, y)`: retorna se `x` e `y` são iguais e quais suas diferenças

Para se aprofundar no tema, recomendamos a leitura do Capítulo 13 *Relational data* de Wickham & Grolemund (2017).

## 6.8 stringr

O pacote `stringr` fornece um conjunto de funções para a manipulação de caracteres ou strings. O pacote concentra-se nas funções de manipulação mais importantes e comumente usadas. Para funções mais específicas, recomenda-se usar o pacote `stringi`, que fornece um conjunto mais abrangente de funções. As funções do `stringr` podem ser agrupadas em algumas operações para tarefas específicas como: i) correspondência de padrões, ii) retirar e acrescentar espaços em branco, iii) mudar maiúsculas e minúsculas, além de muitas outras operações com caracteres.

Todas as funções deste pacote são listadas na [página de referência](#) do pacote.

Demonstraremos algumas funções para algumas operações mais comuns, utilizando um vetor de um elemento, com o string “penguins”.

Podemos explorar o comprimento de strings com a função `stringr::str_length()`.

```
## Comprimento
stringr::str_length(string = "penguins")

## [1] 8
```

Extrair um string por sua posição usando a função `stringr::str_sub()` ou por um padrão com `stringr::str_extract()`.

```
## Extrair pela posição
stringr::str_sub(string = "penguins", end = 3)

## [1] "pen"

## Extrair por padrão
stringr::str_extract(string = "penguins", pattern = "p")

## [1] "p"
```

Substituir strings por outros strings com `stringr::str_replace()`.

```
## Substituir
stringr::str_replace(string = "penguins", pattern = "i", replacement = "y")

## [1] "penguyns"
```

Separar strings por um padrão com a função `stringr::str_split()`.

```
## Separar
stringr::str_split(string = "p-e-n-g-u-i-n-s", pattern = "-", simplify = TRUE)

##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,] "p"  "e"  "n"  "g"  "u"  "i"  "n"  "s"
```

Inserir espaços em brancos pela esquerda, direita ou ambos com a função `stringr::str_pad()`.

```
## Inserir espacos em branco
stringr::str_pad(string = "penguins", width = 10, side = "left")

## [1] " penguins"
stringr::str_pad(string = "penguins", width = 10, side = "right")
```

```
## [1] "penguins "
stringr::str_pad(string = "penguins", width = 10, side = "both")
```

```
## [1] " penguins "
```

Também podemos remover espaços em branco da esquerda, direita ou ambos, utilizando `stringr::str_trim()`.

```
## Remover espacos em branco
stringr::str_trim(string = " penguins ", side = "left")

## [1] "penguins "
stringr::str_trim(string = " penguins ", side = "right")
```

```
## [1] " penguins"
stringr::str_trim(string = " penguins ", side = "both")
```

```
## [1] "penguins"
```

Podemos também alterar minúsculas e maiúsculas em diferentes posições do string, com várias funções.

```
## Alterar minúsculas e maiúsculas
stringr::str_to_lower(string = "Penguins")

## [1] "penguins"
stringr::str_to_upper(string = "penguins")

## [1] "PENGUINS"
stringr::str_to_sentence(string = "penGuins")

## [1] "Penguins"
stringr::str_to_title(string = "penGuins")

## [1] "Penguins"
```

Podemos ainda ordenar os elementos de um vetor por ordem alfabética de forma crescente ou decrescente, usando `stringr::str_sort()`.

```
## Ordenar
stringr::str_sort(x = letters)

## [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r"
## [19] "s" "t" "u" "v" "w" "x" "y" "z"
stringr::str_sort(x = letters, dec = TRUE)

## [1] "z" "y" "x" "w" "v" "u" "t" "s" "r" "q" "p" "o" "n" "m" "l" "k" "j" "i"
## [19] "h" "g" "f" "e" "d" "c" "b" "a"
```

Podemos ainda utilizar essas funções em complemento com o pacote `dplyr`, para alterar os strings de colunas ou nome das colunas.

```
## Alterar valores das colunas
penguins_stringr_valores <- penguins |>
  dplyr::mutate(species = stringr::str_to_lower(species))

## Alterar nome das colunas
penguins_stringr_nomes <- penguins |>
  dplyr::rename_with(stringr::str_to_title)
```

Para se aprofundar no tema, recomendamos a leitura do Capítulo 14 `Strings` de Wickham & Grolemund (2017).

## 6.9forcats

O pacote `forcats` fornece um conjunto de ferramentas úteis para facilitar a manipulação de fatores. Como dito no Capítulo 5, usamos fatores geralmente quando temos dados categóricos, que são variáveis que possuem um conjunto de valores fixos e conhecidos. As funções são utilizadas principalmente para: i) mudar a ordem dos níveis, ii) mudar os valores dos níveis, iii) adicionar e remover níveis, iv) combinar múltiplos níveis, além de outras operações.

Todas as funções deste pacote são listadas na [página de referência](#) do pacote.

Vamos utilizar ainda os dados `penguins` e `penguins_raw` para exemplificar o uso do pacote `forcats`.

```
## Carregar o pacote palmerpenguins
library(palmerpenguins)
```

Primeiramente, vamos converter dados de string para fator, utilizando a função `forcats::as_factor()`.

```
## String  
forcats::as_factor(penguins_raw$Species) |> head()  
  
## [1] Adelie Penguin (Pygoscelis adeliae) Adelie Penguin (Pygoscelis adeliae)  
## [3] Adelie Penguin (Pygoscelis adeliae) Adelie Penguin (Pygoscelis adeliae)  
## [5] Adelie Penguin (Pygoscelis adeliae) Adelie Penguin (Pygoscelis adeliae)  
## 3 Levels: Adelie Penguin (Pygoscelis adeliae) ...
```

Podemos facilmente mudar o nome dos níveis utilizando a função `forcats::fct_recode()`.

```
## Mudar o nome dos níveis  
forcats::fct_recode(penguins$species, a = "Adelie", c = "Chinstrap", g = "Gentoo") |> head()  
  
## [1] a a a a a a  
## Levels: a c g
```

Para inverter os níveis, usamos a função `forcats::fct_rev()`.

```
## Inverter os níveis  
forcats::fct_rev(penguins$species) |> head()  
  
## [1] Adelie Adelie Adelie Adelie Adelie  
## Levels: Gentoo Chinstrap Adelie
```

Uma operação muito comum com fatores é mudar a ordem dos níveis. Quando precisamos especificar a ordem dos níveis, podemos fazer essa operação manualmente com a função `forcats::fct_relevel()`.

```
## Especificar a ordem dos níveis  
forcats::fct_relevel(penguins$species, "Chinstrap", "Gentoo", "Adelie") |> head()  
  
## [1] Adelie Adelie Adelie Adelie Adelie  
## Levels: Chinstrap Gentoo Adelie
```

Como vimos, a reordenação dos níveis pode ser feita manualmente. Mas existem outras formas automáticas de reordenação seguindo algumas regras, para as quais existem funções específicas.

- `forcats::fct_inorder()`: pela ordem em que aparecem pela primeira vez
- `forcats::fct_infreq()`: por número de observações com cada nível (decrescente, i.e., o maior primeiro)
- `forcats::fct_inseq()`: pelo valor numérico do nível

```
## Níveis pela ordem em que aparecem  
forcats::fct_inorder(penguins$species) |> head()
```

```
## [1] Adelie Adelie Adelie Adelie Adelie  
## Levels: Adelie Gentoo Chinstrap  
  
## Ordem (decrescente) de frequência  
forcats::fct_infreq(penguins$species) |> head()
```

```
## [1] Adelie Adelie Adelie Adelie Adelie  
## Levels: Adelie Gentoo Chinstrap
```

Por fim, podemos fazer a agregação de níveis raros em um nível utilizando a função `forcats::fct_lump()`.

```
## Agregação de níveis raros em um nível  
forcats::fct_lump(penguins$species) |> head()
```

```
## [1] Adelie Adelie Adelie Adelie Adelie  
## Levels: Adelie Gentoo Other
```

Podemos ainda utilizar essas funções em complemento com o pacote `dplyr` para fazer manipulações de fatores nas colunas de `tibbles`.

```
## Transformar várias colunas em fator
penguins_raw_multi_factor <- penguins_raw |>
  dplyr::mutate(across(where(is.character),forcats::as_factor))
```

Para se aprofundar no tema, recomendamos a leitura do Capítulo 15 Factors de Wickham & Grolemund (2017).

## 6.10 lubridate

O pacote `lubridate` fornece um conjunto de funções para a manipulação de dados de data e horário. Dessa forma, esse pacote facilita a manipulação dessa classe de dado no R, pois geralmente esses dados não são intuitivos e mudam dependendo do tipo de objeto de data e horário. Além disso, os métodos que usam datas e horários devem levar em consideração fusos horárioss, anos bissextos, horários de verão, além de outras particularidades. Existem diversas funções nesse pacote, sendo as mesmas focadas em: i) transformações de data/horário, ii) componentes, iii) arredondamentos, iv) durações, v) períodos, vi) intervalos, além de muitas outras funções específicas.

Todas as funções deste pacote são listadas na [página de referência](#) do pacote.

Apesar de estar inserido no escopo do `tidyverse`, este pacote não é carregado com os demais, requisitando seu carregamento solo.

```
## Carregar
library(lubridate)
```

Existem três tipos de dados data/horário:

- **Data:** tempo em dias, meses e anos `<date>`
- **Horário:** tempo dentro de um dia `<time>`
- **Data-horário:** tempo em um instante (data mais tempo) `<dttm>`

Para trabalhar exclusivamente com horários, podemos utilizar o pacote `hms`.

É fundamental também destacar que algumas letras terão um significado temporal, sendo abreviações de diferentes períodos em inglês: `year` (ano), `month` (mês), `weak` (semana), `day` (dia), `hour` (hora), `minute` (minuto), e `second` (segundo).

Para acessar a informação da data e horários atuais podemos utilizar as funções `lubridate::today()` e `lubridate::now()`.

```
## Extrair a data nesse instante
lubridate::today()

## [1] "2023-12-18"

## Extrair a data e tempo nesse instante
lubridate::now()

## [1] "2023-12-18 10:27:42 -03"
```

Além dessas informações instantâneas, existem três maneiras de criar um dado de data/horário.

- De um string
- De componentes individuais de data e horário
- De um objeto de data/horário existente

Os dados de data/horário geralmente estão no formato de strings. Podemos transformar os dados especificando a ordem dos seus componentes, ou seja, a ordem em que ano, mês e dia aparecem no string, usando as letras `y` (ano), `m` (mês) e `d` (dia) na mesma ordem, por exemplo, `lubridate::dmy()`.

```
## Strings e números para datas
lubridate::dmy("03-03-2021")
```

```
## [1] "2021-03-03"
```

Essas funções também aceitam números sem aspas, além de serem muito versáteis e funcionarem em outros diversos formatos.

```
## Strings e números para datas
lubridate::dmy("03-Mar-2021")
lubridate::dmy(03032021)
lubridate::dmy("03032021")
lubridate::dmy("03/03/2021")
lubridate::dmy("03.03.2021")
```

Além da data, podemos especificar horários atrelados a essas datas. Para criar uma data com horário adicionamos um underscore (\_) e h (hora), m (minuto) e s (segundo) ao nome da função, além do argumento tz para especificar o *fuso horário* (tema tratado mais adiante nessa seção).

```
## Especificar horários e fuso horário
lubridate::dmy_h("03-03-2021 13")
```

```
## [1] "2021-03-03 13:00:00 UTC"
```

```
lubridate::dmy_hm("03-03-2021 13:32")
```

```
## [1] "2021-03-03 13:32:00 UTC"
```

```
lubridate::dmy_hms("03-03-2021 13:32:01")
```

```
## [1] "2021-03-03 13:32:01 UTC"
```

```
lubridate::dmy_hms("03-03-2021 13:32:01", tz = "America/Sao_Paulo")
```

```
## [1] "2021-03-03 13:32:01 -03"
```

Podemos ainda ter componentes individuais de data/horário em múltiplas colunas. Para realizar essa transformação, podemos usar as funções lubridate::make\_date() e lubridate::make\_datetime().

```
## Dados com componentes individuais
```

```
dados <- tibble::tibble(
  ano = c(2021, 2021, 2021),
  mes = c(1, 2, 3),
  dia = c(12, 20, 31),
  hora = c(2, 14, 18),
  minuto = c(2, 44, 55))
```

```
## Data de componentes individuais
```

```
dados |>
  dplyr::mutate(data = lubridate::make_datetime(ano, mes, dia, hora, minuto))
```

```
## # A tibble: 3 x 6
##       ano     mes     dia     hora   minuto data
##   <dbl>   <dbl>   <dbl>   <dbl>   <dbl> <dttm>
## 1   2021     1      12      2       2 2021-01-12 02:02:00
## 2   2021     2      20     14      44 2021-02-20 14:44:00
## 3   2021     3      31     18      55 2021-03-31 18:55:00
```

Por fim, podemos criar datas modificando entre data/horário e data, utilizando as funções lubridate::as\_datetime() e lubridate::as\_date().

```

## Data para data-horário
lubridate::as_datetime(today())

## [1] "2023-12-18 UTC"
## Data-horário para data
lubridate::as_date(now())

## [1] "2023-12-18"

```

Uma vez que entendemos como podemos criar dados de data/horário, podemos explorar funções para acessar e definir componentes individuais. Para essa tarefa existe uma grande quantidade de funções para acessar partes específicas de datas e horários.

- `year()`: acessa o ano
- `month()`: acessa o mês
- `month()`: acessa o dia
- `yday()`: acessa o dia do ano
- `mday()`: acessa o dia do mês
- `wday()`: acessa o dia da semana
- `hour()`: acessa as horas
- `minute()`: acessa os minutos
- `second()`: acessa os segundos

```

## Extrair
agora <- lubridate::now()
lubridate::year(agora)

```

```

## [1] 2023
lubridate::month(agora)

```

```

## [1] 12
lubridate::day(agora)

```

```

## [1] 18
lubridate::wday(agora)

```

```

## [1] 2
lubridate::second(agora)

```

```

## [1] 42.22976

```

Além de acessar componentes de datas e horários, podemos usar essas funções para fazer a inclusão de informações de datas e horários.

```

## Data
data <- dmy_hms("04-03-2021 01:04:56")

## Incluir
lubridate::year(data) <- 2020
lubridate::month(data) <- 01
lubridate::hour(data) <- 13

```

Mais convenientemente, podemos utilizar a função `update()` para alterar vários valores de uma vez.

```

## Incluir vários valores
update(data, year = 2020, month = 1, mday = 1, hour = 1)

```

```
## [1] "2020-01-01 01:04:56 UTC"
```

Muitas vezes precisamos fazer operações com datas, como: adição, subtração, multiplicação e divisão. Para tanto, é preciso entender três classes importantes que representam intervalos de tempo.

- **Durações:** representam um número exato de segundos
- **Períodos:** representam unidades humanas como semanas e meses
- **Intervalos:** representam um ponto inicial e final

Quando fazemos uma subtração de datas, criamos um objeto da classe `difftime`. Essa classe pode ser um pouco complicada de trabalhar, então dentro do `lubridate` usamos funções que convertem essa classe em duração, da classe `Duration`. As durações sempre registram o intervalo de tempo em segundos, com alguma unidade de tempo maior entre parênteses. Há uma série de funções para tratar dessa classe.

- `duration()`: cria data em duração
- `as.duration()`: converte datas em duração
- `dyears()`: duração de anos
- `dmonths()`: duração de meses
- `dweeks()`: duração de semanas
- `ddays()`: duração de dias
- `dhours()`: duração de horas
- `dminutes()`: duração de minutos
- `dseconds()`: duração de segundos

```
## Subtração de datas
tempo_estudando_r <- lubridate::today() - lubridate::dmy("30-11-2011")

## Conversão para duração
tempo_estudando_r_dur <- lubridate::as.duration(tempo_estudando_r)

## Criando durações
lubridate::duration(90, "seconds")

## [1] "90s (~1.5 minutes)"

lubridate::duration(1.5, "minutes")

## [1] "90s (~1.5 minutes)"

lubridate::duration(1, "days")

## [1] "86400s (~1 days)"

## Transformação da duração
lubridate::dseconds(100)

## [1] "100s (~1.67 minutes)"

lubridate::dminutes(100)

## [1] "6000s (~1.67 hours)"

lubridate::dhours(100)

## [1] "360000s (~4.17 days)"

lubridate::ddays(100)

## [1] "8640000s (~14.29 weeks)"

lubridate::dweeks(100)
```

```

## [1] "60480000s (~1.92 years)"
lubridate::dyears(100)

## [1] "3155760000s (~100 years)"

Podemos ainda utilizar as durações para fazer operações aritméticas com datas como adição, subtração e multiplicação.

## Somando durações a datas
lubridate::today() + lubridate::ddays(1)

## [1] "2023-12-19"

## Subtraindo durações de datas
lubridate::today() - lubridate::dyears(1)

## [1] "2022-12-17 18:00:00 UTC"

## Multiplicando durações
2 * dydays(2)

## [1] "126230400s (~4 years)"

```

Além das durações, podemos usar períodos, que são extensões de tempo não fixados em segundos como as durações, mas flexíveis, com o tempo em dias, semanas, meses ou anos, permitindo uma interpretação mais intuitiva das datas. Novamente, há uma série de funções para realizar essas operações.

- `period()`: cria data em período
- `as.period()`: converte datas em período
- `seconds()`: período em segundos
- `minutes()`: período em minutos
- `hours()`: período em horas
- `days()`: período em dias
- `weeks()`: período em semanas
- `months()`: período em meses
- `years()`: período em anos

```

## Criando períodos
period(c(90, 5), c("second", "minute"))

## [1] "5M 90S"

period(c(3, 1, 2, 13, 1), c("second", "minute", "hour", "day", "week"))

## [1] "20d 2H 1M 3S"

## Transformação de períodos
lubridate::seconds(100)

## [1] "100S"

lubridate::minutes(100)

## [1] "100M 0S"

lubridate::hours(100)

## [1] "100H 0M 0S"

lubridate::days(100)

## [1] "100d 0H 0M 0S"

```

```

lubridate::weeks(100)

## [1] "700d 0H 0M 0S"

lubridate::years(100)

## [1] "100y 0m 0d 0H 0M 0S"

Além disso, podemos fazer operações com os períodos, somando e subtraindo.

## Somando datas
lubridate::today() + lubridate::weeks(10)

## [1] "2024-02-26"

## Subtraindo datas
lubridate::today() - lubridate::weeks(10)

## [1] "2023-10-09"

## Criando datas recorrentes
lubridate::today() + lubridate::weeks(0:10)

## [1] "2023-12-18" "2023-12-25" "2024-01-01" "2024-01-08" "2024-01-15"
## [6] "2024-01-22" "2024-01-29" "2024-02-05" "2024-02-12" "2024-02-19"
## [11] "2024-02-26"

```

Por fim, intervalos são períodos de tempo limitados por duas datas, possuindo uma duração com um ponto de partida, que o faz preciso para determinar uma duração. Intervalos são objetos da classe `Interval`. Da mesma forma que para duração e períodos, há uma série de funções para realizar essas operações.

- `interval()`: cria data em intervalo
- `%--%`: cria data em intervalo
- `as.interval()`: converte datas em intervalo
- `int_start()`: acessa ou atribui data inicial de um intervalo
- `int_end()`: acessa ou atribui data final de um intervalo
- `int_length()`: comprimento de um intervalo em segundos
- `int_flip()`: inverte a ordem da data de início e da data de término em um intervalo
- `int_shift()`: desloca as datas de início e término de um intervalo
- `int_aligns()`: testa se dois intervalos compartilham um ponto final
- `int_standardize()`: garante que todos os intervalos sejam positivos
- `int_diff()`: retorna os intervalos que ocorrem entre os elementos de data/horário
- `int_overlaps()`: testa se dois intervalos se sobrepõem
- `%within%`: testa se o primeiro intervalo está contido no segundo

```

## Criando duas datas - início de estudos do R e nascimento do meu filho
r_inicio <- lubridate::dmy("30-11-2011")
filho_nascimento <- lubridate::dmy("26-09-2013")
r_hoje <- lubridate::today()

## Criando intervalos - interval
r_intervalo <- lubridate::interval(r_inicio, r_hoje)

## Criando intervalos - interval %--%
filho_intervalo <- filho_nascimento %--% lubridate::today()

## Operações com intervalos
lubridate::int_start(r_intervalo)

```

```

## [1] "2011-11-30 UTC"
lubridate::int_end(r_intervalo)

## [1] "2023-12-18 UTC"
lubridate::int_length(r_intervalo)

## [1] 380246400
lubridate::int_flip(r_intervalo)

## [1] 2023-12-18 UTC--2011-11-30 UTC
lubridate::int_shift(r_intervalo, duration(days = 30))

## [1] 2011-12-30 UTC--2024-01-17 UTC

Uma operação de destaque é verificar a sobreposição entre dois intervalos.

## Verificar sobreposição - int_overlaps
lubridate::int_overlaps(r_intervalo, filho_intervalo)

## [1] TRUE

## Verificar se intervalo está contido
r_intervalo %within% filho_intervalo

## [1] FALSE
filho_intervalo %within% r_intervalo

## [1] TRUE

Podemos ainda calcular quantos períodos existem dentro de um intervalo, utilizando as operações de / e %/%.

## Períodos dentro de um intervalo - anos
r_intervalo / lubridate::years()

## [1] 12.04918
r_intervalo %/% lubridate::years()

## [1] 12
## Períodos dentro de um intervalo - dias e semandas
filho_intervalo / lubridate::days()

## [1] 3735
filho_intervalo / lubridate::weeks()

## [1] 533.5714

Ainda podemos fazer transformações dos dados para períodos e ter todas as unidades de data e tempo que o
intervalo comprehende.

## Tempo total estudando R
lubridate::as.period(r_intervalo)

## [1] "12y 0m 18d 0H 0M 0S"
## Idade do meu filho
lubridate::as.period(filho_intervalo)

## [1] "10y 2m 22d 0H 0M 0S"

```

Por fim, fusos horários tendem a ser um fator complicador quando precisamos analisar informações instantâneas de tempo (horário) de outras partes do planeta, ou mesmo fazer conversões dos horários. No `lubridate` há funções para ajudar nesse sentido. Para isso, podemos utilizar a função `lubridate::with_tz()` e no argumento `tzone` informar o fuso horário para a transformação do horário.

Podemos descobrir o fuso horário que o R está considerando com a função `Sys.timezone()`.

```
## Fuso horário no R
Sys.timezone()
```

```
## [1] "America/Cayenne"
```

No R há uma listagem dos nomes dos fusos horários que podemos utilizar no argumento `tzone` para diferentes fusos horários.

```
## Verificar os fuso horários
length(OlsonNames())
```

```
## [1] 596
```

```
head(OlsonNames())
```

```
## [1] "Africa/Abidjan"      "Africa/Accra"        "Africa/Addis_Ababa"
## [4] "Africa/Algiers"       "Africa/Asmara"        "Africa/Asmera"
```

Podemos nos perguntar que horas são em outra parte do globo ou fazer as conversões facilmente no `lubridate`.

```
## Que horas são em...
lubridate::with_tz(lubridate::now(), tzone = "GMT")
```

```
## [1] "2023-12-18 13:27:42 GMT"
```

```
lubridate::with_tz(lubridate::now(), tzone = "Europe/Berlin")
```

```
## [1] "2023-12-18 14:27:42 CET"
```

```
## Altera o fuso sem mudar a hora
```

```
lubridate::force_tz(lubridate::now(), tzone = "GMT")
```

```
## [1] "2023-12-18 10:27:42 GMT"
```

Para se aprofundar no tema, recomendamos a leitura do Capítulo 16 Dates and times de Wickham & Grolemund (2017).

## 6.11 purrr

O pacote `purrr` implementa a *Programação Funcional* no R, fornecendo um conjunto completo e consistente de ferramentas para trabalhar com funções e vetores. A programação funcional é um assunto bastante extenso, sendo mais conhecido no R pela família de funções `purrr::map()`, que permite substituir muitos *loops for* por um código mais sucinto e fácil de ler. Não focaremos aqui nas outras funções, pois esse é um assunto extremamente extenso.

Todas as funções deste pacote são listadas na [página de referência](#) do pacote.

Um *loop for* pode ser entendido como uma iteração: um bloco de códigos é repetido mudando um contador de uma lista de possibilidades. Vamos exemplificar com uma iteração bem simples, onde imprimiremos no console os valores de 1 a 10, utilizando a função `for()`, um contador `i` em um vetor de dez números `1:10` que será iterado, no bloco de códigos definido entre {}, usando a função `print()` para imprimir os valores.

A ideia é bastante simples: a função `for()` vai atribuir o primeiro valor da lista ao contador `i`, esse contador será utilizado em todo o bloco de códigos. Quando o bloco terminar, o segundo valor é atribuído ao contador

`i` e entra no bloco de códigos, repetindo esse processo até que todos os elementos da lista tenham sido atribuídos ao contador.

```
## Loop for
for(i in 1:10){
    print(i)
}
```

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

Com essa ideia em mente, a programação funcional faz a mesma operação utilizando a função `purrr::map()`. O mesmo *loop for* ficaria dessa forma.

```
## Loop for com map
purrr::map(.x = 1:10, .f = print)
```

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

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

```

## [[8]]
## [1] 8
##
## [[9]]
## [1] 9
##
## [[10]]
## [1] 10

```

Nessa estrutura, temos:

```

map(.x, .f)

• .x: um vetor, lista ou data frame
• .f: uma função

```

Num outro exemplo, aplicaremos a função `sum()` para somar os valores de vários elementos de uma lista.

```

## Função map
x <- list(1:5, c(4, 5, 7), c(1, 1, 1), c(2, 2, 2, 2))
purrr::map(x, sum)

```

```

## [[1]]
## [1] 15
##
## [[2]]
## [1] 16
##
## [[3]]
## [1] 3
##
## [[4]]
## [1] 10

```

Há diferentes tipos de retornos da família `purrr::map()`.

- `map()`: retorna uma lista
- `map_chr()`: retorna um vetor de strings
- `map_dbl()`: retorna um vetor numérico (double)
- `map_int()`: retorna um vetor numérico (integer)
- `map_lgl()`: retorna um vetor lógico
- `map_dfr()`: retorna um data frame (por linhas)
- `map_dfc()`: retorna um data frame (por colunas)

```

## Variações da função map
purrr::map_dbl(x, sum)

```

```

## [1] 15 16 3 10
purrr::map_chr(x, paste, collapse = " ")

```

```

## [1] "1 2 3 4 5" "4 5 7"      "1 1 1"      "2 2 2 2"

```

Essas funcionalidades já eram conhecidas no *R Base* pelas funções da *família apply()*: `apply()`, `lapply()`, `sapply()`, `vapply()`, `mapply()`, `rapply()` e `tapply()`. Essas funções formam a base de combinações mais complexas e ajudam a realizar operações com poucas linhas de código, para diferentes retornos.

Temos ainda duas variantes da função `map()`: `purrr::map2()` e `purrr::pmap()`, para duas ou mais listas, respectivamente. Como vimos para a primeira função, existem várias variações do sufixo para modificar o retorno da função.

```
## Listas
x <- list(3, 5, 0, 1)
y <- list(3, 5, 0, 1)
z <- list(3, 5, 0, 1)
```

```
## Função map2
purrr::map2_dbl(x, y, prod)
```

```
## [1] 9 25 0 1
```

```
## Função pmap
purrr::pmap_dbl(list(x, y, z), prod)
```

```
## [1] 27 125 0 1
```

Essas funções podem ser usadas em conjunto para implementar rotinas de manipulação e análise de dados com poucas linhas de código, mas que não exploraremos em sua completude aqui. Listamos dois exemplos simples.

```
## Resumo dos dados
penguins |>
  dplyr::select(where(is.numeric)) |>
  tidyr::drop_na() |>
  purrr::map_dbl(mean)
```

	bill_length_mm	bill_depth_mm	flipper_length_mm	body_mass_g
##	43.92193	17.15117	200.91520	4201.75439
##	year			
##	2008.02924			

```
## Análise dos dados
penguins |>
  dplyr::group_split(island, species) |>
  purrr::map(~ lm(bill_depth_mm ~ bill_length_mm, data = .x)) |>
  purrr::map(summary) |>
  purrr::map("r.squared")
```

```
## [[1]]
## [1] 0.2192052
##
## [[2]]
## [1] 0.4139429
##
## [[3]]
## [1] 0.2579242
##
## [[4]]
## [1] 0.4271096
##
## [[5]]
## [1] 0.06198376
```

Para se aprofundar no tema, recomendamos a leitura do Capítulo 21 Iteration de Wickham & Grolemund (2017).

## 6.12 Para se aprofundar

Listamos a seguir livros que recomendamos para seguir com sua aprendizagem em R e tidyverse.

### 6.12.1 Livros

Recomendamos aos (às) interessados(as) os livros: i) Oliveira e colaboradores (2018) [Ciência de dados com R](#), ii) Grolemund (2018) [The Essentials of Data Science: Knowledge Discovery Using R](#), iii) Holmes e Huber (2019) [Modern Statistics for Modern Biology](#), iv) Ismay e Kim (2020) [Statistical Inference via Data Science: A ModernDive into R and the Tidyverse](#), v) Wickham e Grolemund (2017) [R for Data Science: Import, Tidy, Transform, Visualize, and Model Data](#), vi) Zumel e Mount (2014) [Practical Data Science with R Paperback](#), vii) Irizarry (2017) [Introduction to Data Science: Data Analysis and Prediction Algorithms with R](#), e viii) Irizarry (2019) [Introduction to Data Science](#).

### 6.12.2 Links

[Ciéncia de Dados em R](#)

[tidyverse](#)

[STHDA - Statistical tools for high-throughput data analysis](#)

[Estatística é com R! - tidyverse](#)

[Tidyverse Skills for Data Science in R](#)

[Getting Started with the Tidyverse](#)

[Tidyverse Basics](#)

[Manipulando Dados com dplyr e tidyverse](#)

## 6.13 Exercícios

**5.1** Reescreva as operações abaixo utilizando pipes |>.

- `log10(cumsum(1:100))`
- `sum(sqrt(abs(rnorm(100))))`
- `sum(sort(sample(1:10, 10000, rep = TRUE)))`

**5.2** Use a função `download.file()` e `unzip()` para baixar e extrair o arquivo do data paper de médios e grandes mamíferos: [ATLANTIC MAMMALS](#). Em seguida, importe para o R, usando a função `readr::read_csv()`.

**5.3** Use a função `tibble::glimpse()` para ter uma noção geral dos dados importados no item anterior.

**5.4** Compare os dados de penguins (`palmerpenguins::penguins_raw` e `palmerpenguins::penguins`). Monte uma série de funções dos pacotes `tidyverse` para limpar os dados e fazer com que o primeiro dado seja igual ao segundo.

**5.5** Usando os dados de penguins (`palmerpenguins::penguins`), calcule a correlação de Pearson entre comprimento e profundidade do bico para cada espécie e para todas as espécies. Compare os índices de correlação para exemplificar o Paradoxo de Simpson.

**5.6** Oficialmente a pandemia de COVID-19 começou no Brasil com o primeiro caso no dia 26 de fevereiro de 2020. Calcule quantos anos, meses e dias se passou desde então. Calcule também quanto tempo se passou até você ser vacinado.

[Soluções dos exercícios.](#)

## 7 Primeiros passos com uma raster

Organize following: <https://jakubnowosad.com/posts/2020-05-26-intro-to-landscape-ecology/>

<https://youtu.be/dCTOGDnDuvM> at 28 minutes Attribute data operations - Vector attribute subsetting, aggregation and joins - Creating new vector attributes - Raster subsetting - Summarizing raster objects

Spatial data operations - Spatial subsetting - Topological relations - Spatial joining - - zonal raster operations

Geometry operations - on vector - on raster - Vector-raster interactions

Coordinate reference systems - Understanding map projections - Reprojecting spatial data - Modifying map projections

Uma raster é um matriz de valores com coordenadas geográficos. Cada pixel de uma raster representa uma região geográfica, e o valor do pixel representa uma característica dessa região (mais sobre [dados raster](#)).

Em geral é necessário baixar alguns pacotes para que possamos fazer as nossas análises. Precisamos os seguintes pacotes, que deve estar instalado antes:

- [tidyverse](#),
- [terra](#),
- [sf](#),
- [mapview](#),
- [tmap](#).

Carregar pacotes:

```
library(tidyverse)
library(terra)
library(sf)
library(mapview)
library(tmap)
```

Inicialmente iremos gerar uma raster representando uma paisagem bem simples, de 6 por 6 pixels. Você já deve saber que pixel é a unidade básica de uma imagem (lembra da camera do seu celular, 10Mb ou algo assim?!). Vocês devem ter visto sobre pixels e resolução no mesmo em aulas de geoprocessamento. Aqui podemos tratar o pixel como a resolução. Vamos dizer que temos um pixel de 10 metros (res=10 no bloco de código), ou seja, uma quadrado de 10 por 10m, sendo essa, a menor unidade mapeável. Assim sendo, a resolução também tem ligação com escala cartográfica!

Vamos gerar e plotar uma paisagem simples em 4 passos. Primeiramente, a função `rast` cria um objeto do tipo raster. E depois a função `values` atribui valores, e na sequência vamos visualizar os valores com `plot` e `text`.

```
#Função "rast" gera a paisagem virtual (paisagem simulado)
pai_sim <- rast(ncols=6, nrows=6,
                 xmin=1, xmax=60,
                 ymin=1, ymax=60,
                 res=10)
#E essa atribui valores ("values") para os pixels criados acima
values(pai_sim) <- 1
plot(pai_sim) #Essa plota
text(pai_sim) #Essa coloca os valores dos pixels
```

### 7.0.1 Obter e carregar dados (raster)

Mais uma vez vamos aproveitar os dados de MapBiomas. Agora baixar arquivo raster com cobertura de terra no entorno dos rios em 2020, (formato “.tif”, tamanho 3.3 MB). Link: <https://github.com/darrenorris/>

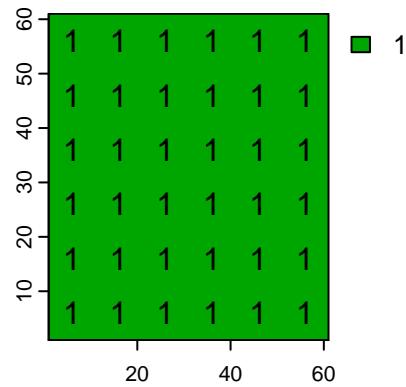


Figura 7.1: Paisagem simples de 36 pixels.

`gisdata/blob/master/inst/raster/mapbiomas_AP_utm_rio/utm_cover_AP_rio_2020.tif`. Lembrando-se de salvar o arquivo (“utm\_cover\_AP\_rio\_2020.tif”) em um local conhecido no seu computador. Agora avisar R sobre onde ficar o arquivo. O código abaixo vai abrir uma nova janela, e você deve buscar e selecionar o arquivo “utm\_cover\_AP\_rio\_2020.tif”:

```
meuSIGr <- file.choose()
```

O código abaixo vai carregar os dados e criar o objeto “mapbiomas\_2020”.

```
mapbiomas_2020 <- rast(meuSIGr)
```

## 7.0.2 Reclassificação

Para simplificar nossa avaliação de escala, reclassificamos a camada mapbiomas\_2020 em uma camada binária de floresta/não-floresta. Essa tarefa de geoprocessamento pode ser realizada anteriormente usando SIG ([QGIS](#)). Aqui vamos reclassificar as categorias de cobertura da terra (agrupando diferentes áreas de cobertura florestal tipos) usando alguns comandos genéricos do R para criar uma nova camada com a cobertura de floresta em toda a região de estudo. Para isso, criamos um mapa do mesmo resolução e extensão, e então podemos redefinir os valores do mapa. Neste caso, queremos agrupar a cobertura da terra categorias 3 e 4 (Formação Florestal e Formação Savânica, respectivamente).

```
# criar uma nova camada de floresta
floresta_2020 <- mapbiomas_2020
# Com valor de 0
values(floresta_2020) <- 0
# Atualizar categorias florestais agrupados com valor de 1
floresta_2020[mapbiomas_2020==3 | mapbiomas_2020==4] <- 1
```

Vizualizar para verificar.

```
# Passo necessário para agilizar o processamento
floresta_2020_modal<-aggregate(floresta_2020, fact=10, fun="modal")
# Plot
tm_shape(floresta_2020_modal) +
  tm_raster(style = "cat",
             palette = c("0" = "#E974ED", "1" ="#129912"),
             legend.show = FALSE) +
  tm_add_legend(type = "fill", labels = c("não-floresta", "floresta"),
                col = c("#E974ED", "#129912"), title = "Classe") +
  tm_scale_bar(breaks = c(0, 25, 50), text.size = 1,
               text.color = "white", position=c("left", "bottom")) +
  tm_layout(legend.position = c("right","top"),legend.bg.color = "white")
```

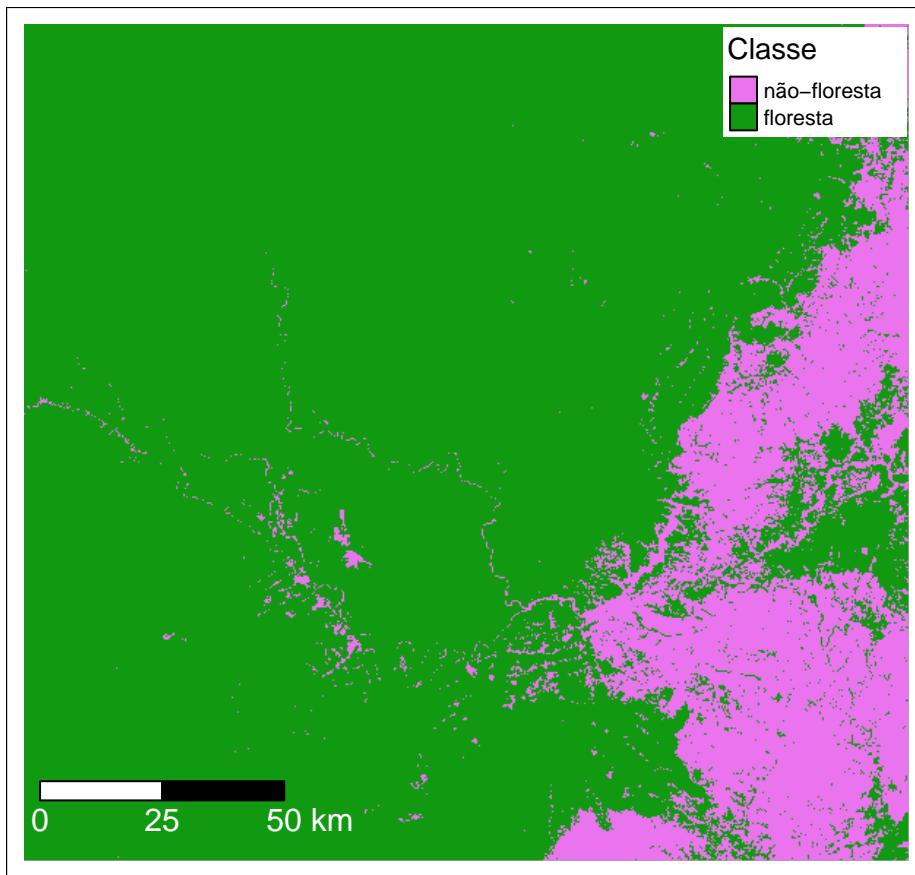


Figura 7.2: Floresta ao redor do Rio Araguari. MapBiomas 2020 reclassificado em floresta e não-floresta. Mostrando os pontos de amostragem (pontos amarelas) cada 5 quilômetros ao longo do rio.

## 8 Primeiros passos com vector

Em geral é necessário baixar alguns pacotes para que possamos fazer as nossas análises. Precisamos os seguintes pacotes, que deve estar instalado antes:

- `tidyverse`,
- `sf`,
- `mapview`,
- `tmap`.

Carregar pacotes:

```
library(tidyverse)
library(sf)
library(mapview)
library(tmap)
```

### 8.1 Obter e carregar dados (vectores)

Precisamos carregar os dados para rios e pontos de amostragem. Baixar arquivo (vector) com os dados (formato “GPKG”, tamanho 54.9 MB). Mais sobre [dados vetoriais](#). O formato aberto [GeoPackage](#) é um contêiner que permite armazenar dados SIG (feições/camadas) em um único arquivo. Por exemplo, um arquivo GeoPackage pode conter vários dados (dados vetoriais e raster) em diferentes sistemas de coordenadas. Todos esses recursos permitem que você compartilhe dados facilmente e evite a duplicação de arquivos.

Baixar o arquivo Link: <https://github.com/darrennorris/gisdata/blob/master/inst/vector/rivers.gpkg> . Lembrando-se de salvar o arquivo (“rivers.gpkg”) em um local conhecido no seu computador.

O formato “GPKG” é diferente de “tif” (raster), o processo de importação é, portanto, diferente. Primeira, avisar R sobre onde ficar o arquivo. O código abaixo vai abrir uma nova janela, e você deve buscar e selecionar o arquivo “rivers.GPKG”:

```
meuSIG <- file.choose()
```

Agora vamos olhar o que tem no arquivo. Depois que vocês rodar o código `st_layers(meuSIG)`, o resultado mostra que o arquivo rivers.GPKG inclui camadas diferentes com pontos (“Point”), linhas (“Line String”) e polígonos (“Polygon”). Além disso, a coluna “crs\_name” mostrar que a sistema de coordenadas é geográfica (WGS84, (EPSG: 4326)

<https://epsg.io/4326>

, e é diferente do arquivo raster:

```
sf::st_layers(meuSIG)
```

```
## Driver: GPKG
## Available layers:
##   layer_name geometry_type features fields crs_name
## 1 centerline   Line String      52     15  WGS 84
## 2 forestloss    Point          276086    12  WGS 84
## 3 canalpoly    Polygon         3       6  WGS 84
## 4 extentpoly50km Polygon         1       0  WGS 84
## 5 midpoints    Point          52     17  WGS 84
## 6 midpoints_hansen Point          52     37  WGS 84
## 7 cachoeira_caldeirao Point         1       2  WGS 84
## 8 porto_grande Point          1       1  WGS 84
## 9 icmbio_base   Point          1       1  WGS 84
## 10 direct_affect Polygon         1       2  WGS 84
## 11 midpoints_hansen_distances Point          52     43  WGS 84
```

```
## 12      midpoints_hansen_ffr      Point      52      82      WGS 84
## 13      midpoints_hansen_ffril    Point      52      91      WGS 84
## 14      direct_affect_line     Line String      1       2      WGS 84
```

Nós só precisamos de duas dessas camadas. O código abaixo vai carregar as camadas que precisamos e criar os objetos “rsm” e “rsl”. Assim, agora temos dados com: pontos cada 5 km ao longo os rios (“rsm”) e a linha central de rios (“rsl”).

```
# pontos cada 5 km
rsm <- sf::st_read(meuSIG, layer = "midpoints")

## Reading layer `midpoints' from data source
##   `C:\Users\user\Documents\Articles\gis_layers\gisdata\inst\vector\rivers.gpkg'
##   using driver `GPKG'
## Simple feature collection with 52 features and 17 fields
## Geometry type: POINT
## Dimension:      XY
## Bounding box:  xmin: -52.01259 ymin: 0.7175827 xmax: -51.29688 ymax: 1.330365
## Geodetic CRS:  WGS 84

# linha central de rios
rsl <- sf::st_read(meuSIG, layer = "centerline")

## Reading layer `centerline' from data source
##   `C:\Users\user\Documents\Articles\gis_layers\gisdata\inst\vector\rivers.gpkg'
##   using driver `GPKG'
## Simple feature collection with 52 features and 15 fields
## Geometry type: LINESTRING
## Dimension:      XY
## Bounding box:  xmin: -52.01443 ymin: 0.7094595 xmax: -51.2924 ymax: 1.352094
## Geodetic CRS:  WGS 84
```

## 8.2 Visualizar os arquivos (camadas vector)

Visualizar para verificar. Mapa com linha central e pontos de rios em trechos de 5km.

```
ggplot(rsl) +  
  geom_sf(aes(color=rio)) +  
  geom_sf(data = rsm, shape=21, aes(fill=zone))
```

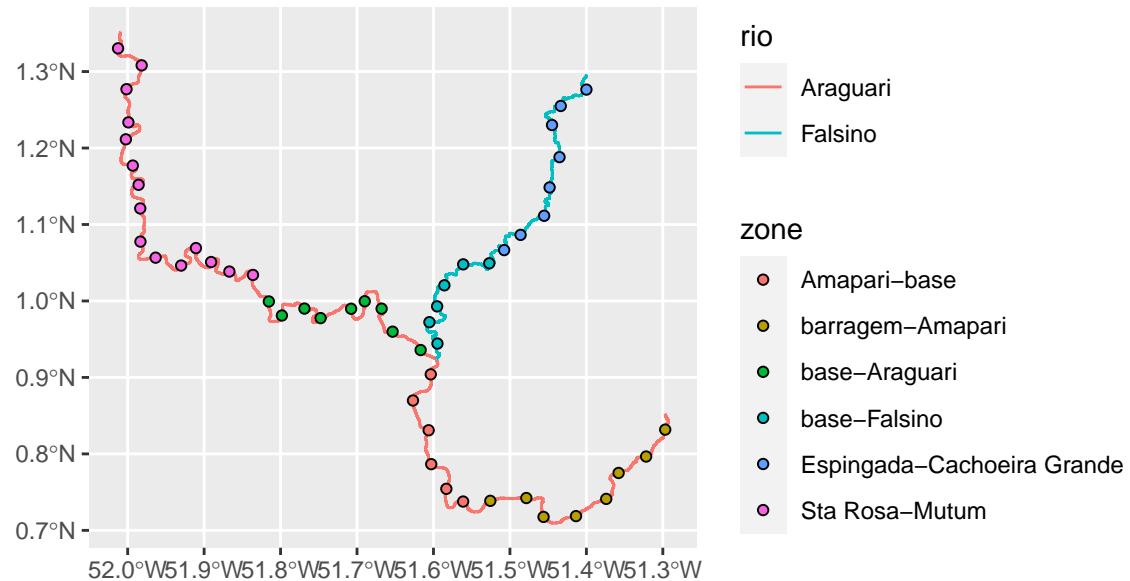


Figura 8.1: Pontos ao longo dos rios a montante das hidrelétricas no Rio Araguari.

Mapa interativo (funcione somente com internet). Mostrando agora com fundo de mapas “base” (OpenStreetMap/ESRI etc)

```
#  
mapview(rsl, zcol = "rio")
```

Em andamento .....

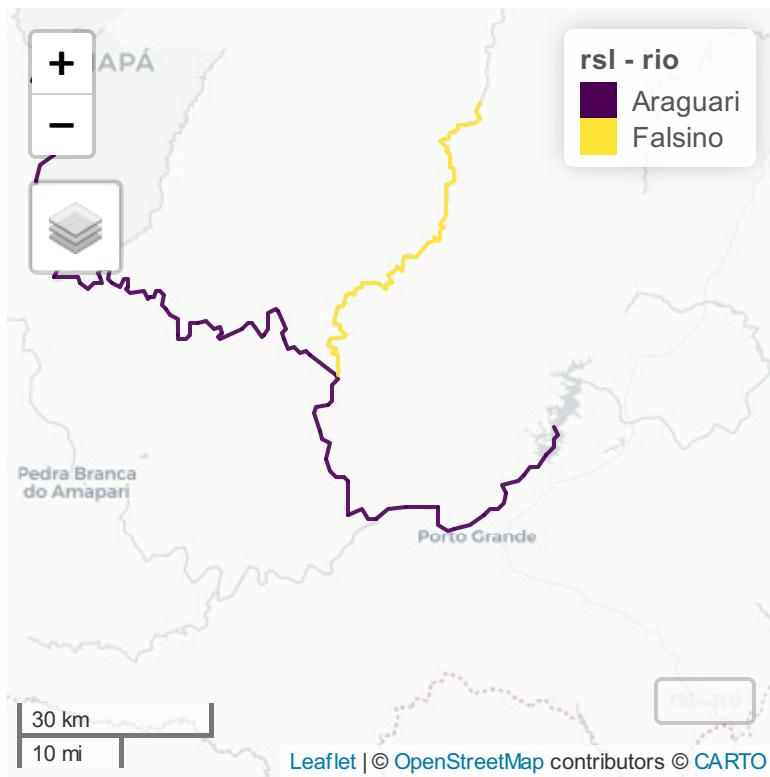


Figura 8.2: Linhas dos rios a montante das hidrelétricas no Rio Araguari.

## Part V

# Encerramento

## Bibliografia

- Fletcher, R., and M.-J. Fortin. 2018. *Spatial ecology and conservation modeling: applications with r*. Cham: Springer International Publishing. <https://doi.org/10.1007/978-3-030-01989-1>.
- Holmes, S., and W. Huber. 2019. *Modern Statistics for Modern Biology*. Cambridge, United Kingdom: Cambridge university press.
- Irizarry, R. A. 2017. *Data Analysis for the Life Sciences with R*. Boca Raton: CRC Press, Taylor & Francis Group.
- Irizarry, Rafael A. 2019. *Introduction to Data Science: Data Analysis and Prediction Algorithms with R*. 1<sup>a</sup> edição. Chapman; Hall/CRC.
- Ismay, C., and A. Y.-S. Kim. 2020. *Statistical Inference via Data Science: A ModernDive into R and the Tidyverse*. Chapman & Hall/CRC the R Series. Boca Raton: CRC Press / Taylor & Francis Group.
- Oliveira, P. F., S. Guerra, and R. McDonnell. 2018. *Ciência de Dados Com R - Introdução*.
- Wickham, H. 2014. “Tidy Data.” *Journal of Statistical Software* 59 (10). <https://doi.org/10.18637/jss.v059.i10>.
- . 2016. *ggplot2: Elegant graphics for data analysis*. Springer-Verlag New York. <https://ggplot2.tidyverse.org>.
- Wickham, H., M. Averick, J. Bryan, W. Chang, L. McGowan, R. François, G. Grolemund, et al. 2019. “Welcome to the Tidyverse.” *Journal of Open Source Software* 4 (43): 1686. <https://doi.org/10.21105/joss.01686>.
- Wickham, H., and G. Grolemund. 2017. *R for data science: import, tidy, transform, visualize, and model data*. First edit. Sebastopol, CA: O'Reilly.
- Wilkinson, L., and G. Wills. 2005. *The grammar of graphics*. 2nd ed. Statistics and Computing. New York: Springer.
- Williams, G. J. 2018. *The Essentials of Data Science: Knowledge Discovery Using R*. Boca Raton: Taylor & Francis, CRC Press.
- Zumel, N., and J. Mount. 2014. *Practical Data Science with R*. Shelter Island, NY: Manning Publications Co.

## A Annexo 1

Exemplos de código R usado para - baixar Mapbiomas, - recortar para uma região de interesse - plotar com legenda

```
library(plyr)
library(tidyverse)
library(terra)
library(sf)
library(readxl)
library(tmap)
```

### A.1 Download mapbiomas cover

Site: <https://brasil.mapbiomas.org/colecoes-mapbiomas/>. Any zeros are NA (NODATA). <http://forum.mapbiomas.ecostage.com.br/t/pixels-com-valor-zero/170/5> . And “27” “Não observado”. No exemplo os arquivos vai ficar no working directory.

```
#Get mapbiomas data
#
url_text <- "https://storage.googleapis.com/mapbiomas-public/initiatives/brasil/collection_8/lclu/cover
layer_names <- paste("brasil_coverage_", 1985:2022, sep="")
file_type <- ".tif"
filenames_mapbioma_8 <- paste(layer_names, file_type, sep="")
urlnames_mapbioma_8 <- paste(url_text, layer_names, file_type, sep="")

#set timeout to 20 minutes as big files and teeny tiny internet connection
options(timeout = max((60*20),getOption("timeout")))
# test with one year - 1985
res <- utils::download.file(url=urlnames_mapbioma_8[1],
                           destfile=filenames_mapbioma_8[1],
                           method="auto", quiet = FALSE, mode = "wb",
                           cacheOK = TRUE)

#make list
dflist <- data.frame(layer_names = layer_names,
                      urls = urlnames_mapbioma_8,
                      filenames = filenames_mapbioma_8)
alist <- split(dflist, f = dflist$layer_names)

#make function to automatically process list elements
get_mapbiomas <- function(x){
  res <- utils::download.file(url=x$urls,
                               destfile=x$filenames,
                               method="auto", quiet = FALSE, mode = "wb",
                               cacheOK = TRUE)
  res
}

#run function in blocks
lapply(alist[2], get_mapbiomas)
lapply(alist[3:5], get_mapbiomas)
lapply(alist[6:8], get_mapbiomas)
lapply(alist[9:20], get_mapbiomas)
lapply(alist[21:30], get_mapbiomas)
lapply(alist[31:37], get_mapbiomas)
```

## A.2 Crop mapbiomas cover

Identify all tif files in a folder.

```
#works one folder at a time
get_files <- function(folder_name = NA) {
  library(tidyverse)
  folder_location <- folder_name
  in_files <- list.files(folder_location,
                         pattern = ".tif", full.names = TRUE)
  data.frame(folder_id = folder_location, file_id = in_files) |>
    group_by(folder_id, file_id) |>
    dplyr::summarise(file_count = dplyr::n()) |>
    ungroup() -> df_muni_tif
  return(df_muni_tif)
}

infolder <- "E:/mapbiomas"
df_muni_tif <- get_files(folder_name = infolder)
#update
df_muni_tif |>
  dplyr::mutate(state_code = str_sub(folder_id, -2, -1),
                ) -> df_muni_tif
df_muni_tif$state_code <- "AP"
```

Area of interest - format extent for use by terra.

```
#extent from 50 km around river upstream of cachoeira caldeirao
meuSIG <- file.choose()
#"C:\\Users\\Darren\\Documents\\gisdata\\vector\\rivers.GPKG"
# "C:\\Users\\user\\Documents\\Articles\\gis_layers\\gisdata\\inst\\vector\\rivers.gpkg"
rsl <- st_read(meuSIG, layer = "centerline")
rsl_50km <- st_union(st_buffer(rsl, dist=50000))
myextent <- ext(vect(rsl_50km))
```

Now to crop and project to desired coordinate system (epsg:31976).

```
# This function reads in file, crops and projects to new coordinate system.
# Then exports result as Geotiff (.tif).
crop_proj <- function(x, state_id = NA,
                      sf_state = NA) {
  library(plyr)
  library(tidyverse)
  library(terra)
  library(sf)
  library(stringi)

  state_sigla = x$state_code
  rin <- x$file_id
  rbig <- terra::rast(rin)
  layer_name <- names(rbig)
  layer_year <- stri_sub(layer_name,-4,-1)
  out_name <- paste("mapbiomas_cover", layer_year, sep="_")

  e2 <- myextent

#Crop
```

```

rtest_mask <- crop(rbig, e2, snap="out")
#rtest_mask <- rbig
rm("rbig")

#Project
new_crs_utm <- "epsg:31976"
rtest_mask <- project(rtest_mask, new_crs_utm, method = "near")
names(rtest_mask) <- out_name

#Export
#folder <- paste(state_sigla, "_", "equalarea", sep = "")
#folder_utm <- paste(state_sigla, "_", "utm", sep = "")

#folder_utm_muninorte <- paste(state_sigla, "_", "utm_muni_norte", sep = "")
#folder_utm_munimaza <- paste(state_sigla, "_", "utm_munis_maza_santana", sep = "")
#folder_utm_cutias <- paste(state_sigla, "_", "utm_muni_cutias", sep = "")
#folder_path <- paste("C:/Users/Darren/Documents/2022_Norris_gdp_deforestation/AmazonConservation/mapb
#outfile <- paste("ea_cover_", state_sigla, "_",
#                  layer_year, ".tif", sep = "")
folder_utm_rio <- paste(state_sigla, "_", "utm_rio", sep = "")
outfile_utm <- paste("utm_cover_AP_rio_", layer_year, ".tif", sep = "")
f <- file.path(folder_utm_rio, outfile_utm)
writeRaster(rtest_mask, f, datatype = "INT1U", NAflag=27,
            gdal=c("COMPRESS=DEFLATE"), overwrite = TRUE)

#clear temporary files
tmpFiles(current =TRUE, remove = TRUE)

endtime <- Sys.time()
textout <- paste(outfile_utm, ":", endtime, sep="")
print(textout)
}

#run
plyr::a_ply(df_muni_tif, .margins = 1,
            .fun = crop_proj)

```

### A.3 Plot with legend

Combine all years into multi layer raster.

```

infolder_rio <- "E:/mapbiomas/AP_utm_rio"
df_tif_rio <- get_files(folder_name = infolder_rio)
r85a22 <- rast(df_tif_rio$file_id)
writeRaster(r85a22, "utm_cover_AP_rio_85a22.tif", datatype = "INT1U", NAflag=27,
            gdal=c("COMPRESS=DEFLATE"), overwrite = TRUE)

```

Legend.

```
mapbiomas8_legenda <- read.csv2("legend/Codigos-da-legenda-colecao-8.csv")
```

Plot.

```

class_vals <- mapbiomas8_legenda$Class_ID
class_color <- mapbiomas8_legenda$Color

```

```

names(class_color) <- mapbiomas8_legenda$Class_ID
#labels
my_label <- mapbiomas8_legenda$Descricao
names(my_label) <- mapbiomas8_legenda$Class_ID

# Plot one year to check
tm_shape(subset(r85a22n_ag, c(1, 2, 37,38))) +
  tm_raster(style = "cat", palette = class_color, labels = my_label, title="") +
  tm_scale_bar(breaks = c(0, 10), text.size = 1,
                position=c("right", "bottom"))

# Now all years together.
tm_shape(r85a22n_ag) +
  tm_raster(style = "cat", palette = class_color, labels = my_label, title="") +
  tm_scale_bar(breaks = c(0, 10), text.size = 1,
                position=c("right", "bottom")) +
  tm_facets(ncol=5) +
  tm_layout(legend.bg.color="white", legend.outside = TRUE,
            legend.outside.position = "right",
            panel.labels = c('1985', '1986', '1987', '1988','1989', '1990',
                            '1991', '1992', '1993', '1994', '1995','1996',
                            '1997', '1998',
                            '1999', '2000', '2001', '2002', '2003',
                            '2004', '2005', '2006', '2007',
                            '2008', '2009', '2010', '2011', '2012',
                            '2013', '2014', '2015',
                            '2016', '2017', '2018', '2019',
                            '2020', '2021', '2022')) -> figmap

png("figmap.png", width=6, height=9,
    units="in", res = 600)
figmap
invisible(dev.off())

```

## B Soluções de exercícios