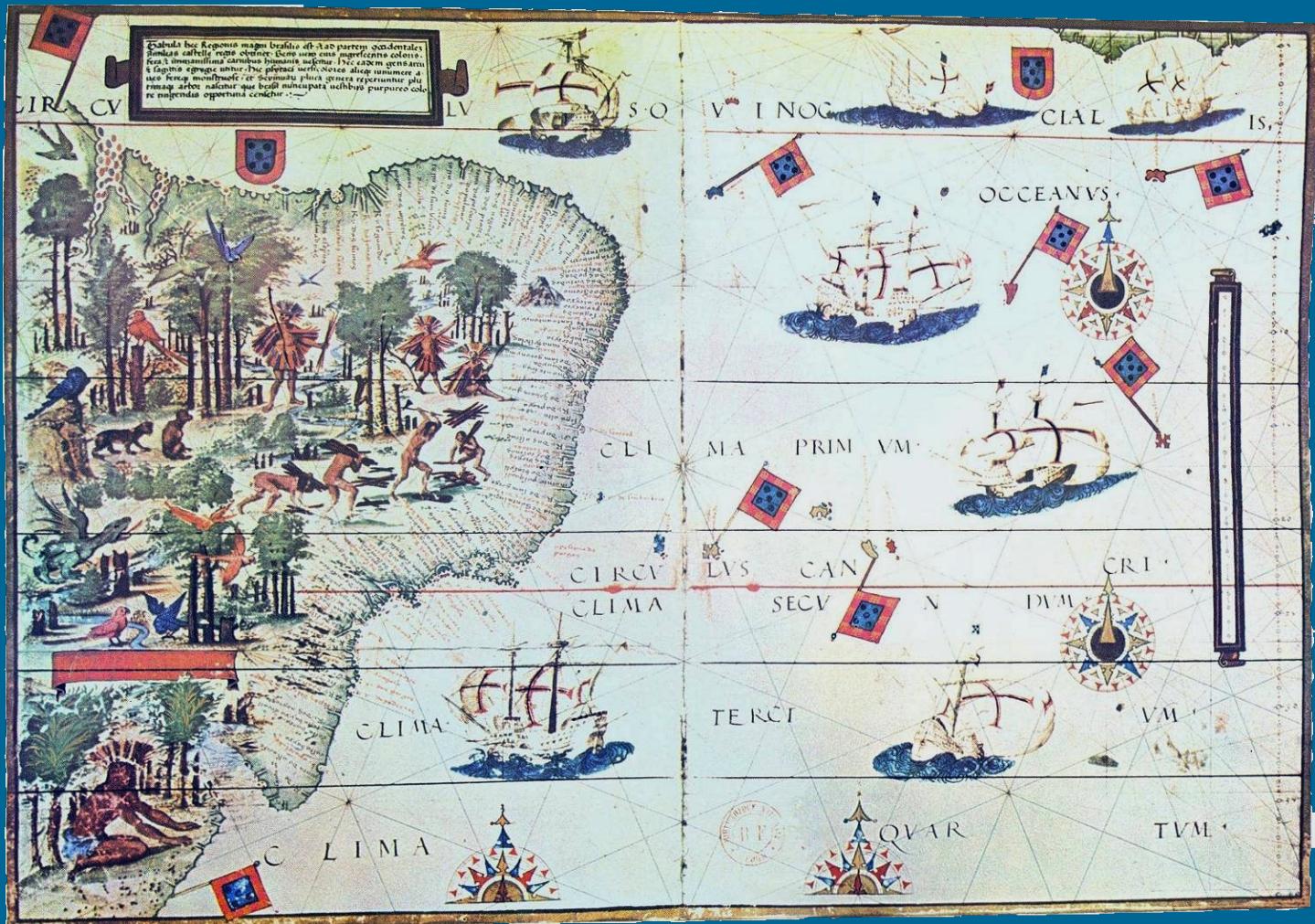


Análise espacial com R

*Jean-François Mas, Marise Barreiros Horta,
Rodrigo Nogueira de Vasconcelos,
& Elaine Cristina Barbosa Cambui*



A obra da capa foi realizada por Pedro Reinel, Jorge Reinel, Lopo Homem (cartógrafos) e António de Holanda (pintor e miniaturista) [Domínio Público]. Foi obtida via Wikimedia Commons https://commons.wikimedia.org/wiki/File:Brazil_16thc_map.jpg

ANÁLISE ESPACIAL COM R



UNIVERSIDADE ESTADUAL DE FEIRA DE SANTANA

Evandro do Nascimento Silva
Reitor
Norma Lucia Fernandes de Almeida
Vice-reitora



Eraldo Medeiros Costa Neto
Diretor
Valdomiro Santana
Editor
Zenailda Novais
Assistente Editorial

CONSELHO EDITORIAL

Adeílalo Manoel Pinto
Antonio César Ferreira da Silva
Antônio Vieira da Andrade Neto
Diógenes Oliveira Senna
Geciara da Silva Carvalho
Gilberto Marcos de Mendonça Santos
Jorge Aliomar Barreiros Dantas
Marluce Nunes Oliveira
Nilo Henrique Neves dos Reis

Jean-François Mas ^{†§}
Marise Barreiros Horta [‡]
Rodrigo Nogueira de Vasconcelos ^{§*}
Elaine Cristina Barbosa Cambui ^{§¶}

[†] Centro de Investigaciones en Geografía Ambiental, Universidad Nacional Autónoma de México, Morelia, Mich., México.

[§] Instituto Nacional de Ciência e Tecnologia em Estudos Interdisciplinares e Transdisciplinares em Ecologia e Evolução (INCT IN-TREE), Salvador, BA, Brasil.

[‡] PPG em Ecologia, Conservação e Manejo da Vida Silvestre, Universidade Federal de Minas Gerais, Belo Horizonte, MG, Brasil.

^{*} PPG em Modelagem e Ciências da Terra e do Ambiente, Universidade Estadual de Feira de Santana, Feira de Santana, BA, Brasil.

[¶] Núcleo de Inovação Tecnológica em Reabilitação, Universidade Federal da Bahia, Salvador, BA, Brasil.

ANÁLISE ESPACIAL COM R



Feira de Santana
2019

Copyright © 2019 by Mas J-F, Barreiros Horta M., Nogueira de Vasconcelos R. e Barbosa Cambu E.C. Essa obra é um derivado de *Mas, J-F., Análisis espacial con R, usa R como un sistema de información geográfica*. Tem uma licença Creative Commons Atribuição-Não Comercial-Compartilha Igual 4.0 Brasil (CC BY-NC-SA 4.0 BR)



Projeto gráfico: Jean-François Mas
Editoração eletrônica: Jean-François Mas
Revisão Textual: Marise Barreiros Horta
Capa: Jean-François Mas
Revisão de provas: Marise Barreiros Horta
Normalização bibliográfica: Jean-François Mas



Ficha catalográfica: Biblioteca Central Julieta Carteado — UEFS

A551a Análise espacial com R / Jean-François Mas...[et. al.] .-
Feira de Santana: UEFS Editora, 2019.
102 p.: il.

ISBN: 978-85-5592-091-2

1. Geografia-Geoprocessamento. 2. Geoestatística. 3. Geomática.
4. Mas, Jean-François, et. al.. I.
Universidade Estadual de Feira de Santana.

CDU: 91

Elaboração: Tatiane Souza Santos — Bibliotecária — CRB- 5/1634

UEFS Editora,
Av. Transnordestina, s/n, Campus da UEFS, CAU III
44.036-900 — Feira de Santana, BA
Telefone: (75) 3161-8380
Email: editora@uefs.br

AGRADECIMENTOS

Esta publicação recebeu apoio do INCT em Estudos Interdisciplinares e Transdisciplinares em Ecologia e Evolução (IN-TREE), da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) no âmbito da Chamada INCT - MCTI/CNPq/CAPES/FAPs nº 16/2014 e do *Programa de Apoyos para la Superación del Personal Académico* (PASPA) da *Dirección General Asuntos del Personal Académico* (DGAPA) da *Universidad Nacional Autónoma de México* (UNAM).

M. Barreiros Horta agradece a CAPES pelo suporte financeiro fornecido durante a realização deste trabalho, concedida através de bolsa de doutoramento do Programa de Pós-Graduação em Ecologia, Conservação e Manejo da Vida Silvestre, Universidade Federal de Minas Gerais (UFMG).

Agradecemos a três revisores anônimos cujos comentários contribuíram para melhorar a qualidade e o rigor do livro.

Agradecemos também aos desenvolvedores dos programas de código aberto que foram utilizados nesse livro (Knitr, L^AT_EX, raster, sf e muitos outros!).

Conteúdo

Introdução	3
0.1 Propósito desse livro	3
0.2 Organização do livro e convenções	3
0.2.1 Organização do livro	3
0.2.2 Convenções de escrita	4
0.2.3 Dados dos exemplos e exercícios	4
1 Instalação e apresentação de R e RStudio	5
1.1 Instalação de R e RStudio	5
1.1.1 Windows	5
1.1.2 Linux	5
1.1.3 Mac	6
1.2 Introdução ao R e ao Rstudio	6
1.3 Instalando pacotes no R	7
2 Operações básicas no R	9
2.1 Operações básicas	9
2.2 Importação de dados em R	12
2.3 Operações com tabelas	13
2.4 Desenho de gráficos	16
2.5 Relação entre duas variáveis	17
2.6 Operações sobre margens: apply	19
2.7 Operações por grupos	20
2.8 Criação de funções	21
2.9 Repetições e condições	21
3 Organização de objetos espaciais em R	25
3.1 Dados vetoriais: modelo "simple feature"	25
3.1.1 Camada de Pontos	26
3.1.2 Camada de linhas	29
3.1.3 Camada de polígonos	32
3.2 Dados raster: RasterLayer no pacote raster	36
4 Importação / exportação de dados espaciais	39
4.1 Importação de arquivos shape	39
4.1.1 Importação com sf	39
4.2 Importação de arquivos vetoriais de outros formatos	42

4.3	Exportação para outros formatos	42
4.4	Importação / exportação de dados raster	43
5	Operações básicas de SIG (formato vetor)	47
5.1	Algumas operações de análise espacial	47
5.2	Análise espacial em formato vetor	51
6	Operações básicas de SIG (dados raster)	59
6.1	Operações de análise espacial	59
6.1.1	Álgebra de mapas	60
6.1.2	Mosaicagem e corte	61
6.1.3	Reclassificação	62
6.1.4	Modificação do arranjo espacial	64
6.1.5	Operações focais e zonais	65
6.2	Análise espacial no formato raster	67
7	Análise geoestatística: Identificação de <i>hot spots</i>	73
7.1	Método de Getis Ord	73
7.2	Aplicação para a detecção de áreas com altas taxas de desmatamento	73
8	Elaboração de mapas	77
8.1	Elaboração de arquivos de imagens	77
8.2	Gerenciamento de cores	78
8.3	Elaboração do mapa de densidade	79
8.4	Elaboração de mapas com base em dados raster	82
9	Interagindo R com QGIS e Dinamica EGO	85
9.1	Sistema de Informação Geográfica QGIS	85
9.2	Plataforma de modelagem Dinamica EGO	86
10	Recursos da internet	91
10.1	Blogs e Tutoriais (inglês)	91
10.2	Tutoriais e teses (português)	91
10.3	Repositórios de pacotes	91
10.4	Livros	91
	Índice de termos	91
	Referências	93

Introdução

0.1 Propósito desse livro

R, uma plataforma de análise estatística com ferramentas gráficas muito avançadas, tem sido uma referência na análise estatística durante muitos anos. A obtenção e distribuição são gratuitas e está sob a Licença Pública Geral (GPL) do projeto colaborativo de Software livre GNU. Esta licença tem como objetivo declarar que o software é gratuito e protegê-lo contra tentativas de apropriação que restrinja essas liberdades a novos usuários, quando o software é distribuído ou modificado. Como é um programa de código aberto, R é, portanto, livre, mas acima de tudo, é o resultado do esforço de milhares de pessoas ao redor do mundo que colaboraram no seu desenvolvimento. Isso permite solucionar erros de programação (*bugs*) mais rapidamente e desenvolver pacotes que são suplementos especializados para temas específicos, elaborados por especialistas em alguma parte do mundo. Esses pacotes são baseados muitas vezes em métodos inovadores aplicados para uma ampla gama de problemas: para processar dados diversos (como censos, séries temporais, sequências genéticas ou informação econômica), implementar uma grande variedade de métodos estatísticos, incrementar interfaces gráficas e permitir a interação com processadores de texto como Latex¹. Este livro é elaborado usando esses pacotes, mais especificamente o pacote Knitr (Xie, 2013).

À primeira vista, R pode parecer hostil aos usuários acostumados a gerenciar programas de computador com menus e opções selecionados com o mouse, porque é baseado em linhas de comando. No entanto, depois de ter (facilmente) superado esse obstáculo, esses usuários perceberão que o uso de pequenos "scripts" permitem executar uma seqüência de operações de forma mais eficiente do que uma longa sequência de "clics", sem esquecer o risco de tendinite. O R permite repetir facilmente o mesmo procedimento com dados diferentes ou com modificações em uma cadeia de processamento já implementada. Além disso, reduz consideravelmente a possibilidade de cometer erros em uma cadeia de operações de rotina e possibilita a documentação do processamento realizado.

Durante os últimos anos, foram criados diferentes pacotes de R dirigidos à análise espacial (Mas *et al.*, 2018). Ainda assim, existem poucos livros focados na análise espacial com R². Este livro destina-se a usuários com conhecimento básico de Sistemas de Informação Geográfica (SIG) que desejam iniciar a gestão e análise de dados espaciais com R. Portanto, não requer nenhum conhecimento prévio deste programa, mas um conhecimento básico do SIG. O livro pretende permitir ao leitor dar os primeiros passos na gestão de R para a análise espacial sem muitos tropeços. Para continuar com aplicativos mais avançados, há um ótimo número de fontes de informação (ver seção 10).

0.2 Organização do livro e convenções

0.2.1 Organização do livro

O livro foi organizado da seguinte maneira: no primeiro capítulo, é explicado como instalar R e RStudio e os principais elementos da interface RStudio são apresentados. É recomendado instalar ambos os programas para experimentar os códigos dos exercícios. No segundo capítulo, é feita uma introdução ao uso básico do R. O leitor com conhecimento prévio de R pode ir diretamente para o capítulo seguinte. No terceiro capítulo, apresentamos

¹Para convencer-se a ver a lista de pacotes em <https://cran.r-project.org/>

²Consultar os livros de Bivand *et al.* (2008) e Brunsdon & Comber (2015)

como os dados espaciais em R estão estruturados utilizando os pacotes *sf* e *raster*, os dois principais pacotes que usaremos ao longo deste livro. Este capítulo pode parecer um pouco árido. Na verdade, você pode lidar com dados espaciais sem entrar nos detalhes da organização da informação. No entanto, é importante, e ajuda muito, conhecer e entender essas informações. No capítulo 4, apresentamos algumas maneiras de interagir com dados geográficos entre R e outros sistemas de gerenciamento de informações geográficas através de procedimentos de importação / exportação de dados em formato vetorial ou de imagem, bem como alguns métodos para converter informações entre vetor e raster. Nos capítulos 5 e 6, são apresentadas operações SIG básicas, respectivamente, com dados em formato vetorial e raster. No Capítulo 7, apresentamos algumas análises do campo da geoestatística que podem ser realizadas com pacotes R. No oitavo capítulo mostramos algumas maneiras de elaborar cartografia. Finalmente, o capítulo 9 apresenta ao leitor as técnicas para fazer R interagir com o sistema de informação geográfica (SIG) de código aberto Q-GIS e a plataforma de modelagem espacial Dinamica EGO.

0.2.2 Convenções de escrita

Para facilitar a leitura deste documento, os nomes de pacotes (como por exemplo **maptools**) estão ressaltados em **negrito** e os nomes de funções em "**negrito/italico**". As linhas de comando, tal como se escreve em um script ou na janela de console, encontram-se em estilo da fonte do tipo **typewriter**. O tipo ou classe dos objetos estão também em **Sans Serif**. Alguns anglicismos estão em **italico** como no seguinte exemplo: a função **stepAIC()**, do pacote **MASS** utiliza por predefinição (*default*) a opção "forward" como observado no código que se segue.

```
stepAIC(fit)
stepAIC(fit, direction="forward")
```

0.2.3 Dados dos exemplos e exercícios

Os dados (tabelas, scripts, mapas e imagens) para levar a cabo as operações apresentadas neste livro estão disponibilizadas em:

- http://lae.ciga.unam.mx/recursos/dados_pt.zip
- <https://bit.ly/2O6BD2s>.

1. Instalação e apresentação de R e RStudio

O R está disponível para sistemas operacionais Linux, Windows e Mac. Existem várias maneiras de obtê-lo e instalá-lo. RStudio é uma interface gráfica muito útil para o uso do R. No entanto, é possível utilizar o R sem interface ou com outras, como R Commander, RKWard ou Tinn-R.

1.1 Instalação de R e RStudio

1.1.1 Windows

- Para obter R para Windows, entre na página *Comprehensive R Archive Network* (CRAN) <https://cran.r-project.org/mirrors.html>.
- Escolha o espelho (*CRAN mirrors*).
- Clique em *Baixar R para Windows* (*Download R for Windows*) e *Instale R pela primeira vez* (*Install R for the first time*).
- Clique en *Baixar R 3.5.1 para Windows* (*Download R 3.5.1 for Windows*) (ou a versão mais atual disponível), salve o arquivo R para Windows e execute-o.

Na página <https://www.rstudio.com/products/rstudio/download/>, achará o executável para instalar a versão gratuita do RStudio para Windows.

1.1.2 Linux

O R está incluído nos repositórios da maioria das distribuições Linux. Por exemplo, no Ubuntu (versão 18.04.1 LST), ele é facilmente instalado através da utilização do centro de software ou usando o código que se segue na janela do terminal:

```
sudo apt-get update  
sudo apt-get install r-base r-base-dev  
R --version # verifica a versão instalada
```

Esses repositórios nem sempre possuem a versão mais recente do R. Para instalar a última versão (versão 3.5 em novembro 2018), siga as etapas abaixo (para mais detalhes, consultar blogs como o de S. Rochette, 2018). Primeiro, precisa verificar se usou um dos servidores espelho do R como <https://cran.r-project.org/mirrors.html>. O nome deste servidor pode ser listado no arquivo /etc/apt/sources.list, que você pode editar com o editor de texto **vi**:

```
sudo vi /etc/apt/sources.list
```

Para mover o cursor, dentro do texto, use as teclas das setas direcionais. Encontre a(s) linha(s) que se parecem com `deb https://mirror.../pub/CRAN/xenial/` e comente com `#`. Para sair gravando, tecle Esc e ir para acessar a linha de comando do editor, onde você pode dar as instruções `:wq`. Para sair sem gravar as alterações feitas no arquivo, use o comando `:q!`.

É também necessário remover as versões anteriores (se houver instalações anteriores) utilizando:

```
sudo apt purge r-base* r-recommended r-cran-*
sudo apt autoremove
sudo apt update
```

Atualize o sistema e importe a chave pública:

```
sudo add-apt-repository 'deb https://cloud.r-project.org/bin/linux/ubuntu bionic-cran35/'
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys E084DAB9
sudo apt update
```

Instale a versão mais recente do R:

```
sudo apt install r-base r-base-core r-recommended
```

Qualquer que seja a distribuição do Ubuntu LTS que você está usando, é melhor instalar o UbuntuGIS PPA para poder trabalhar com pacotes R para análise de dados geográficos.

```
sudo add-apt-repository 'deb http://ppa.launchpad.net/ubuntugis/ubuntugis-unstable/
                           ubuntu bionic main'
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 314DF160
sudo apt update
sudo apt install libgdal1-dev libproj-dev libgeos-dev libudunits2-dev libv8-dev
                  libcairo2-dev libnetcdf-dev
```

Em <https://www.rstudio.com/products/rstudio/download/>, encontram-se os arquivos de instalação RStudio para diferentes distribuições Linux. A instalação pode ser realizada usando programas como Synaptic, tanto o centro de software como o terminal.

1.1.3 Mac

Os arquivos de instalação R para Mac podem ser baixados de <https://cran.r-project.org/bin/macosx/>, os arquivos de RStudio de <http://www.rstudio.com/products/rstudio/download/>.

1.2 Introdução ao R e ao Rstudio

Como já mencionado, os usuários acostumados a outros programas notarão a falta de "menus" (opções para *clicar*): para usar R é necessário digitar os comandos. No entanto, uma vez familiarizados, notarão que o mecanismo de comando é mais flexível e possibilita a programação.

É importante levar em conta alguns detalhes: R é "*case-sensitive*", ou seja, é sensível à diferença entre minúsculas e maiúsculas e, portanto, "Nome" é diferente de "nome". O separador decimal é o ponto ".", a vírgula é usada para separar os elementos de uma lista. Recomenda-se evitar o uso de acentos nos diretórios e nos nomes dos arquivos. Nos exercícios do livro, usaremos R e Rstudio para analisar dados espaciais. RStudio é uma interface gráfica do R que, por sua vez, é a linguagem de programação. Vejamos esta interface (Figura 1):

- A janela no canto inferior esquerdo é o *console*, onde as linhas de comando são digitadas. Ela interpreta qualquer entrada como um comando a ser executado. Esses comandos e sua sintaxe fornecem uma maneira bastante natural e intuitiva de acessar dados e executar operações de processamento e estatística. No entanto, é mais fácil escrever seu código como um *script*: um texto com uma seqüência de operações a ser executada.
- A janela na parte superior esquerda é o editor de texto dos *scripts*. Um *script* é um arquivo de texto com uma série de instruções. Você pode executar uma única linha do *script*, um conjunto (bloco) de linhas ou o *script* inteiro.
- Os gráficos elaborados aparecem na janela à direita. A área de trabalho e o histórico estão no canto superior direito.

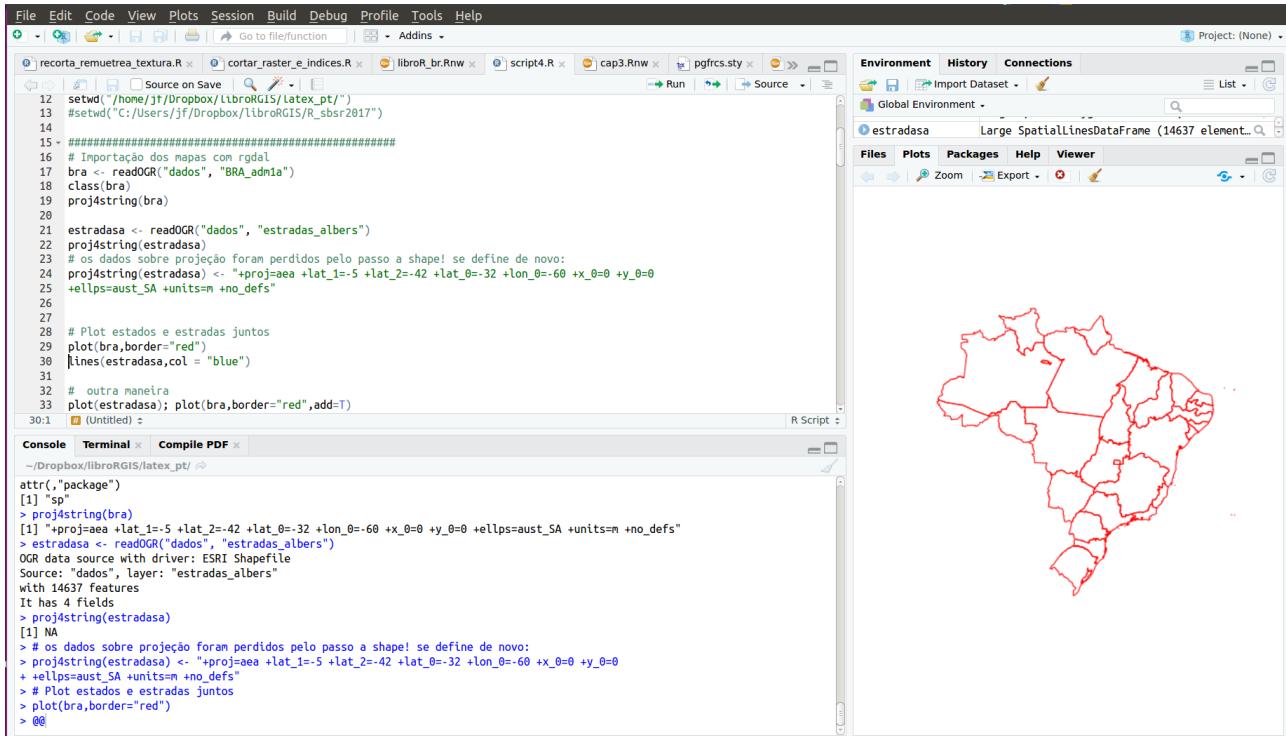


Figura 1. Interface de RStudio

1.3 Instalando pacotes no R

Ao instalar o R, apenas uma configuração mínima está disponível para operação básica do software. Para executar tarefas mais específicas, muitas vezes é necessário instalar pacotes adicionais. Há mais de 10.000 pacotes disponíveis no site R (www.r-project.org). Para instalar um pacote, existem várias opções. Na primeira, há uma conexão à Internet e R se conecta diretamente a um repositório. Desta forma, é possível instalar pacotes com o comando `install.packages("nomedopacote")`. Por exemplo, `install.packages("maptools")` instala o pacote chamado **maptools**.

A segunda opção é obter os arquivos de instalação (arquivos zip para Windows ou tar.gz para Linux) a partir da página do R e instalar os pacotes desses arquivos locais (sem conexão à Internet) com a função `install.packages()` com a seguinte Sintaxe:

Para Linux:

```
install.packages("maptools_0.9-2.tar.gz", repos = NULL, type="source")
```

Para Windows:

```
install.packages("maptools_0.9-2.zip", repos = NULL, type="source")
```

2. Operações básicas no R

Neste capítulo daremos os primeiros passos com R. Os leitores com experiência prévia do nível básico podem passar para o próximo capítulo. Aos iniciantes em R, convidamos a testar os códigos, lembrando que esta é a melhor forma de aprender R como linguagem de programação. Para realizar as análises apresentadas neste livro e executar o código R, você pode editar seu próprio script no RStudio, (Novo Arquivo R script) digitando o código (ou usando as funções copiar e colar no arquivo pdf do livro) ou abrir o script correspondente ao capítulo disponível na pasta de recursos (ver seção 0.2.3).

2.1 Operações básicas

R é uma linguagem orientada a objetos: um objeto pode ser compreendido como um *container* de informações. Essa informação é guardada como objeto, utilizando os símbolos <- (símbolo "maior que" seguido de "menos") ou simplesmente o símbolo =, seguido da informação que se deseja atribuir ao objeto. Os objetos são guardados na memória ativa do computador, sem utilizar arquivos temporários (embora existam funções que possibilitam a utilização de arquivos temporários ao invés da memória ativa). No exemplo abaixo atribuímos os valores 6 e 4, respectivamente, a dois objetos chamados primeirovalor e segundovalor. Em seguida, criamos um novo objeto, chamado soma, que recebe o resultado da soma dos dois primeiros. A função **print()** permite visualizar o conteúdo de um objeto ou operações entre objetos (o mesmo resultado é obtido simplesmente pelo nome do objeto).

```
primeirovalor <- 6
primeirovalor = 6 # O símbolo = é aceitado mas não recomendado
segundovalor = 4 # equivalente a segundovalor <- 4
soma <- primeirovalor + segundovalor
print(soma)

## [1] 10

soma

## [1] 10
```

R é uma linguagem interpretada e não compilada como C ou Fortran. Isso significa que os comandos são executados diretamente sem a necessidade de criar executáveis. Além disso, a sintaxe de R é muito simples e intuitiva. Por exemplo, a função **sum()** executa a soma de todos os argumentos (elementos dentro dos parênteses). Nas linhas do código abaixo, o resultado dessa operação resulta no objeto soma, que já criamos na operação anterior e, portanto, é substituído.

```
soma <- sum(primeirovalor,segundovalor,primeirovalor)
print(soma)

## [1] 16
```

Além de um nome e conteúdo, os objetos criados no R possuem atributos que especificam o tipo básico de dados representados pelo objeto. Os tipos de dados em R podem ser numéricos, caracteres, lógicos (TRUE / FALSE) e números complexos. Existem também tipos derivados, como fatores ou números inteiros que apontam

para níveis de categorias. Por padrão, todos os caracteres se tornam fatores quando os dados são importados para R. Existem muitos tipos de objetos; neste capítulo vamos ver alguns deles:

1. **vetor**: seqüência de valores numéricos ou de caracteres (letras, palavras).
2. **dataframe**: uma tabela formada por linhas (observações) e colunas (variáveis), que aceita colunas de diferentes tipos (por exemplo, colunas numéricas e outras com caracteres).
3. **matriz**: como o *dataframe* tem uma disposição bidimensional. É indexada por linhas e colunas, o que significa que uma célula é identificada pelo número de linhas e colunas. Todos os elementos devem ser do mesmo tipo (todos numéricos, por exemplo).
4. **lista**: usado para reunir objetos que podem ter diferentes estruturas.

Cada objeto tem pelo menos dois atributos: tipo (modo) e tamanho (comprimento). Veremos que esta informação é bastante importante durante a manipulação dos dados.

Abra RStudio em seu computador e comece um novo Script em “File” > “New File” > “New Rscript” (“Arquivo” > “Novo Arquivo” > “Novo Rscript”). Para fazer o seu script, você pode capturar os comandos manualmente no editor de script ou copiar e colar a partir deste documento. Além disso, você pode usar o script Cap2.R encontrado na pasta de scripts do livro.

Vamos criar um vetor, chamado Prec, que contém os valores mensais de precipitação observados na estação meteorológica de Irecê, Bahia, durante 1961-1990. No script, você pode escrever comentários, que começam com ”#” (Hashtag). Como já vimos, a operação de atribuição tem a sintaxe objeto (Prec) recebe (<-) valor(es). Usamos a função *c()* que permite combinar elementos. Da mesma forma, vamos criar mais dois vetores:

- Um vetor chamado meses, que contém o nome dos primeiros quatro meses do ano. Observe que esses nomes estão escritos entre aspas, caso contrário, R interpreta esses nomes como nomes de objetos e não como texto (*strings*),
- outro vetor chamado numeros, que receberá uma seqüência de números consecutivos entre um e quatro escritos 1:4.

```
# Vetores
# Dados de precipitação mensal em Irecê, Bahia
# www.cnpm.embrapa.br/projetos/bdclima/balanco/index/index_ba.html
Prec <- c(110, 91, 101, 53, 12, 6, 3, 3, 13, 39, 90, 132)

meses <- c("Janeiro", "Fevereiro", "Março", "Abril")
numeros <- 1:4
```

A função *ls()* nos dá uma lista dos objetos contidos no diretório de trabalho. Para conhecer os tipos de elementos contidos nesses objetos, você pode usar a função *class()*, colocando como argumento o nome do objeto, por exemplo, *class(Prec)*. É preciso ficar atento porque R faz a diferença entre maiúsculas e minúsculas (*case sensitive*). Portanto, prec não é o mesmo que Prec e, neste caso, o objeto prec não existe. Da mesma forma, podemos perguntar ao R se um objeto é um vetor com a função *is.vector()*, a resposta será VERDADEIRA ou FALSA (TRUE/FALSE). A função *length()* indicará o comprimento do objeto (número de elementos).

```
# Lista dos objetos
ls()

## [1] "hook_output"      "meses"           "numeros"          "Prec"
## [5] "primeirovalor"   "segundovalor"    "soma"

# Mostra o tipo de dados
class(Prec)

## [1] "numeric"
```

```
class(meses)

## [1] "character"

class(numeros)

## [1] "integer"

print(prec) # prec não existe, é Prec

## Error in print(prec): object 'prec' not found

# Pergunta é um vetor?
is.vector(Prec)

## [1] TRUE

is.vector(meses)

## [1] TRUE

is.vector(numeros)

## [1] TRUE

# Indica o comprimento do vetor
length(Prec)

## [1] 12

length(meses)

## [1] 4

length(numeros)

## [1] 4
```

Podemos exibir na tela ("imprimir") o conteúdo de um objeto com **print()** e executar estatísticas básicas com **max()**, **min()**, **mean()** e **sum()**.

```
# Mostra o conteúdo do objeto na tela
print(Prec)

## [1] 110 91 101 53 12 6 3 3 13 39 90 132

print(meses)

## [1] "Janeiro"   "Fevereiro"  "Março"      "Abril"

print(numeros)

## [1] 1 2 3 4

# Calcula estatísticas básicas (Máxima, mínima, média e soma)
max(Prec) # máximo
```

```
## [1] 132

min(Prec) # mínimo

## [1] 3

mean(Prec) # média

## [1] 54.41667

sum(Prec) # soma

## [1] 653
```

2.2 Importação de dados em R

Muitas vezes é trabalhoso capturar dados, sendo mais conveniente usar informações em planilhas ou em formato de texto armazenados em uma pasta. R usa o diretório de trabalho, ou seja, o caminho de uma pasta, para ler e gravar arquivos. Para definir este espaço, você pode usar a função `setwd()`. Para saber qual é o espaço de trabalho, a função `getwd()` é usada. Os caminhos do arquivo são definidos em relação ao caminho do espaço de trabalho. Por exemplo, na figura 2 se definimos C:\User\Documentos\livro como o diretório de trabalho, o caminho do arquivo irece.csv que está na pasta dados seria dados\irece.csv. No Windows é necessário utilizar o símbolo de barra invertida / ou barra dupla \\ como por exemplo C:/User/Documentos/livro ou C:\\User\\Documentos\\livro.

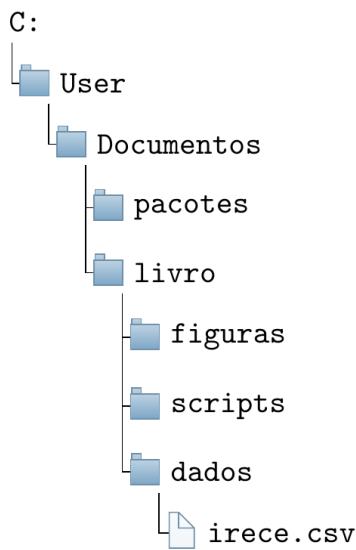


Figura 2. Caminho do arquivo irece.csv

No código abaixo, definimos e verificamos o caminho do espaço de trabalho com as funções `setwd()` e `getwd()`. Carregamos o arquivo de texto irece.csv graças a função `read.csv()`. O produto deste processo de importação é atribuído a um objeto em formato tabular (*dataframe*) que chamamos tab. A tabela mostra, para cada mês, os valores de precipitação (P), temperatura média (T), evapotranspiração potencial (ETP), evapotranspiração real ou efetiva (ETR) e período chuvoso e seco em Irecê, Bahia, no período de 1961-1990¹.

¹Dados climáticos do NMET obtidos a través da plataforma web do EMBRAPA (<https://www.cnpm.embrapa.br/projetos/bdclima/balanco/index>)

2.3 Operações com tabelas

O comando `class(tab)` nos permite ver se o objeto é do tipo *dataframe*. Através do comando `print(tab)` é possível visualizar a tabela inteira e com `head(tab)` visualizamos apenas as primeiras linhas da tabela, sendo este comando muito prático no manuseio de tabelas muito grandes.

Os *dataframes* possuem cabeçalhos nas colunas, que podem ser obtidos ou definidos com `names()`. Você também pode lidar com uma única coluna da tabela, invocando o nome do *dataframe*, seguido de `$` e o nome da coluna, por exemplo `tab$P`. No R esta coluna é um vetor simples como indicado pelo resultado da consulta `is.vector(tab$P)`.

```
# Lê uma tabela em formato .csv (separado por vírgula)
# Determina o caminho do espaço de trabalho
setwd("/home/jf/Dropbox/libroRGIS/latex_pt/dados")
# Mostra o caminho do espaço de trabalho
getwd()

## [1] "/home/jf/Dropbox/libroRGIS/latex_pt/dados"

# Carrega a tabela "irece.csv" no objeto tab
tab <- read.csv("irece.csv")

# Define o tipo do objeto tab
class(tab)

## [1] "data.frame"

# Mostra a tabela completa
print(tab)

##      Mes     T    P   ETP   ETR Periodo
## 1 Jan 23.8 110 107 107    chuva
## 2 Fev 24.0  91 101  93    chuva
## 3 Mar 23.8 101 107 102    chuva
## 4 Abr 23.3  53  95  59    seca
## 5 Mai 22.4  12  86  18    seca
## 6 Jun 21.3   6  71   9    seca
## 7 Jul 21.0   3  70   4    seca
## 8 Ago 21.6   3  77   4    seca
## 9 Set 23.2   13  93  13    seca
## 10 Out 24.4   39 113  39    seca
## 11 Nov 24.4   90 113  90    chuva
## 12 Dez 24.1 132 114 114    chuva

# Mostra as três primeiras linhas da tabela
head(tab,3)

##      Mes     T    P   ETP   ETR Periodo
## 1 Jan 23.8 110 107 107    chuva
## 2 Fev 24.0  91 101  93    chuva
## 3 Mar 23.8 101 107 102    chuva

# Mostra nomes (cabeçalhos) das colunas
names(tab)
```

```
## [1] "Mes"      "T"        "P"        "ETP"      "ETR"      "Periodo"

# Mostra apenas uma coluna (P)
print(tab$P)

## [1] 110  91 101  53  12   6   3   3  13  39  90 132

# Uma coluna de tabela é um vetor
is.vector(tab$P)

## [1] TRUE
```

É muito fácil executar operações matemáticas entre elementos de diferentes colunas, semelhante ao que realizamos em uma planilha de dados. No exemplo a seguir, calculamos o déficit de precipitação, subtraindo a evaporação potencial e a evapotranspiração real para cada mês. Para fazer isso, uma nova coluna é criada na tabela chamada def. Finalmente, a função `write.table()` permite salvar o *dataframe*.

```
# Calcula o deficit (ETP - ETR)
# Nova coluna na tabela: def
tab$def <- tab$ETP - tab$ETR

# Mostra as primeiras linhas da tabela
head(tab)

##   Mes     T     P   ETP   ETR Periodo def
## 1 Jan 23.8 110 107 107    chuva    0
## 2 Fev 24.0  91 101  93    chuva    8
## 3 Mar 23.8 101 107 102    chuva    5
## 4 Abr 23.3  53  95  59    seca   36
## 5 Mai 22.4  12  86  18    seca   68
## 6 Jun 21.3   6  71   9    seca   62
```

Vejamos agora os dados na coluna *Periodo* que indica se estamos dentro ou fora do período da chuva. Esta coluna tem apenas dois tipos de resposta "chuva" ou "seca". R reconheceu que eram dados qualitativos, com duas categorias e foram importados como um fator conforme indicado por `class(tab$Periodo)`. Internamente, cada categoria é codificada numericamente.

```
# Fator
tab$Periodo

## [1] chuva chuva chuva seca  seca  seca  seca  seca  seca  chuva
## [12] chuva
## Levels: chuva seca

class(tab$Periodo)

## [1] "factor"

as.numeric(tab$Periodo)

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

A qualquer momento é possível salvar o *dataframe* em um arquivo de texto usando, entre várias opções, a função `write.table()`.

```
# Salva a tabela num arquivo de texto
write.table(tab, file="tabla.txt")
```

Um outro aspecto importante no R é a possibilidade de converter um tipo de dado do objeto. Como exemplo podemos citar a conversão da tabela do tipo *dataframe* em matriz, através do comando ***as.matrix()***.

A vantagem das matrizes é que elas apresentam um sistema de indexação que permite um acesso eficiente e flexível aos elementos de um objeto. No caso das matrizes, o nome da matriz, seguido do número de linha e coluna, separados por uma vírgula, e entre colchetes, permite o acesso a uma célula específica da matriz. No nosso exemplo, **m[2, 4]** permite o acesso a célula localizada na segunda linha e quarta coluna. A seleção **m[1:3, 4]** cobre os elementos das linhas 1 a 3 na quarta coluna; **m[, 4]** a quarta coluna inteira; **m[1,]** a primeira linha.

```
# Converte de data.frame em matriz
m <- as.matrix(tab)
class(m)

## [1] "matrix"

print(m)

##      Mes     T     P    ETP    ETR Periodo def
## [1,] "Jan" "23.8" "110" "107" "107" "chuva" " 0"
## [2,] "Fev" "24.0"  "91"  "101" " 93" "chuva" " 8"
## [3,] "Mar" "23.8" "101" "107" "102" "chuva" " 5"
## [4,] "Abr" "23.3"  "53"  " 95" " 59" "seca"  "36"
## [5,] "Mai" "22.4"  "12"  " 86" " 18" "seca"  "68"
## [6,] "Jun" "21.3"  " 6"  " 71" " 9"  "seca"  "62"
## [7,] "Jul" "21.0"  " 3"  " 70" " 4"  "seca"  "66"
## [8,] "Ago" "21.6"  " 3"  " 77" " 4"  "seca"  "73"
## [9,] "Set" "23.2"  "13"  " 93" " 13" "seca"  "80"
## [10,] "Out" "24.4"  "39"  "113" " 39" "seca"  "74"
## [11,] "Nov" "24.4"  "90"  "113" " 90" "chuva" "23"
## [12,] "Dez" "24.1"  "132" "114" "114" "chuva" " 0"

print(m[2,4]) # indexação: 2a linha, 4a coluna

##    ETP
## "101"

print(m[1:3,4]) # linhas 1 a 3, 4a coluna

## [1] "107" "101" "107"

print(m[,7]) # 7a coluna

## [1] " 0" " 8" " 5" "36" "68" "62" "66" "73" "80" "74" "23" " 0"

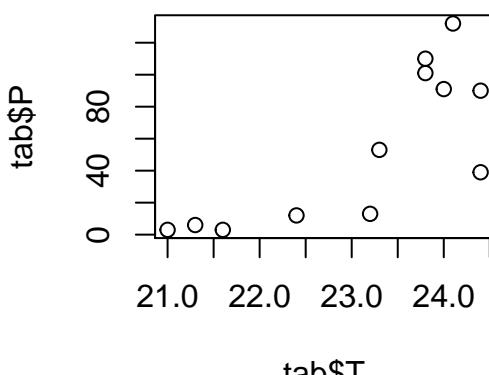
print(m[1,]) # 1a linha

##      Mes     T     P    ETP    ETR Periodo def
## "Jan" "23.8" "110" "107" "107" "chuva" " 0"
```

2.4 Desenho de gráficos

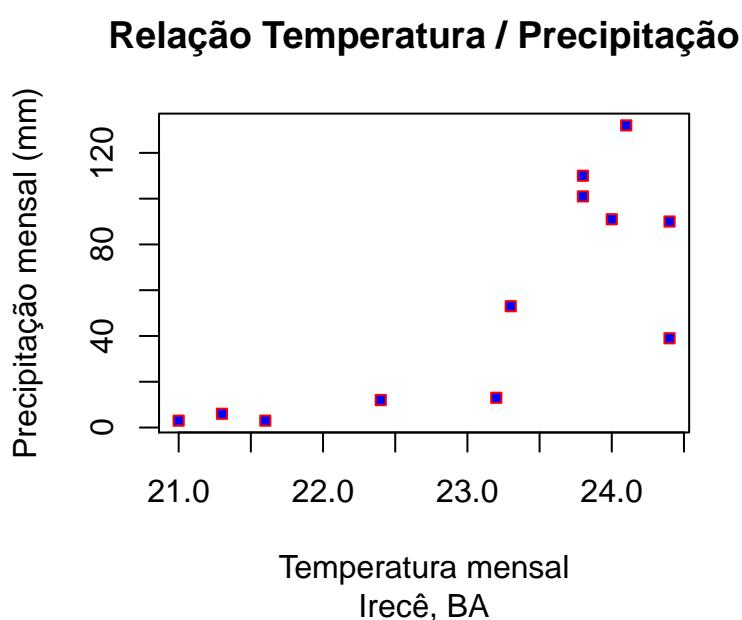
O R é famoso por permitir a elaboração de gráficos e análises estatísticas em diferentes níveis. Vamos criar dois diagramas de dispersão simples usando a função `plot()`. Os dois argumentos básicos são os nomes de dois vetores, o primeiro para o eixo horizontal (x) e o segundo para o eixo vertical (y), sendo que ambos os vetores devem ter o mesmo comprimento.

```
# Um diagrama de dispersão: núm de dias de chuva em X, Prec em Y
# Muito simples
plot(tab$T,tab$P)
```



Usando alguns argumentos adicionais, o gráfico pode ser facilmente aprimorado incluindo outras opções. Por exemplo, com `xlab = "texto do eixo x"` e `ylab = "texto do eixo y"` são criados os títulos de ambos os eixos. Os argumentos `pch` permite escolher o tipo de símbolo para os pontos, `col` e `bg` definem respectivamente a cor de linhas e recheio dos símbolos, `principal` e `sub` possibilitam a definição de um título principal e um subtítulo; e o `cex` permite controlar o tamanho da fonte (`cex = 0.8` reduz 20% o tamanho das letras em relação ao `cex = 1`, `cex = 1.5` aumenta em 50%).

```
# Um pouco mais elaborado
plot(tab$T,tab$P, xlab="Temperatura mensal", ylab="Precipitação mensal (mm)",
      pch=22, col="red", bg="blue",
      main="Relação Temperatura / Precipitação", sub = "Irecê, BA", cex=0.8)
```



2.5 Relação entre duas variáveis

O gráfico mostrado anteriormente sugere que há uma relação positiva entre a temperatura e a quantidade de precipitação. Vamos realizar algumas análises estatísticas básicas, o que nos permitirá compreender melhor as funções, suas opções e resultados. A função `cor()` permite calcular o coeficiente de correlação entre duas variáveis. No entanto, sabemos que existem várias maneiras para calcular esse coeficiente, mas nem sempre fica evidente qual delas é utilizada por padrão no R. Você pode obter ajuda em qualquer comando com `help(nome-da-função)`. Na ajuda, podemos ver que `cor()` usa o método de *Pearson* se o usuário não fornecer nenhuma especificação (Figura 3). Com o argumento `method = "nome_método"` você pode escolher entre três métodos diferentes (*Pearson*, *Kendall* e *Spearman*).

`cor {stats}` R Documentation

Correlation, Variance and Covariance (Matrices)

Description

`var`, `cov` and `cor` compute the variance of `x` and the covariance or correlation of `x` and `y` if these are vectors. If `x` and `y` are matrices then the covariances (or correlations) between the columns of `x` and the columns of `y` are computed.

`cov2cor` scales a covariance matrix into the corresponding correlation matrix *efficiently*.

Usage

```
var(x, y = NULL, na.rm = FALSE, use)
cov(x, y = NULL, use = "everything",
     method = c("pearson", "kendall", "spearman"))
cor(x, y = NULL, use = "everything",
     method = c("pearson", "kendall", "spearman"))
cov2cor(V)
```

Figura 3. Documento de ajuda da função `cor()`

```
# Correlação
cor(tab$T,tab$P)

## [1] 0.780685

# Para qualquer dúvida, pedir ajuda!
help(cor)
?cor()
cor(tab$T,tab$P, method = "pearson")

## [1] 0.780685

cor(tab$T,tab$P, method = "spearman")

## [1] 0.7557129
```

Você pode realizar uma regressão linear com a função `lm()` indicando a variável dependente, e depois do símbolo `~` as variáveis independentes ou explicativas (fórmula). Um resumo dos resultados da análise de

R pode ser obtido com `summary()`. O comando `summary(reg)` nos fornece informações sobre o ajuste do modelo linear chamado reg. Geralmente, os resultados são apresentados de tal forma que é possível extrair um valor específico através da indexação que vimos no caso das matrizes. Nesse caso, os coeficientes `resumo$coefficients` representam uma matriz que contém valores diferentes que resultam da regressão (coeficientes de regressão, erro, significado, etc.). É possível extrair um elemento específico nesta matriz. Por exemplo, `resumo$coefficients[1,3]` nos dá aqui o valor de t para a interseção no eixo vertical (intercepto).

```
# Uma regressão lineal entre a prec e o número de dias de chuva
reg <- lm(tab$P ~ tab$T)

# Os resultados do ajuste lineal
summary(reg)

##
## Call:
## lm(formula = tab$P ~ tab$T)
##
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -54.75 -10.65    1.45   15.97   47.39 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -649.245    178.352   -3.64  0.00453 **  
## tab$T        30.451      7.708    3.95  0.00273 **  
## ---        
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1 
##
## Residual standard error: 31.41 on 10 degrees of freedom
## Multiple R-squared:  0.6095, Adjusted R-squared:  0.5704 
## F-statistic: 15.61 on 1 and 10 DF,  p-value: 0.002728 

resumo <- summary(reg)

# Novas classes de objeto: lm (linear model) e summary.lm
class(reg)

## [1] "lm"

class(resumo)

## [1] "summary.lm"

# summary.lm contém a informação em uma matriz chamada coefficients
resumo$coefficients

##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -649.24489 178.351955 -3.640245 0.004534944
## tab$T        30.45055   7.708096  3.950464 0.002728236

# Resgata um elemento da matriz (valor do t de Student do intercepto)
resumo$coefficients[1,3]

## [1] -3.640245
```

Finalmente, é possível criar listas (list) para agrupar objetos de diferentes tipos. No exemplo que segue, o objeto chamado *lista* contém um vetor (Prec), um modelo de regressão linear (reg) e caracteres.

```
# Uma lista (list) é uma relação de objetos de diferentes tipos
lista <- list(Prec, reg, "lista rara")
lista

## [[1]]
## [1] 110 91 101 53 12 6 3 3 13 39 90 132
##
## [[2]]
##
## Call:
## lm(formula = tab$P ~ tab$T)
##
## Coefficients:
## (Intercept)      tab$T
## -649.24        30.45
##
## 
## [[3]]
## [1] "lista rara"

# Mostra o primeiro e o terceiro elemento da lista
lista[[1]]

## [1] 110 91 101 53 12 6 3 3 13 39 90 132

lista[[3]]

## [1] "lista rara"
```

2.6 Operações sobre margens: apply

Existem operações, chamadas operações sobre margens, que são executadas para todas as colunas ou todas as linhas de uma matriz. Por exemplo, *colSums()* permite a adição das células de cada coluna; *rowSums()* a soma das células em cada linha; *colMeans()* calcula a média das células em cada coluna.

```
# Cria uma matriz de 3 x 3
# byrow=T: os números do vetor entram por linha
m <- matrix(c(1,2,2,3,6,0,4,7,9),ncol=3,byrow=T)
print(m)

##      [,1] [,2] [,3]
## [1,]    1    2    2
## [2,]    3    6    0
## [3,]    4    7    9

colSums(m)

## [1] 8 15 11

rowSums(m)
```

```
## [1] 5 9 20
colMeans(m)
## [1] 2.666667 5.000000 3.666667
```

O R fornece a possibilidade de desenvolver funções sobre as margens de uma tabela, usando a função **apply()** (família de funções de repetição, baseado em vetorização). Este comando possui três argumentos: o objeto no qual o cálculo é feito; um número que indica se a operação é executada por linha (1) ou por coluna (2) e o operador. Por exemplo, `apply(m, 1, sum)` executa a soma das células em cada linha e `apply(m, 2, sd)` calcula o desvio padrão dos valores das células em cada coluna.

```
apply(m, 1, sum)
## [1] 5 9 20
apply(m, 2, sum)
## [1] 8 15 11
apply(m, 1, mean)
## [1] 1.666667 3.000000 6.666667
apply(m, 1, max)
## [1] 2 6 9
apply(m, 2, sd)
## [1] 1.527525 2.645751 4.725816
```

Existem vários comandos na família "apply", sendo que **lapply()** e **sapply()** possibilitam operações que agem sobre as margens das listas.

2.7 Operações por grupos

Em certos casos, é necessário realizar o cálculo de algum índice estatístico por grupos de observações. Na tabela de dados climáticos de Irecê é interessante calcular a diferença de precipitação entre os períodos de chuva e seca. Este tipo de cálculo pode ser feito com a função **aggregate()** seguindo diferentes sintaxes.

```
aggregate(P ~ Periodo, data = tab, FUN = "sum")
##   Periodo   P
## 1     chuva 524
## 2     seca 129

aggregate(x = tab$P, by = list(tab$Periodo), FUN = "sum")
##   Group.1   x
## 1     chuva 524
## 2     seca 129
```

```
aggregate(T ~ Periodo, data = tab, FUN = "mean")
##   Periodo      T
## 1    chuva 24.02000
## 2     seca 22.45714
```

2.8 Criação de funções

R possibilita a criação de comandos e funções próprios. Para criar uma função, o comando “function” é usado numa expressão que contém a definição dos argumentos da função, uma ou várias expressões válidas e o resultado. Uma vez que uma função foi definida, ela pode ser chamada tantas vezes quanto for necessário durante uma sessão de R.

```
# Define uma função para somar a um valor numérico
# com o dobro de um segundo valor
Func <- function (a, b) {
  resultado <- a + 2 * b
  resultado
}
# Execução da função
Func(3,7) # 3 + 2 * 7 = 3 + 14 = 17
## [1] 17
```

2.9 Repetições e condições

Como qualquer linguagem de programação o R permite operações iterativas, como *loops*. A função mais simples para lidar com *loops* é *for()*. A sintaxe é a seguinte:

```
for(i in listavalores) {script para executar para cada valor de i}
```

No exemplo número 1, a variável i receberá sucessivamente os valores 3, 4, 5 ... até 7 e *print(i)* imprime cada um deles.

```
# Loops (iterações)
# Exemplo número 1
for (i in 3:7){print(i)}

## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
```

No segundo exemplo, o valor do objeto fac é atualizado consecutivamente, multiplicando seu valor na iteração anterior pelo valor de i. Portanto, o resultado final é $1 \times 2 \times 3 \times 4 \times 5 = 120$.

```
# Exemplo número 2
fac <- 1
  for (i in 1:5){
    fac <- fac * i
  }
print(fac)
```

```
## [1] 120
```

No exemplo 3, o cálculo é feito em um vetor e i serve tanto para gerar um valor, quanto para escolher a posição do vetor em que o valor resultante será colocado. Por exemplo, na primeira iteração ($i = 1$), a instrução `vetor[i] <- i*2` colocará o valor 2 no primeiro elemento do vetor.

```
# Exemplo número 3
vetor <- c(0,0,0,0)
for (i in 1:4){
  vetor[i] <- i*2
}
print(vetor)

## [1] 2 4 6 8
```

No quarto exemplo, uma condição foi aninhada dentro do loop. A sintaxe das condições é semelhante àquela das iterações:

```
if(condição){script quando a condição for preenchida} else
{sequência script quando a condição não for cumprida}
```

i equivale a valores de 1 a 5, uma vez que o comprimento do vetor é 5. Portanto, para cada iteração, um elemento do vetor é comparado com o valor cinco: se o valor for maior que cinco (`if (vetor[i] > 5)`), o valor de soma é atualizado ao receber a adição do seu valor na iteração anterior e do elemento vetorial na posição i (`soma <- soma + vetor[i]`). Caso contrário, o valor é subtraído por um (`soma <- soma - 1`).

```
# Exemplo número 4
vetor <- c(3,3,2,8,3); soma <- 0
for (i in 1:length(vetor)){
  if (vetor[i] > 5) {soma <- soma + vetor[i]} else {soma <- soma -1}
  print(soma)
}

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

No último exemplo, a função `paste()` é usada para concatenar texto e variáveis numéricas. Isso permite criar nomes "dinâmicos" combinando as palavras "ImagenSPOT-banda" e ".tif" com i .

```
# Exemplo número 5
i <- 1
nome <- paste("ImagenSPOT-banda", i, ".tif", sep="")
print(nome)

## [1] "ImagenSPOT-banda1.tif"

for (i in 1:3){
  nome <- paste("ImagenSPOT-banda", i, ".tif", sep="")
  print(nome)
}
```

```
## [1] "ImagenSPOT-banda1.tif"
## [1] "ImagenSPOT-banda2.tif"
## [1] "ImagenSPOT-banda3.tif"
```

Ao trabalhar com R, todos os dados geralmente estão na memória ativa do computador. Apesar da capacidade dos computadores atuais, isso pode ser um problema ao manipular bancos de dados muito grandes. A função `rm()` possibilita a exclusão de objetos para liberar memória. Por exemplo, `rm(Prec)` excluirá o objeto chamado Prec. O comando `rm(list = ls())`, por outro lado, apagará todos os objetos no espaço de trabalho.

```
ls()

## [1] "fac"           "Func"          "hook_output"   "i"
## [5] "lista"         "m"              "meses"        "nome"
## [9] "numeros"       "Prec"          "primeirovalor" "reg"
## [13] "resumo"        "segundovalor"  "soma"         "tab"
## [17] "vetor"

rm(Prec)
ls()

## [1] "fac"           "Func"          "hook_output"   "i"
## [5] "lista"         "m"              "meses"        "nome"
## [9] "numeros"       "primeirovalor" "reg"          "resumo"
## [13] "segundovalor"  "soma"         "tab"          "vetor"
```

Ao deixar uma sessão do R, a mensagem "Salvar a imagem do espaço de trabalho?" viabilizará o salvamento dos dados e sua utilização numa sessão futura.

3. Organização de objetos espaciais em R

Em R existem várias classes de objetos para representar informações espaciais. Neste capítulo, apresentaremos algumas opções para manuseio de arquivos vetoriais e *raster*, utilizando os pacotes **sf** e **raster**, respectivamente. Embora o pacote **sf** não seja o mais difundido, o escolhemos tendo em vista sua simplicidade e a descontinuidade gradual do pacote **sp**. Para dados no formato *raster*, apresentaremos as funções do pacote do mesmo nome.

3.1 Dados vetoriais: modelo "simple feature"

O modelo geométrico de recursos simples (*simple feature*) é um padrão de código aberto desenvolvido e suportado pelo *Open Geospatial Consortium* (OGC) para representar uma ampla gama de informações geográficas. É um modelo de dados hierárquico, que simplifica dados geográficos condensando uma ampla gama de formas geográficas em uma única classe de geometria. O *simple feature* é um modelo de dados amplamente suportado. Esse modelo está subjacente a estrutura de dados de muitos aplicativos SIG, incluindo QGIS e PostGIS, permitindo a transferência de dados entre configurações.

O pacote **sf** (Pebesma, 2017) é totalmente compatível com os recursos *simple feature* usados majoritariamente nas operações de análise espacial: pontos, linhas, polígonos e suas respectivas versões "múltiplas". O **sf** também permite criar coleções de geometrias, que podem conter diferentes tipos em um único objeto. Este pacote também integra a funcionalidade dos três pacotes principais: **sp** (Pebesma & Bivand, 2018) para o sistema de classe de objetos espaciais, **rgdal** (Bivand *et al.*, 2017a) para ler e escrever dados e **rgeos** (Bivand & Rundel, 2017) para operações espaciais. Lovelace *et al.* (2018) mencionam alguns motivos para usar o pacote **sf** em vez de **sp**, mesmo que o último tenha sido amplamente testado:

- O **sf** fornece uma interface quase direta para as funções GDAL e GEOS C ++.
- A leitura e a escrita de dados são rápidas.
- O desempenho da implantação foi melhorado.
- Os nomes das funções **sf** são relativamente consistentes e intuitivas (todos começam com `st_`).
- Os objetos **sf** podem ser processados da mesma forma que *dataframe* de dados na maioria das operações.
- As funções **sf** podem ser combinadas usando o operador `%>%` e funcionam bem com a coleção de pacotes **tidyverse** do R.

Em R, os objetos *simple feature* são armazenados em uma tabela de *dataframe*, na qual os dados geográficos ocupam uma coluna especial, que contém uma lista. Esta coluna geralmente é chamada de "geom" ou "geometry". O pacote precisa ser instalado (consulte a seção 1.3 para a instalação de pacotes). No Linux (Ubuntu), a instalação é feita seguindo as etapas abaixo:

```
sudo add-apt-repository ppa:ubuntugis/ubuntugis-unstable
sudo apt-get update
sudo apt-get install libgdal-dev libgeos-dev libproj-dev libudunits2-dev
sudo apt-get install liblwgeom-dev
```

No Windows, recomenda-se instalar previamente o Rtools seguindo as instruções contidas na seção 1.3.

No **sf**, existem diferentes tipos de objetos espaciais, dependendo do tipo de informação. As classes Point e Multipoint, Linestring e Multilinestring e as classes Polygon e Multipolygon permitem gerenciar pontos, linhas e polígonos, respectivamente. A classe Geometrycollection permite que geometrias diferentes sejam combinadas no mesmo objeto.

Nas seções a seguir, construiremos objetos espaciais muito simples dessas diferentes classes, para entender melhor a maneira como estão estruturados. Na prática, trabalharemos usualmente com objetos espaciais obtidos através da importação de bancos de dados. No entanto, a construção de objetos, a partir da base, nos permitirá vislumbrar a hierarquia dos elementos que compõem cada objeto. Nós vamos utilizar um sistema de coordenadas arbitrárias com valores, tanto em x como em y, entre zero e dez, para lidar com dados simples. Uma descrição detalhada desses objetos é encontrada na página do pacote **sf** (<https://cran.rstudio.com/web/packages/sf/vignettes/sf1.html>). Iniciaremos com a ativação da biblioteca com **library(sf)**.

3.1.1 Camada de Pontos

O pacote **sf** permite manipular dados em duas, três ou quatro dimensões. Normalmente, a terceira dimensão é a altura (*z*) e a quarta (*m*) alguma medida (temperatura, por exemplo). As coordenadas são apresentadas em forma de vetor ou tabela e são transformadas em uma cobertura pontual da classe **sfg**: simple feature geometry com a função **st_point()** (um único ponto) ou **st_multipoint()** (vários pontos).

```
# Instalação de sf
# Não rodar se já estiver instalado, para roder tirar o # de comentário
# install.packages("sf", dependencies = TRUE)

library(sf)

## Linking to GEOS 3.5.1, GDAL 2.2.2, PROJ 4.9.2

# Geometrias Simple Features (sfg: simple feature geometry)
# Point: coordenadas de um ponto em 2, 3 o 4 dimensões
P <- st_point(c(2,6)) # 2 dimensões (XY)
class(P)

## [1] "XY"     "POINT"   "sfg"

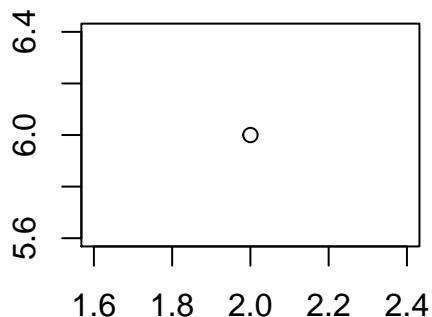
P <- st_point(c(2,6,18,41),"XYZM") # 4 dimensões (XYZM)
class(P)

## [1] "XYZM"   "POINT"   "sfg"

str(P)

## Classes 'XYZM', 'POINT', 'sfg' num [1:4] 2 6 18 41

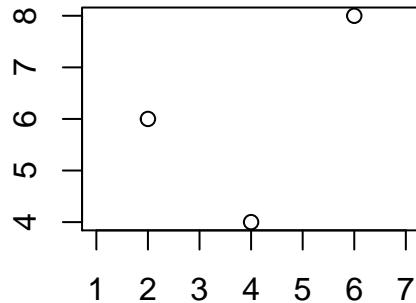
plot(P, axes = TRUE)
```



```
# Multipoint: coordenadas de vários pontos em 2, 3 ou 4 dimensões
# Cria um vetor de coordenadas em x
Xs <- c(2,4,6)
# Cria um vetor de coordenadas em y
Ys <- c(6,4,8)
# Cola Xs e Ys para criar uma tabela de coordenadas
coords <- cbind(Xs,Ys)
print(coords)

##          Xs   Ys
## [1,]    2    6
## [2,]    4    4
## [3,]    6    8

# Cria um objeto Multipoint (MP)
MP <- st_multipoint(coords)
plot(MP, axes = TRUE)
```



```
class(MP)

## [1] "XY"           "MULTIPOINT" "sfg"

print(MP)

## MULTIPPOINT (2 6, 4 4, 6 8)

# Multipoint em 3 dimensões
xyz <- cbind(coords,c(17, 22, 32))
print(xyz)

##          Xs   Ys   Zs
## [1,]    2    6   17
## [2,]    4    4   22
## [3,]    6    8   32

MP <- st_multipoint(xyz)
print(MP)

## MULTIPPOINT Z (2 6 17, 4 4 22, 6 8 32)
```

Com a função `st_sfc()`, é possível juntar objetos em coleções (sfc). Por exemplo, `geometrial` é um objeto da classe `simple feature collection` (sfc) que une os pontos P1, P2 e P3. Nessa etapa, é possível determinar o sistema de coordenadas de dados com a opção `crs`. O sistema de coordenadas pode ser descrito pelo código numérico EPSG ou pelo sistema Proj4string.

```
# Coleções de Simple Features (sfc: simple feature collection)
# Cria vários sfg
P1 <- st_point(c(2,6)); P2 <- st_point(c(4,4)); P3 <- st_point(c(6,8))

# Junta vários sfg em um sfc
geometria1 = st_sf(P1,P2,P3) # st_sfc(P1,P2, crs = 4326)
```

Os objetos sfc (geometrias) que acabamos de criar têm apenas a informação das coordenadas e possivelmente os atributos *z* e *m*. Vamos agora criar uma tabela de atributos com informações sobre cada um dos pontos. A associação desta tabela com o objeto anterior nos permite criar um objeto mais "sofisticado" da classe simple feature.

Comandos diferentes nos possibilitam conhecer as características desse objeto. Em particular, *class()* revela a natureza híbrida do objeto: *dataframe* e *sf*. A função *print()* expõe a tabela de atributos com a coluna em que a informação espacial de cada ponto é armazenada.

```
# Associando uma geometria sfc com uma tabela de atributos (dataframe)
# Tabela com ID (campo num) e informação adicional (tabela de atributos)
# Pontos de localização Poços (Poco) e postos de gasolina (Posto)
num <- c(1,2,3)
nome <- c("Poco","Posto","Poco")
tabpontos <- data.frame(cbind(num,nome))
class(tabpontos)

## [1] "data.frame"

print(tabpontos)

##   num nome
## 1   1  Poco
## 2   2 Posto
## 3   3  Poco

# sf object
SFP = st_sf(tabpontos, geometry = geometria1)
class(SFP) # classe dupla: simple feature e dataframe

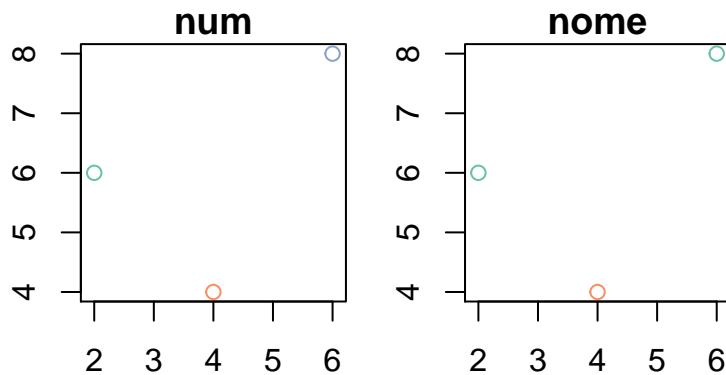
## [1] "sf"          "data.frame"

print(SFP) # olhar a coluna-lista "geometry"

## Simple feature collection with 3 features and 2 fields
## geometry type: POINT
## dimension:      XY
## bbox:           xmin: 2 ymin: 4 xmax: 6 ymax: 8
## epsg (SRID):    NA
## proj4string:    NA
##   num nome   geometry
## 1   1  Poco POINT (2 6)
## 2   2 Posto POINT (4 4)
## 3   3  Poco POINT (6 8)
```

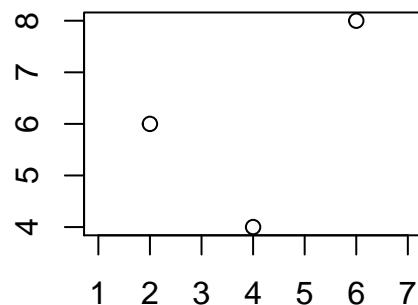
A função `plot()` exibe o mapa na tela: um mapa é criado para cada atributo da tabela. No caso da exibição restrita da localização dos pontos, a função `st_geometry()` é usada.

```
plot(SFP, axes=TRUE)
```



```
# Só geometria
```

```
plot(st_geometry(SFP), axes=TRUE)
```



Eventualmente é possível obter a tabela de atributos, perdendo a informação espacial, com a função `as.data.frame()`. É muito simples extrair determinados elementos da camada usando a tabela de atributos. Por exemplo, `Pocos <- SFP[nome=="Poco",]` cria um novo objeto sf com os pontos cujo nome é "Poco" na tabela de atributos.

```
# Extrai a tabela de atributos de um objeto sfc com:
as.data.frame(SFP)
```

```
##   num  nome geometry
## 1   1  Poco      2, 6
## 2   2 Posto      4, 4
## 3   3  Poco      6, 8
```

```
# Seleção de elementos dentro do mapa (poços)
Pocos <- SFP[nome=="Poco",]
```

3.1.2 Camada de linhas

No formato vetorial, uma linha simples é definida pelas coordenadas dos vértices. Para manipular esse tipo de dados, o sf utiliza segmentos da classe `LineString` que descrevem segmentos simples (que não possuem interseções ou bifurcações). Os objetos `MultilineString` unem segmentos `LineString`. No código abaixo, criamos três objetos `LineString` (`L1`, `L2` e `L3`) com base em uma tabela de coordenadas, usando a função `st_linestring()`. Da mesma forma que para pontos, pode-se juntar objetos em coleções (`sfC`) com a função `st_sf()`.

```

# Cria 3 objetos "Linestring": simple cadeia de coordenadas (vértices)
# Linha L1
X1s <- c(0,3,5,8,10)
Y1s <- c(0,3,4,8,10)
Coord1 <- cbind(X1s,Y1s)
# Cria objeto de Classe Linestring
L1 <- st_linestring(Coord1)
class(L1)

## [1] "XY"           "LINESTRING" "sfg"

print(L1)

## LINESTRING (0 0, 3 3, 5 4, 8 8, 10 10)

# Linha L2
X2s <- c(2,1,1)
Y2s <- c(2,4,5)
Coord2 <- cbind(X2s,Y2s)
# Cria objeto de Classe Linestring
L2 <- st_linestring(Coord2)

# Linha 3
X3s <- c(8,10)
Y3s <- c(8,6)
Coord3 <- cbind(X3s,Y3s)
# Cria objeto de Classe Linestring
L3 <- st_linestring(Coord3)

# Cria um objeto Multilines: conjunto de objetos Linestring
L1L2 <- st_multilinestring(list(L1,L2))

# Junta vários sfg num sfc (coleção de simple features)
geometria2 <- st_sfc(L1,L2,L3)

```

Para criar um objeto sf, uma tabela de atributos está associada às linhas de uma maneira semelhante a que vimos anteriormente para a camada de pontos.

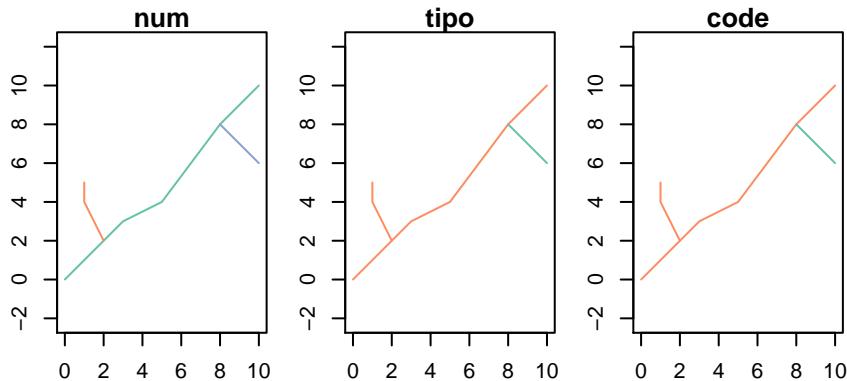
```

# Tabela de atributos
num <- c(1,2,3)
code <- c("t","t","p")
tipo <- c("Terra","Terra","Pavimentada")
tabLinhas <- data.frame(cbind(num,tipo,code))
print(tabLinhas)

##   num      tipo code
## 1   1      Terra    t
## 2   2      Terra    t
## 3   3 Pavimentada    p

# sf object
SFL <- st_sf(tabLinhas, geometry = geometria2)
plot(SFL,axes=TRUE)

```



```
class(SFL) # dobre classe: simple feature e dataframe
## [1] "sf"          "data.frame"

print(SFL) # ver coluna lista "geometry"

## Simple feature collection with 3 features and 3 fields
## geometry type: LINESTRING
## dimension: XY
## bbox: xmin: 0 ymin: 0 xmax: 10 ymax: 10
## epsg (SRID): NA
## proj4string: NA
##   num      tipo code               geometry
## 1  1      Terra  t LINESTRING (0 0, 3 3, 5 4, ...
## 2  2      Terra  t LINESTRING (2 2, 1 4, 1 5)
## 3  3 Pavimentada  p LINESTRING (8 8, 10 6)
```

Quanto aos pontos, a tabela de atributos pode ser usada para selecionar linhas com certas características. No exemplo abaixo, as estradas pavimentadas e de terra são selecionadas e plotadas empregando cores diferentes.

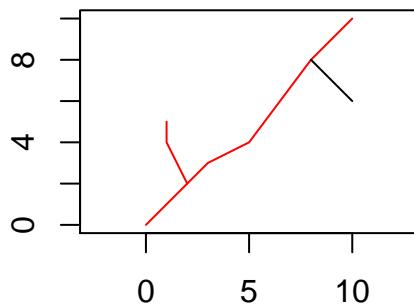
```
# Extrair a tabela de atributos de um sfc com:
as.data.frame(SFL)

##   num      tipo code               geometry
## 1  1      Terra  t LINESTRING (0 0, 3 3, 5 4, ...
## 2  2      Terra  t LINESTRING (2 2, 1 4, 1 5)
## 3  3 Pavimentada  p LINESTRING (8 8, 10 6)

# Seleciona certos elementos usando a tabela de atributos
print(SFL[tipo=="Pavimentada",])

## Simple feature collection with 1 feature and 3 fields
## geometry type: LINESTRING
## dimension: XY
## bbox: xmin: 8 ymin: 6 xmax: 10 ymax: 8
## epsg (SRID): NA
## proj4string: NA
##   num      tipo code               geometry
## 3  3 Pavimentada  p LINESTRING (8 8, 10 6)

plot(st_geometry(SFL[tipo=="Terra",]), col="red", axes=TRUE)
plot(st_geometry(SFL[tipo=="Pavimentada",]), add=TRUE)
```



3.1.3 Camada de polígonos

Um mapa está constituído por vários polígonos que representam diferentes categorias. Para criarmos um polígono, obtemos objetos `Polygon` a partir das coordenadas dos vértices, que constituem uma forma fechada usando a função `st_polygon()`. Para criar um polígono com espaços internos vazios, se utiliza a seqüência dos vértices: o contorno externo está definido por vértices no sentido anti-horário e as "lacunas" dentro desse polígono são definidas por vértices no sentido dos ponteiros do relógio. Finalmente, vários objetos são agrupados em uma coleção usando a função `st_sfc()`.

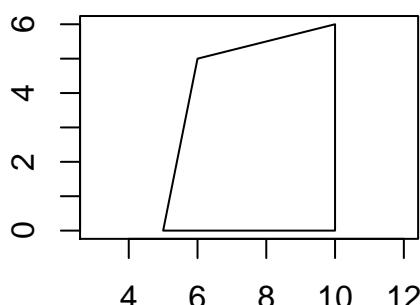
```
## P1 Polígono florestal no SudEste
## Polygon
# Cria uma cadeia de coordenadas em X
X1 <- c(5,10,10,6,5)
# Cria uma cadeia de coordenadas em Y
# Cuidado! tem que fechar (último par de coord = primeiro)
Y1 <- c(0,0,6,5,0)
# Cola X e Y para criar uma tabela de coordenadas
c1 <- cbind(X1,Y1)
print(c1)

##      X1   Y1
## [1,]  5   0
## [2,] 10   0
## [3,] 10   6
## [4,]  6   5
## [5,]  5   0

class(c1)

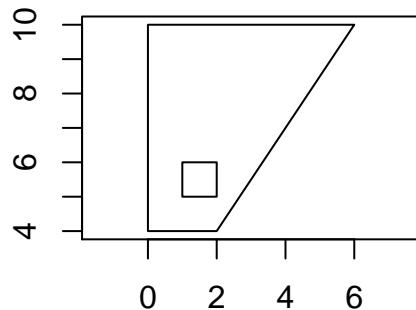
## [1] "matrix"

# Cria um objeto Polygon. Um polygon é uma forma simple fechada
# eventualmente com buraco(s)
P1 = st_polygon(list(c1))
plot(P1, axes=T)
```



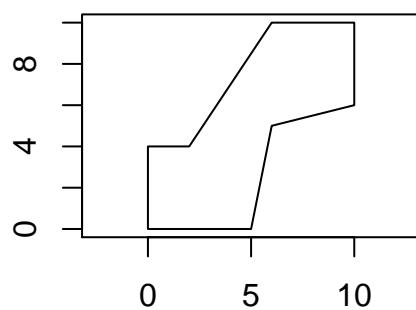
```
# P2 Polígono florestal ao Nordeste
# Cria cadeias de coordenadas em X e Y
X2 <- c(0,2,6,0,0)
Y2 <- c(4,4,10,10,4)
# Cola X e Y para criar uma tabela de coordenadas
c2 <- cbind(X2,Y2)

# Polígono vazio %%%%%% ordem coordenadas !!!!!!
# Cria cadeias de coordenadas em X e Y
X3 <- c(1,1,2,2,1)
Y3 <- c(5,6,6,5,5)
# Cola X e Y para criar uma tabela de coordenadas
c3 <- cbind(X3,Y3)
P2 = st_polygon(list(c2,c3))
plot(P2,axes=T)
```



```
# P4 Polígono de agricultura
c3i <- c3[nrow(c3):1, ] # inverte a ordem das coordenadas
P4 = st_polygon(list(c3i)) # essa vez não é vazio

# P5 Polígono de área urbana
X5 <- c(0,5,6,10,10,6,2,0,0)
Y5 <- c(0,0,5,6,10,10,4,4,0)
c5 <- cbind(X5,Y5)
P5 = st_polygon(list(c5))
plot(P5,axes=T)
```

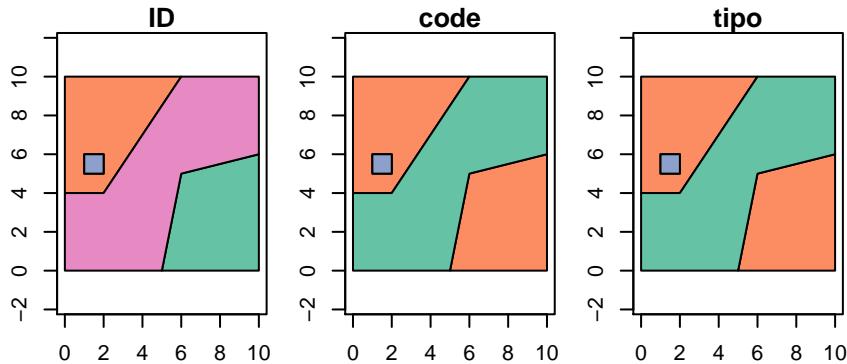


```
# Junta vários sfg num sfc (coleção de simple features)
geometria3 = st_sf(P1,P2,P4,P5)
```

Essa camada de polígonos está associada a uma tabela de atributos, para criar um objeto da classe sf de forma semelhante àquela utilizada para camadas de pontos e linhas.

```
# Tabela de atributos
ID <- c(1,2,3,4); code <- c("F","F","U","A")
tipo <- c("Floresta","Floresta","Urbano","Agricultura")
tabpol <- data.frame(cbind(ID,code,tipo))

# sf object
SFPol = st_sf(tabpol, geometry = geometria3)
plot(SFPol,axes=TRUE)
```



```
class(SFPol) # Tem duas classes: simple feature e dataframe

## [1] "sf"           "data.frame"

print(SFPol) # Olha coluna lista "geometry"

## Simple feature collection with 4 features and 3 fields
## geometry type: POLYGON
## dimension: XY
## bbox: xmin: 0 ymin: 0 xmax: 10 ymax: 10
## epsg (SRID): NA
## proj4string: NA
##   ID code      tipo      geometry
## 1  1   F    Floresta POLYGON ((5 0, 10 0, 10 6, ...
## 2  2   F    Floresta POLYGON ((0 4, 2 4, 6 10, 0...
## 3  3   U     Urbano POLYGON ((1 5, 2 5, 2 6, 1 ...
## 4  4   A Agricultura POLYGON ((0 0, 5 0, 6 5, 10...

summary(SFPol)

##   ID   code      tipo      geometry
## 1:1  A:1  Agricultura:1  POLYGON:4
## 2:1  F:2    Floresta :2    epsg:NA:0
## 3:1  U:1    Urbano  :1
## 4:1
```

A tabela de atributos possibilita a extração de determinados elementos. As últimas linhas do código abaixo mostram que uma coleção pode eventualmente reunir diferentes tipos de geometria, como pontos, linhas e polígonos.

```
# Extrai a tabela de atributos de um sfc com:
as.data.frame(SFPol)
```

```

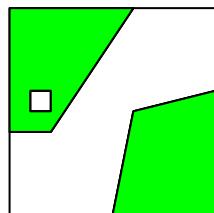
##   ID code      tipo             geometry
## 1  1     F Floresta POLYGON ((5 0, 10 0, 10 6, ...
## 2  2     F Floresta POLYGON ((0 4, 2 4, 6 10, 0...
## 3  3     U Urbano POLYGON ((1 5, 2 5, 2 6, 1 ...
## 4  4     A Agricultura POLYGON ((0 0, 5 0, 6 5, 10...

# Seleciona certos elementos usando a tabela de atributos
print(SFPol[tipo=="Floresta",])

## Simple feature collection with 2 features and 3 fields
## geometry type: POLYGON
## dimension: XY
## bbox: xmin: 0 ymin: 0 xmax: 10 ymax: 10
## epsg (SRID): NA
## proj4string: NA
##   ID code      tipo             geometry
## 1  1     F Floresta POLYGON ((5 0, 10 0, 10 6, ...
## 2  2     F Floresta POLYGON ((0 4, 2 4, 6 10, 0...

Floresta <- SFPol[tipo=="Floresta",]
plot(st_geometry(SFPol), axes=FALSE)
plot(Floresta, add=TRUE, col = "green")

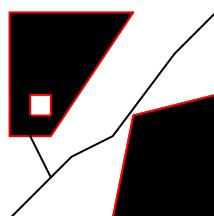
```



```

# Uma coleção pode eventualmente reunir vários tipos de geometria
Detodo <- st_sfc(P1,P2,L1,L2,P1)
plot(Detodo,border="red",axes=FALSE)

```



```

# Salva os objetos em formato shape
setwd("/home/jf/dados")
st_write(SFP,"SFP.shp",delete_layer = TRUE)

## Deleting layer `SFP' using driver
`ESRI Shapefile'
## Writing layer `SFP' to data source `SFP.shp' using driver
`ESRI Shapefile'
## features:      3
## fields:        2
## geometry type: Point

```

```

st_write(SFL,"SFL.shp",delete_layer = TRUE)

## Deleting layer `SFL' using driver
`ESRI Shapefile'
## Writing layer `SFL' to data source `SFL.shp' using driver
`ESRI Shapefile'
## features:      3
## fields:        3
## geometry type: Line String

st_write(SFPol,"SFPOL.shp",delete_layer = TRUE)

## Deleting layer `SFPOL' using driver
`ESRI Shapefile'
## Writing layer `SFPOL' to data source `SFPOL.shp' using driver
`ESRI Shapefile'
## features:      4
## fields:        3
## geometry type: Polygon

```

3.2 Dados raster: RasterLayer no pacote raster

O pacote **raster** viabiliza a manipulação dos dados *raster* com objetos da classe *RasterLayer*. O mapa *raster* pode ser obtido a partir de uma matriz que contém os valores de cada célula. A extensão espacial é definida pela função *extent()*, inserindo as coordenadas extremas do mapa *raster* (x mínimo, x máximo, y mínimo e y máximo).

```

library(raster)

## Loading required package: sp

# Cria uma matriz
m <- matrix(c(1,2,3,4,2,NA,2,2,3,3,3,1),ncol=4,nrow=3,byrow=TRUE)
m

##      [,1] [,2] [,3] [,4]
## [1,]     1     2     3     4
## [2,]     2    NA     2     2
## [3,]     3     3     3     1

r <- raster(m)
extent(r) <- extent(c(0,4,0,3))
class(r)

## [1] "RasterLayer"
## attr(,"package")
## [1] "raster"

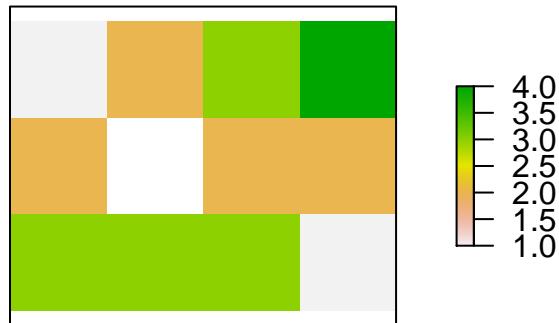
print(r)

## class       : RasterLayer
## dimensions : 3, 4, 12  (nrow, ncol, ncell)

```

```
## resolution : 1, 1 (x, y)
## extent      : 0, 4, 0, 3 (xmin, xmax, ymin, ymax)
## coord. ref. : NA
## data source : in memory
## names       : layer
## values      : 1, 4 (min, max)

plot(r, axes=F)
```



4. Importação / exportação de dados espaciais

4.1 Importação de arquivos *shape*

O *shapefile* é um formato de arquivo para dados vetoriais amplamente utilizado, desenvolvido pela empresa ESRI. É composto por pelo menos três arquivos que possuem as seguintes extensões:

- .shp: armazena as entidades geométricas dos objetos.
- .shx: armazena o índice de entidades geométricas.
- .dbf: armazena a tabela de atributos dos objetos.

É comum encontrar um quarto arquivo com extensão .prj que contém as informações referidas no sistema de coordenadas.

Existem vários pacotes para importar arquivos *shape*. Apresentamos aqui o procedimento de importação com o pacote **sf**. Como mencionado no capítulo anterior, esse pacote foi escolhido porque a estrutura de objetos espaciais é simples e está planejado como uma substituição para o pacote **sp**. O pacote **sf** permite gerenciar e representar a informação espacial de variadas formas, mas nos limitaremos àquelas mais comuns apresentadas no capítulo anterior: geometrias de pontos, linhas e polígonos associados (ou não) a uma tabela de atributos (ver script cap4.R).

4.1.1 Importação com **sf**

Para trabalhar com o pacote **sf**, devemos carregar a biblioteca com `library(sf)`. Definimos o espaço de trabalho como a pasta na qual os arquivos *shape* estão localizados de modo que essa pasta seja utilizada *por default*. A função `st_read()` é para importar mapas de qualquer geometria; `class()` é adequada para averiguar se o objeto espacial criado é da classe **sf** e `summary()` apropriada para descrever o objeto.

A função `plot()` exibe um mapa para cada atributo do objeto. Para mapear apenas a geometria dos objetos (neste caso, o limite dos polígonos), usa-se a função `st_geometry()`.

```
# Determina o caminho do espaço de trabalho
setwd("/home/jf/dados")
library(sf)

# Importação dos mapas com st_read (pacote sf)
# Mapa dos estados
br <- st_read("brasil.shp")

## Reading layer `brasil` from data source `/home/jf/dados/brasil.shp` using driver
`ESRI Shapefile'
## Simple feature collection with 27 features and 3 fields
## geometry type: MULTIPOLYGON
## dimension: XY
## bbox: xmin: -73.99045 ymin: -33.75208 xmax: -28.83591 ymax: 5.271841
## epsg (SRID): NA
## proj4string: +proj=longlat +ellps=GRS80 +no_defs

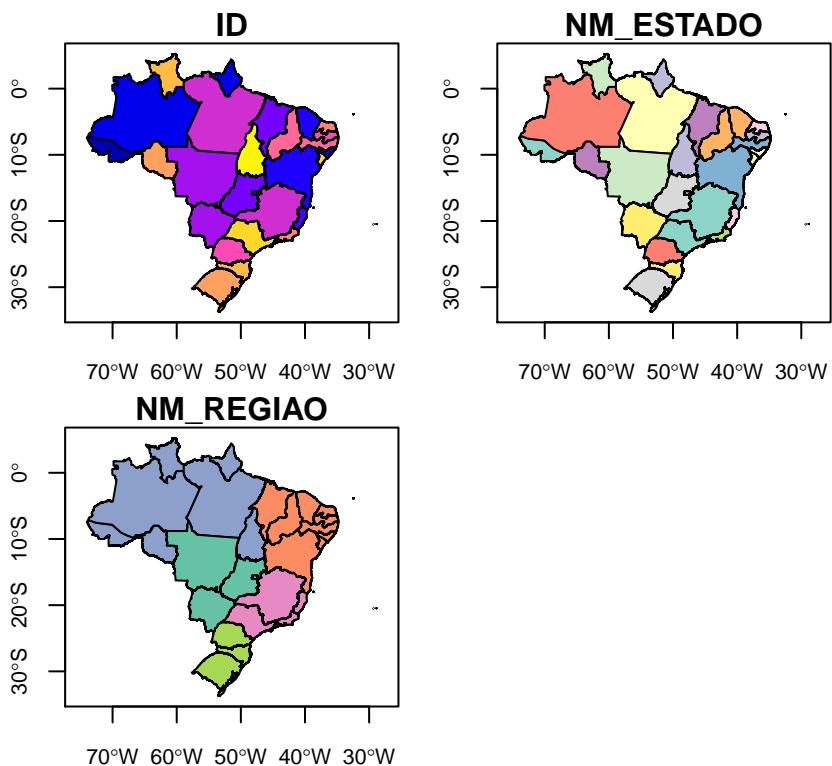
# Pergunta a classe do objeto espacial
class(br) # E um simple feature "sf"
```

```
## [1] "sf"           "data.frame"

summary(br) # um dataframe com uma coluna "lista" sobre a geometria

##      ID      NM_ESTADO      NM_REGIAO      geometry
## Min.   : 1.0   ACRE     : 1   CENTRO-OESTE:4   MULTIPOLYGON :27
## 1st Qu.: 7.5   ALAGOAS  : 1   NORDESTE    :9    epsg:NA       : 0
## Median :14.0   AMAPA    : 1   NORTE       :7    +proj=long...: 0
## Mean   :14.0   AMAZONAS: 1   SUDESTE     :4
## 3rd Qu.:20.5   BAHIA    : 1   SUL        :3
## Max.   :27.0   CEARA    : 1
##          (Other) :21

# plota o mapa (um para cada atributo)
plot(br, axes = T, cex.axis=0.8)
```



```
# Plota somente a geometria
plot(st_geometry(br))
```



Muitas vezes, é necessário verificar as características e validade dos dados importados. As funções *st_geometry()* e *st_crs()* propiciam a verificação da geometria e da projeção de um dado espacial. Certos arquivos não contêm as informações do sistema de coordenadas. Neste caso, é importante determinar isso usando a função *st_crs()*. O sistema de coordenada de referência (CRS) fornece uma maneira padronizada de descrever o espaço. Há muitos CRS diferentes para descrever dados geográficos. Existem vários atributos do CRS, como projeção, datum e elipsóide. "+proj=aea +lat_1=-5 +lat_2=-42 +lat_0=-32+lon_0=-60 +x_0=0 +y_0=0 +ellps=aust_SA +units=m +no_defs" representa uma projeção "Albers Equal Area" utilizando os seguintes parâmetros: Longitude origem: -60°, Latitude origem: -32°, Paralelo padrão 1: -5°, Paralelo padrão 2: -42° e elipsóide *Australian Natl & S. Amer. 1969*.

```
# Define o sistema de projeção
st_geometry(br) # equivalente a print(br$geometry)

## Geometry set for 27 features
## geometry type: MULTIPOLYGON
## dimension: XY
## bbox: xmin: -73.99045 ymin: -33.75208 xmax: -28.83591 ymax: 5.271841
## epsg (SRID): NA
## proj4string: +proj=longlat +ellps=GRS80 +no_defs
## First 5 geometries:

## MULTIPOLYGON (((-63.32721 -7.97672, -63.11838 -...
## MULTIPOLYGON (((-73.18253 -7.335496, -73.1368 -...
## MULTIPOLYGON (((-67.32609 2.029714, -67.31682 2...
## MULTIPOLYGON (((-60.20051 5.264343, -60.19828 5...
## MULTIPOLYGON (((-46.06095 -1.094701, -46.06666 ...

st_crs(br) # por código EPSG e proj4string

## Coordinate Reference System:
##   No EPSG code
##   proj4string: "+proj=longlat +ellps=GRS80 +no_defs"

estradas <- st_read("estradas_albers.shp")

## Reading layer `estradas_albers' from data source `/home/jf/dados/
estradas_albers.shp' using driver
`ESRI Shapefile'
## Simple feature collection with 14637 features and 4 fields
## geometry type: MULTILINESTRING
## dimension: XY
## bbox: xmin: -1465854 ymin: -214657.6 xmax: 3066889 ymax: 4195116
## epsg (SRID): NA
## proj4string: NA

st_crs(estradas) # Não há informação sobre o sistema de coordenadas

## Coordinate Reference System: NA

# Se define:
st_crs(estradas) <- "+proj=aea +lat_1=-5 +lat_2=-42 +lat_0=-32
+lon_0=-60 +x_0=0 +y_0=0 +ellps=aust_SA +units=m +no_defs"

# Teste da validade dos mapas
st_is_valid(br)
```

```
## [1] TRUE TRUE
## [14] TRUE TRUE
## [27] TRUE

# st_is_valid(estradas)
```

4.2 Importação de arquivos vetoriais de outros formatos

A função `st_read()`, que usa o GDAL, é capaz de importar uma grande quantidade de formatos vetoriais, que podem ser visualizados através da função `sf_drivers`. Em geral, `st_read()` determina automaticamente qual *driver* usar com base na extensão do arquivo.

```
drivers_suportados <- st_drivers()
names(drivers_suportados)

## [1] "name"      "long_name"   "write"       "copy"        "is_raster"
## [6] "is_vector" "vsi"

# Lista dos 15 primeiros drivers da lista
head(drivers_suportados[,-2], n = 15)

##          name write copy is_raster is_vector vsi
## PCIDSK     PCIDSK  TRUE FALSE    TRUE    TRUE  TRUE
## netCDF     netCDF  TRUE  TRUE    TRUE   FALSE  FALSE
## JP2OpenJPEG JP2OpenJPEG FALSE  TRUE    TRUE    TRUE  TRUE
## PDF         PDF    TRUE  TRUE    TRUE    TRUE  TRUE
## ESRI Shapefile ESRI Shapefile TRUE FALSE   FALSE  TRUE  TRUE
## MapInfo File MapInfo File  TRUE FALSE   FALSE  TRUE  TRUE
## UK .NTF      UK .NTF FALSE FALSE  FALSE  TRUE FALSE
## OGR_SDTs    OGR_SDTs FALSE FALSE  FALSE  TRUE FALSE
## S57         S57    TRUE FALSE  FALSE  TRUE  TRUE
## DGN         DGN    TRUE FALSE  FALSE  TRUE FALSE
## OGR_VRT     OGR_VRT FALSE FALSE  FALSE  TRUE  TRUE
## REC         REC    FALSE FALSE  FALSE  TRUE FALSE
## Memory      Memory TRUE FALSE  FALSE  TRUE FALSE
## BNA         BNA    TRUE FALSE  FALSE  TRUE  TRUE
## CSV         CSV    TRUE FALSE  FALSE  TRUE  TRUE
```

4.3 Exportação para outros formatos

Podemos salvar um objeto espacial da classe sf em formato *shape* ou em outro formato com a função `st_write()`.

```
# Salva o objeto em formato shape
st_write(br, "Edos_brasil.shp", delete_layer = TRUE)

## Deleting layer `Edos_brasil' using driver
`ESRI Shapefile'
## Writing layer `Edos_brasil' to data source `Edos_brasil.shp' using driver
`ESRI Shapefile'
## features:      27
```

Tabela 1. Tipo de codificação de imagem

Tipo de dado	Valor mínimo	Valor máximo
LOG1S	FALSE (0)	TRUE (1)
INT1S	-127	127
INT1U	0	255
INT2S	-32,767	32,767
INT2U	0	65,534
INT4S	-2,147,483,647	2,147,483,647
INT4U	0	4,294,967,296
FLT4S	-3.4e+38	3.4e+38
FLT8S	-1.7e+308	1.7e+308

```
## fields:      3
## geometry type: Multi Polygon

# Salva em formato OGC GeoPackage (GPKG)
st_write(br, dsn = "Edos_brasil.gpkg", delete_layer = TRUE)

## Deleting layer `Edos_brasil` using driver
`GPKG'
## Updating layer `Edos_brasil` to data source `Edos_brasil.gpkg` using driver
`GPKG'
## features:    27
## fields:      3
## geometry type: Multi Polygon
```

Também é possível transformar objetos espaciais de outros pacotes para **sf**. Esta opção pode ser particularmente útil para usar determinados pacotes que apenas manipulam objetos **sp**. Em **sp** a função **as()** permite importar um objeto **sf** na classe **Spatial** de **sp**. Por outro lado, **st_as_sf()** transforma objetos de **sp** a **sf**.

```
library(sp)
br_sp = as(br, Class = "Spatial")
# Retorna a sf com st_as_sf():
br_sf = st_as_sf(br_sp, "sf")
```

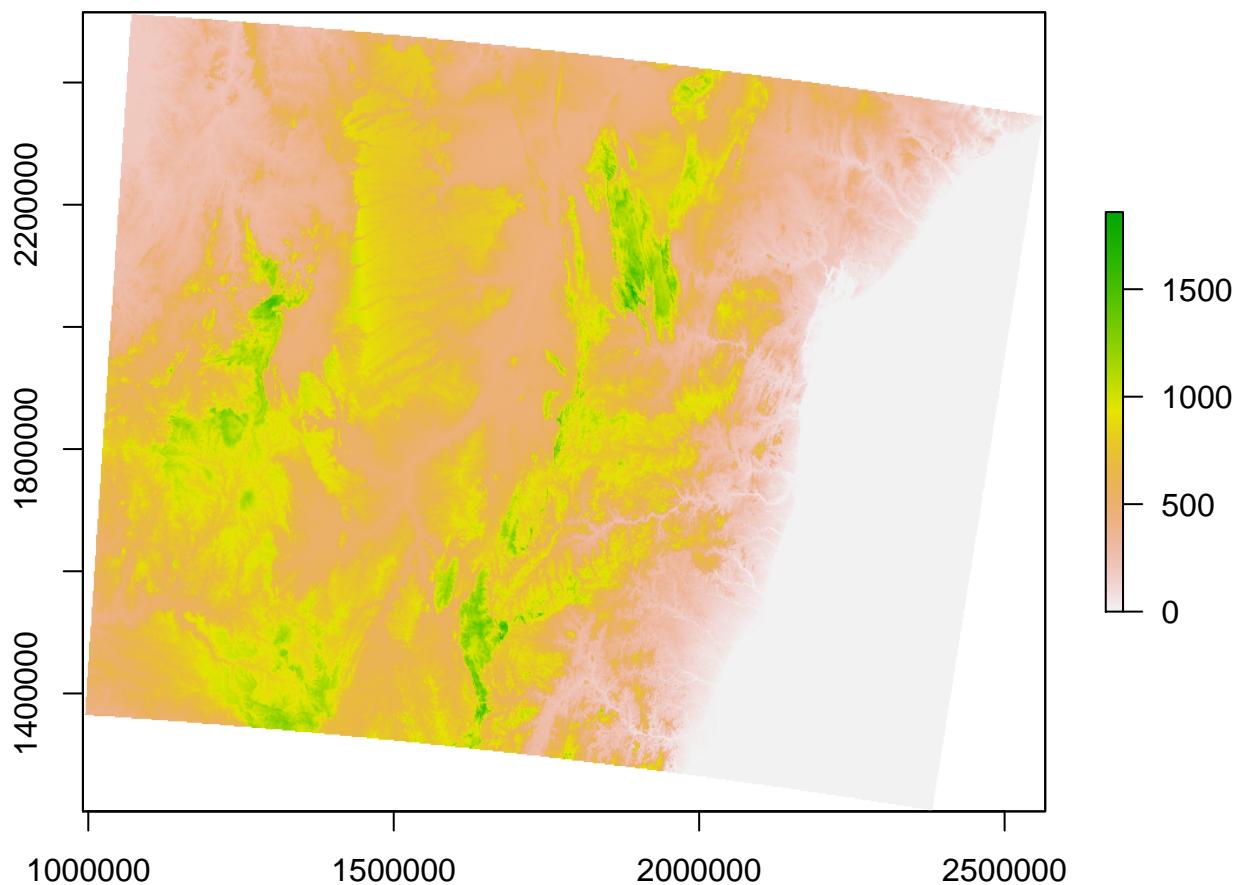
4.4 Importação / exportação de dados raster

Vamos importar um modelo digital de elevação utilizando o pacote **raster**. A função **raster()** importa imagens de muitos formatos, para objetos RasterLayer.

Finalmente, a função **writeRaster()** permite salvar os resultados em vários formatos de imagem, incluindo ENVI, ESRI, ERDAS, GeoTiff, IDRISI e SAGA (opção de formato). A opção de tipo de dados viabiliza a escolha da codificação destes. Por exemplo, INT1U, é a codificação de 8 bits com valores de 0 a 255 (Tabela 1).

```
library(raster)

# Importa o MDE (imagem em formato TIF)
dem <- raster("dem_albers.tif")
plot(dem)
```



```

extent(dem)

##   class      : Extent
##   xmin      : 992814.6
##   xmax      : 2564859
##   ymin      : 1206958
##   ymax      : 2515006

# Formatos disponíveis para salvar (Lista dos 40 primeiros)
formatos <- writeFormats(); head(formatos, 40)

##       name      long_name
## [1,] "raster"    "R-raster"
## [2,] "SAGA"      "SAGA GIS"
## [3,] "IDRISI"    "IDRISI"
## [4,] "IDRISIold" "IDRISI (img/doc)"
## [5,] "BIL"        "Band by Line"
## [6,] "BSQ"        "Band Sequential"
## [7,] "BIP"        "Band by Pixel"
## [8,] "ascii"      "Arc ASCII"
## [9,] "CDF"        "NetCDF"
## [10,] "big"        "big.matrix"
## [11,] "ADRG"      "ARC Digitized Raster Graphics"
## [12,] "BMP"        "MS Windows Device Independent Bitmap"
## [13,] "BT"         "VTP .bt (Binary Terrain) 1.3 Format"
## [14,] "CTable2"    "CTable2 Datum Grid Shift"
## [15,] "EHdr"      "ESRI .hdr Labelled"

```

```
## [16,] "ELAS"      "ELAS"
## [17,] "ENVI"       "ENVI .hdr Labelled"
## [18,] "ERS"        "ERMapper .ers Labelled"
## [19,] "GPKG"       "GeoPackage"
## [20,] "GS7BG"     "Golden Software 7 Binary Grid (.grd)"
## [21,] "GSBG"       "Golden Software Binary Grid (.grd)"
## [22,] "GTiff"      "GeoTIFF"
## [23,] "GTX"        "NOAA Vertical Datum .GTX"
## [24,] "HDF4Image"  "HDF4 Dataset"
## [25,] "HFA"         "Erdas Imagine Images (.img)"
## [26,] "IDA"         "Image Data and Analysis"
## [27,] "ILWIS"       "ILWIS Raster Map"
## [28,] "INGR"       "Intergraph Raster"
## [29,] "ISCE"        "ISCE raster"
## [30,] "ISIS2"      "USGS Astrogeology ISIS cube (Version 2)"
## [31,] "ISIS3"      "USGS Astrogeology ISIS cube (Version 3)"
## [32,] "KRO"         "KOLOR Raw"
## [33,] "LAN"         "Erdas .LAN/.GIS"
## [34,] "Leveller"    "Leveller heightfield"
## [35,] "MBTiles"     "MBTiles"
## [36,] "MRF"         "Meta Raster Format"
## [37,] "netCDF"      "Network Common Data Format"
## [38,] "NITF"        "National Imagery Transmission Format"
## [39,] "NTv2"         "NTv2 Datum Grid Shift"
## [40,] "NWT_GRD"    "Northwood Numeric Grid Format .grd/.tab"

# Salva o raster em formato LAN
writeRaster(dem, filename="DEM_br.lan", format="LAN", overwrite=TRUE,
            datatype="INT2S")
```

5. Operações básicas de SIG (formato vetor)

Neste capítulo revisaremos algumas funções de análise espacial de `sf` utilizando, na primeira seção, os mapas simples indicados no capítulo 3 e, na segunda seção, dados mais realistas.

5.1 Algumas operações de análise espacial

Apresentamos aqui algumas funções de análise espacial em `sf`, que são mais bem entendidas com o uso de dados muito simples. Algumas funções de `sf` são mostradas, visando a verificação da geometria de objetos, o cálculo do comprimento de linhas simples e da área dos polígonos. Na segunda parte do código, elaboramos um mapa com os objetos espaciais que vamos analisar. Na figura elaborada pelo script, os números indicam o número dos pontos e das linhas.

```
# Determina o caminho do espaço de trabalho
setwd("/home/jf/dados")
library(sf)
# Importação dos mapas com st_read (pacote sf)
# Mapa pontos
SFP <- st_read("SFP.shp")

## Reading layer `SFP` from data source `/home/jf/dados/SFP.shp` using driver
`ESRI Shapefile'
## Simple feature collection with 3 features and 2 fields
## geometry type:  POINT
## dimension:      XY
## bbox:            xmin: 2 ymin: 4 xmax: 5 ymax: 8
## epsg (SRID):    NA
## proj4string:    NA

# Mapa linhas
SFL <- st_read("SFL.shp")

## Reading layer `SFL` from data source `/home/jf/dados/SFL.shp` using driver
`ESRI Shapefile'
## Simple feature collection with 3 features and 3 fields
## geometry type:  LINESTRING
## dimension:      XY
## bbox:            xmin: 0 ymin: 0 xmax: 10 ymax: 10
## epsg (SRID):    NA
## proj4string:    NA

# Mapa polígonos
SFPol <- st_read("SFPol.shp")

## Reading layer `SFPol` from data source `/home/jf/dados/SFPol.shp` using driver
`ESRI Shapefile'
## Simple feature collection with 4 features and 3 fields
```

```

## geometry type: POLYGON
## dimension: XY
## bbox: xmin: 0 ymin: 0 xmax: 10 ymax: 10
## epsg (SRID): NA
## proj4string: NA

# Teste da validade dos polígonos
st_is_valid(SFPol)

## [1] TRUE TRUE TRUE TRUE

# Teste se linhas são simples
st_is_simple(SFL)

## [1] TRUE TRUE TRUE

# Calcula área dos polígonos
st_area(SFPol)

## [1] 22.5 23.0 1.0 53.5

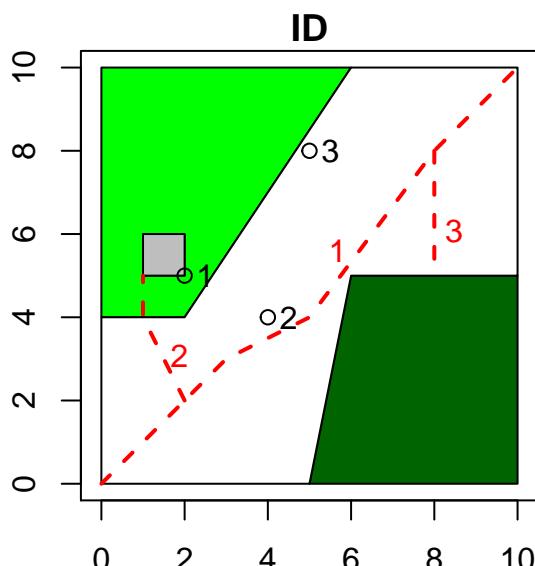
# Calcula comprimento das linhas (não funciona para multiline)
st_length(SFL)

## [1] 14.307136 3.236068 3.000000

# Mapa dos dados
plot(SFPol["ID"], col=c("Dark Green", "green", "grey", "white"), axes=T, reset = FALSE)
plot(st_geometry(SFL), lty=2, lwd=2, col="red", add=TRUE)
plot(st_geometry(SFP), add=TRUE)
plot(st_geometry(SFP) + c(0.5, 0), pch=c(49, 50, 51), cex=1, add=TRUE)

# Adiciona o número das linhas
text(5, 5.5, "1", pos=4, col="red", cex=1)
text(1.2, 3, "2", pos=4, col="red", cex=1)
text(7.8, 6, "3", pos=4, col="red", cex=1)

```



Primeiramente, é apresentado `st_intersects()`, que possibilita verificar se os elementos de objetos espaciais se intersectam ou não. A opção `sparse` permite manejar duas formas de apresentação dos resultados, a saber: como uma matriz (`sparse = FALSE`), neste caso de 3 x 4 (3 linhas, 4 polígonos) ou de uma forma condensada (`sparse = TRUE`). A função `st_distance()` propicia o cálculo da distância euclidiana mais curta entre elementos de dois objetos e a apresentação dos resultados em forma de matriz.

```
## Operadores lógicos binários
# Interseção de pontos e polígonos
st_intersects(SFP, SFPol, sparse = FALSE)

##      [,1]  [,2]  [,3]  [,4]
## [1,] FALSE TRUE  TRUE FALSE
## [2,] FALSE FALSE FALSE TRUE
## [3,] FALSE FALSE FALSE TRUE

st_intersects(SFP, SFPol, sparse = TRUE)

## Sparse geometry binary predicate list of length 3, where the predicate was
`intersects'
## 1: 2, 3
## 2: 4
## 3: 4

# Interseção entre linhas e polígonos
st_intersects(SFL, SFPol, sparse = FALSE)

##      [,1]  [,2]  [,3]  [,4]
## [1,] FALSE FALSE FALSE TRUE
## [2,] FALSE TRUE  TRUE TRUE
## [3,] TRUE FALSE FALSE TRUE

st_intersects(SFL, SFPol, sparse = TRUE)

## Sparse geometry binary predicate list of length 3, where the predicate was
`intersects'
## 1: 4
## 2: 2, 3, 4
## 3: 1, 4

# Distância mais curta entre elementos de dois objetos
st_distance(SFP, SFPol)

##      [,1]      [,2]      [,3]      [,4]
## [1,] 3.922323 0.0000000 0.0000000 0.5547002
## [2,] 1.765045 1.6641006 2.236068 0.0000000
## [3,] 3.162278 0.2773501 3.605551 0.0000000
```

Finalmente, através de `st_intersection()` é possível realizar uma interseção geométrica entre dois objetos, obtendo-se como resultado um objeto espacial, cuja tabela de atributos agrupa a informação de ambos, onde estes se sobrepõem. No código a seguir, esta função permite calcular o comprimento dos segmentos que coincidem espacialmente com as diferentes categorias dos polígonos.

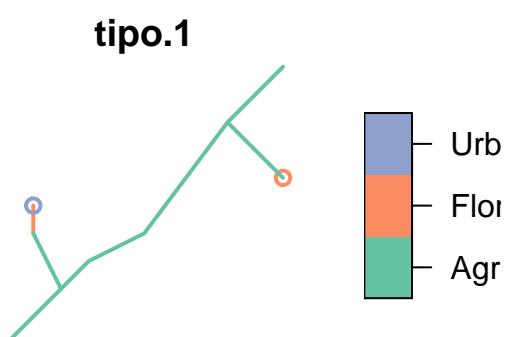
```
# Operadores geométricos (interseção geométrica)
SFPxSFPol <- st_intersection(SFP, SFPol)
print(SFPxSFPol) # ponto 1 é duplicado porque pertence a 2 polígonos

## Simple feature collection with 4 features and 5 fields
## geometry type: POINT
## dimension: XY
## bbox: xmin: 2 ymin: 4 xmax: 6 ymax: 8
## epsg (SRID): NA
## proj4string: NA
##     num nome ID code      tipo      geometry
## 1    1 Poco  2   F Floresta POINT (2 6)
## 1.1  1 Poco  3   U   Urbano POINT (2 6)
## 2    2 Posto 4   A Agricultura POINT (4 4)
## 3    3 Poco  4   A Agricultura POINT (6 8)

SFLxSFPol <- st_intersection(SFL, SFPol)
print(SFLxSFPol) # tem pontos e linhas

## Simple feature collection with 6 features and 6 fields
## geometry type: GEOMETRY
## dimension: XY
## bbox: xmin: 0 ymin: 0 xmax: 10 ymax: 10
## epsg (SRID): NA
## proj4string: NA
##     num      tipo code ID code.1      tipo.1
## 3    3 Pavimentada p  1   F Floresta
## 2    2 Terra     t  2   F Floresta
## 2.1  2 Terra     t  3   U   Urbano
## 1    1 Terra     t  4   A Agricultura
## 2.2  2 Terra     t  4   A Agricultura
## 3.1  3 Pavimentada p  4   A Agricultura
##                         geometry
## 3                  POINT (10 6)
## 2                  LINESTRING (1 4, 1 5)
## 2.1                 POINT (1 5)
## 1 LINESTRING (0 0, 3 3, 5 4, ...
## 2.2             LINESTRING (2 2, 1 4)
## 3.1             LINESTRING (8 8, 10 6)

plot(SFLxSFPol["tipo.1"], lwd=2)
```



```
# Extrai somente as linhas
SFLxSFPol_1 <- st_collection_extract(SFLxSFPol,type="LINESTRING")
compri <- st_length(SFLxSFPol_1) # Cálculo comprimento
SFLxSFPol_1$compri <- compri # Resultados em uma nova coluna da tabela
# Soma o comprimento dos segmentos de cada tipo de coberta
Suma_c <- aggregate(SFLxSFPol_1$compri ~ tipo.1, FUN = sum, data = SFLxSFPol_1)
print(Suma_c)

##          tipo.1 SFLxSFPol_1$compri
## 1 Agricultura      19.37163
## 2 Floresta         1.00000
```

5.2 Análise espacial em formato vetor

Neste item, vamos importar dois mapas em formato *shape*: um dos estados brasileiros e o outro das principais estradas. Vamos visualizar a tabela de atributos destes mapas e reprojetar o mapa dos estados que está em Lat/Long para a Cônica de Albers, que corresponde à projeção na qual se encontra o mapa de estradas.

```
# Carrega o pacote sf
library(sf)
# Determina a rota do espaço de trabalho
setwd("/home/jf/dados")
# Importação dos mapas com st_read (pacote sf)
# Mapa dos estados do Brasil
br <- st_read("brasil.shp")

## Reading layer `brasil` from data source `/home/jf/dados/brasil.shp` using driver
`ESRI Shapefile'
## Simple feature collection with 27 features and 3 fields
## geometry type: MULTIPOLYGON
## dimension: XY
## bbox: xmin: -73.99045 ymin: -33.75208 xmax: -28.83591 ymax: 5.271841
## epsg (SRID): NA
## proj4string: +proj=longlat +ellps=GRS80 +no_defs

st_crs(br)

## Coordinate Reference System:
##   No EPSG code
##   proj4string: "+proj=longlat +ellps=GRS80 +no_defs"

summary(br)

##           ID        NM_ESTADO       NM_REGIAO      geometry
## Min.    : 1.0     ACRE     : 1     CENTRO-OESTE:4     MULTIPOLYGON :27
## 1st Qu.: 7.5     ALAGOAS  : 1     NORDESTE   :9     epsg:NA      : 0
## Median :14.0     AMAPA    : 1     NORTE      :7     +proj=long...: 0
## Mean   :14.0     AMAZONAS: 1     SUDESTE    :4
## 3rd Qu.:20.5     BAHIA    : 1     SUL        :3
## Max.   :27.0     CEARA    : 1     (Other)    :21
```

```

# Mapa de estradas
estrad <- st_read("estradas_albers.shp")

## Reading layer `estradas_albers` from data source `/home/jf/dados/
estradas_albers.shp` using driver
`ESRI Shapefile'
## Simple feature collection with 14637 features and 4 fields
## geometry type: MULTILINESTRING
## dimension: XY
## bbox: xmin: -1465854 ymin: -214657.6 xmax: 3066889 ymax: 4195116
## epsg (SRID): NA
## proj4string: NA

st_crs(estrad) <- "+proj=aea +lat_1=-5 +lat_2=-42 +lat_0=-32 +lon_0=-60 +x_0=0
+y_0=0
+ellps=aust_SA +units=m +no_defs"
# Atributos do mapa de estradas
names(estrad)

## [1] "SL_ID"      "MED_DESCRI" "RTT_DESCRI" "F_CODE.Des" "geometry"

summary(estrad)

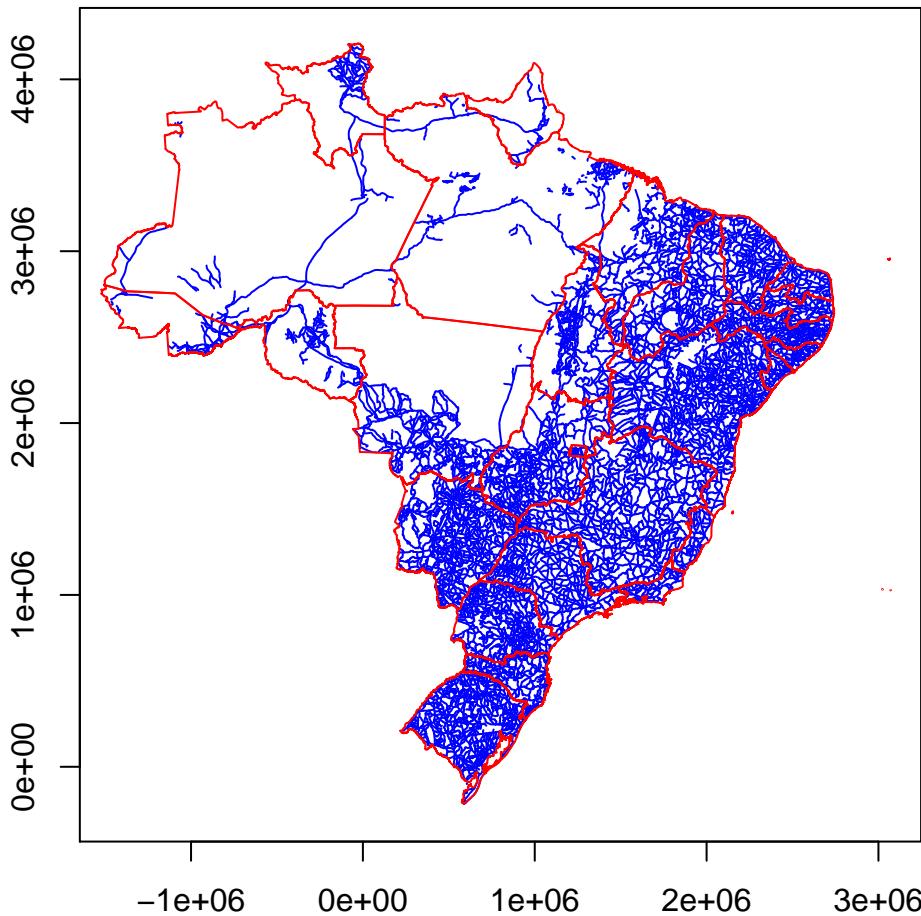
##          SL_ID           MED_DESCRI          RTT_DESCRI
## 0       : 1   Unknown      : 926 Primary Route  :2473
## 1       : 1   With Median  : 61 Secondary Route:9494
## 10      : 1 Without Median:11906 Unknown       : 926
## 100     : 1    NA's        :1744 NA's          :1744
## 1000    : 1
## 10000   : 1
## (Other):14631
##          F_CODE_Des          geometry
## Road :12893 MULTILINESTRING:14637
## Trail: 1744  epsg:NA      :     0
##                  +proj=aea ... :     0
## 
## 
## 
## 

# Reprojeta br a Albers
br_a <- st_transform(br, st_crs(estrad))
st_crs(br_a)

## Coordinate Reference System:
##   No EPSG code
##   proj4string: "+proj=aea +lat_1=-5 +lat_2=-42 +lat_0=-32 +lon_0=-60 +x_0=0
+y_0=0 +ellps=aust_SA +units=m +no_defs"

# Calcula a área dos estados
br_a$AreaEdo <- st_area(br_a) / 1000000 # km2
# Plota os estados e as estradas juntos
plot(st_geometry(estrad), col="blue", axes=TRUE)
plot(st_geometry(br_a), border="red", add=TRUE)

```



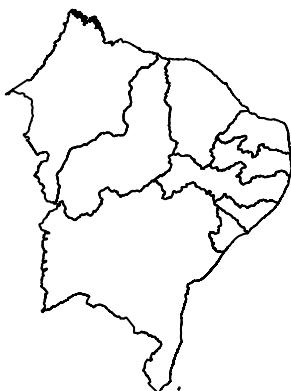
Em seguida, vamos calcular o comprimento das estradas principais em cada estado do nordeste do Brasil (Alagoas, Bahia, Ceará, Maranhão, Paraíba, Piauí, Pernambuco, Rio Grande do Norte e Sergipe). Em sf, a seleção de certas características de um mapa se realiza com base na tabela de atributos. Por exemplo, `carr[RTT_DESCRI=="Primary Road",]` nos possibilita selecionar as linhas que correspondem às estradas principais. Porém, o resultado da seleção é um objeto espacial sf e não uma simples tabela.

A função `st_intersection()` viabiliza a realização da interseção entre dois objetos espaciais. Neste caso, o objeto obtido é compreendido por linhas (estradas principais). A informação do estado, na qual se encontram os segmentos das estradas, está na tabela de atributos. Se calcula então, o comprimento de cada segmento, e se somam os valores de comprimento para cada estado.

```
# Qual é o comprimento das principais estradas em cada estado do nordeste?
# Seleciona as estradas principais ("Primary roads")
estrad_p <- estrad[RTT_DESCRI=="Primary Road",]
# Seleciona os estados do nordeste (Alagoas, Bahia, Ceará, Maranhão, Paraíba,
# Piauí, Pernambuco, Rio Grande do Norte e Sergipe)
nordeste <- br_a[br_a$NM_ESTADO %in% c("ALAGOAS", "BAHIA", "CEARA", "MARANHAO", "PARAIBA",
                                         "PIAUI", "PERNAMBUCO", "RIO GRANDE DO NORTE", "SERGIPE"),]
# Alternativa mais simples
# porque já existe um campo indicando a região
nordeste <- br_a[br_a$NM_REGIAO == "NORDESTE",]
class(nordeste)

## [1] "sf"           "data.frame"

plot(st_geometry(nordeste))
```



```
# Interseção entre mapas de estradas principais e Estados do Nordeste
estradpXedos <- st_intersection(estrada_p,nordeste)
names(estradpXedos)

## [1] "SL_ID"      "MED_DESCRI" "RTT_DESCRI" "F_CODE.Des" "ID"
## [6] "NM_ESTADO"  "NM_REGIAO"  "AreaEdo"     "geometry"

summary(estradpXedos)

##      SL_ID                  MED_DESCRI                  RTT_DESCRI
## 2290 : 3    Unknown       : 196 Primary Route   : 940
## 4213 : 3 With Median    :  0 Secondary Route:3899
## 1051 : 2 Without Median:4839 Unknown        : 196
## 1068 : 2    NA's          : 231 NA's           : 231
## 1176 : 2
## 1283 : 2
## (Other):5252
##      F_CODE_Des      ID      NM_ESTADO      NM_REGIAO
## Road :5035 Min.   : 2.0   BAHIA    :2056 CENTRO-OESTE: 0
## Trail: 231 1st Qu.: 5.0   PERNAMBUCO: 678 NORDESTE   :5266
##            Median : 6.0   PIAUI     : 672 NORTE      : 0
##            Mean    :10.3  MARANHAO   : 556 SUDESTE     : 0
##            3rd Qu.:17.0  CEARA     : 505 SUL        : 0
##            Max.   :26.0  PARAIBA    : 239
##                      (Other)   : 560
##      AreaEdo      geometry
## Min.   : 21915  LINESTRING :5168
## 1st Qu.: 98149  MULTILINESTRING: 98
## Median :251580  epsg:NA     :  0
## Mean   :320861  +proj=aea ... :  0
## 3rd Qu.:564737
## Max.   :564737
## 

# Converte as multilines em lines (para cálculo do comprimento)
estradpXedos_l <- st_collection_extract(estradpXedos,type="LINESTRING")
comprimento <- st_length(estradpXedos_l) / 1000 # km
head(comprimento)

## Units: [m]
## [1] 5.741655 4.340072 15.547179 20.835949 19.300506 27.675411
```

```
# Adiciona coluna compri à tabela de atributos
estradpXedos_1$compri <- comprimento
# Soma o comprimento dos segmentos de cada estado
Soma_compri <- aggregate(compri ~ NM_ESTADO, FUN = sum, data = estradpXedos_1)
print(Soma_compri)

##           NM_ESTADO      compri
## 1          ALAGOAS 3109.105 []
## 2          BAHIA 43712.672 []
## 3          CEARA 11358.465 []
## 4        MARANHAO 14849.949 []
## 5         PARAIBA 4603.082 []
## 6 PERNAMBUCO 10547.611 []
## 7        PIAUI 15950.623 []
## 8 RIO GRANDE DO NORTE 4284.618 []
## 9        SERGIPE 2219.499 []
```

Posteriormente, se calcula a proporção do território da região nordeste, que se encontra a menos de 30 km de uma estrada principal. Para esse fim, se elabora uma região de *buffer* de 30 km ao redor das estradas. As etapas seguintes são similares ao cálculo realizado para o comprimento de estradas. É possível observar que a maioria das operações são tabulares, se realizando da mesma forma que com qualquer tabela *dataframe*.

```
# Que proporção do território estadual está a menos de 30 km de uma
# estrada principal?
# Cria um buffer de 30 km ao redor das estradas principais
buf <- st_buffer(estrad_p,dist=30000)
plot(st_geometry(buf),axes=FALSE)
```



```
# Interseção entre buffer e Estados do Nordeste
bufXedos <- st_intersection(buf,nordeste)
area <- st_area(bufXedos) / 1000000 # km2
head(area)

## Units: [m^2]
## [1] 889.33619 95.58438 1785.55215 1827.03439 1779.13313 2538.16114
```

```

bufXedos$area <- area
# Soma da área em cada estado
Soma_area <- aggregate(area ~ NM_ESTADO, FUN = sum, data = bufXedos)
print(Soma_area)

##           NM_ESTADO      area
## 1          ALAGOAS 707284.5 []
## 2          BAHIA  8008202.8 []
## 3          CEARA 1933706.8 []
## 4        MARANHAO 2346673.9 []
## 5        PARAIBA  866903.1 []
## 6      PERNAMBUCO 2320139.3 []
## 7         PIAUI 2741918.2 []
## 8 RIO GRANDE DO NORTE 701900.2 []
## 9        SERGIPE 499431.2 []

# Junta ao mapa nordeste a tabela Soma_area (por chave em comum)
nordeste <- merge(nordeste,Soma_area,by="NM_ESTADO")
# Calcula a proporção da área do buffer / área do estado
nordeste$prop <- nordeste$area / nordeste$AreaEdo

```

O código apresentado nesse trecho final, relaciona a tabela de atributos do mapa dos estados com uma tabela externa (população e produto interno bruto dos estados), onde se calculam índices baseados em variáveis representadas por diferentes colunas da tabela. Finalmente, se exporta o objeto espacial obtido ao formato *shape*.

```

# Determina a rota do espaço de trabalho
setwd("/home/jf/dados")
# População
tab_pop <- read.csv("populacao.csv")
head(tab_pop)

##   NumEdo Sigla   Estado     Pop      PIB
## 1      1    AC    Acre  732793  8477000
## 2      2    AL  Alagoas 3120922 24575000
## 3      3    AP  Amapá  668689  8266000
## 4      4    AM Amazonas 3480937 59779000
## 5      5    BA Bahia 14021432 154340000
## 6      6    CE  Ceara 8448055  77865000

head(br_a)

## Simple feature collection with 6 features and 4 fields
## geometry type: MULTIPOLYGON
## dimension: XY
## bbox: xmin: -1523283 ymin: 2127159 xmax: 1582934 ymax: 4209948
## epsg (SRID): NA
## proj4string: +proj=aea +lat_1=-5 +lat_2=-42 +lat_0=-32 +lon_0=-60 +x_0=0
## +y_0=0 +ellps=aust_SA +units=m +no_defs
##   ID NM_ESTADO NM_REGIAO           geometry
## 1 22 RONDANIA    NORTE MULTIPOLYGON (((-361668.7 2...
## 2  1      ACRE    NORTE MULTIPOLYGON (((-1437496 27...
## 3  4 AMAZONAS    NORTE MULTIPOLYGON (((-849007.6 3...

```

```

## 4 23     RORAIMA      NORTE MULTIPOLYGON (((-23694.93 4...
## 5 14       PARA        NORTE MULTIPOLYGON (((1582934 346...
## 6 3       AMAPA        NORTE MULTIPOLYGON (((1034081 404...
##               AreaEdo
## 1 237591.8 [m^2]
## 2 164123.0 [m^2]
## 3 1559172.8 [m^2]
## 4 224302.5 [m^2]
## 5 1247966.2 [m^2]
## 6 142829.4 [m^2]

# Junta as duas tabelas com a chave numérica
br_a <- merge(br_a,tab_pop,by.x="ID",by.y="NumEdo")
head(br_a)

## Simple feature collection with 6 features and 8 fields
## geometry type: MULTIPOLYGON
## dimension: XY
## bbox: xmin: -1523283 ymin: 1452958 xmax: 2671844 ymax: 4095118
## epsg (SRID): NA
## proj4string: +proj=aea +lat_1=-5 +lat_2=-42 +lat_0=-32 +lon_0=-60 +x_0=0
+y_0=0 +ellps=aust_SA +units=m +no_defs
##   ID NM_ESTADO NM_REGIAO          AreaEdo Sigla    Estado      Pop
## 1  1     ACRE     NORTE  164123.0 [m^2]    AC     Acre  732793
## 2  2    ALAGOAS   NORDESTE  27778.7 [m^2]    AL  Alagoas 3120922
## 3  3     AMAPA     NORTE  142829.4 [m^2]    AP   Amapá  668689
## 4  4   AMAZONAS   NORTE 1559172.8 [m^2]    AM Amazonas 3480937
## 5  5     BAHIA   NORDESTE  564737.4 [m^2]    BA   Bahia 14021432
## 6  6     CEARA   NORDESTE 148921.5 [m^2]    CE  Ceara  8448055
##           PIB          geometry
## 1 8477000 MULTIPOLYGON (((-1437496 27...
## 2 24575000 MULTIPOLYGON (((2640007 246...
## 3 8266000 MULTIPOLYGON (((1034081 404...
## 4 59779000 MULTIPOLYGON (((-849007.6 3...
## 5 154340000 MULTIPOLYGON (((2227770 256...
## 6 77865000 MULTIPOLYGON (((2090475 303...

names(br_a)

## [1] "ID"          "NM_ESTADO"   "NM_REGIAO"   "AreaEdo"     "Sigla"
## [6] "Estado"      "Pop"         "PIB"         "geometry"

# Produto Interno Bruto per capita (ou por pessoa)
br_a$PIBC <- br_a$PIB;br_a$Pop
# Densidade de população (núm habitantes por km2)
br_a$dens <- br_a$Pop;br_a$AreaEdo

# Salva o objeto em formato shape
st_write(br_a,"br_a.shp",delete_layer = TRUE)

```

6. Operações básicas de SIG (dados *raster*)

6.1 Operações de análise espacial

Apresentamos aqui algumas funções para análise espacial presentes no pacote **raster**, que são mais fáceis de entender com dados de pequenas dimensões. Para uma revisão mais profunda sobre o pacote **raster**, consultar Hijmans (2016).

Vamos criar alguns dados *raster* de dimensão reduzida (3 x 4 células) para mostrar de forma fácil os resultados obtidos, observando os mapas *raster* como matrizes. Os dados *raster* podem ser eventualmente tratados (manejados) como imagens multibanda com o uso das funções *stack()* ou *brick()*. As funções *res()*, *dim()*, *xmin()*, *xmax()*, *ymin()* e *ymax()* possibilitam a obtenção de algumas das características do mapa como resolução (tamanho das células), dimensão (número de linhas, colunas e bandas) e coordenadas extremas. A função *cellStats()* permite calcular índices sobre o conjunto da imagem (levando-se em consideração todas as células).

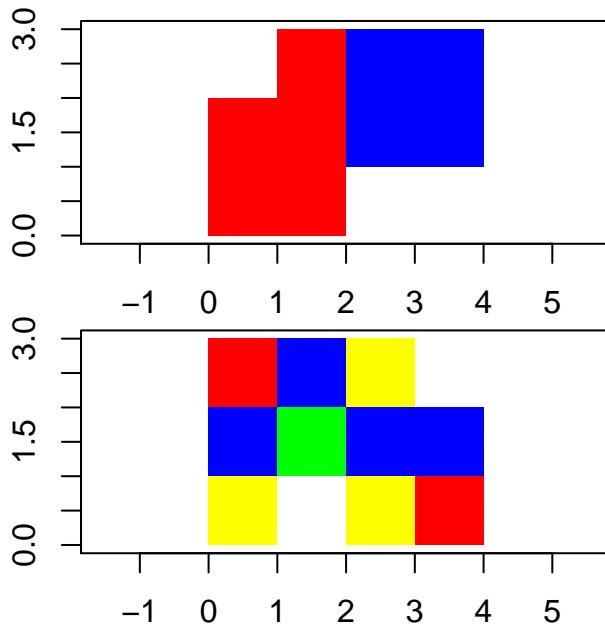
```
library(raster)
#####
# Dados muito simples
# Cria duas matrizes
m1 <- matrix(c(3,1,2,2,1,1,2,2,1,1,3,3), ncol=4, nrow=3, byrow=TRUE)
print(m1)

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

m2 <- matrix(c(1,2,3,4,2,NA,2,2,3,4,3,1), ncol=4, nrow=3, byrow=TRUE)
print(m2)

##      [,1] [,2] [,3] [,4]
## [1,]     1     2     3     4
## [2,]     2    NA     2     2
## [3,]     3     4     3     1

r1 <- raster(m1)
r2 <- raster(m2)
extent(r1) <- extent(r2) <- extent(c(0,4,0,3))
### stack
colortable(r1) <- c("green", "red", "blue")
colortable(r2) <- c("green", "red", "blue", "yellow")
r12 <- stack(r1, r2)
plot(r1, axes=TRUE); plot(r2, axes=TRUE)
```



```
# Características do raster
res(r1) # Resolução
## [1] 1 1

dim(r12) # Dimensão
## [1] 3 4 2

xmax(r1) # Existem também xmin, ymin y ymax
## [1] 4

cellStats(r1,"sum") # soma de todos os pixels
## [1] 22

cellStats(r1,"sd") # desvio padrão de todos os pixels
## [1] 0.8348471
```

6.1.1 Álgebra de mapas

O pacote **raster** viabiliza a realização de todas as operações de SIG. As operações de álgebra de mapa se aplicam *pixel a pixel*. No primeiro exemplo, se adiciona o valor 2 ao valor de cada célula do mapa r1. No segundo, se somam, pixel a pixel, o valor do mapa r1 com o dobro do valor do mapa r2.

```
# Algebra de Mapa
print(as.matrix(r1)) # mapa r1
##      [,1] [,2] [,3] [,4]
## [1,]     3     1     2     2
## [2,]     1     1     2     2
## [3,]     1     1     3     3
```

```

print(as.matrix(r2)) # mapa r2

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    NA    2    2
## [3,]    3    4    3    1

sum1 <- r1 + 2; print(as.matrix(sum1)) # soma valor 2 a r1

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

sum2 <- r1 + 2*r2; print(as.matrix(sum2)) # soma r1 e o dobro de r2

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

```

A função ***overlay()*** possibilita a utilização de funções definidas pelo usuário. No exemplo que segue, a função `x + 2*y -z` permite combinar três mapas de entrada (soma ponderada).

```

sum3 <- overlay(r1, r2, sum1, fun=function(x, y, z){ x + 2*y -z} )
print(as.matrix(sum3))

##      [,1] [,2] [,3] [,4]
## [1,]    0    2    4    6
## [2,]    2    NA    2    2
## [3,]    4    6    4    0

```

6.1.2 Mosaicagem e corte

É possível reunir vários mapas para elaborar mosaicos com a função ***merge()***. Por outro lado, ***crop()*** permite recortar uma imagem com base em coordenadas extremas. Estas operações se baseiam na extensão (*extent*) dos mapas, que contêm as coordenadas extremas (esquinas inferiores e superiores) na sequência xmin, xmax, ymin e ymax. A figura 4 ilustra as operações de elaboração e recorte de um mosaico realizadas com o código que se segue.

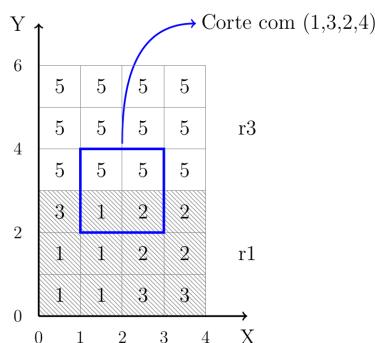


Figura 4. Mosaico e corte de rasters

```

# Operações tipo SIG
# Mosaico (merge) e corte (crop)
m3 <- matrix(c(5,5,5,5,5,5,5,5,5,5,5,5), ncol=4, nrow=3, byrow=TRUE)
print(m3)

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

r3 <- raster(m3)
extent(r3) <- extent(c(0,4,3,6)) # r3 está ao norte de r1
# Mosaico (merge)
mosaico <- merge(r1,r3)
extent(mosaico)

## class       : Extent
## xmin        : 0
## xmax        : 4
## ymin        : 0
## ymax        : 6

print(as.matrix(mosaico))

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

# Corte (crop)
extent_corte <- extent(c(1,3,2,4))
corte <- crop(mosaico, extent_corte)
print(as.matrix(corte))

##      [,1] [,2]
## [1,]     5     5
## [2,]     1     2

```

6.1.3 Reclassificação

A seguir, apresentamos algumas opções para realizar reclassificações de valores de um mapa disponível em `raster`. Com `r2[r2 > 2] <- 6`, estamos atribuindo o valor 6 às células de `r2` que têm um valor estritamente maior que 2. Para reclassificações envolvendo vários intervalos, é mais fácil usar tabelas de reclassificação. Essas tabelas são matrizes com três colunas: as duas primeiras indicam os limites do intervalo de reclassificação e a terceira o valor de saída. O primeiro valor não está incluído no intervalo, o segundo está incluído. Por exemplo, a primeira linha da tabela de reclassificação 0 2 0 indica que valores estritamente acima de zero e menores ou iguais a 2 são reclassificados para zero. A segunda fila 2 5 1 permite reclassificar os valores entre 2 (não incluído) e 5 (incluído) no valor 1.

```

# Reclassificações
print(as.matrix(r2))

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    3    4
## [2,]    2    NA    2    2
## [3,]    3    4    3    1

r2[r2 > 2] <- 6
print(as.matrix(r2))

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    6    6
## [2,]    2    NA    2    2
## [3,]    6    6    6    1

# Tabela de reclassificação (intervalos)
m <- c(0, 2, 0, 2, 5, 1)
tabla_reclas <- matrix(m, ncol=3, byrow=TRUE)
print(tabla_reclas)

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

reclas <- reclassify(mosaico, tabla_reclas)
print(as.matrix(reclas))

##      [,1] [,2] [,3] [,4]
## [1,]    1    1    1    1
## [2,]    1    1    1    1
## [3,]    1    1    1    1
## [4,]    1    0    0    0
## [5,]    0    0    0    0
## [6,]    0    0    1    1

```

Para substituir valores de um mapa por outros, podem-se usar ambos os métodos apresentados anteriormente. Com `r2[r2 == 6] <- 9`, estamos atribuindo o valor 9 às células de r2 que têm um valor igual a 6. Para reclassificações envolvendo várias substituições, é mais simples usar a função `subs()` com uma tabela de reclassificação com duas colunas: a primeira indica o valor de entrada e a segunda o valor de saída.

```

# Substituição de valores
print(as.matrix(r2))

##      [,1] [,2] [,3] [,4]
## [1,]    1    2    6    6
## [2,]    2    NA    2    2
## [3,]    3    4    3    1

r2[r2 == 6] <- 9
print(as.matrix(r2))

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

```

```

## [1,]    1    2    9    9
## [2,]    2    NA   2    2
## [3,]    9    9    9    1

# Com tabela
tab_subs <- data.frame(id=c(1,2,3),v=c(1,56,84))
print(tab_subs) # Tabela de substituição

##   id  v
## 1  1  1
## 2  2 56
## 3  3 84

sub <- subs(r1, tab_subs)
print(as.matrix(r1))

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

print(as.matrix(sub))

##      [,1] [,2] [,3] [,4]
## [1,]   84    1   56   56
## [2,]    1    1   56   56
## [3,]    1    1   84   84

```

6.1.4 Modificação do arranjo espacial

Caso seja de interesse, pode-se mudar o arranjo espacial das células. A função `aggregate()` permite reagrupar vários pixels, para a obtenção de um mapa *raster* de menor resolução espacial. A função `fun` controla o cálculo do valor do pixel de baixa resolução, que corresponde a várias celulas nos arquivos de entrada. Nos dois exemplos abaixo, são reunidos pixels em conjuntos 2 x 2. Na primeira opção somamos os valores de quatro células, na segunda escolhemos o valor mais comum (majoritário).

```

# Reamostragens
# Agregação espacial
print(as.matrix(mosaico)) # raster de entrada

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

agreg <- aggregate(mosaico, fact=2, fun=sum) # agregação com soma
print(as.matrix(agreg))

##      [,1] [,2]

```

```
## [1,]    20    20
## [2,]    14    14
## [3,]     4    10

agreg2 <- aggregate(mosaico, fact=2, fun=modal,na.rm = TRUE) # moda
print(as.matrix(agreg2))

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

O pacote **raster** permite também realizar reamostragens com os métodos mais comumente utilizados na área de sensoriamento remoto, como o método do vizinho mais próximo ou de interpolação bilinear.

```
# Reamostragem
# Imagem do mesmo tamanho que mosaico com menos linhas
m4 <- matrix(rep(1,20),ncol=4,nrow=5)
r4 <- raster(m4)
extent(r4) <- extent(mosaico)
resv <- resample(mosaico,r4,method="nrb") # vizinho mais próximo
print(as.matrix(resv))

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

resb <- resample(mosaico,r4,method="bilinear") # bilinear
print(as.matrix(resb))

##      [,1] [,2] [,3] [,4]
## [1,]  5.0   5  5.0  5.0
## [2,]  5.0   5  5.0  5.0
## [3,]  4.0   3  3.5  3.5
## [4,]  1.6   1  2.0  2.0
## [5,]  1.0   1  2.9  2.9
```

6.1.5 Operações focais e zonais

O pacote **raster** viabiliza a realização de operações de filtro espacial conhecido como convolução (kernel ou núcleo). O filtro de matriz de convolução usa uma matriz de pesos de ponderação, comumente de 3 x 3 ou 5 x 5 pixels. O filtro analisa cada célula da imagem sucessivamente. Para cada uma delas, o valor dessa célula será a soma do produto do valor de cada pixel ao redor da mesma e do valor correspondente na matriz de convolução. Por exemplo, as duas operações focais abaixo são equivalentes e calculam os valores médios das células em uma janela móvel de 3 x 3 pixels. O cálculo é feito apenas para células que possuem oito vizinhos.

```
# Operações focais
help(focal)
media <- focal(r1, w=matrix(1/9, ncol=3, nrow=3))
```

```

media <- focal(r1, w=matrix(1,3,3), fun=mean) # outra forma
print(as.matrix(r1))

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

print(as.matrix(media))

##      [,1]      [,2]      [,3]      [,4]
## [1,]    NA        NA        NA        NA
## [2,]    NA 1.666667 1.888889    NA
## [3,]    NA        NA        NA        NA

```

No entanto, a opção "pad = TRUE" permite evitar esse efeito nas bordas da imagem (veja a ajuda). No exemplo que segue, o cálculo com as opções pad=TRUE e padValue=NA permite criar colunas e filas virtuais ao redor da imagem com valores NA e calcular a média para células do bordo da imagem.

```

# Efeito das opções PAD e padValue
media <- focal(r1, w=matrix(1,3,3),fun=mean)
# com PAD e padValue
media2 <- focal(r1, w=matrix(1,3,3), fun=mean,pad=TRUE,padValue=NA,na.rm=TRUE)
print(as.matrix(media))

##      [,1]      [,2]      [,3]      [,4]
## [1,]    NA        NA        NA        NA
## [2,]    NA 1.666667 1.888889    NA
## [3,]    NA        NA        NA        NA

print(as.matrix(media2))

##      [,1]      [,2]      [,3]      [,4]
## [1,] 1.500000 1.666667 1.666667 2.000000
## [2,] 1.333333 1.666667 1.888889 2.333333
## [3,] 1.000000 1.500000 2.000000 2.500000

```

O pacote **raster** também permite realizar operações zonais, que consistem em calcular algum índice em um mapa, estratificando o cálculo com base em outro mapa. No exemplo abaixo, os valores das células do mapa r2 são adicionados para cada categoria (valor) do mapa r1 (Figura 5). O resultado obtido é uma tabela que indica o resultado do cálculo para cada zona.

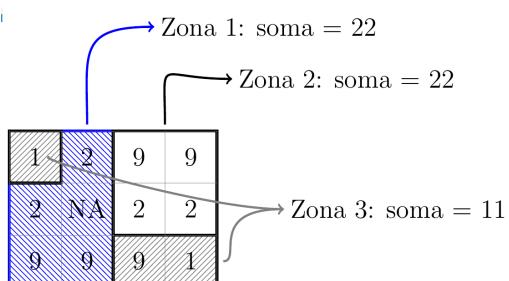


Figura 5. Soma zonal baseada no mapa r1 com 3 categorias

```
# Operações zonais
print(as.matrix(r1))

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

print(as.matrix(r2))

##      [,1] [,2] [,3] [,4]
## [1,]     1     2     9     9
## [2,]     2    NA     2     2
## [3,]     9     9     9     1

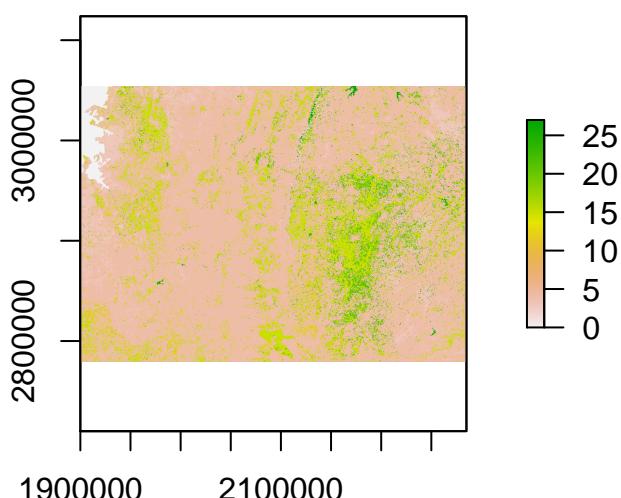
zonal(r2,r1,"sum") # mapa de valores, mapa de zonas

##      zone sum
## [1,]     1   22
## [2,]     2   22
## [3,]     3   11
```

6.2 Análise espacial no formato raster

Na presente seção, vamos aplicar alguns dos métodos anteriormente apresentados para elaborar um mapa de áreas desmatadas, baseado em mapas de cobertura e uso do solo de uma região do nordeste brasileiro, nos anos 2004 e 2014¹. Para tanto, vamos importar e cortar os mapas que se encontram no formato *TIF* e reclassificar nas categorias florestal e não florestal. Em seguida, combinamos os mapas de 2002 e 2012 para gerar um mapa das áreas desmatadas durante este período.

```
# Determina a rota do espaço de trabalho
setwd("/home/jf/dados")
# Importa mapas no formato tiff (mapas modificados do MapBiomass)
mapa2002 <- raster("mapa2002.tif")
plot(mapa2002, axes = T)
```

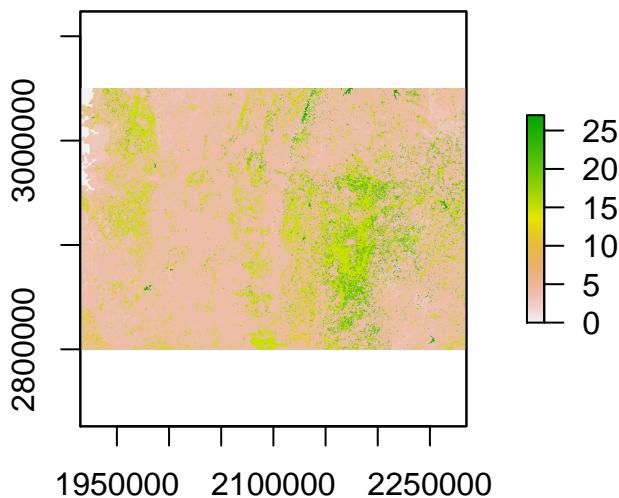


¹Mapas obtidos do projeto MapBiomass <http://mapbiomas.org/>

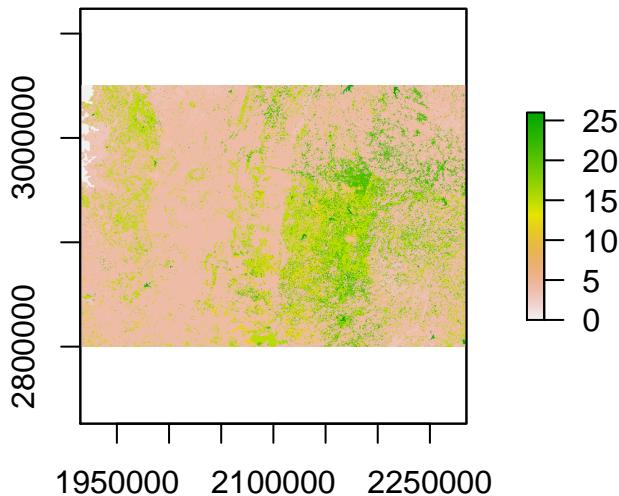
```
extent(mapa2002) # Extensão original do mapa

## class      : Extent
## xmin      : 1900000
## xmax      : 2285000
## ymin      : 2780000
## ymax      : 3054000

# Define uma janela para cortar
extensao_corte <- extent(c(1915000,2285000,2800000,3050000))
mapa2002 <- crop(mapa2002,extensao_corte)
plot(mapa2002, axes = T)
```



```
# Para o mapa 2012 se importa e corta o mapa com funções aninhadas
mapa2012 <- crop(raster("mapa2012.tif"),extensao_corte)
plot(mapa2012, axes = T)
```



```
res(mapa2002) # Resolução espacial 250 m por 250 m

## [1] 250 250
```

Como podemos observar na tabela 2, as classes florestais estão todas codificadas com valores entre 1 e 9. Portanto, a reclassificação para elaborar um mapa binário floresta / não floresta é obtida através da utilização de uma tabela com dois intervalos.

Tabela 2. Códigos da legenda para os valores de pixel na Coleção 3 do MapBiomas

Classe	ID	Classe	ID
1. Floresta	1	3.1.2. Pastagem Plantada	17
1.1. Floresta Natural	2	3.2. Agricultura	18
1.1.1. Formação Florestal	3	3.3. Mosaico de Agricultura e Pastagem	21
1.1.2. Formação Savânica	4	4. Área não vegetada	22
1.1.3. Mangue	5	4.1. Praia e Duna	23
1.2. Floresta Plantada	9	4.2. Infraestrutura Urbana	24
2. Formação Natural não Florestal	10	4.3. Afloramento Rochoso	29
2.1. Área Úmida Natural não Florestal	11	4.4. Mineração	30
2.2. Formação Campestre	12	4.5. Outra Área não Vegetada	25
2.3. Apicum	32	5. Corpos D'água	26
2.4. Outra Formação Natural não Florestal	13	5.1 Rio Lago e Oceano	33
3. Agropecuária	14	5.2 Aquicultura	31
3.1. Pastagem	15	6. Não observado	27
3.1.1. Pastagem Natural	16		

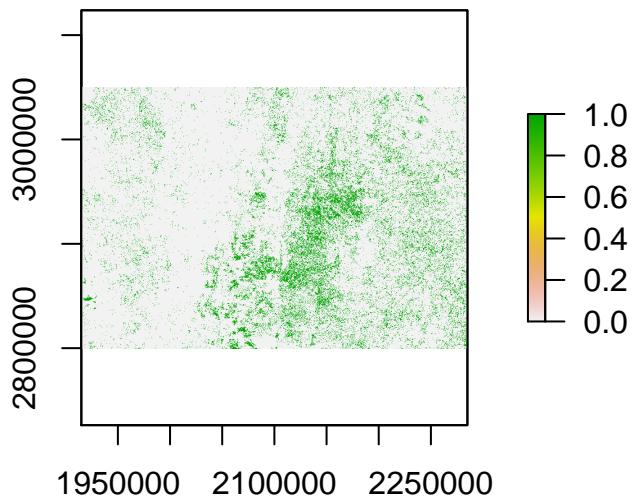
```
# Reclassificação para elaborar um mapa binário floresta / não floresta
# 1-9 são classes florestais, as demais não florestais
# Matriz tabela para reclassificar
# Por padrão, o valor inferior do intervalo não está incluído
m <- c(0, 9, 1, 10, 29, 0)
rclmat <- matrix(m, ncol=3, byrow=TRUE)
print(rclmat)

##      [,1] [,2] [,3]
## [1,]    0    9    1
## [2,]   10   29    0

# Mapas binários (Floresta 1, não floresta 0)
F2002 <- reclassify(mapa2002, rclmat)
F2012 <- reclassify(mapa2012, rclmat)
```

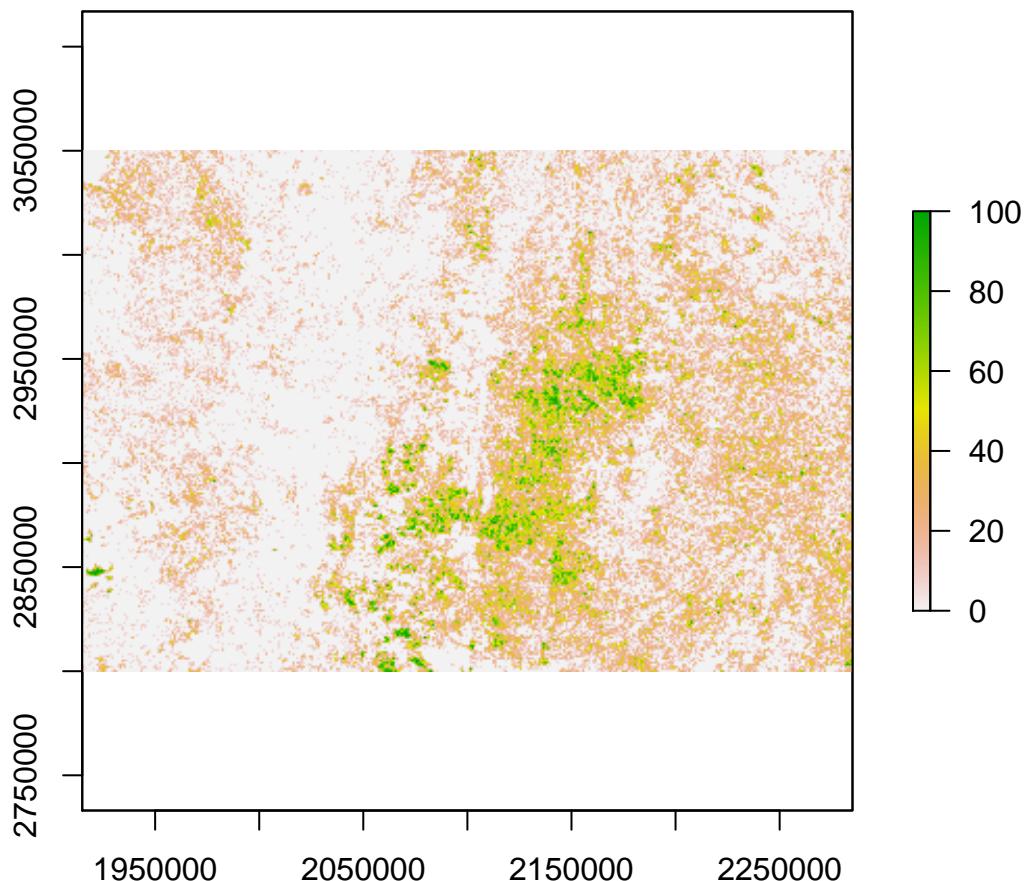
Uma simples operação lógica de álgebra de mapas, permite obter as áreas desmatadas: células com floresta em 2002 (codificadas 1 no mapa binário F2002) já sem cobertura florestal em 2012 (codificadas 0 no mapa binário F2012). A função `ifelse()`, que contém uma condição e os valores de saída no caso da condição se cumprir ou não: `ifelse(x == 1 & y == 0, 1, 0)`, permite obter um mapa binário das áreas desmatadas entre 2002 e 2012 utilizando-se a função `overlay()`.

```
## Algebra de mapas: Desmatamento
# Floresta em 2002 que já não é floresta em 2012
def <- overlay(F2002, F2012, fun = function(x,y) {ifelse( x == 1 & y == 0, 1, 0) })
plot(def)
```



Por último, geramos um mapa de taxa de desmatamento (área desmatada por km^2) agregando as células em grupos de 4×4 pixels (4×4 células de $250 \text{ m} \times 250 \text{ m}$ é equivalente a um km^2). Devido ao fato da resolução espacial do mapa binário de desmatamento ser de 250 m , as células correspondem a 6.25 ha . Assim, é necessário multiplicar a soma das células por 6.25 , para obtenção de um mapa indicativo da quantidade de hectares desmatados por km^2 .

```
## Agrega pixels por "pacotes" de 4 x 4 pixels (soma). Cada pixel é de 6.25 ha
# O mapa indica o número de ha desmatada por km2
def2 <- aggregate(def, fact=4, fun=sum) * 6.25
plot(def2)
```



```
dim(def2)

## [1] 250 370    1

res(def2) # a resolução é 1 km por 1 km

## [1] 1000 1000

# Salva o raster como def.tif
writeRaster(def2, filename="def.tif", format="GTiff", datatype="INT2U", overwrite=TRUE)
```

7. Análise geoestatística: Identificação de *hot spots*

7.1 Método de Getis Ord

Algumas análises geoestatísticas buscam examinar os padrões de distribuição espacial de variáveis quantitativas. Por exemplo, pontos quentes (*hot spots*) e pontos frios (*cold spots*) são áreas que apresentam uma agregação espacial de valores altos ou baixos da variável considerada, que não pode ser explicada por uma distribuição aleatória dos valores. Neste capítulo, identificaremos zonas com alta (*hot spots*) ou baixa (*cold spots*) incidência de desmatamento, nas quais ocorrem mais ou menos desmatamentos do que seria esperado se as mudanças fossem distribuídas aleatoriamente no território.

Para esta análise, se utilizará o índice estatístico Getis-Ord Gi* (Getis & Ord, 1992). Este índice avalia a agregação espacial dos dados através da comparação das médias locais (um sítio e seus arredores) e globais (para toda a área de estudo). Essa comparação é baseada no cálculo do valor padronizado z ou z-score (equações 7.1 a 7.3). Os valores de z (desvios padrão) e de p indicam se as características são agregadas estatisticamente em uma determinada distância. Um valor de z maior que 1,96 ou menor que -1,96 significa que a variável avaliada mostra um ponto quente ou um ponto frio estatisticamente significativo em um determinado nível de significância p (geralmente 0,05). No nosso caso de estudo, o índice avalia se, numa pequena região, se agregam mais ou menos áreas desmatadas do que se esperaria, caso as áreas desmatadas apresentem-se distribuídas de maneira aleatória.

$$G_i^* = \frac{\sum_{j=1}^n w_{i,j}x_j - \bar{X} \sum_{j=1}^n w_{i,j}}{S \sqrt{\frac{[n \sum_{j=1}^n w_{i,j}^2 - (\sum_{j=1}^n w_{i,j})^2]}{n-1}}} \quad (7.1)$$

$$\bar{X} = \frac{\sum_{j=1}^n x_i}{n} \quad (7.2)$$

$$S = \sqrt{\frac{\sum_{j=1}^n x_i^2}{n} - (\bar{X})^2} \quad (7.3)$$

onde x_j é o valor da variável considerada (nesse caso o índice de desmatamento), $w_{i,j}$ é o peso de ponderação espacial entre o sítio i e j, n é o número total de sítios. \bar{X} e S são respectivamente a média e o desvio padrão do valor da variável para toda a área de estudo (estatística "global"). G_i^* é, portanto, baseado na diferença entre a média dos valores da variável do ponto e seus vizinhos, $\sum_{j=1}^n w_{i,j}x_j$, e o valor que poderia ser esperado se a distribuição fosse homogênea (valor baseado na média global, $\bar{X} \sum_{j=1}^n w_{i,j}$). Essa diferença é normalizada para avaliar se excede a variabilidade que poderia ser atribuída a efeitos aleatórios (valores de z). Estes cálculos são feitos com o pacote spdep (Bivand & Piras, 2015).

7.2 Aplicação para a detecção de áreas com altas taxas de desmatamento

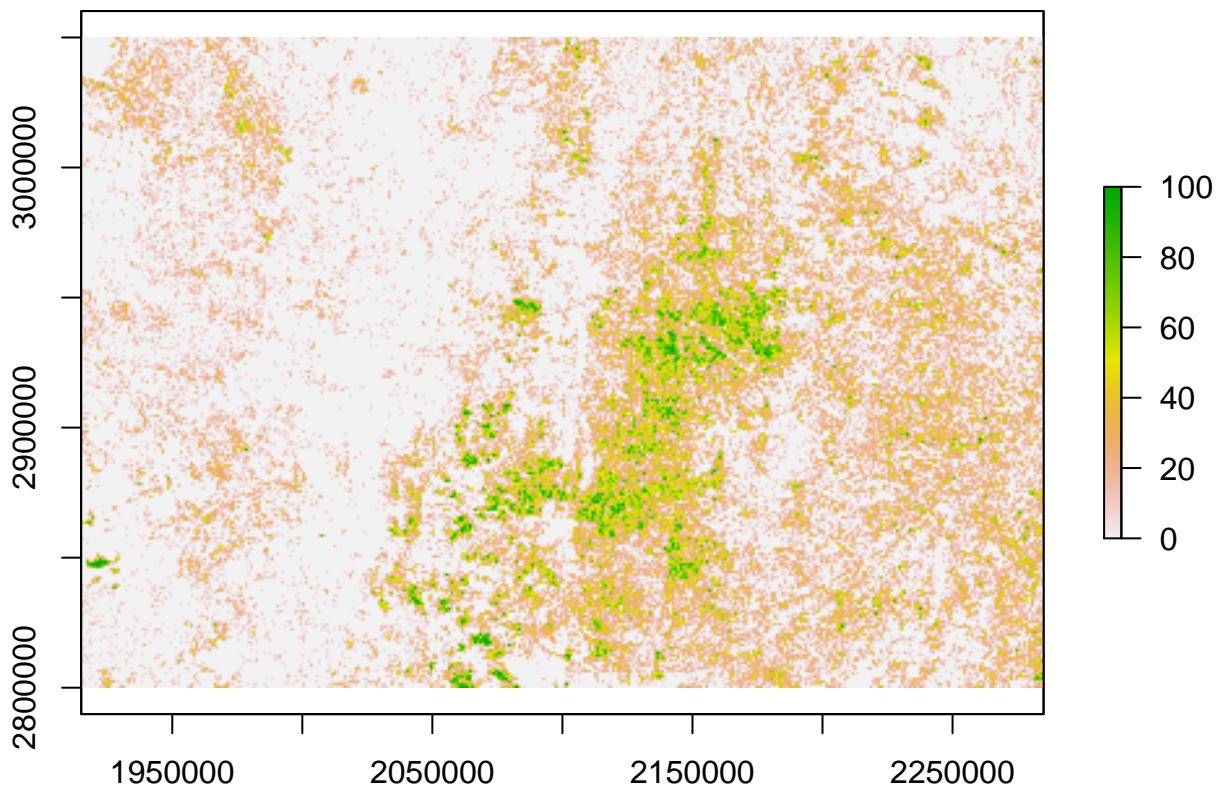
Vamos usar o mapa de desmatamento elaborado no capítulo 6 (seção 6.2). Este mapa tem uma resolução espacial de um quilômetro e representa a área desmatada (em hectare - ha) durante 2002-2012. Em um primeiro passo, o mapa de desmatamento é importado no formato *raster* e convertido em um arquivo de ponto (baseado no centro

de cada célula) usando a função ***rasterToPoints()*** (Veja scrip7.R). Uma tabela é gerada das coordenadas de cada ponto (tabela xy) e outra do valor da taxa de desmatamento (tabela de taxas). Com base na tabela de coordenadas e na distância máxima (5000 m), os pontos vizinhos de cada ponto dentro dessa distância são determinados usando a função ***dnearest()***.

```
library(raster)
library (rgdal)
library(maptools)
# Instalar spdep se necessário:
# install.packages("spdep", dependencies=TRUE)
library(spdep)

# Determina a rota do espaço de trabalho
setwd("/home/jf/dados")

# Desmata indica o número de ha desmatadas m em quadrantes de 1 x 1 km
desmata <- raster("def.tif")
plot(desmata)
```



```
dim(desmata)
## [1] 250 370    1

res(desmata)
## [1] 1000 1000

# Transforma o raster em pontos
desma_pts <- rasterToPoints(desmata)
head(desma_pts)
```

```

##           x      y def
## [1,] 1915500 3049500  0
## [2,] 1916500 3049500  0
## [3,] 1917500 3049500  0
## [4,] 1918500 3049500  0
## [5,] 1919500 3049500  0
## [6,] 1920500 3049500  0

summary(desma_pts)

##           x                  y                  def
## Min.   :1915500   Min.   :2800500   Min.   : 0.00
## 1st Qu.:2007500   1st Qu.:2862500   1st Qu.: 0.00
## Median :2100000   Median :2925000   Median : 0.00
## Mean    :2100000   Mean   :2925000   Mean   : 10.27
## 3rd Qu.:2192500   3rd Qu.:2987500   3rd Qu.: 12.00
## Max.    :2284500   Max.   :3049500   Max.   :100.00

class(desma_pts)

## [1] "matrix"

# Distância
dist_G <- 5000

# Gera matriz com 2 colunas para coordenadas (x e y)
xy <- as.matrix(desma_pts[,1:2])
head(xy)

##           x      y
## [1,] 1915500 3049500
## [2,] 1916500 3049500
## [3,] 1917500 3049500
## [4,] 1918500 3049500
## [5,] 1919500 3049500
## [6,] 1920500 3049500

taxa <- desma_pts[,3]

# Identifica vizinhos de cada ponto entre distâncias de 0 a dist_G
vizinho <- dnearest(xy, 0, dist_G)
print(vizinho)

## Neighbour list object:
## Number of regions: 92500
## Number of nonzero links: 7292456
## Percentage nonzero weights: 0.08522958
## Average number of links: 78.83736

```

A função ***nb2listw()*** suplementa uma lista de vizinhos com pesos espaciais para o esquema de codificação escolhido. Neste caso, a opção binária (**Style="B"**) foi usada, sendo o peso, portanto, binário. Os pontos distantes de menos de 5000 m (**dist_G**) são considerados vizinhos (peso = 1). Finalmente, o valor do índice

estatístico Getis-Ord Gi* (valor padronizado z) para cada ponto é calculado com a função *localG()* (equação 7.1).

```
Getis<- localG(taxa, nb2listw(vizinho, style="B"))
class(Getis)

## [1] "localG"

head(Getis)

## [1] -3.159619 -3.461282 -3.738713 -3.996961 -4.192139 -4.239532

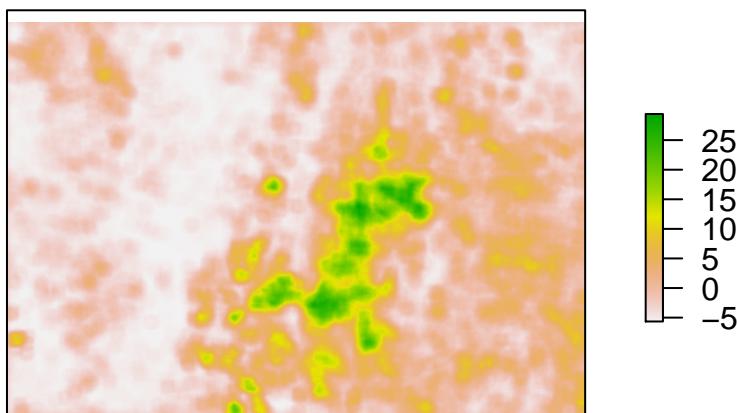
z <- as.numeric(Getis)
```

O resultado é um vetor que contém os valores z de cada ponto. Para visualizar a distribuição espacial de z, temos que criar um objeto espacial. Vamos primeiro criar uma tabela *Dataframe* juntando as coordenadas e os valores de z. Então, esta tabela é transformada em um objeto *SpatialPointsDataFrame* e *RasterLayer* no pacote **raster**. O mapa final mostra, em verde, as áreas onde a agregação espacial de manchas (patches) de desmatamento excede a variabilidade que pode ser atribuída a efeitos aleatórios (áreas *Hot spots* de desmatamento).

```
# Gera uma tabela com x, y e z
spz <- as.data.frame(cbind(xy,z))
head(spz)

##           x       y       z
## 1 1915500 3049500 -3.159619
## 2 1916500 3049500 -3.461282
## 3 1917500 3049500 -3.738713
## 4 1918500 3049500 -3.996961
## 5 1919500 3049500 -4.192139
## 6 1920500 3049500 -4.239532

# Cria um objeto "spatial points data frame"
coordinates(spz) <- ~ x + y
# converte a SpatialPixelsDataFrame
gridded(spz) <- TRUE
# Converte em raster
rasterz <- raster(spz)
plot(rasterz, axes=F)
```



8. Elaboração de mapas

R é conhecido pela produção de gráficos bem elaborados e vem agregando, cada vez mais, ferramentas que viabilizam a feitura de mapas. No presente capítulo, vamos começar apresentando umas funções para salvar os gráficos em arquivos de imagem e para manipular as cores. Em seguida, elaboraremos um mapa muito simples para apresentar a densidade populacional dos estados (script8.R e figura 6). Para esse fim, utilizaremos o mapa das entidades federativas do Brasil, através do qual calculamos anteriormente a densidade populacional de cada estado (capítulo 5). Esta informação se encontra na coluna "dens" da tabela de atributos.

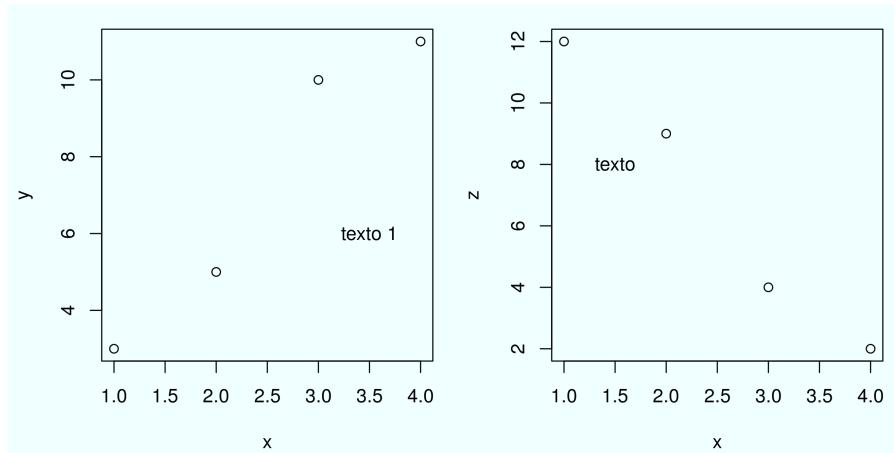
8.1 Elaboração de arquivos de imagens

No R, é possível salvar os gráficos elaborados em vários formatos, como bmp, jpeg, png, postscript, tiff e pdf. Há funções para cada formato de arquivo. Para criar um arquivo png, por exemplo, existe a função *png()* no qual se deve especificar pelo menos o nome do arquivo de saída (*png(filename="saída.png")*). O R enviará todos os resultados de comandos gráficos para este arquivo, que é fechado com a função *dev.off()*.

No exemplo que segue, se elabora um diagrama de dispersão no qual a função *text()* permite colocar texto com base nas coordenadas do gráfico. Nesse caso, usamos algumas opções adicionais que permitem controlar o tamanho (height, width, units), a resolução (res) e a cor de fundo (bg) da figura.

É possível modificar as margens por padrão de um gráfico em R chamando a função *par()* com o argumento *mar* (para margem!). Por exemplo, *par(mar = c(4.1,4.1,1.1,1.1))* define as margens inferior, esquerda, superior e direita, respectivamente, da região de plotagem em número de linhas de texto. Por outro lado, a função *par()* com a opção *mfrow=c(nfila,ncol)* permite combinar os dois *plots* numa fila.

```
png(filename="saída.png",width=20,height=10,units="cm",res=300,bg="azure")
par(mar=c(4.1,4.1,1.1,1.1))
par(mfrow=c(1,2)) # plots em uma fila, 2 colunas
x <- c(1,2,3,4)
y <- c(3,5,10,11)
z <- c(12,9,4,2)
plot(x,y)
# Põe texto nas coordenadas (3.5, 6)
text(3.5,6,"texto 1")
plot(x,z)
# Põe texto nas coordenadas (1.5, 8)
text(1.5,8,"texto")
dev.off()
```



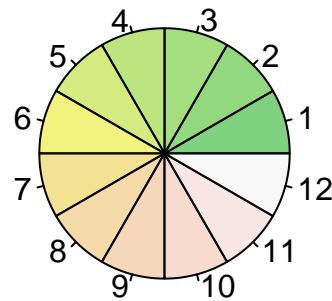
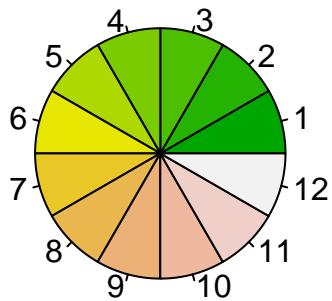
8.2 Gerenciamento de cores

R tem várias opções para o manuseio de paletas de cores e de tons de cinza. Por exemplo, a função `terrain.colors()` viabiliza a criação de um vetor de cores contíguas. A opção `alpha` permite controlar a transparência.

```
# Manuseio de paletas de cores
# Cria uma escala de 12 cores
cores <- terrain.colors(12, alpha = 1)
cores2 <- terrain.colors(12, alpha = 0.5) # com transparência
print(cores) # Vetor de códigos de cores

## [1] "#00A600FF" "#24B300FF" "#4CBF00FF" "#7ACCO0FF" "#ADD900FF"
## [6] "#E6E600FF" "#E8C727FF" "#EAB64EFF" "#ECB176FF" "#EEB99FFF"
## [11] "#FOFCFC8FF" "#F2F2F2FF"

# Elabora gráfico de pizza com as cores
par(mfrow=c(1,2))
pie(rep(1, 12), col = cores)
pie(rep(1, 12), col = cores2)
```



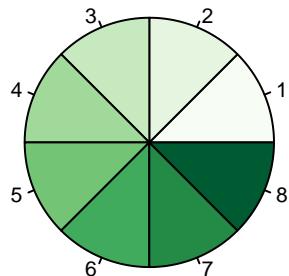
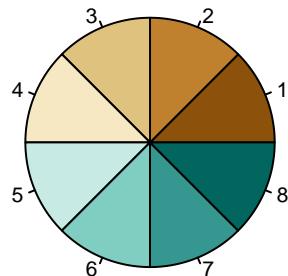
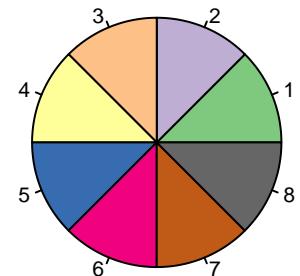
As funções `gray.colors()` (para tons de cinza) e `rainbow()` funcionam de forma semelhante.

RColorBrewer é um pacote R que usa o trabalho de <http://colorbrewer2.org/> para criar paletas de cores para R. A função `brewer.pal()` viabiliza a criação de três tipos de paletas (sequenciais, divergentes e qualitativas) caracterizadas da seguinte forma:

- As paletas sequenciais são adequadas para dados ordenados que progridem de baixo para alto, com cores claras para valores baixos e cores escuras para valores altos.
- As paletas divergentes colocam ênfase igual nos valores críticos e extremos intermediários em ambas as extremidades do intervalo de dados. A classe crítica ou quebra no meio da legenda é enfatizada com cores claras e os baixos e altos extremos são enfatizados com cores escuras que possuem tons contrastantes.

- As paletas qualitativas não indicam diferenças de magnitude entre as classes de legenda, e as matizes são usadas para criar diferenças visuais entre as classes. Os esquemas qualitativos são mais adequados para representar dados nominais ou categóricos.

```
# Manuseio de paletas de cores
library(RColorBrewer)
sequencial <- brewer.pal(8,"Greens")
divergente <- brewer.pal(8,"BrBG")
qualitativa <- brewer.pal(8,"Accent")
# Elabora gráfico de pizza com as 3 paletas
par(mfrow=c(1,3))
pie(rep(1, 8), col = sequencial,main="sequencial")
pie(rep(1, 8), col = divergente,main="divergente")
pie(rep(1, 8), col = qualitativa,main="qualitativa")
```

sequencial**divergente****qualitativa**

8.3 Elaboração do mapa de densidade

Importamos o mapa de densidade de população em formato *shape* e visualizamos os valores do campo "dens".

```
library(maptools)
library(RColorBrewer)
library(classInt)
library(sf)
# Determina a rota do espaço de trabalho
setwd("/home/jf/dados")
bra <- st_read("br_a.shp")

## Reading layer `br_a` from data source `/home/jf/dados/br_a.shp` using driver
`ESRI Shapefile'
## Simple feature collection with 27 features and 10 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:            xmin: -1523283 ymin: -215522 xmax: 3071472 ymax: 4209948
## epsg (SRID):    NA
## proj4string:    +proj=aea +lat_1=-5 +lat_2=-42 +lat_0=-32 +lon_0=-60 +x_0=0
+y_0=0 +ellps=aust_SA +units=m +no_defs

# Nomes das columnas da tabela de atributos
names(bra)

## [1] "ID"          "NM_ESTADO"   "NM_REGIAO"   "AreaEdo"     "Sigla"
```

```

## [6] "Estado"     "Pop"        "PIB"        "PIBC"       "dens"
## [11] "geometry"

# Valores de densidade dos estados
print(bra$dens)

## [1]  4.464901 112.349456  4.681732  2.232554 24.828235 56.728233
## [7] 443.416237 76.203550 17.653025 19.791826 3.358523  6.858072
## [13] 33.409093  6.080356 66.704841 52.378886 89.619153 12.397730
## [19] 365.313046 59.989544 39.794615  6.567991 2.011690 65.279800
## [25] 166.188350 94.364828  4.981419

```

Os valores de densidade variam de 2 (dois) a mais de 443 habitantes por km². Vamos representar a densidade através de uma escala de sete cores, abrangendo do amarelo ao vermelho (o valor sete é arbitrário). Assim, geraremos um conjunto de sete cores utilizando o pacote **RColorBrewer** (Função *brewer.pal()*) e reclassificaremos os valores de densidade em sete intervalos. Esta reclassificação é feita determinando-se os limiares entre as sete categorias de forma automática, aplicando a função *classIntervals()* do pacote **classInt**. Neste caso, usamos o método dos quantis, ou seja, de modo a se determinar os limites para obter o mesmo número de observações (estados) em cada intervalo. Finalmente, calculamos um vetor que indica o código da cor que corresponde a cada estado.

```

## Paleta de cores e categorização da variável densidade
numclass <- 7 # número de categorias
# Gera 7 níveis de cores na escala entre amarelo e vermelho
cores <- brewer.pal(numclass, "YlOrRd")
print(cores) # Códigos das cores

## [1] "#FFFFB2" "#FED976" "#FEB24C" "#FD8D3C" "#FC4E2A" "#E31A1C"
## [7] "#B10026"

# Determina os limiares entre categorias (intervalos, método quantil)
var <- bra$dens
brks<-classIntervals(var, n=numclass, style="quantile")
brks <- brks$brks
print(brks)

## [1]  2.011690  4.619780  6.692311 20.511313 50.581133 66.094109
## [7] 99.503293 443.416237

class(brks)

## [1] "numeric"

class(bra$dens)

## [1] "numeric"

# Determina a cor correspondente para cada valor de densidade
codigos_num <- findInterval(var, brks, all.inside=TRUE)
print(codigos_num) # indica a classe (intervalo) de densidade

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

```

```

codigos_cor <- cores[codigos_num] # código da cor de cada classe
print(codigos_cor)

## [1] "#FFFFB2" "#B10026" "#FED976" "#FFFFB2" "#FD8D3C" "#FC4E2A"
## [7] "#B10026" "#E31A1C" "#FEB24C" "#FEB24C" "#FFFFB2" "#FEB24C"
## [13] "#FD8D3C" "#FED976" "#E31A1C" "#FC4E2A" "#E31A1C" "#FEB24C"
## [19] "#B10026" "#FC4E2A" "#FD8D3C" "#FED976" "#FFFFB2" "#FC4E2A"
## [25] "#B10026" "#E31A1C" "#FED976"

```

Após a obtenção dos intervalos de densidade e a paleta de cores, vamos criar o mapa que será salvo em formato png com o nome "brasil_densidade.png".

O comando `plot(bra["dens"], col=codigos_cor, axes=T, main="")` ploteia o mapa com base na densidade associando um código de cor da paleta a cada classe de densidade. As funções *SpatialPolygonsRescale()*, *layout.north.arrow()* e *layout.scale.bar()* permitem criar o símbolo da bússola, que indica a orientação cartográfica e uma escala gráfica, conservando a correta relação de proporcionalidade.

```

png(file="brasil_densidade.png", height=20, width=20, units="cm", res=400)
par(mar=c(10.1,4.1,4.1,10.1))
plot(bra["dens"], col=codigos_cor, axes=T, main="")
## Põe um título
title(main="Densidade populacional", cex.main=1.5)
# Põe texto em certas coordenadas
text(2500000,1000000,"Atlântico", pos = 1, cex = 1.2)
# Põe Norte
SpatialPolygonsRescale(layout.north.arrow(1), offset= c(2600000,3500000),
                       scale = 500000, plot.grid=F)
# Legenda
# Gera texto da legenda (intervalos de valores de densidade)
intervalos <- leglabs(as.vector(round(brks, digit = 0)),
                      under="Menos de", over="Mais de")
class(intervalos)

## [1] "character"

legend("bottomleft", inset=.05, title=expression("Densidade (hb/km"^-2*)) ,
       legend=intervalos, fill=cores, horiz=FALSE, box.col = NA, cex = 1.2)
# Escala gráfica
SpatialPolygonsRescale(layout.scale.bar(), offset= c(1900000,0),
                       scale= 1000000, fill=c("transparent", "black"), plot.grid= F)
text(1900000,150000,"0"); text(3000000,150000,"1000 km") # texto da escala
par(mar=c(5.1,4.1,4.1,2.1))
dev.off()

```

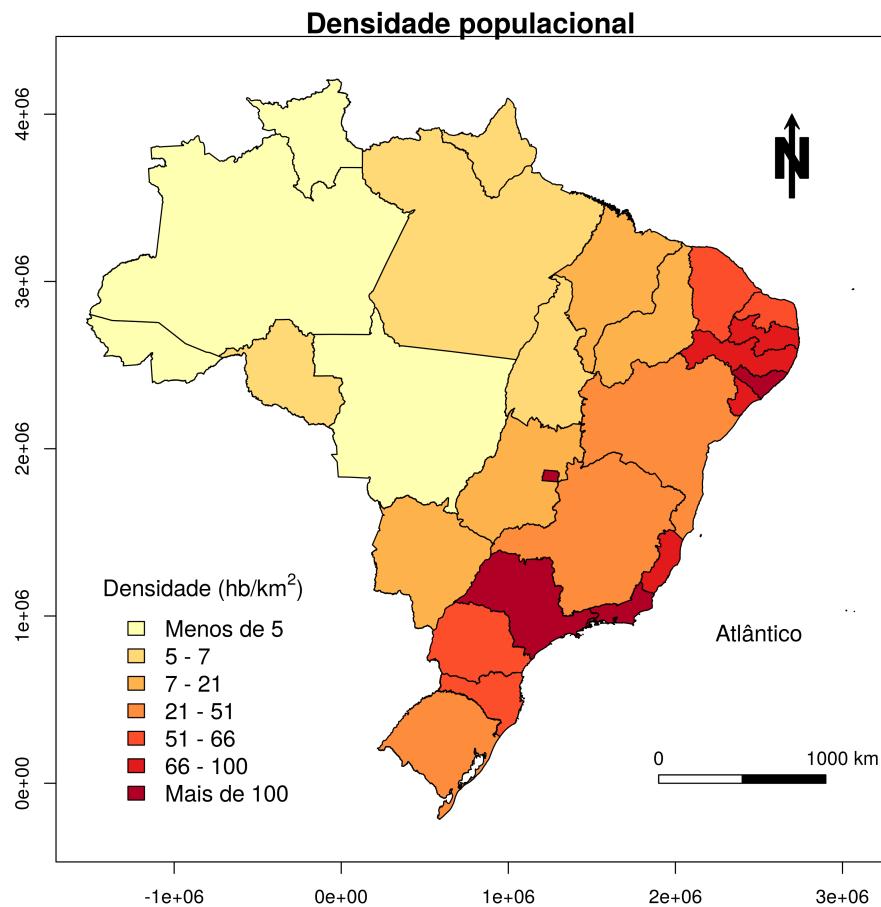


Figura 6. Mapa de densidade populacional

8.4 Elaboração de mapas com base em dados raster

Já em um segundo exemplo, elaboraremos um mapa com base no arquivo *raster* (*dem_albers.tif*) utilizando um dos conjuntos de cor disponível no R, porém sem fazer uso do **RColorBrewer**.

```
# Importa e corta o MDE
extension <- extent(1880000, 2095000, 1900000, 2059000)
dem <- crop(raster("dem_albers.tif"), extension)
minValue(dem); maxValue(dem)

## [1] 140
## [1] 1877

# Cria uma escada de cores
cores <- terrain.colors(7, alpha = 1)
print(cores)

## [1] "#00A600FF" "#63C600FF" "#E6E600FF" "#E9BD3AFF" "#ECB176FF"
## [6] "#EFC2B3FF" "#F2F2F2FF"

# Elabora o mapa
png(file="dem.png", height=15.65, width=20, units="cm", res=400)
plot(dem, col=cores, breaks = c(100, 200, 300, 500, 700, 1000, 1500, 2000),
     main = "Altitude")
SpatialPolygonsRescale(layout.scale.bar(), offset= c(1885000, 1907000), scale= 40000,
```

```

fill=c("transparent", "black"), plot.grid= F)
text(1885000,1904000,"0"); text(1885000+45000,1904000,"40 km")
dev.off()

```

No exemplo abaixo, mostramos os passos para gerar uma imagem de sombreado (exibida em tons de cinza) com base no modelo digital de elevação. Em seguida, adicionamos o raster de elevação com transparência (opção **alpha**). Ver as figuras 7 e 8.

```

# Prepara o relevo sombreado
decliv <- terrain(dem, opt = "slope")
orient <- terrain(dem, opt = "aspect")
sombra <- hillShade(decliv, orient, 40, 270)

# Plota o sombreado e a elevação com transparência
png(file="sombreado.png", height=15.65, width=20, units="cm", res=400)
plot(sombra, col = grey(0:100/100), legend = FALSE,
     main = "Altitude com relevo sombreado")
plot(dem, col = rainbow(25, alpha = 0.3), add = TRUE)
SpatialPolygonsRescale(layout.scale.bar(), offset= c(1885000,1907000), scale= 40000,
                       fill=c("transparent", "black"), plot.grid= F)
text(1885000,1904000,"0"); text(1885000+45000,1904000,"40 km")
dev.off()

```

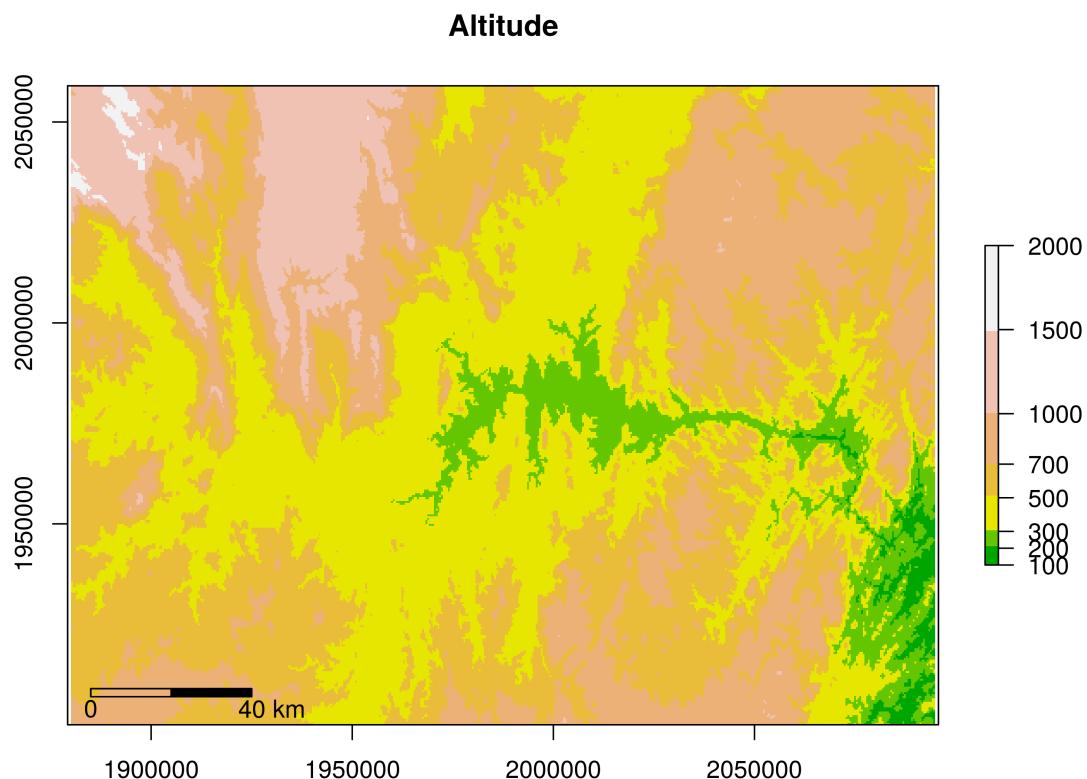


Figura 7. Mapa de elevação

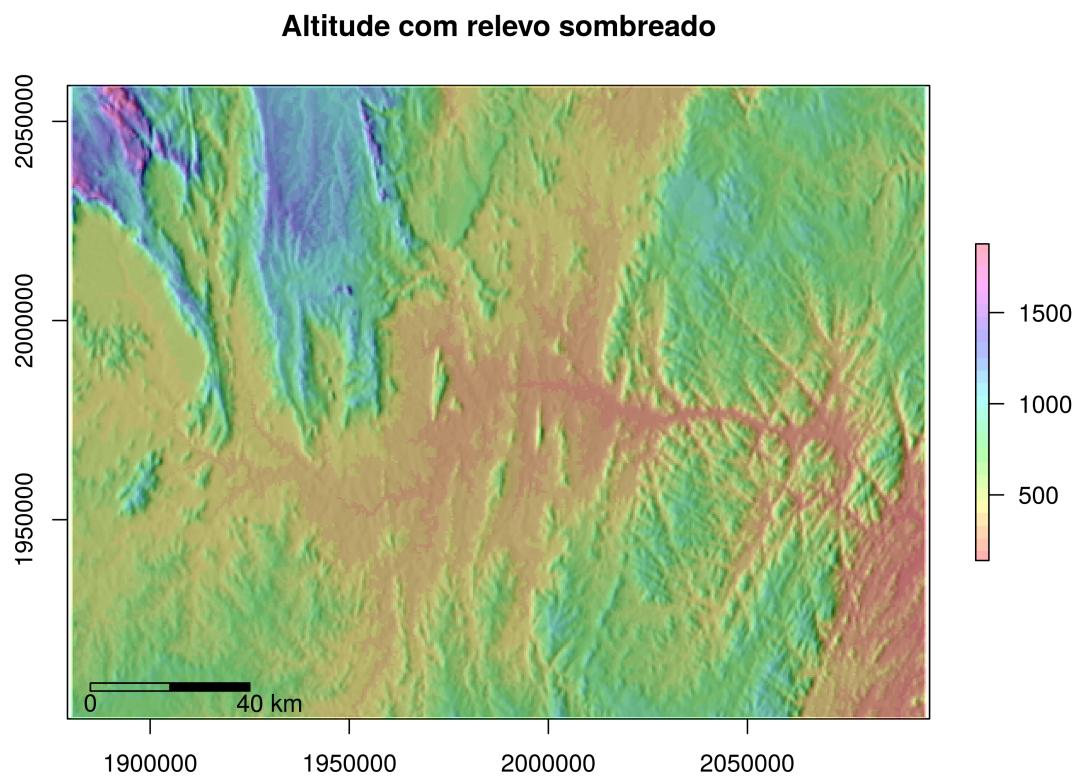


Figura 8. Mapa de sombreado

Existem muitas outras opções para a elaboração de mapas no R, dentre os quais é possível mencionar a função **spplot** de **sp** e os pacotes **Choroplethr** (Lamstein & Johnson, 2018), **Prettymapr** (Dunnington, 2017) e **tmap** (Tennekes *et al.*, 2017). Cabe mencionar também **RgoogleMaps** (Loecher, 2016) e **Rosm** (Dunnington & Giraud, 2017), que permitem utilizar-se como fundo, os mapas do GoogleMap, Open Street e Bing.

9. Interagindo R com QGIS e Dinamica EGO

Vários programas computacionais de manejo de informação espacial permitem a interação com R. Neste capítulo vamos revisar os mecanismos implementados em dois deles: Sistema de Informação Geográfica QGIS e a plataforma de modelagem espacial Dinamica EGO.

9.1 Sistema de Informação Geográfica QGIS

O programa Q-GIS é um sistema de informação geográfica de código aberto que permite o manejo de dados em formato raster e vetorial através das bibliotecas GDAL e OGR. O Q-GIS pode executar scripts de R utilizando Python, e os resultados podem ser incorporados como camada de informação. Para tal, se deve configurar o programa em Processar > Opções>_Provedores>_Rscripts. Aparecerá uma janela, na qual se deve preencher os quadros de Ativar e Usar a versão 64bit indicando a rota (*path*) do executor do R. Aperte então o OK, feche o Q-GIS e abra-o novamente. Para escrever um novo script, selecione Processar>_caixadeferramentas>_Rscripts>_Tools>_CreatenewRscript.

Ao início do script precisamos definir os argumentos das funções de R nas linhas iniciadas com o símbolo duplo ## e que são utilizadas para criar a interface gráfica. Por exemplo, o script mostrado a seguir, permite calcular o histograma de um raster aberto no QGIS. O script gera uma interface que permite escolher o mapa raster e a rota para salvar o histograma (Figura 9).

```
##Layer = raster
##showplots
hist(as.matrix(Layer),main="Histograma",xlab="Mapa")
```

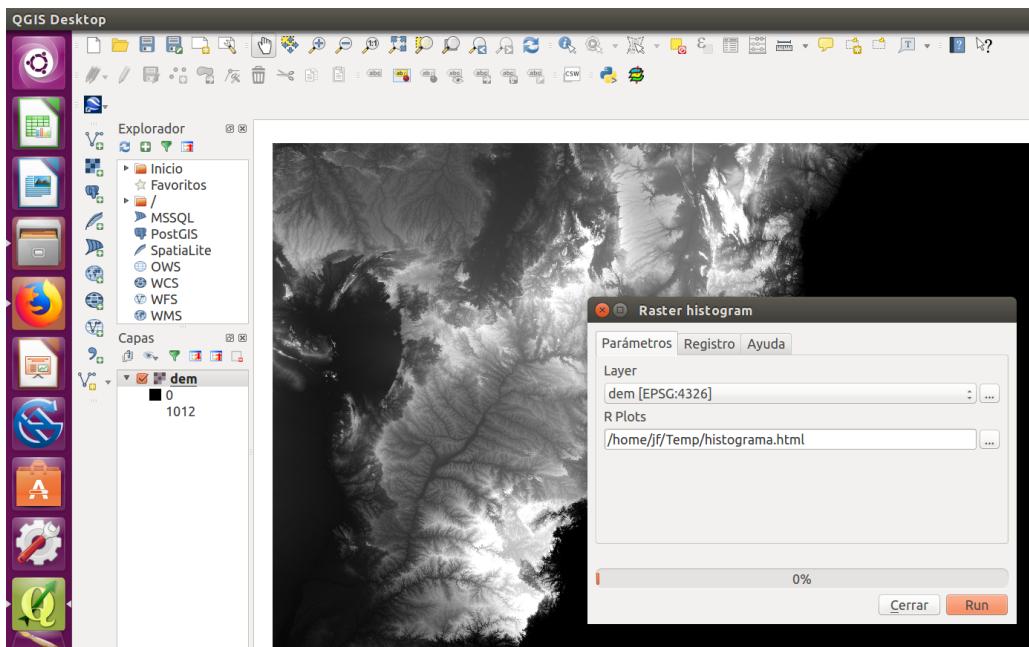


Figura 9. Interface gráfica

Neste segundo exemplo, o script tem como entradas um mapa de polígonos e um número cujo valor, *por default* ou padronização é 10, e gera pontos aleatórios distribuídos na camada de polígonos. A linha `##sp=group` permite organizar os scripts em grupos. Neste caso, o script é armazenado em uma pasta denominada `sp`. A interface gráfica associada ao script permite capturar o nome da camada de polígonos, o número de pontos e o nome e rota da cobertura de pontos aleatórios (Figura 10)

```
##polyg=vector
##numpoints=number 10
##output=output vector
##sp=group

pts <- spsample(polyg,numpoints,type="random")
output=SpatialPointsDataFrame(pts, as.data.frame(pts))
```

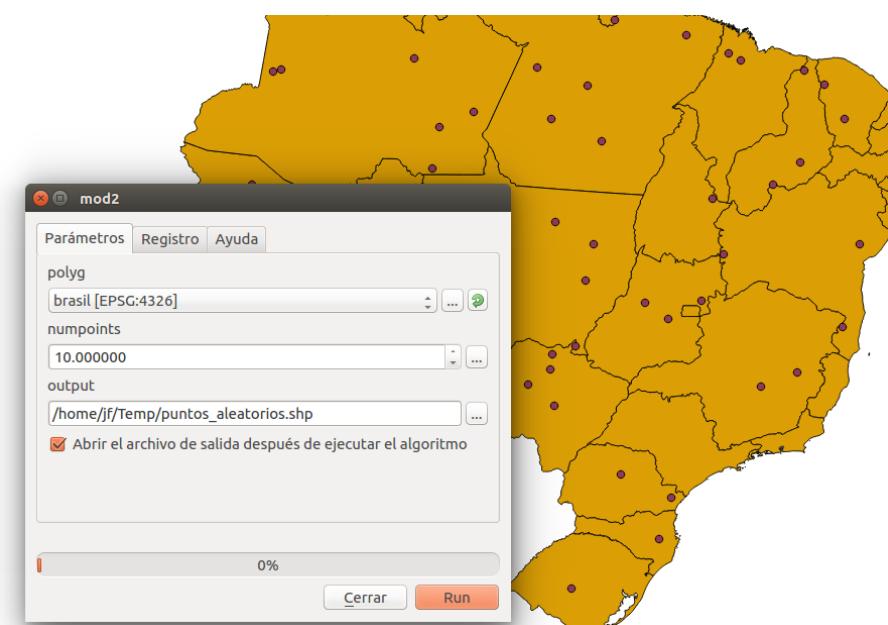


Figura 10. Interface gráfica do segundo modelo

É possível, portanto, gerar interfaces gráficas que possibilitam aos usuários não familiarizados com o R executar scripts de forma simples e interativa. Além disso, o pacote RQGIS permite realizar processamentos do QGIS a partir do R.

9.2 Plataforma de modelagem Dinamica EGO

Dinamica EGO é uma plataforma de modelagem espaço-temporal muito flexível e amigável para o usuário. Tem sido utilizada para desenvolver uma grande variedade de modelos ambientais, dentre os quais, aqueles de simulação de mudança de uso e cobertura do solo, de crescimento urbano, de incêndios, de renda, entre outros (Mas *et al.*, 2014, Rodrigues & Soares-Filho, 2018). Este programa pode ser obtido gratuitamente, estando disponível para Windows e Linux, em: <http://csr.ufmg.br/dinamica/downloads/>.

A partir da versão 4 foram desenvolvidos para o programa Dinamica, módulos especialmente desenhados para integrar scripts de R dentro do processo de modelagem (biblioteca *Integration*). Existem duas formas de obter interações entre Dinamica e R:

- Realizar una sessão compartilhada entre R e Dinamica, na qual os dois programas são executados de forma paralela e trocam informações (valores, tabelas).
- Executar um script de R dentro de Dinamica através da função *Calculate R expression*.

Nesta seção, vamos explorar esta segunda opção que é mais simples e geralmente mais eficiente que a primeira, permitindo executar um ou vários scripts de R, dentro do fluxo de operações de um modelo de Dinamica. Primeiramente, deve-se instalar alguns pacotes de R que se encontram disponíveis nos repositórios (**Rcpp**, **RcppProgress**, **rbenchmark**, **inline**).

```
install.packages(c("Rcpp", "RcppProgress", "rbenchmark", "inline"))
```

É necessário também instalar um pacote especialmente desenhado para a interação entre R e Dinamica. Este deve ser obtido na página http://www.csr.ufmg.br/dinamica/dokuwiki/lib/exe/fetch.php?media=dinamica_1.0.2.tar.gz, e instalado localmente com RStudio ou com o comando de linha (modificar a rota indicando a pasta onde se encontra o arquivo):

```
install.packages("/home/jf/Downloads/dinamica_1.0.2.tar.gz", repos=NULL, type="source")
```

Deve-se também indicar ao programa Dinamica onde encontrar o executor de R, denominado Rscript (Rscript.exe no Windows). Para isso, faz-se necessário definir a rota de acesso a este arquivo, em uma janela que se obtém em Tools > Options > Integration (ver figura 11).

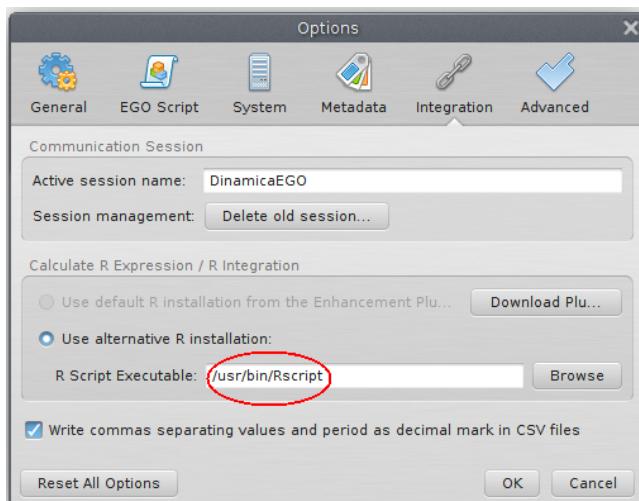


Figura 11. Definição da rota do executor de R

No primeiro exemplo que se segue, (modelo elabora_hist_R.egoml), o programa Dinamica transfere a um script de R a tabela dos valores de uma imagem, para elaborar um histograma e salvar a figura (Figura 12). Esta tabela entra no functor (operador do programa Dinamica) **Calculate R Expression** como em qualquer outro functor de Dinamica (Number table: Table #1), ou seja internamente a tabela denomina-se t1. Com o edit Functor, se pode editar o script de R. Na primeira linha, se atribui esta tabela ao objeto denominado tab. As linhas seguintes são exclusivamente de R e permitem a atribuição de nomes às colunas da tabela, assim como a realização de um gráfico, que se salva em formato png.

```
tab <- t1
colnames(tab) <- c("valores", "n")
png("histograma.png")
plot(tab$n, tabla$valores, main="Histograma banda 2", xlab="", 
     ylab="Número de pixels")
dev.off()
```

No segundo exemplo a seguir (modelo regressao_linear.egoml), vamos ver como Dinamica pode recuperar dados gerados pelo R. Neste caso, o modelo realiza uma análise da relação entre os valores de duas bandas de uma imagem de satélite (Figura 13). Para tal, empreende uma amostra aleatória de pixels, ajustando um modelo

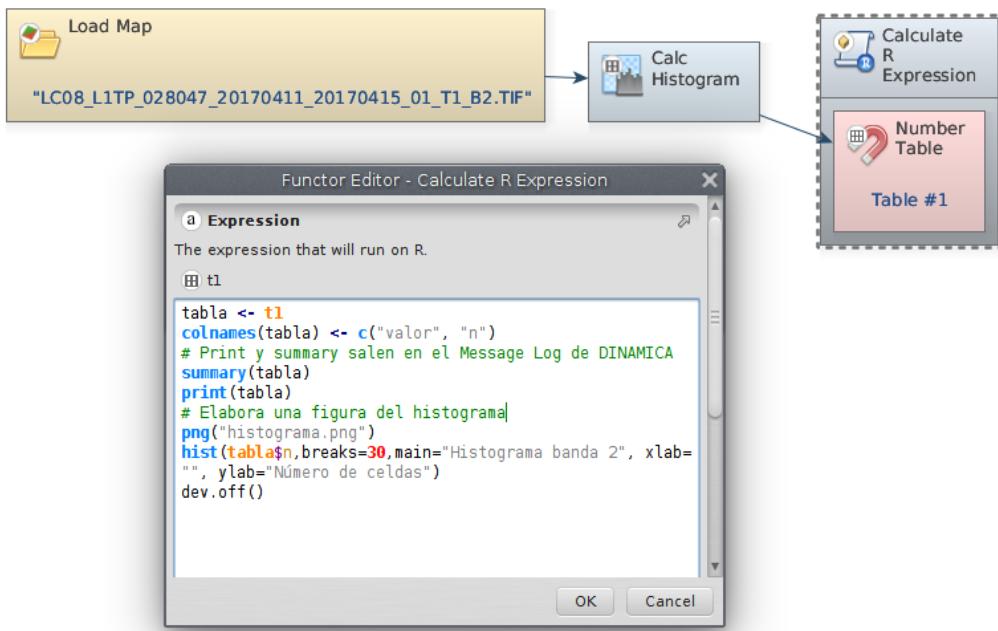


Figura 12. Modelo de Dinamica (primeiro exemplo)

de regressão para explicar os valores de uma banda com base na outra. Então calcula, pixel a pixel, os resíduais (diferenças entre os valores de uma banda calculada pelo modelo e valores observados).

O modelo de Dinamica acessa ambas as imagens e extrai o número de linhas e colunas (coordenadas), sendo que estes valores fazem parte do primeiro script de R, juntamente com o tamanho amostral desejado. No R a função ***runif()*** gera valores aleatórios dentro de um certo intervalo. ***Floor()*** possibilita o arredondamento dos valores gerados a números inteiros. Estas coordenadas aleatórias são agrupadas em uma tabela *dataframe*, exportadas ao Dinamica pela função ***outputTable()*** e ao functor de Dinamica ***Extract Struct Table***.

```

# Recebe dados de DINAMICA
numptos <- v1
numlin <- v2
numcol <- v3

# Gera coordenadas aleatórias
y <- floor(runif(numptos, min=1, max=numlin))
x <- floor(runif(numptos, min=1, max=numcol))
tab <- cbind(seq(1:numptos), x, y)

# Transfere a tabela a Dinamica
outputTable("tabxy", tab)
  
```

A tabela de linhas/colunas permite ao Dinamica extrair os valores espectrais dos pixels correspondentes nas duas bandas, para criar duas tabelas que entram no segundo script de R (estas tabelas têm uma coluna "key" com um número consecutivo, e uma segunda coluna com o valor espectral). O R cria uma tabela *dataframe* com os valores espectrais das duas bandas. Com base nestes dados se elabora um diagrama de dispersão e se ajusta um modelo linear. As funções ***outputDouble("b", coefficients[1])*** e ***outputDouble("a", coefficients[2])*** possibilitam a exportação dos dois coeficientes em um mesmo objeto de Dinamica (os quais são diferenciados pela designação das letras "a" e "b"). O functor de Dinamica ***Extract Struct Number*** permite a recuperação destes dois valores, através da utilização dos nomes designados. Posteriormente, Dinamica utiliza estes valores para calcular os resíduais com operações de álgebra de mapas.

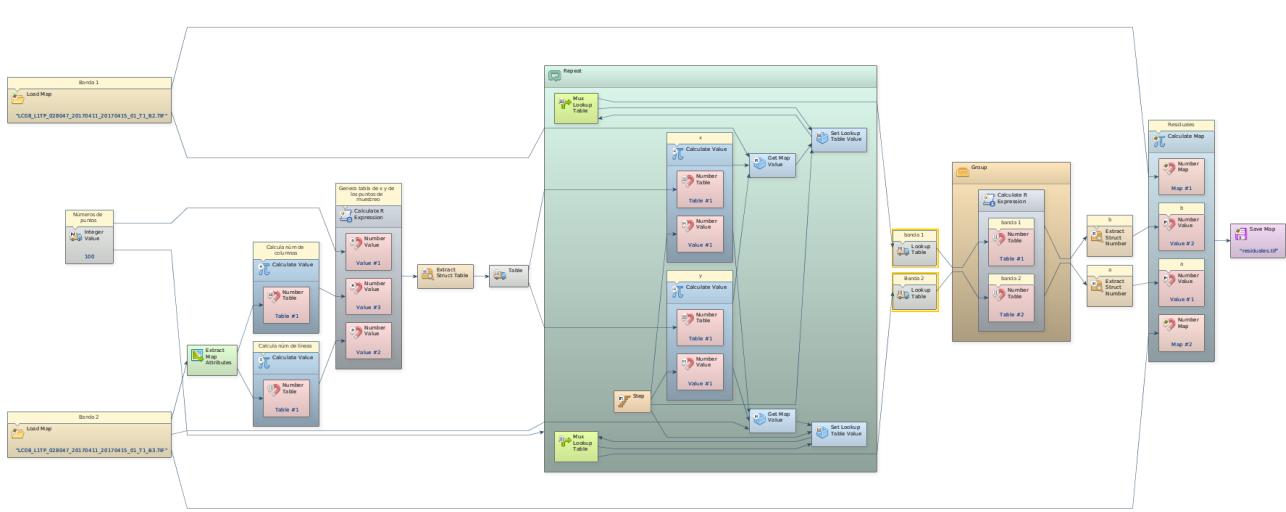


Figura 13. Modelo de Dinamica (Segundo exemplo)

```

# Importa as duas tabelas de Dinamica
tabla1 <- t1
tabla2 <- t2
colnames(tabla1) <- colnames(tabla2) <- c("key", "ND")
# Extrai a coluna com os valores espectrais
x <- tabla1$ND
y <- tabla2$ND
# Cria um dataframe
tab <- data.frame(cbind(x,y))!

# Elabora o diagrama de dispersão
png("diag_dispersion.png")
plot(x,y,main="Diagrama de dispersión banda 2 versus 1", xlab="banda 1",
      ylab="banda 2")
dev.off()

# Ajusta o modelo linear
regression <- lm(formula=y~x, data=tab)
print(regression)
coefficients <- coef(regression)
#  $y = a x + b$ 

# Exporta a Dinamica os coeficientes da equação
outputDouble("b", coefficients[1])
outputDouble("a", coefficients[2])

```

Existem outros pacotes que efetuam operações de R conjuntamente a programas de processamento de imagens e SIG, como spgrass6 (Bivand, 2016) e rgrass7 (Bivand *et al.*, 2017b, Bivand, 2007, programa grass); RSAGA (programa SAGA, Brenning, 2008) e RPyGeo (programa ArcGis, Brenning, 2012).

10. Recursos da internet



10.1 Blogs e Tutoriais (inglês)

- Geospatial Use Cases, R Analytical Environment: <https://www.nceas.ucsb.edu/scicomp/usecases>
- GeogR Geographic Data Analysis Using R <http://www.geog.uoregon.edu/GeogR/>
- Geospatial Use Cases, R Analytical Environment <http://www.latextemplates.com/cat/books>
- Making Maps with R <http://www.molecularecologist.com/2012/09/making-maps-with-r/>
- Maps and Data Visualisations with R <http://spatialanalysis.co.uk/r/>
- Robinlovelace/Creating-maps-in-R <https://github.com/Robinlovelace/Creating-maps-in-R>
- <https://rseek.org/>
- Working with Spatial Data <https://casoilresource.lawr.ucdavis.edu/software/r-advanced-statistical-package/working-spatial-data/>

10.2 Tutoriais e teses (português)

- HumbertoSubiza/Intro_analise_espacial_R https://github.com/HumbertoSubiza/Intro_analise_espacial_R
- Galas, S., Estatística espacial utilizando o software R aplicado em dados de precipitação pluviométrica do estado do Ceará, Universidade Federal do Ceará <http://www.repositorio.ufc.br/handle/riufc/24732>
- Dall’Agnol, R.W., 2017, Aplicação web integrada ao software R para análise espacial e geração de mapas temáticos em agricultura de precisão <http://bdtd.ibict.br>

10.3 Repositórios de pacotes

- The Comprehensive R Archive Network <https://cran.r-project.org/>
- Github <https://github.com/topics/r>

10.4 Livros

- Books related to R <https://www.r-project.org/doc/bib/R-books.html>
- R Documentation <https://www.r-project.org/other-docs.html>
- R manuals and contributed documentation <https://cran.r-project.org>
- Lovelace, R., J. Nowosad, J. Muenchow, 2018, Geocomputation with R, <https://geocompr.robinlovelace.net>

Referências

- Bivand, R. (2007). “Using the R–GRASS Interface: Current Status”. Em: *OSGeo Journal* 1: 36–38.
- (2016). *spgrass6: Interface between GRASS 6 and R*. R package version 0.8-9.
- Bivand, R. & G. Piras (2015). “Comparing Implementations of Estimation Methods for Spatial Econometrics”. Em: *Journal of Statistical Software* 63(18): 1–36.
- Bivand, R. & C. Rundel (2017). *Rgeos: Interface to Geometry Engine - Open Source ('GEOS')*. <https://CRAN.R-project.org/package=rgeos>.
- Bivand, R., T. Keitt & B. Rowlingson (2017a). *rgdal: Bindings for the 'Geospatial' Data Abstraction Library*. R package version 1.2-15.
- Bivand, R., R. Krug, M. Neteler & S. Jeworutzki (2017b). *rgrass7: Interface Between GRASS 7 Geographical Information System and R*. R package version 0.1-10.
- Bivand, R. S., E. J. Pebesma & V. Gómez-Rubio (2008). *Applied Spatial Data Analysis with R*. New York: Springer.
- Brenning, A. (2008). “Statistical geocomputing combining R and SAGA: The example of landslide susceptibility analysis with generalized additive models”. Em: *SAGA – Seconds Out (= Hamburger Beitraege zur Physischen Geographie und Landschaftsoekologie, vol. 19)*. J. Boehner, T. Blaschke, L. Montanarella. 23–32.
- (2012). *RPyGeo: ArcGIS Geoprocessing in R via Python*. R package version 0.9-3.
- Brudson, C. & L. Comber (2015). *An Introduction to R for Spatial Analysis and Mapping*. New York, USA: SAGE.
- Dunnington, D. (2017). *prettymapr: Scale Bar, North Arrow, and Pretty Margins in R*. CRAN.R-project.org/.
- Dunnington, D. & T. Giraud (2017). *rosm: Plot Raster Map Tiles from Open Street Map and Other Sources*. <https://CRAN.R-project.org/package=rosm>.
- Getis, A. & J. K. Ord (1992). “The Analysis of Spatial Association by Use of Distance Statistics”. Em: *Geographical Analysis* 24: 189–206.
- Lamstein, A. & B. P. Johnson (2018). *choroplethr: Simplify the Creation of Choropleth Maps in R*. <https://CRAN.R-project.org/package=choroplethr>.
- Loecher, M. (2016). *RgoogleMaps: Overlays on Static Maps*. <https://CRAN.R-project.org/package=RgoogleMaps>.
- Lovelace, R., J. Nowosad & J. Muenchow (2018). *Geocomputation with R*. <http://geocompr.robinlovelace.net/>. CRS Press.
- Mas, J.-F., U. Jiménez-Pelágo & G. Ramírez-Hernández (2018). “El programa R como herramienta para el análisis espacial”. Em: *Memorias del XVIII Simposio Internacional de la Sociedad Latinoamericana de Percepción Remota y Sistemas de Información Espacial, La Habana, Cuba (CD)*. SELPER.
- Mas, J., M. Kolb, M. Paegelow, M. C. Olmedo & T. Houet (2014). “Inductive pattern-based land use/cover change models: A comparison of four software packages”. Em: *Environmental Modelling and Software* 51: 94–111.
- Pebesma, E. (2017). *Map overlay and spatial aggregation in sp*. cran.r-project.org/web/packages/sp/vignettes/over.pdf. Institute for Geoinformatics, University of Münster. Alemania.
- Pebesma, E. & R. Bivand (2018). *Sp: Classes and Methods for Spatial Data*. <https://CRAN.R-project.org/package=sp>.
- Rochette, S. (2018). *Installation of R 3.5 on Ubuntu 18.04 LTS and tips for spatial packages*. URL: <https://rtask.thinkr.fr/blog/installation-of-r-3-5-on-ubuntu-18-04-lts-and-tips-for-spatial-packages/>.

- Rodrigues, H. & B. Soares-Filho (2018). “A Short Presentation of Dinamica EGO”. Em: *Geomatic Approaches for Modeling Land Change Scenarios*. Ed. por M. T. Camacho Olmedo, M. Paegelow, J.-F. Mas & F. Escobar. Cham: Springer International Publishing. 493–498.
- Tennekes, M., J. Gombin, S. Jeworutzki, K. Russell & R. Zijdeman (2017). *tmap: Thematic Maps*. R package version 1.10.
- Xie, Y. (2013). *Dynamic Documents with R and Knitr*. Chapman & Hall/CRC.