Capítulo
5

Apresentação de Resultados Experimentais para Processamento de Alto Desempenho em `R`

Vinícius Garcia Pinto
Centro de Ciências Computacionais, Universidade Federal do Rio Grande
Rio Grande, Brasil

Lucas Leandro Nesi, Lucas Mello Schnorr
Instituto de Informática, Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Resumo

A apresentação de resultados de experimentos que avaliam desempenho é uma etapa onipresente nas pesquisas de processamento de alto desempenho (PAD). Uma apresentação adequada é essencial para compreensão e reprodutibilidade dos resultados relatados. Tal etapa, entretanto, é frequentemente subestimada, sendo conduzida de maneira ad hoc*, utilizando ferramentas inapropriadas e com pouca rastreabilidade, como por exemplo, editores de planilhas eletrônicas. Neste capítulo, apresentaremos uma abordagem baseada na linguagem* `R` *abrangendo desde o processamento dos dados brutos até a representação visual. A metodologia a ser apresentada explora os pacotes* `dplyr`*,* `tidyr`*,* `readr`*,* `patchwork`*,* `ggplot2` *e* `plotly` *para permitir que todas as transformações e manipulações aplicadas aos dados sejam documentadas e reprodutíveis. O conteúdo abordado está organizado em três seções principais: introdução à linguagem* `R`*, processamento de dados brutos e visualização de dados. A discussão de cada seção é acompanhada de exemplos práticos dos tópicos apresentados.*

5.1. Considerações Iniciais

O processo metodológico para realização de experimentos em computação de alto desempenho pode ser organizado em duas grandes etapas: a coleta dos dados e a análise dos resultados. A coleta dos dados é o momento em que ocorre a execução de fato do experimento no ambiente computacional de alto desempenho. O sucesso desta etapa depende dos apropriados controle e coleta dos parâmetros da plataforma computacional e

da definição de um projeto experimental [Pinto et al. 2020]. Já a etapa de análise de resultados, onde acontece o pré-processamento, filtragem, transformação, interpretação e apresentação das conclusões, é realizada *a posteriori*. É nessa etapa que se situa este minicurso.

A reprodutibilidade da análise e da apresentação dos resultados é um aspecto chave do processo científico. Entretanto é frequentemente subestimada ou realizada com ferramentas com pouca rastreabilidade. Em diferentes áreas do conhecimento, existem relatos de problemas [Hutson 2010, Herndon et al. 2014], equívocos ou retratações decorrentes de falhas metodológicas ou de manipulação e interpretação dos dados. Uma forma de fortalecer a reprodutibilidade da apresentação de resultados e reduzir riscos é investir em automatização de processos (*scripting*), na rastreabilidade, no versionamento e na apresentação. Tais conceitos nem sempre são plenamente acessíveis ou praticáveis em ferramentas interativas como editores de planilhas eletrônicas. Entretanto, são largamente utilizados no desenvolvimento de *software*. Neste minicurso, ilustramos como podemos usar a linguagem `R` e seu ambiente para ir desde o processamento dos dados brutos até a representação visual. Tal abordagem é facilmente documentável, rastreável e reprodutível.

Este capítulo está organizado em três grandes seções. Primeiramente, a Seção 5.2 apresenta uma introdução à linguagem `R` discutindo os comandos e funcionalidades essenciais. Nessa seção, são abordados apenas recursos providos pela biblioteca de base. Em seguida, a Seção 5.3 trata do processamento de dados brutos em formato texto. As transformações apresentadas nessa seção são providas pelos seguintes pacotes: `readr` [Wickham and Hester 2021], `magrittr` [Bache and Wickham 2022], `dplyr` [Wickham et al. 2021], `tidyr` [Wickham and Girlich 2022] e, finalmente, o pacote `tibble` [Müller and Wickham 2021]. Esses pacotes são disponibilizados pelo metapacote `tidyverse` [Wickham et al. 2019]. Por fim, a visualização de dados é discutida na Seção 5.4 onde empregamos funcionalidades dos pacotes `ggplot2` [Wickham 2016], `patchwork` [Pedersen 2020] e `plotly` [Sievert 2020].

5.2. Introdução à linguagem `R`

Nas próximas seções discutiremos alguns aspectos fundamentais da linguagem `R`. O objetivo é introduzir a linguagem e suas características a leitores e leitoras que tenham pouca ou nenhuma experiência com a mesma. O conteúdo abordado restringe-se ao necessário no escopo deste minicurso e não abrange outras funcionalidades da biblioteca base da linguagem `R`. Por exemplo, para a leitura de dados e construção de gráficos, utilizaremos pacotes específicos.

5.2.1. Comandos básicos

`R` é uma linguagem interpretada, logo, a interação com o usuário acontece primordialmente por uma interface de linha de comandos (CLI), dispensando comandos específicos para entrada e saída básicas. Os comandos de atribuição (`<-` e `->`) possibilitam o armazenamento de valores na memória, resultando na criação ou alteração de um objeto `R`. Para simplificar, vamos considerar um objeto `R` como equivalente a uma variável em linguagens de propósito geral como `C`. Operações aritméticas básicas podem ser obtidas

com os operadores usuais como −, +, ⋆, / e ^, assim como as relacionais com <, >, ==, !=, >= e <=. O caractere # é usado para inserção de comentários de linha.

```
x <- 100            # x recebe 100
100/3 + 2^2 -> y    # y recebe 37.33333...
z <- "escola regional de alto desempenho"
```

Código 5.1. Exemplo de comandos básicos em R.

5.2.2. Funções e Pacotes

Além dos diversos operadores e funções disponíveis por padrão no R, podemos definir nossas próprias funções atribuindo uma construção `function` a um nome. Os argumentos são opcionais e podem conter valores predefinidos (*default values*). A construção `return` pode ser usada para retornar explicitamente algum valor. Todavia, na ausência desta, por padrão, R retorna o valor da última linha de código avaliada da função.

```
aceleracao <- function(tseq, tpar){
    return(tseq/tpar)
}
```

Código 5.2. Exemplo de definição de função.

Funções R usualmente são agrupadas e distribuídas em pacotes. O CRAN[1] (*Comprehensive R Archive Network*) é o repositório oficial da linguagem e contém milhares de pacotes desenvolvidos pela comunidade. Pacotes do CRAN podem ser instalados com o comando `install.packages` e uma vez instalados podem ser carregados com o comando `library`, atualizados com o comando `update.packages` ou removidos com `remove.packages`. A documentação de um pacote ou de uma função pode ser acessada por meio do comando `?` seguido do respectivo nome (*e.g.*, `?round`).

```
install.packages("ggplot2")
library(ggplot2)

ggplot(mtcars, aes(x = mpg, y = cyl, color = gear)) + geom_point()
```

Código 5.3. Exemplo de instalação e uso do pacote `ggplot2`.

5.2.3. Tipos básicos

Vetores são o tipo mais simples de objeto R e podem armazenar apenas valores de um mesmo tipo básico. Os tipos básicos possíveis são `logical`, `integer`, `numeric`, `complex`, `character` e `raw` [R Core Team 2021]. Objetos com um único valor, como `x` e `z` declarados no Código 5.1, são vetores de tamanho 1. Vetores de tamanho zero também são permitidos. A indexação de vetores R inicia em 1 e estes podem ter mais de uma dimensão, constituindo matrizes ou de maneira mais geral, vetores multidimensionais. Em contrapartida, uma lista é similar a um vetor porém admite valores de diferentes tipos básicos. Tanto listas quanto vetores podem ter seus valores internos nomeados, possibilitando que os nomes dos valores sejam usados para indexação. O código 5.4 ilustra três maneiras para definição de vetores. A função `c` permite combinar valores individuais em um vetor. Enquanto o operador `:` e a função `seq` são usados para geração de sequências.

[1] `https://cran.r-project.org/`

```
d <- c(1, 5, 3)   # d recebe 1 3 5
e <- 2:6          # e recebe 2 3 4 5 6
f <- seq(9, 5)    # f recebe 9 8 7 6 5
```

Código 5.4. Exemplos de definição de vetores.

Data frames são as estruturas mais usadas para armazenamento de dados experimentais representando casos (linhas) por variáveis (colunas). Cada *data frame* é uma lista de vetores, todos com o mesmo tamanho. Cada vetor desta lista possui um nome e é usado para representar uma variável [Grolemund 2014]. No restante deste capítulo, usaremos *Tibbles* [Müller and Wickham 2021] que são uma implementação moderna dos *Data frames*.

5.2.4. Aritmética com vetores

Vetores `R` podem ser usados como operandos em expressões aritméticas. Nestes casos, a operação é dita vetorial, ou seja, a mesma operação é aplicada em cada um dos valores armazenados. O Código 5.5 ilustra este comportamento. O vetor `b` recebe o resultado da expressão `a^3`, isto é, um novo vetor de mesmo tamanho que `a` onde cada valor deste é elevado ao cubo. Por fim, no caso de `c`, cada posição receberá a soma dos valores armazenados nas mesmas posições de `a` e `b`.

```
a <- 1:5      # a recebe 1  2  3  4   5
b <- a ^ 3    # b recebe 1  8 27 64 125
c <- a + b    # c recebe 2 10 30 68 130
```

Código 5.5. Exemplo do uso de vetores em expressões aritméticas.

No exemplo acima, os três vetores possuem o mesmo tamanho. Entretanto, quando vetores diferentes estão envolvidos na mesma expressão, o resultado da avaliação será um vetor com o tamanho do maior dos vetores presentes na expressão. Os vetores menores serão *reciclados* quantas vezes necessário, até atingir o maior tamanho [Venables et al. 2021]. Um aviso é emitido toda vez que o tamanho do maior vetor não é múltiplo do tamanho do menor. O Código 5.6 exemplifica estas situações.

```
i <- 3:8      # i recebe 3 4 5 6 7 8
j <- 0:1      # j recebe 0 1
k <- i * j    # k recebe 0 4 0 6 0 8

m <- 5:7      # m recebe 5 6 7
n <- m - j    # n recebe 5 5 7 (um aviso é emitido)
```

Código 5.6. Exemplo de expressões aritméticas com vetores de tamanhos distintos.

5.2.5. *Scripts* R

Uma maneira apropriada para disponibilização das etapas e procedimentos realizados durante a análise de resultados experimentais é através de um arquivo `.R` contendo todos os comandos necessários. A criação destes arquivos de comandos (*script*) permite automatizar a reexecução da análise dos dados e das etapas posteriores como a geração de gráficos. Tal automatização contribui para melhor reprodutibilidade visto que há um rastreamento integral desde a leitura dos dados brutos até a apresentação final. O Código 5.7

demonstra como os argumentos informados na linha de comando podem ser recuperados no `R` enquanto o Código 5.8 ilustra a invocação do *script* em um `shell` Linux.

```
#!/usr/bin/Rscript
args <- commandArgs(trailingOnly = TRUE)

print(paste("primeiro argumento:", args[1]))
print(paste("segundo argumento:",  args[2]))
print(paste("terceiro argumento:", args[3]))
```

Código 5.7. Exemplo de recuperação de argumentos da linha de comando.

```
./meuscript.R arg1 arg2 arg3
```

Código 5.8. Invocação de um *script* R.

5.3. Manipulação de resultados em texto puro

Nesta seção abordaremos o processamento dos dados brutos, usualmente obtidos em texto puro após a execução de um experimento. Iniciaremos pela leitura dos dados utilizando o pacote `readr`. Em seguida, empregaremos o pacote `dplyr` para ilustrar uma série de manipulações gerais como a seleção de um subconjunto de colunas (variáveis), reordenamento e filtragem de linhas (observações). O pacote `tidyr` é usado para pivotar o conjunto de dados, reorganizando-o com mais ou menos colunas o que facilita algumas operações. Por fim, utilizaremos novamente o pacote `dplyr` para realização de derivações, criando uma nova coluna a partir das outras colunas ou de dados externos, sumarizações, sintetizando dados de um grupo ou ainda combinações, unindo diferentes conjuntos de dados a partir de uma chave em comum.

5.3.1. Leitura

O pacote `readr` pode ser utilizado para a leitura de dados "retangulares". Estes dados estão estruturados em uma tabela, onde cada linha do documento é uma linha da tabela, e as células estão separadas por um delimitador comum. Cada coluna é uma variável, e cada linha é um caso. Um exemplo clássico deste tipo de documento é o CSV (*Comma-separated values* – valores separados por vírgula). O Código 5.9 apresenta um exemplo de documento CSV com quatro variáveis (nas colunas: Maquina, A, B e Tempo) explicitadas na primeira linha do arquivo, e oito casos, da segunda até a oitava linha. Por exemplo, o caso da última linha é a situação onde Máquina é `Mac2`, A é `1`, B é `1` e Tempo é `18.0`.

```
Maquina,A,B,Tempo
Mac1,0,0,20.0
Mac1,0,1,18.0
Mac1,1,0,16.0
Mac1,1,1,12.0
Mac2,0,0,25.0
Mac2,0,1,20.0
Mac2,1,0,25.0
Mac2,1,1,18.0
```

Código 5.9. Exemplo de um arquivo CSV.

A leitura de um arquivo CSV em R pode ser realizada com a função `read_csv`. Esta função recebe como parâmetro obrigatório o caminho do arquivo. Outros parâmetros adicionais úteis são: *col_types* para especificar o nome e o tipo de cada coluna, e *progress* para mostrar ou não o progresso da leitura. Os tipos de coluna comumente utilizados são *col_character* para *strings*, *col_integer* para inteiros, e *col_double* para números decimais. O seguinte bloco de código mostra um exemplo de utilização desta função para a leitura de dados que se encontram no arquivo `exemplo.csv` (com dados do Código 5.9). A função, após leitura dos dados do arquivo, retorna um `tibble`, que é um tipo para dados tabulares sendo atribuído para a variável `df`:

```
read_csv("exemplo.csv",
         col_types = list(Maquina = col_character(),
                          A       = col_integer(),
                          B       = col_integer(),
                          Tempo   = col_double()),
         progress = FALSE) -> df_ex
df_ex
```

```
# A tibble: 8 x 4
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
2 Mac1        0     1    18
3 Mac1        1     0    16
4 Mac1        1     1    12
5 Mac2        0     0    25
6 Mac2        0     1    20
7 Mac2        1     0    25
8 Mac2        1     1    18
```

Para realizar a leitura de um arquivo retangular estruturado que não seja um CSV, mas que tenha um delimitador comum, pode-se utilizar a função `read_delim`. Um segundo parâmetro obrigatório desta função é *delim*, o delimitador entre células. Basicamente, o `read_csv` é uma chamada para o `read_delim` com *delim=","*. Todas estas funções retornam um objeto do tipo `tibble`.

5.3.2. Encadeamento

O operador `%>%` é um operador provido pelo pacote `magrittr`[2] que possibilita o encadeamento de operações de maneira similar ao mecanismo *pipe* dos sistemas do tipo Unix. Este é um operador binário onde o operando esquerdo é uma variável (incluindo os retornos de funções) e o operando direito é uma função com pelo menos um parâmetro. O operador permite a passagem do operando da esquerda (variável) como o primeiro argumento do operador da direita (função). Desta forma, uma chamada originalmente escrita como `fun(x, y)` pode ser realizada com o operador pipe como `x %>% fun(y)`. Este operador permite uma melhor redação de código, já que as várias operações estarão escritas (concatenadas) na ordem em que elas ocorrem.

[2]Recentemente, na versão R 4.1.0, foi adicionado na linguagem um comando de *pipe* nativo com a sintaxe `|>`.

5.3.3. Transformações

O comando `select` é utilizado para selecionar colunas dos dados. Para tal, basta passar como parâmetro o nome das colunas desejadas para seleção assim:

```
df_ex %>% select(Maquina, Tempo)
```

```
# A tibble: 8 x 2
  Maquina Tempo
  <chr>   <dbl>
1 Mac1       20
2 Mac1       18
3 Mac1       16
4 Mac1       12
5 Mac2       25
6 Mac2       20
7 Mac2       25
8 Mac2       18
```

Outra operação recorrente é a ordenação dos dados. Para isto, pode-se utilizar o comando `arrange`. A ordenação pode ser realizada utilizando múltiplas colunas, onde os dados primeiramente são ordenados pela primeira coluna informada, em seguida pela segunda, e assim por diante. Então, para os casos com a primeira coluna com valor repetido, acontece a ordenação pelos valores das colunas subsequentes.

```
df_ex %>% arrange(A, B)
```

```
# A tibble: 8 x 4
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
2 Mac2        0     0    25
3 Mac1        0     1    18
4 Mac2        0     1    20
5 Mac1        1     0    16
6 Mac2        1     0    25
7 Mac1        1     1    12
8 Mac2        1     1    18
```

5.3.4. Filtragem

O comando `filter` é utilizado para realizar a filtragem de casos que satisfazem alguma condição nas suas variáveis. O seu argumento principal é a condição lógica que os casos devem satisfazer para passarem pelo filtro. Por exemplo, considerando o conjunto de dados anterior, gostaríamos de filtrar os dados que foram executados na máquina `Mac1`. Desta forma, pode-se utilizar o seguinte comando:

```
df_ex %>% filter(Maquina == "Mac1")
```

```
# A tibble: 4 x 4
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
```

```
2 Mac1         0     1    18
3 Mac1         1     0    16
4 Mac1         1     1    12
```

Outros operadores lógicos podem ser utilizados para especificar as necessidades de filtragem. Um exemplo mais elaborado é a filtragem de casos na máquina `Mac1` ou `Mac2` onde o tempo de execução é maior que 18s. Em outras linguagens de programação, a operação lógica de máquinas seria descrita usualmente utilizando `Maquina = "Mac1" OU Maquina = "Mac2"`. Entretanto, em `R` podemos utilizar o operador binário `%in%` , ou seja, `váriavel %in% vetor` onde a operação é verdade se a variável está presente no conjunto de elementos do vetor.

```
df_ex %>% filter(Maquina %in% c("Mac1", "Mac2") & Tempo > 18)
```

```
# A tibble: 4 x 4
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
2 Mac2        0     0    25
3 Mac2        0     1    20
4 Mac2        1     0    25
```

Ainda é possível selecionar linhas específicas dos dados. Isso é realizado com o comando `slice(linhas)`, onde `linhas` é um vetor que tem os números das linhas que devem ser selecionadas. Por exemplo, para selecionar as linhas de 2 até 4, pode-se utilizar:

```
df_ex %>% slice(2:4)
```

```
# A tibble: 3 x 4
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     1    18
2 Mac1        1     0    16
3 Mac1        1     1    12
```

5.3.5. Pivotagem

A organização de dados recomendada é conhecida como *tidy* [Wickham 2014]. Nesta abordagem, cada coluna é uma variável, cada linha é uma observação, e cada célula contêm um valor único. Entretanto, existem duas abordagens principais de organização dos dados que são chamadas de "longas" e "largas". A organização de dados "larga" utiliza mais colunas e diminui a quantidade de linhas. Nesta abordagem, cada linha contém diversas observações. Na organização "longa", cada coluna é uma variável. A Tabela 5.1 apresenta estas duas organizações.

Pode-se utilizar os comandos `pivot_wider` e `pivot_longer` para transformar dados de uma organização em outra. Com o comando `pivot_longer`, transformamos uma organização larga em longa informando: as colunas que desejamos unificar, a coluna que vai receber o nome destas variáveis (*names_to*), e a coluna que vai receber o valor destas variáveis (*values_to*). No exemplo abaixo, construímos duas tabelas através

Nome	Caso_A	Caso_B	Caso_C
α	1	2	3
β	4	5	6

Nome	Variavel_1	Valor
α	Caso_A	1
α	Caso_B	2
α	Caso_C	3
β	Caso_A	4
β	Caso_B	5
β	Caso_C	6

Tabela 5.1. Dados organizados de maneira larga (esquerda) e longa (direita), sendo esta organização também conhecida como *tidy*.

da função `tibble` que recebe como parâmetro uma lista de elementos nomeados, onde os nomes são as variáveis das tabelas (`Nome` `Caso_A`, e assim por diante) e o conteúdo são vetores de dados.

```
df_wider <- tibble(Nome = c("Alpha", "Beta"),
                   Caso_A = c(1, 4),
                   Caso_B = c(2, 5),
                   Caso_C = c(3, 6))
pivot_longer(df_wider,
             c(Caso_A, Caso_B, Caso_C),
             names_to = "Variavel_1",
             values_to = "Valor") -> df_longer
df_longer
```

```
# A tibble: 6 x 3
  Caso  Variavel_1 Valor
  <chr> <chr>      <dbl>
1 Alpha Caso_A         1
2 Alpha Caso_B         2
3 Alpha Caso_C         3
4 Beta  Caso_A         4
5 Beta  Caso_B         5
6 Beta  Caso_C         6
```

A operação reversa é realizada com a função `pivot_wider` e com os argumentos opostos. Informando a coluna que contêm os nomes das variáveis (*names_from*) e a coluna que contêm os valores (*values_from*).

```
df_longer %>% pivot_wider(names_from = `Variavel_1`,
                          values_from = Valor)
```

```
# A tibble: 2 x 4
  Caso  Caso_A Caso_B Caso_C
  <chr>  <dbl>  <dbl>  <dbl>
1 Alpha      1      2      3
2 Beta       4      5      6
```

5.3.6. Derivações

A criação de uma nova coluna pode ser realizada com a operação `mutate`, informando o nome da nova coluna e seus valores. Os valores podem ser derivados de outras colunas ou utilizando vetores externos ao *dataset*.

```
df_ex %>% mutate(D = A+B)
```

```
# A tibble: 8 x 5
  Maquina     A     B Tempo     D
  <chr>   <int> <int> <dbl> <int>
1 Mac1        0     0    20     0
2 Mac1        0     1    18     1
3 Mac1        1     0    16     1
4 Mac1        1     1    12     2
5 Mac2        0     0    25     0
6 Mac2        0     1    20     1
7 Mac2        1     0    25     1
8 Mac2        1     1    18     2
```

Para sumarizar os dados, i.e., realizar uma transformação sobre um determinado grupo para gerar um valor final, pode-se utilizar a operação `summarize`. Uma função comumente utilizada é a função de média sobre os valores:

```
df_ex %>% summarize(media_a = mean(A))
```

```
# A tibble: 1 x 1
  media_a
    <dbl>
1     0.5
```

5.3.7. Combinando Tabelas

Dados podem ser organizados em diferentes tabelas que compartilhem informações. Estes compartilhamentos de dados são relações entre as tabelas. Por exemplo, podemos ter uma tabela descrevendo as máquinas e uma tabela de experimentos nas máquinas (conforme podemos visualmente observar na Tabela 5.2):

```
maquinas <- tibble(Nome = c("Mac1", "Mac2"),
                   Cores = c(20, 24))
experimentos <- tibble(Maquina = c("Mac1", "Mac1", "Mac2", "Mac2"),
                       Parametro = c(0, 1, 0, 1),
                       Tempo = c(20, 10, 24, 12))
```

Se desejamos adicionar os valores de *Cores* para a tabela de experimentos, podemos realizar a operação de união de tabelas utilizando o comando `inner_join` (Vários outros comandos `*_join` estão disponíveis para necessidades variadas). A operação recebe duas tabelas para serem unidas, e um argumento especificando quais colunas devem ser utilizadas para a combinação. No exemplo de máquinas e experimentos, devemos realizar a combinação da coluna `Maquina` da tabela `experimentos` com a coluna `Nome` da tabela `maquinas`. Informamos esta combinação com o parâmetro *by* passando um vetor nomeado onde cada elemento tem como nome a coluna da tabela e o valor a coluna da segunda tabela. Tais combinações podem ser feitas com mais de uma variável.

Nome	Cores
Mac1	20
Mac2	24

Máquina	Parâmetro	Tempo
Mac1	0	20
Mac1	1	10
Mac2	0	24
Mac2	1	12

Tabela 5.2. Tabela das máquinas (esquerda) e de experimentos (direita).

```
inner_join(experimentos, maquinas, by = c("Maquina" = "Nome"))
```

```
# A tibble: 4 x 4
  Maquina Parametro Tempo Cores
  <chr>       <dbl> <dbl> <dbl>
1 Mac1            0    20    20
2 Mac1            1    10    20
3 Mac2            0    24    24
4 Mac2            1    12    24
```

5.3.8. Grupos

Os dados podem ser organizados em grupos, ou seja, casos que compartilham o mesmo valor em uma ou mais variáveis. Esse agrupamento é realizado com a função `group_by` e ele é importante já que os comandos do `dplyr` podem atuar sobre grupos. Por exemplo, considerando o dataset anterior, podemos criar grupos para cada máquina utilizada com o seguinte comando `group_by`, que recebe dois argumentos: os dados e as variáveis que serão utilizadas para o agrupamento (neste caso, apenas a variável `Maquina`):

```
df_ex %>% group_by(Maquina)
```

```
# A tibble: 8 x 4
# Groups:   Maquina [2]
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
2 Mac1        0     1    18
3 Mac1        1     0    16
4 Mac1        1     1    12
5 Mac2        0     0    25
6 Mac2        0     1    20
7 Mac2        1     0    25
8 Mac2        1     1    18
```

O *tibble* resultante contêm o seguinte cabeçalho:

```
# A tibble: 8 x 4
# Groups:   Maquina [2]
```

Assim, casos que compartilham o mesmo valor para a variável Maquina estarão logicamente organizados e divididos em grupos. A mensagem `# Groups: Maquina`

`[2]` mostra que existem dois grupos, já que existem dois valores possíveis para máquina. Considerando os grupos, podemos realizar o comando `slice(1)`, onde cada grupo será recortado individualmente:

```
df_ex %>% group_by(Maquina) %>% slice(1)
```

```
# A tibble: 2 x 4
# Groups:   Maquina [2]
  Maquina     A     B Tempo
  <chr>   <int> <int> <dbl>
1 Mac1        0     0    20
2 Mac2        0     0    25
```

5.4. Construção de gráficos

Com uma perspectiva histórica, a linguagem `R` possui um extenso histórico de geração de gráficos estatísticos sendo este suporte majoritariamente nativo, embutido na biblioteca de base da linguagem. Ao longo dos anos, surgiu um pacote para esta linguagem chamado `ggplot2` que incorpora uma importante evolução no que diz respeito a forma como os gráficos são construídos. Este pacote implementa o que é conhecido por uma Gramática de Gráficos [Wilkinson 2012], que foi uma primeira proposta para prover uma teoria formal para criar visualização de dados. Nós veremos aqui a visão em camadas desta gramática de gráficos [Wickham 2010].

A ideia principal por trás do conceito de gramática de gráficos é que, ao construir um gráfico, se realiza de maneira explícita uma relação entre as variáveis dos dados (colunas de tabelas) com os elementos gráficos. Por exemplo, em um gráfico de pontos, o tamanho do ponto será associado a uma determinada coluna da tabela que contém os dados de origem. Esse mesmo tipo de mapeamento acontece para todos os demais elementos estéticos tais como forma, cor, eixos horizontal e vertical, e assim por diante. Frequentemente, as características estéticas obrigatórias dependem do tipo de gráfico. Com pontos, precisamos apenas de duas coordenadas para posicioná-los, para um gráfico de retângulos, no entanto, precisamos de quatro coordenadas.

Não temos por objetivo ser exaustivos na apresentação de todos os tipos de gráficos e estéticas obrigatórias, afinal de contas temos a documentação em mãos[3]. Precisamos deixar claro que, independente do tipo de gráfico que desejamos construir, os dados devem estar organizados de maneira "longa", ou *tidy*, como já apresentado (veja Seção 5.3.3). Na terminologia do pacote `ggplot2`, os tipos de gráficos são chamados de *geoms* e o mapeamento de dados para as estéticas são definidas com a função `aes()`.

No que segue, apresentamos as principais funcionalidades do pacote `ggplot2` para a construção de gráficos. Em seguida, veremos como combinar gráficos diferentes com o pacote `patchwork`. Enfim, veremos como estes objetos gráficos podem ser transformados em gráficos interativos, com a ferramenta `plotly`. Para este percurso, adotamos um conjunto de dados disponível publicamente na internet[4] que consiste em

[3]Disponível em `https://ggplot2.tidyverse.org/reference/`

[4]Disponível em `https://github.com/dlenski/top500/blob/master/TOP500_history.csv`

dados históricos do Top500. Uma vez este arquivo estando localmente disponível, podemos ler seu conteúdo da seguinte forma:

```
FILE <- "TOP500_history.csv"
df_t500 <- read_csv(FILE, progress = FALSE, show_col_types = FALSE)
```

Os dados do Top500 são bastante numerosos. São 29 mil observações e 53 variáveis. Simplificamos os dados por razões pedagógicas selecionando apenas as sete colunas seguintes (percebam o encadeamento de comandos com o `%>%` como explicado na Seção 5.3.2), criando uma nova coluna (utilizando a função `ymd` do pacote `lubridate`) para termos uma única coluna com a data, e filtrando para obter apenas sistemas que tenham uma região definida (onde na coluna `Region` temos um valor diferente de `NA`):

```
library(lubridate)
df_t500 %>%
    filter(!is.na(Region)) %>%
    mutate(Data = ymd(paste(Year, Month, Day))) %>%
    select(Data, Rank, RPeak, Region, Nmax) %>%
    print(n=2) -> df.simple
```

```
# A tibble: 27,000 x 5
  Data        Rank RPeak Region         Nmax
  <date>     <dbl> <dbl> <chr>         <dbl>
1 1993-06-01     1 131   North America 52224
2 1993-06-01     2  69.6 North America 36864
#...with 26,998 more rows
```

Os dados do Top500 contém a posição dos diferentes sistemas computacionais de alto desempenho que foram ranqueados ao longo do tempo. Cada linha da tabela consiste em um sistema, sendo que as colunas representam as diferentes variáveis que o sistema teve naquele instante de tempo.

5.4.1. Gráficos Básicos

Para a construção de gráficos devemos realizar chamadas ao comando `ggplot` do pacote `ggplot2`. Como `ggplot` faz parte do universo de pacotes que integram o `dplyr`, ele pode ser utilizado com encadeamento de comandos (veja abaixo). Primeiro filtramos para obter apenas os sistemas que foram ranqueados em primeiro lugar (`Rank == 1`), para em seguida construir o gráfico. Usando a função `aes()`, nós mapeamos para o eixo horizontal a coluna `Data` (`x = Data`) e para o eixo vertical a coluna `RPeak` (y = `RPeak`). Isso permite definir duas estéticas básicas para o gráfico, sendo estas obrigatórias para um gráfico de pontos `geom_point()`.

```
df.simple %>%
    filter(Rank == 1) %>%
    ggplot(aes(x = Data, y = RPeak)) +
    theme_bw(base_size = 12) +
    geom_point() -> p1
p1
```

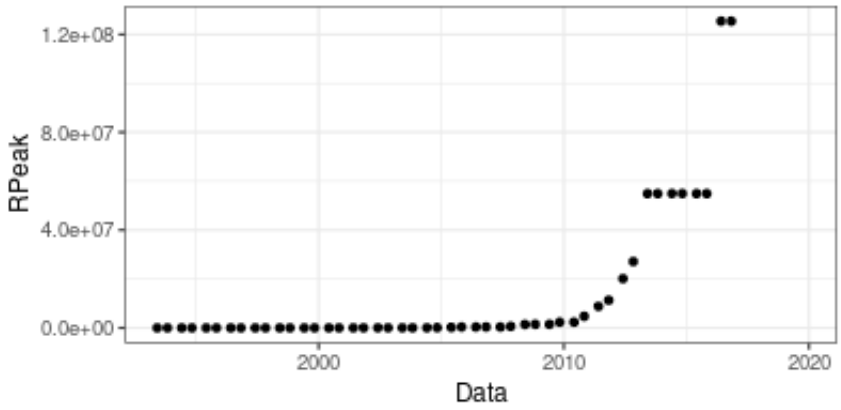
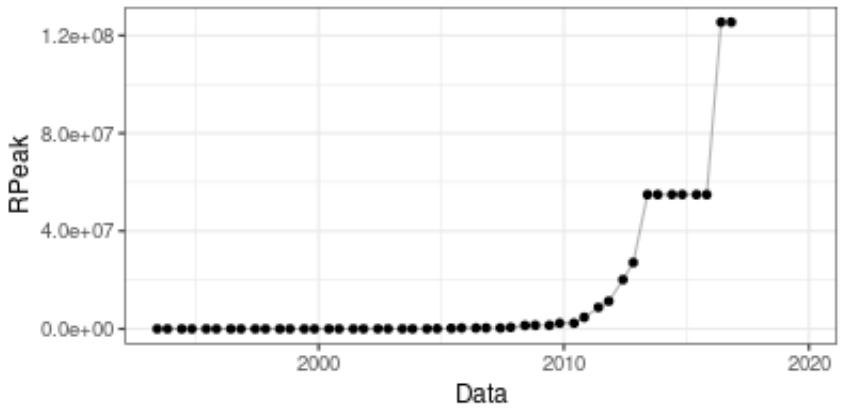


Figura 5.1. Exemplo de gráfico de pontos construído com o pacote `ggplot2`.

Um objeto `ggplot` consiste em camadas sobrepostas de tipos de gráficos potencialmente diferentes. Por exemplo, podemos unir os pontos do gráfico anterior simplesmente adicionando uma chamada `geom_line()`, que herdará a estética definida globalmente na chamada `ggplot()`. A ordem das camadas importa, portanto adicionar `geom_point() + geom_line()` é diferente graficamente do que fazer o contrário. No primeiro caso, as linhas são desenhadas por cima dos pontos. O exemplo abaixo faz o contrário, mantendo as linhas desenhadas por trás com uma certa transparência para distinguirmos a diferença:

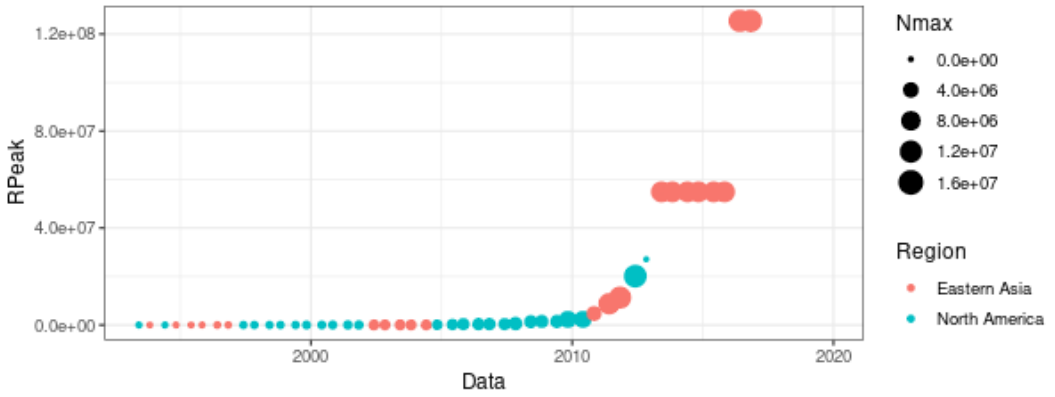
```
df.simple %>%
    filter(Rank == 1) %>%
    ggplot(aes(x = Data, y = RPeak)) +
    theme_bw(base_size = 12) +
    geom_line(alpha = 0.3) +
    geom_point()
```



Outras estéticas incluem `size` e `color`, por exemplo. Elas podem ser empregadas de maneira análoga ao que já fizemos com as coordenadas x e y. No exemplo abaixo,

mapeamos a região para a cor (`color = Region`) e o valor de `Nmax` para o tamanho (`size = Nmax`).

```
df.simple %>%
    filter(Rank == 1) %>%
    ggplot(aes(x = Data, y = RPeak, size = Nmax, color = Region)) +
    theme_bw(base_size = 12) +
    geom_point() -> p2
p2
```



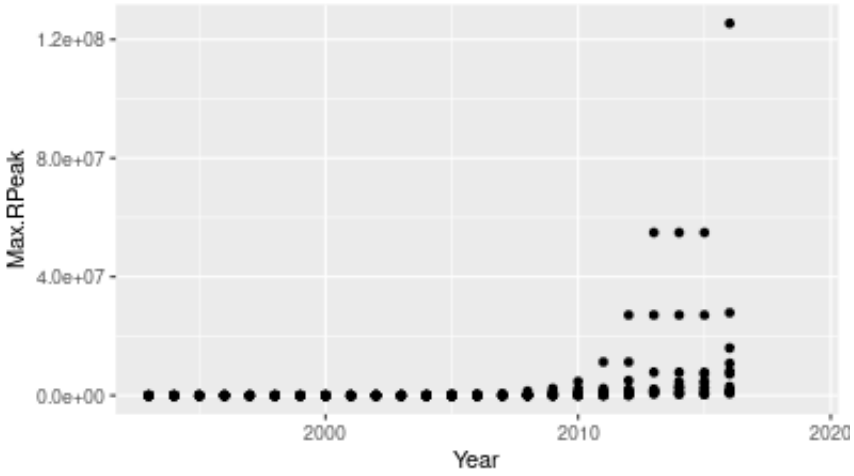
Podemos observar que automaticamente o pacote `ggplot2` define as escalas de cores e de tamanho para `Region` e `NMax` conforme os dados da tabela fornecida como entrada (`df.simple`). Observamos também que alguns sistemas que foram posicionados em primeiro lugar ao longo do tempo tem a região não definida `NA`.

5.4.2. Comparações com Facetas

O emprego de facetas permite a criação de painéis definidos por linha e coluna dependendo de variáveis categóricas que temos em nossos dados. Existem dois tipos de facetas que podem ser criadas, em grade (`facet_grid`) ou em linha (`facet_wrap`). A primeira é utilizada quando desejamos posicionar paineis em uma grade criada a partir de duas variáveis dos dados, enquanto que a segunda é utilizada quando desejamos paineis baseados apenas em uma variável, em cenários onde esta possui vários valores possíveis.

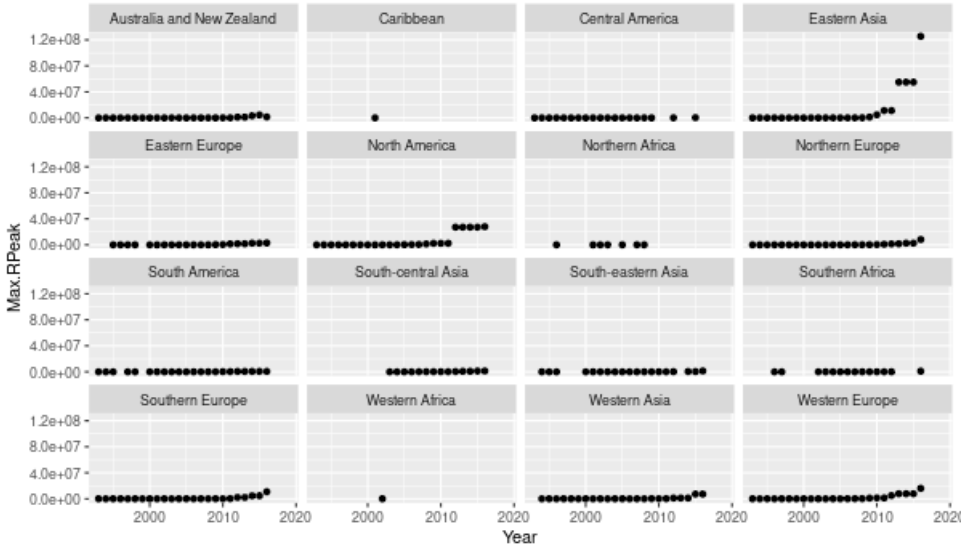
Para exemplificar o emprego de facetas, vamos primeiro agrupar os dados por ano e por região para então calcular o máximo `RPeak` observado em cada um desses grupos. Em seguida, vamos criar um gráfico onde mapeamos o ano para o eixo horizontal (`x = Year`) e o máximo `RPeak` para o eixo vertical (`y = Max.RPeak`). Construindo um gráfico de pontos (com a camada `geom_point()`), observamos que para um mesmo ano temos vários pontos representando as diferentes regiões.

```
df.simple %>%
    mutate(Year = year(Data)) %>%
    group_by(Year, Region) %>%
    summarize(Max.RPeak = max(RPeak), .groups = "keep") %>%
    ggplot(aes(x = Year, y = Max.RPeak)) +
    geom_point()
```



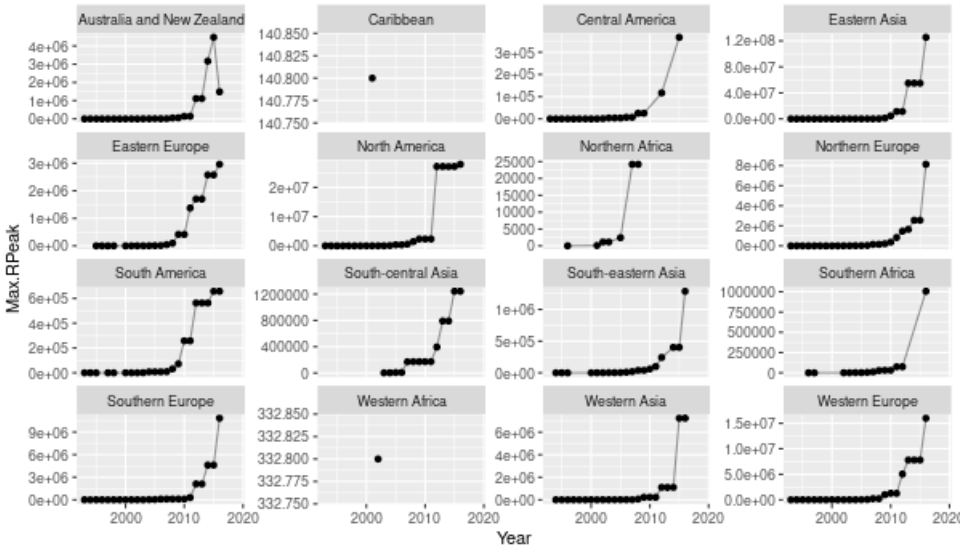
Para empregar facetas para diferenciar as regiões no gráfico acima, basta adicionarmos uma nova camada `facet_wrap()` informando sobre qual variável desejamos que tais facetas sejam criadas:

```
df.simple %>%
    mutate(Year = year(Data)) %>%
    group_by(Year, Region) %>%
    summarize(Max.RPeak = max(RPeak), .groups = "keep") %>%
    ggplot(aes(x = Year, y = Max.RPeak)) +
    geom_point() +
    facet_wrap(~Region)
```



Observamos que os gráficos compartilham as escalas de x (`Year`) e y (`Max.RPeak`). Isso normalmente é o desejado quando se analisam dados através desses paineis, mas por vezes gostaríamos de ter escalas livres para cada uma das facetas. Para fazer isso no eixo y, adicionamos um novo parâmetro ao `facet_wrap` e adicionamos linhas:

```
df.simple %>%
    mutate(Year = year(Data)) %>%
    group_by(Year, Region) %>%
    summarize(Max.RPeak = max(RPeak), .groups = "keep") %>%
    ggplot(aes(x = Year, y = Max.RPeak)) +
    geom_line(alpha = .4) +
    geom_point() +
    facet_wrap(~Region, scales = "free_y") -> p3
p3
```

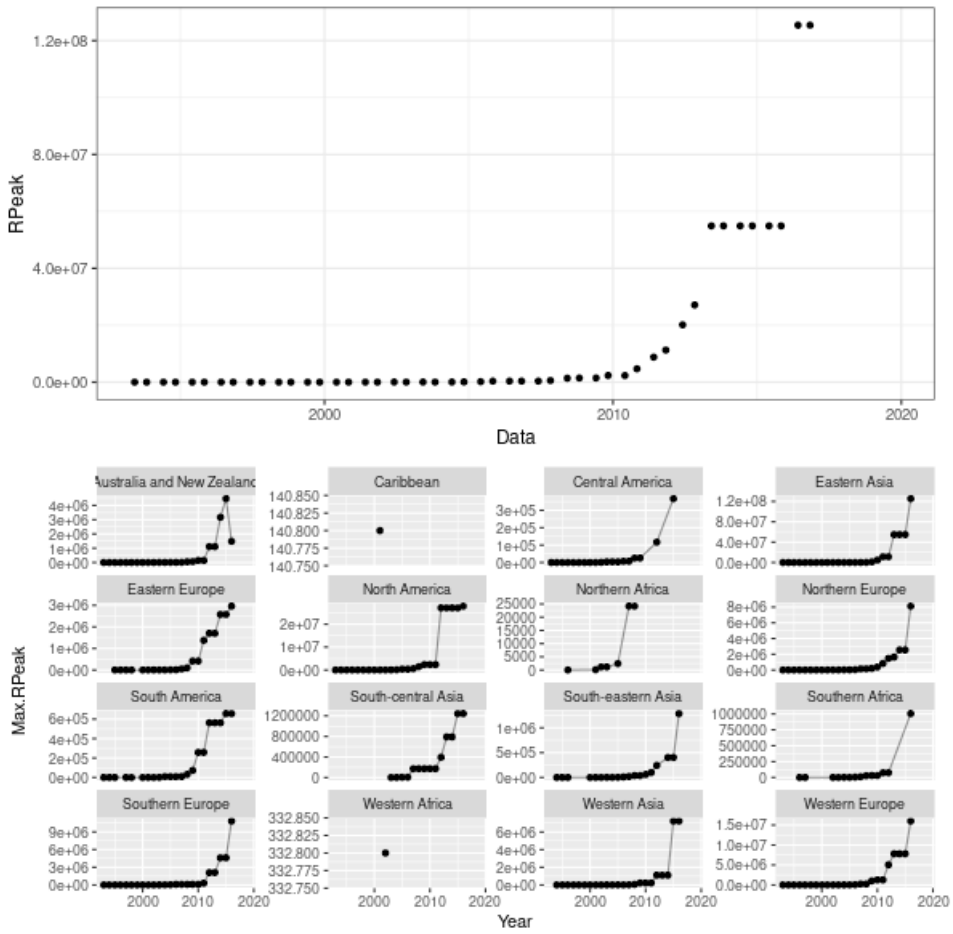


Agora temos gráficos que não são mais comparáveis visualmente entre si, pelo menos no que diz respeito ao eixo Y, mas onde podemos ver a evolução do máximo de RPeak em cada ano ao longo do tempo para cada região.

5.4.3. Composição de Gráficos

O pacote `patchwork` é utilizado para se combinar, ou compor, gráficos que tenha sido criados com `ggplot` mas que tenham características diferentes. Seu objetivo é associar os vários gráficos para compor uma única visualização. Para exemplificar seu uso, faremos uma combinação de dois gráficos gerados na seção anterior, identificados pelas variáveis `p1` e `p3`. No código abaixo, utilizamos o operador `/` para informar ao patchwork que os gráficos devem ficar uma acima do outro. Caso os quiséssemos lado a lado, poderíamos usar o operador `+`.

```
library(patchwork)
p1 / p3
```



5.4.4. Gráficos interativos

Os gráficos apresentados anteriormente são entidades estáticas. Embora suficientes para a maioria das situações, há casos em que é preciso ir além da visão geral. Tomando como exemplo o gráfico da Figura 5.1 podemos observar que é difícil distinguir entre os valores da série histórica no intervalo entre a data inicial até 2010. Tal dificuldade é consequência dos valores mais recentes, que por serem de maior grandeza, acabam por "achatar" os demais pontos. Ainda que possamos gerar novos gráficos estáticos reduzindo a faixa de valores, tal ação ainda será limitada a necessidade momentânea e pouco produtiva.

Uma solução para este e outros problemas é a utilização de gráficos interativos. Tais gráficos não substituem os equivalentes estáticos, porém são úteis para complementar as possibilidades de visualização e análise dos dados. Acrescentar interatividade permite ao usuário explorar o gráfico sob demanda, fazendo *zoom* em algumas partes, rolando o mouse para visualizar os valores absolutos de cada ponto ou ainda desativando temporariamente partes do conjunto de dados que possam estar em sobreposição com outras. O pacote `plotly` possibilita a geração de gráficos interativos que podem ser explorados em um navegador web. A função `ggplotly` realiza a tradução automática de um gráfico `ggplot2` para `plotly`. Desta maneira, podemos obter facilmente versões interativas dos gráficos estáticos criados anteriormente. A Figura 5.2 mostra uma captura de tela dos

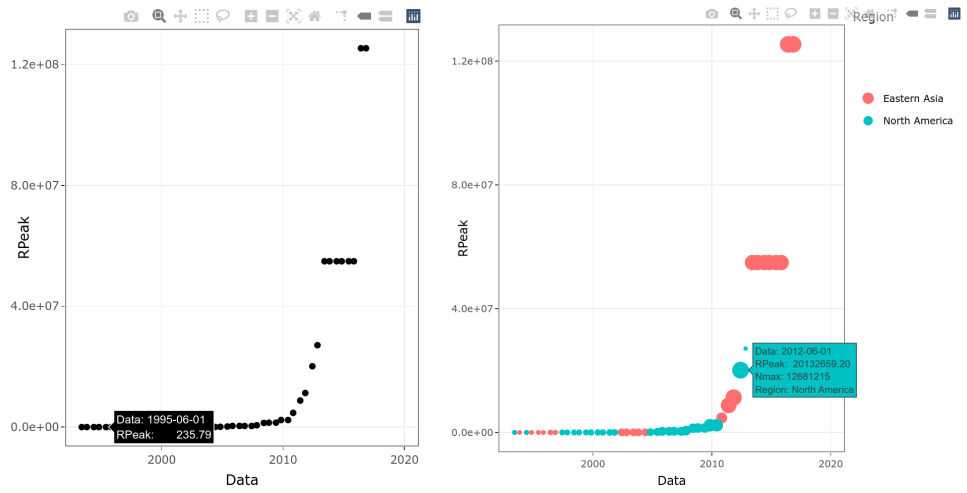


Figura 5.2. Captura de tela de dois gráficos interativos construídos com `plotly`.

dois gráficos interativos criados. As versões interativas podem ser acessadas na internet nos endereços `http://exp-hpc.gitlab.io/erad-2022/plotly-1.html` e `http://exp-hpc.gitlab.io/erad-2022/plotly-2.html`.

```
library(plotly)

ggplotly(p1)
ggplotly(p2)
```

5.5. Considerações Finais

Este capítulo ilustrou uma abordagem em `R` para apresentação de resultados experimentais para processamento de alto desempenho. Acreditamos que esta abordagem contribui para melhor documentação e reprodutibilidade dos resultados uma vez que todas as transformações aplicadas desde o processamento dos dados brutos até a construção dos gráficos pode ser facilmente rastreadas. A título de exemplo, todos os trechos de código apresentados neste texto podem ser acessados em: `http://exp-hpc.gitlab.io/erad-2022/companion.R`. Ao executar o mesmo é possível reproduzir as saídas dos trechos de código e as figuras.

Por fim, gostaríamos de lembrar que o assunto tratado neste capítulo tem estreita relação com outros três minicursos apresentados em edições recentes da ERAD/RS: “Boas Práticas para Experimentos” [Schnorr and Pinto 2019], “Boas Práticas para Experimentos Computacionais de Alto Desempenho” [Pinto et al. 2020] e “Are you root? Experimentos Reprodutíveis em Espaço de Usuário” [Dagostini et al. 2021]. Nós encorajamos as pessoas que desejam empregar essas boas práticas em seus experimentos a entrar em contato conosco para melhoramos esta série de cursos. Materiais complementares destes minicursos estão disponíveis em `https://exp-hpc.gitlab.io`.

Referências

[Bache and Wickham 2022] Bache, S. M. and Wickham, H. (2022). *magrittr: A Forward-Pipe Operator for R*. R package version 2.0.2.

[Dagostini et al. 2021] Dagostini, J. I., Pinto, V. G., Nesi, L. L., and Schnorr, L. M. (2021). Are you root? Experimentos Reprodutíveis em Espaço de Usuário. In Charão, A. and Serpa, M., editors, *Minicursos da XXI Escola Regional de Alto Desempenho da Região Sul*, chapter 3, pages 70–87. Sociedade Brasileira de Computação - SBC, Porto Alegre.

[Grolemund 2014] Grolemund, G. (2014). *Hands-On Programming with R*. O'Reilly Media, 1 edition.

[Herndon et al. 2014] Herndon, T., Ash, M., and Pollin, R. (2014). Does high public debt consistently stifle economic growth? a critique of reinhart and rogoff. *Cambridge Journal of Economics*, 38(2):257–279.

[Hutson 2010] Hutson, S. (2010). Data handling errors spur debate over clinical trial. *Nature Medicine*, 16(6):618–618.

[Müller and Wickham 2021] Müller, K. and Wickham, H. (2021). *tibble: Simple Data Frames*. R package version 3.1.6.

[Pedersen 2020] Pedersen, T. L. (2020). *patchwork: The Composer of Plots*. R package version 1.1.1.

[Pinto et al. 2020] Pinto, V. G., Nesi, L. L., and Schnorr, L. M. (2020). Boas Práticas para Experimentos Computacionais de Alto Desempenho. In du Bois, A. R. and Castro, M. B., editors, *Minicursos da XX Escola Regional de Alto Desempenho da Região Sul*, pages 1–19. Sociedade Brasileira de Computação - SBC.

[R Core Team 2021] R Core Team (2021). *R Language Definition*.

[Schnorr and Pinto 2019] Schnorr, L. M. and Pinto, V. G. (2019). Boas Práticas para Experimentos. In *Anais da XIX Escola Regional de Alto Desempenho da Região Sul*, pages 45–64. Sociedade Brasileira de Computação - SBC.

[Sievert 2020] Sievert, C. (2020). *Interactive Web-Based Data Visualization with R, plotly, and shiny*. Chapman and Hall/CRC.

[Venables et al. 2021] Venables, W. N., Smith, D. M., and R Core Team (2021). An Introduction to R. Technical report.

[Wickham 2010] Wickham, H. (2010). A layered grammar of graphics. *Journal of Computational and Graphical Statistics*, 19(1):3–28.

[Wickham 2014] Wickham, H. (2014). Tidy data. *The Journal of Statistical Software*, 59.

[Wickham 2016] Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York.

[Wickham et al. 2019] Wickham, H., Averick, M., Bryan, J., Chang, W., McGowan, L. D., François, R., Grolemund, G., Hayes, A., Henry, L., Hester, J., Kuhn, M., Pedersen, T. L., Miller, E., Bache, S. M., Müller, K., Ooms, J., Robinson, D., Seidel, D. P., Spinu, V., Takahashi, K., Vaughan, D., Wilke, C., Woo, K., and Yutani, H. (2019). Welcome to the tidyverse. *Journal of Open Source Software*, 4(43):1686.

[Wickham et al. 2021] Wickham, H., François, R., Henry, L., and Müller, K. (2021). *dplyr: A Grammar of Data Manipulation*. R package version 1.0.7.

[Wickham and Girlich 2022] Wickham, H. and Girlich, M. (2022). *tidyr: Tidy Messy Data*. R package version 1.2.0.

[Wickham and Hester 2021] Wickham, H. and Hester, J. (2021). *readr: Read Rectangular Text Data*. R package version 2.0.1.

[Wilkinson 2012] Wilkinson, L. (2012). The grammar of graphics. In *Handbook of computational statistics*, pages 375–414. Springer.