

Curso de Programação Estatística

Andressa Cerqueira, Rafael Izbicki, Thiago Ramos

2024-10-12

Table of contents

Curso de Programação Estatística	8
Objetivo Geral	8
1 Breve introdução ao Python	9
1.1 Variáveis e Tipos de Dados	9
1.2 O Comando <code>help()</code> em Python	10
1.3 Armazenando Resultados em Variáveis	11
1.4 Variáveis Lógicas	12
1.5 Caracteres/Strings em Python	14
1.6 Listas	16
1.7 Tuplas	17
1.8 Dicionários	17
1.9 Funções em Python	18
1.10 Loops	20
1.11 Estruturas Condicionais: <code>if</code> , <code>else</code> e <code>elif</code> em Python	22
1.12 Bibliotecas Essenciais para Estatística: NumPy, Pandas e Matplotlib	24
1.13 Exercícios	41
2 Breve Introdução ao R	44
2.1 Variáveis Numéricas	44
2.2 Variáveis Lógicas	46
2.3 Caracteres/Strings	48
2.4 Vetores e Sequências	49
2.5 Funções	52
2.6 Laços	54
2.7 If/Else	56
2.8 Listas	57
2.9 Estatística Descritiva	58
2.10 O operador <code>%>%</code> (pipe)	62
2.11 O pacote <code>dplyr</code>	64
2.12 O pacote <code>ggplot2</code>	70
2.13 O pacote <code>tidyr</code>	79
2.14 Leitura de Arquivos com <code>readr</code>	83
2.15 Exercícios	85

3	Geração de Números Aleatórios e Aplicação em Estatística	87
3.1	Objetivo da Aula	87
3.2	Conteúdo Teórico	87
3.3	Exemplo de Problema	87
4	R	88
5	Python	90
5.1	Exemplo: Simulação de um Jogo de Dados com Dado “Viciado”	91
6	R	93
7	Python	95
7.1	Pergunta: precisamos da função np.random.choice/set.seed?	96
8	R	97
9	Python	101
10	Números Pseudoaleatórios	104
10.1	O que é um número pseudoaleatório?	104
10.2	Geração de Números Pseudoaleatórios com o Gerador Linear Congruente (LCG)	104
11	R	106
12	Python	107
12.1	Por que o Gerador Linear Congruente Funciona?	107
13	R	110
14	Python	112
15	R	115
16	Python	117
16.1	Gerando números uniformes com uma moeda	118
17	R	119
18	Python	121
19	Técnica da Inversão para Variáveis Discretas	123
19.1	Geração de Variáveis Aleatórias Discretas Genéricas	123
20	R	124

21 Python	126
21.1 Inversa da CDF	127
21.2 Geração de Variáveis Aleatórias com Distribuição Geométrica	127
22 R	130
23 Python	133
23.1 Geração de Variáveis Aleatórias com Distribuição Poisson	135
24 R	137
25 Python	140
25.1 Exercícios	142
26 Técnica da Inversão para Variáveis Contínuas	144
26.1 Função Inversa	144
27 R	145
28 Python	147
28.1 Método da Inversão	149
28.2 Exemplo 1	149
29 R	150
30 Python	152
31 R	154
32 Python	156
32.1 Simulação de transformações de variáveis aleatórias	157
33 R	158
34 Python	160
35 R	162
36 Python	164
36.1 Exercícios	165
37 Método da Rejeição para Variáveis Discretas	167
38 Método da Rejeição para Variáveis Contínuas	168
38.1 Algoritmo	168

39 R	169
40 Python	171
40.1 Exemplo	173
41 R	174
42 Python	176
42.1 Resultados Teóricos	177
42.2 Exercícios	180
43 Transformações e Misturas	181
44 R	182
45 Python	183
45.1 Exercícios	183
46 Método de Box-Muller	184
47 R	185
48 Python	186
48.1 Exercícios	186
49 Método de Monte Carlo	187
49.1 Exemplo 1: Estimativa de uma Integral	188
50 R	189
51 Python	190
52 R	191
53 Python	193
53.1 Exemplo 2: Aproximando uma Probabilidade	194
54 R	195
55 Python	196
55.1 Exemplo 3: Aproximando o valor de π	196
56 R	198
57 Python	199

58 R	202
59 Python	204
59.1 Intervalos de Confiança para Estimativas de Monte Carlo	205
60 R	207
61 Python	208
61.1 Exemplo 4: Problema das Figurinhas	209
62 R	210
63 Python	211
64 R	212
65 Python	213
66 R	215
67 Python	216
68 R	217
69 Python	218
70 R	219
71 Python	220
71.1 Exercícios	220
72 Técnicas de Redução de Variância	222
73 R	223
74 Python	224
74.1 Exercícios	224
75 Amostragem por Importância	225
75.1 Exemplo 1: Estimativa de $\mathbb{P}(Z > 4.5)$	225
76 R	226
77 Python	227
77.1 Amostragem por Importância	228

78 R	229
79 Python	230
79.1 Exemplo 2: Estimativa de $\mathbb{E}[\sqrt{X}]$ com Amostragem por Importância	230
80 R	231
81 Python	232
82 R	234
83 Python	235
84 R	236
85 Python	238
86 R	240
87 Python	242
87.1 Exercícios	243
References	244

Curso de Programação Estatística

Objetivo Geral

Este curso visa explorar o impacto das representações numéricas nos resultados de algoritmos de análise estatística. O foco será na programação, visualização e preparação de dados, além de discutir tópicos importantes como aleatoriedade, pseudoaleatoriedade, erros de truncamento e arredondamento, entre outros. O curso inclui ainda uma introdução à inferência por simulação estocástica, utilizando métodos como Monte Carlo e integrações numéricas. O material do curso foi amplamente baseado nas discussões apresentadas em Ross (2006).

Este livro apresenta códigos tanto em [R](#) quanto em [Python](#)

Autores: [Andressa Cerqueira](#), [Rafael Izbicki](#), [Thiago Rodrigo Ramos](#)

1 Breve introdução ao Python

Aqui faremos uma breve introdução ao Python. Para uma introdução mais detalhada, você pode explorar: - [Documentação Oficial do Python](#) - [Real Python](#)

1.1 Variáveis e Tipos de Dados

Em Python, variáveis são usadas para armazenar informações que podem ser manipuladas ao longo do código. Existem diferentes tipos de dados que podem ser atribuídos a uma variável. Vamos explorar os principais tipos básicos em Python: `int`, `float`, `str`, `bool`, e `None`.

```
## Exemplo de variáveis e tipos de dados
nome = "Thiago"           ## String
idade = 35                ## Inteiro (int)
saldo_bancario = 1023.75  ## Float
estudante = True          ## Booleano (bool)
endereco = None           ## None, indicando ausência de valor

## Usando o método .format() para formatar a string
print("Nome: {} (Tipo: {})".format(nome, type(nome)))
print("Idade: {} (Tipo: {})".format(idade, type(idade)))
print("Saldo Bancário: {} (Tipo: {})".format(saldo_bancario, type(saldo_bancario)))
print("Estudante: {} (Tipo: {})".format(estudante, type(estudante)))
print("Endereço: {} (Tipo: {})".format(endereco, type(endereco)))
```

```
Nome: Thiago (Tipo: <class 'str'>)
Idade: 35 (Tipo: <class 'int'>)
Saldo Bancário: 1023.75 (Tipo: <class 'float'>)
Estudante: True (Tipo: <class 'bool'>)
Endereço: None (Tipo: <class 'NoneType'>)
```

1.1.1 f-strings: Formatação de Strings em Python (*)

Introduzidas na versão Python 3.6, as f-strings (ou “formatted string literals”) são uma forma eficiente e legível de formatar strings, permitindo a inclusão de expressões e variáveis diretamente dentro de uma string.

A sintaxe das f-strings utiliza a letra f antes da string e permite a inclusão de expressões dentro de chaves {}. Essas expressões são avaliadas em tempo de execução, e seus resultados são inseridos na string.

```
## Usando f-strings para formatar a string
print(f"Nome: {nome} (Tipo: {type(nome)})")
print(f"Idade: {idade} (Tipo: {type(idade)})")
print(f"Saldo Bancário: {saldo_bancario} (Tipo: {type(saldo_bancario)})")
print(f"Estudante: {estudante} (Tipo: {type(estudante)})")
print(f"Endereço: {endereco} (Tipo: {type(endereco)})")
```

```
Nome: Thiago (Tipo: <class 'str'>)
Idade: 35 (Tipo: <class 'int'>)
Saldo Bancário: 1023.75 (Tipo: <class 'float'>)
Estudante: True (Tipo: <class 'bool'>)
Endereço: None (Tipo: <class 'NoneType'>)
```

1.2 O Comando help() em Python

Python possui uma função embutida chamada help(), que é muito útil para obter informações sobre funções, módulos, objetos, e classes. Ele fornece uma explicação detalhada de como determinado elemento funciona, quais parâmetros aceita e o que retorna, entre outros detalhes. Essa função é especialmente útil quando você está começando ou precisa de uma rápida referência sem ter que sair do ambiente de desenvolvimento.

```
help(len)
```

```
Help on built-in function len in module builtins:
```

```
len(obj, /)
    Return the number of items in a container.
```

```
def exemplo():  
    """Esta é uma função de exemplo."""  
    pass  
  
help(exemplo)
```

Help on function exemplo in module __main__:

```
exemplo()  
    Esta é uma função de exemplo.
```

1.3 Armazenando Resultados em Variáveis

Em Python, as variáveis são usadas para armazenar resultados de cálculos ou operações para uso posterior no código. Ao armazenar um valor em uma variável, você pode acessá-lo facilmente quando precisar, sem ter que repetir o cálculo ou operação.

Quando você atribui um valor ou resultado de uma operação a uma variável, o Python guarda esse valor na memória e o associa ao nome que você escolheu para a variável. Isso permite que você reutilize o valor sempre que necessário.

```
resultado = 5 + 3  ## Armazenando a soma de 5 e 3 em uma variável  
print(resultado)   ## Resultado: 8
```

8

```
x = 2 + 3  
print(x)
```

5

```
y = 2 * x  
y
```

10

1.4 Variáveis Lógicas

Python, assim como R, possui operadores lógicos para comparar valores e trabalhar com variáveis booleanas. Abaixo estão alguns dos principais operadores lógicos usados em Python:

##	Operador	Descrição
1	<code>x < y</code>	x é menor que y?
2	<code>x <= y</code>	x é menor ou igual a y?
3	<code>x > y</code>	x é maior que y?
4	<code>x >= y</code>	x é maior ou igual a y?
5	<code>x == y</code>	x é igual a y?
6	<code>x != y</code>	x é diferente de y?
7	<code>not x</code>	Negativa de x (inverte o valor lógico)
8	<code>x or y</code>	x ou y são verdadeiros? (ou inclusivo)
9	<code>x and y</code>	x e y são verdadeiros? (e lógico)
10	<code>x ^ y</code>	x ou y, mas não ambos, são verdadeiros? (xor lógico)

```
a = 5
b = 10
print(a < b)  ## True
print(a == b) ## False
```

True
False

```
x = True
print(not x)  ## False
```

False

```
x = False
y = True
print(x or y)  ## True
```

True

```
x = True
y = False
print(x and y)  ## False
```

False

```
x = True
y = False
print(x ^ y)  ## True (apenas um dos dois é verdadeiro)
```

True

1.4.1 Cuidado!

Em Python, o operador \wedge não é usado para exponenciação. Em vez disso, ele é o operador bitwise XOR (ou exclusivo) para manipulação de bits.

O operador XOR (ou exclusivo) compara os bits de dois números. Ele retorna 1 quando os bits são diferentes e 0 quando os bits são iguais.

```
a = 5  ## Em binário: 101
b = 3  ## Em binário: 011

resultado = a ** b  ## Faz XOR bit a bit

print(resultado)  ## Saída: 6 (Em binário: 110)
```

125

```
a = 5  ## Em binário: 101
b = 3  ## Em binário: 011

resultado = a ^ b  ## Faz XOR bit a bit

print(resultado)  ## Saída: 6 (Em binário: 110)
```

6

1.5 Caracteres/Strings em Python

Em Python, uma string é uma sequência de caracteres que pode ser usada para armazenar e manipular textos. Strings são um dos tipos de dados mais comuns e úteis, e Python oferece uma grande variedade de métodos e operações para trabalhar com elas.

As strings em Python podem ser definidas de diferentes maneiras, usando aspas simples (') ou aspas duplas (").

```
## Definindo strings
nome = 'Thiago'
cidade = "São Carlos"
```

- Concatenar Strings: Você pode unir duas ou mais strings usando o operador +.

```
## Concatenação
nome_completo = "Thiago" + " " + "Rodrigo"
print(nome_completo) ## Resultado: Thiago Rodrigo
```

Thiago Rodrigo

- Repetir Strings: Você pode repetir uma string múltiplas vezes usando o operador *.

```
repeticao = "Oi! " * 3
print(repeticao) ## Resultado: Oi! Oi! Oi!
```

Oi! Oi! Oi!

- Acessar Caracteres por Índice: As strings em Python são indexadas, e você pode acessar um caractere específico usando o índice (começando em 0).

```
nome = "Thiago"
print(nome[0]) ## Resultado: T
print(nome[-1]) ## Resultado: o (último caractere)
```

T
o

- Fatiar Strings (Slicing): Você pode pegar uma parte da string usando fatias (substrings).

```
nome = "Thiago"
print(nome[0:3]) ## Resultado: Thi (caracteres do índice 0 ao 2)
```

Thi

- Comprimento de Strings: Para saber quantos caracteres uma string tem, use a função `len()`.

```
nome = "Thiago"
print(len(nome)) ## Resultado: 6
```

6

1.5.1 Métodos Úteis para Strings

- `lower()` e `upper()`: Convertem todas as letras para minúsculas ou maiúsculas, respectivamente.

```
nome = "Thiago"
print(nome.lower()) ## Resultado: thiago
print(nome.upper()) ## Resultado: THIAGO
```

thiago

THIAGO

- `strip()`: Remove espaços em branco no início e no final da string.

```
frase = " Olá! "
```

```
print(frase.strip()) ## Resultado: Olá!
```

Olá!

- `replace()`: Substitui parte de uma string por outra.

```
frase = "Eu gosto de R"
nova_frase = frase.replace("R", "Python")
print(nova_frase) ## Resultado: Eu gosto de programação
```

Eu gosto de Python

- `split()`: Divide uma string em uma lista, utilizando um delimitador (por padrão, espaço).

```
frase = "Eu gosto de Python"
palavras = frase.split()
print(palavras)  ## Resultado: ['Eu', 'gosto', 'de', 'Python']
```

```
['Eu', 'gosto', 'de', 'Python']
```

- `join()`: Une uma lista de strings em uma única string, usando um delimitador.

```
lista = ['Thiago', 'Rodrigo', 'Ramos']
nome_completo = " ".join(lista)
print(nome_completo)  ## Resultado: Thiago Rodrigo Ramos
```

Thiago Rodrigo Ramos

1.6 Listas

Em Python, listas são coleções ordenadas e mutáveis, o que significa que você pode modificar os elementos após sua criação. Elas são definidas usando colchetes `[]` e podem armazenar múltiplos tipos de dados, como inteiros, strings ou até mesmo outras listas.

Características principais:

- Mutáveis: você pode adicionar, remover ou modificar elementos.
- Ordenadas: os elementos mantêm a ordem em que são inseridos.
- Acesso por índice: os elementos podem ser acessados pelo índice, começando por 0.

```
## Criando uma lista
frutas = ['maçã', 'banana', 'laranja']

## Acessando elementos
print(frutas[1])  ## banana

## Modificando a lista
frutas.append('uva')  ## Adiciona 'uva' ao final
frutas[0] = 'kiwi'    ## Substitui 'maçã' por 'kiwi'

## Removendo um elemento
frutas.remove('banana')

print(frutas)  ## ['kiwi', 'laranja', 'uva']
```



```
banana
['kiwi', 'laranja', 'uva']
```

1.7 Tuplas

As tuplas em Python são semelhantes às listas, porém, diferentemente das listas, são imutáveis, ou seja, seus elementos não podem ser modificados após a criação. Elas são úteis quando você deseja garantir que uma sequência de valores permaneça inalterada. As tuplas são definidas usando parênteses (). Características principais:

- Imutáveis: uma vez criadas, seus elementos não podem ser alterados, adicionados ou removidos.
- Ordenadas: os elementos mantêm a ordem de inserção.
- Acesso por índice: assim como nas listas, os elementos podem ser acessados por índices, começando do 0.

```
## Criando uma tupla
coordenadas = (10, 20)

## Acessando elementos
print(coordenadas[0])  ## 10

## Tuplas podem armazenar diferentes tipos de dados
dados = ('Thiago', 30, True)

print(dados)  ## ('Thiago', 30, True)
```

```
10
('Thiago', 30, True)
```

```
### Tentando modificar a tupla (resultará em erro)
## coordenadas[0] = 15  # TypeError: 'tuple' object does not support item assignment
```

1.8 Dicionários

Os dicionários em Python são coleções não ordenadas de pares chave-valor. Eles permitem associar valores a uma chave específica, sendo muito úteis quando você precisa acessar elementos por meio de uma chave, em vez de um índice. Eles são definidos com chaves {}.

Características principais:

- Mutáveis: você pode adicionar, remover ou modificar pares chave-valor.
- Não ordenados: a ordem dos elementos não é garantida nas versões anteriores ao Python 3.7.
- Acesso por chave: os valores são acessados por suas chaves, que podem ser de tipos imutáveis (como strings ou números).

```
## Criando um dicionário
estudante = {
    'nome': 'Ana',
    'idade': 22,
    'curso': 'Estatística'
}

## Acessando valores
print(estudante['nome']) ## Ana

## Modificando o dicionário
estudante['idade'] = 23 ## Atualiza o valor da chave 'idade'

## Adicionando um novo par chave-valor
estudante['matricula'] = 12345

## Removendo um elemento
del estudante['curso']

print(estudante) ## {'nome': 'Ana', 'idade': 23, 'matricula': 12345}
```

Ana

```
{'nome': 'Ana', 'idade': 23, 'matricula': 12345}
```

1.9 Funções em Python

As funções em Python são blocos de código reutilizáveis que realizam uma tarefa específica. Elas ajudam a organizar o código, tornando-o mais modular e legível. Você pode definir suas próprias funções usando a palavra-chave `def`, e elas podem receber parâmetros, retornar valores ou simplesmente executar uma ação. Características principais:

- Reutilizáveis: uma vez definida, a função pode ser chamada várias vezes no código.
- Modulares: permitem dividir o código em partes menores e mais organizadas.

- Parâmetros opcionais ou obrigatórios: funções podem receber parâmetros (ou argumentos) para realizar operações com base neles.

```
def saudacao(nome):  
    """Exibe uma saudação personalizada."""  
    print(f"Olá, {nome}!")  
  
## Chamando a função  
saudacao("Thiago")  ## Olá, Thiago!
```

Olá, Thiago!

1.9.1 Return

Funções podem retornar valores usando a palavra-chave `return`. Isso permite que o resultado da função seja usado em outras partes do código.

```
def soma(a, b):  
    """Retorna a soma de dois números."""  
    return a + b  
  
resultado = soma(3, 5)  
print(resultado)  ## 8
```

8

1.9.2 Parâmetros Opcionais e Valores Padrão

Você pode definir parâmetros opcionais em uma função, atribuindo valores padrão a eles.

```
def saudacao(nome="amigo"):  
    """Exibe uma saudação personalizada, com um nome padrão."""  
    print(f"Olá, {nome}!")  
  
saudacao()          ## Olá, amigo!  
saudacao("Rafael")  ## Olá, Rafael!
```

Olá, amigo!
Olá, Rafael!

1.9.3 Lambda

Python também oferece funções anônimas (ou funções lambda), que são funções curtas e sem nome. Elas são úteis quando você precisa de uma função simples e temporária.

```
## Função lambda para multiplicar dois números
multiplicar = lambda x, y: x * y
print(multiplicar(3, 4))  ## 12
```

12

1.10 Loops

Os loops são estruturas fundamentais de programação que permitem executar um bloco de código repetidamente enquanto uma condição for verdadeira ou para cada item de uma sequência. Python oferece dois tipos principais de loops: for e while.

1.10.1 for

O loop for é usado para iterar sobre uma sequência (como uma lista, tupla, string ou range) e executar um bloco de código para cada item dessa sequência. É especialmente útil quando o número de iterações é conhecido ou quando você deseja percorrer uma estrutura de dados.

```
## Iterando sobre uma lista
frutas = ['maçã', 'banana', 'laranja']
for fruta in frutas:
    print(fruta)
```

maçã
banana
laranja

```
## Iterando de 0 a 4
for i in range(5):
    print(i)
```

0
1
2
3
4

```
frutas = ['maçã', 'banana', 'laranja']  
for i, fruta in enumerate(frutas):  
    print(i, fruta)
```

0 maçã
1 banana
2 laranja

1.10.2 while

O loop while executa um bloco de código repetidamente enquanto uma condição for verdadeira. Ele é útil quando o número de iterações não é conhecido antecipadamente e depende de uma condição dinâmica.

```
## Loop enquanto o valor de x for menor que 5  
x = 0  
while x < 5:  
    print(x)  
    x += 1
```

0
1
2
3
4

1.10.3 Controle de Loops: break e continue

break: interrompe o loop imediatamente, mesmo que a condição ainda seja verdadeira (no caso do while) ou ainda restem itens na sequência (no caso do for).

continue: pula para a próxima iteração do loop, ignorando o código restante naquela iteração.

```
for i in range(10):
    if i == 5:
        break ## Interrompe o loop quando i é igual a 5
    print(i)
```

0
1
2
3
4

```
for i in range(5):
    if i == 3:
        continue ## Pula a iteração quando i é igual a 3
    print(i)
```

0
1
2
4

1.11 Estruturas Condicionais: if, else e elif em Python

As estruturas condicionais em Python permitem que o programa tome decisões e execute blocos de código diferentes com base em condições específicas. As principais instruções condicionais são if, else e elif.

1.11.1 if

A instrução if é usada para executar um bloco de código se uma condição for verdadeira. Se a condição avaliada for True, o bloco de código indentado após o if será executado.

```
idade = 20
if idade >= 18:
    print("Você é maior de idade")
```

Você é maior de idade

1.11.2 else

A instrução else é usada para executar um bloco de código se a condição do if for falsa. É como uma segunda opção, caso a primeira condição não seja atendida.

```
idade = 16
if idade >= 18:
    print("Você é maior de idade")
else:
    print("Você é menor de idade")
```

Você é menor de idade

1.11.3 elif

A instrução elif (abreviação de else if) permite testar múltiplas condições. Se a condição if for falsa, o Python verificará a condição do elif. Você pode ter vários blocos elif em uma estrutura condicional.

```
nota = 85
if nota >= 90:
    print("Aprovado com excelência")
elif nota >= 70:
    print("Aprovado")
else:
    print("Reprovado")
```

Aprovado

1.11.4 Operadores lógicos

As condições podem usar operadores lógicos para combinar mais de uma verificação:

- and: todas as condições devem ser verdadeiras.
- or: pelo menos uma condição deve ser verdadeira.
- not: inverte o resultado da condição.

```
idade = 20
possui_carteira = True

if idade >= 18 and possui_carteira:
    print("Você pode dirigir")
else:
    print("Você não pode dirigir")
```

Você pode dirigir

1.12 Bibliotecas Essenciais para Estatística: NumPy, Pandas e Matplotlib

Em Python, as bibliotecas NumPy, Pandas, e Matplotlib são amplamente utilizadas para análise de dados e computação científica. Elas fornecem ferramentas poderosas para lidar com grandes volumes de dados, realizar cálculos matemáticos eficientes e criar visualizações de alta qualidade. Vamos detalhar cada uma delas:

1.12.1 NumPy

NumPy (Numerical Python) é uma biblioteca fundamental para cálculos matemáticos em Python. Ela fornece suporte para arrays e matrizes multidimensionais, além de uma coleção de funções para operações com esses arrays. Principais características:

- Array multidimensional (ndarray): O ndarray é a estrutura central da NumPy, oferecendo suporte a vetores e matrizes de várias dimensões, com operações eficientes em termos de memória e tempo de execução.
- Operações vetorizadas: Permite realizar operações em todos os elementos de um array sem a necessidade de loops explícitos.
- Funções matemáticas: Inclui uma vasta gama de funções para álgebra linear, estatística, trigonometria, entre outros.

```
import numpy as np

## Criando um array NumPy
array = np.array([1, 2, 3, 4, 5])

## Operações elementares
print(array * 2)  ## Multiplicação por escalar: [2 4 6 8 10]
```



```
## Matrizes e operações matriciais
matriz = np.array([[1, 2], [3, 4]])
print(np.dot(matriz, matriz)) ## Multiplicação de matrizes
```

```
[ 2  4  6  8 10]
[[ 7 10]
 [15 22]]
```

```
dados = np.array([1, 2, 3, 4, 5])
media = np.mean(dados)
print(media) ## Saída: 3.0
```

3.0

```
mediana = np.median(dados)
print(mediana) ## Saída: 3.0
```

3.0

```
variancia = np.var(dados)
print(variancia) ## Saída: 2.0
```

2.0

```
desvio_padrao = np.std(dados)
print(desvio_padrao) ## Saída: 1.4142135623730951
```

1.4142135623730951

```
minimo = np.min(dados)
maximo = np.max(dados)
print(minimo, maximo) ## Saída: 1 5
```

1 5

```
p25 = np.percentile(dados, 25)
p50 = np.percentile(dados, 50)
p75 = np.percentile(dados, 75)
print(p25, p50, p75)  ## Saída: 2.0 3.0 4.0
```

2.0 3.0 4.0

1.12.2 Pandas

Pandas é uma biblioteca poderosa para manipulação de dados em Python, frequentemente usada em projetos de ciência de dados, análise estatística e processamento de dados tabulares. Ela oferece estruturas de dados como DataFrames e Series que facilitam a organização, limpeza e análise de dados, tornando-a uma das ferramentas mais populares para lidar com grandes volumes de dados.

1.12.2.1 Séries

Uma Series é uma coluna de dados unidimensional, semelhante a um array de NumPy, mas com rótulos (índices) associados a cada valor. A Series pode armazenar qualquer tipo de dado, como inteiros, floats, strings, ou objetos.

```
import pandas as pd

## Criando uma Series
s = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])
print(s)

## Acessando elementos por índice
print(s['b'])  ## Saída: 2
```

```
a    1
b    2
c    3
d    4
e    5
dtype: int64
2
```

1.12.2.2 DataFrame

O DataFrame é a estrutura de dados mais importante do Pandas. Ele é uma tabela bidimensional (semelhante a uma planilha ou tabela SQL) com rótulos para as linhas e colunas. Cada coluna de um DataFrame é uma Series, e as colunas podem ter diferentes tipos de dados.

```
## Criando um DataFrame a partir de um dicionário
dados = {
    'Nome': ['Ana', 'Pedro', 'João'],
    'Idade': [23, 34, 19],
    'Cidade': ['São Paulo', 'Rio de Janeiro', 'Curitiba']
}

df = pd.DataFrame(dados)
print(df)
```

	Nome	Idade	Cidade
0	Ana	23	São Paulo
1	Pedro	34	Rio de Janeiro
2	João	19	Curitiba

1.12.2.3 Operações Essenciais com DataFrames

1. Selecionar Colunas
2. Filtragem de Dados
3. Alteração de Dados
4. Agrupamento e Agregação
5. Tratamento de Dados Faltantes

```
## Selecionando uma única coluna
print(df['Nome'])
```

```
0      Ana
1    Pedro
2     João
Name: Nome, dtype: object
```

```
## Selecionando múltiplas colunas
print(df[['Nome', 'Cidade']])
```

	Nome	Cidade
0	Ana	São Paulo
1	Pedro	Rio de Janeiro
2	João	Curitiba

```
## Filtrando o DataFrame para mostrar apenas pessoas com mais de 20 anos
filtro = df[df['Idade'] > 20]
print(filtro)
```

	Nome	Idade	Cidade
0	Ana	23	São Paulo
1	Pedro	34	Rio de Janeiro

```
## Alterando o valor de uma célula
df.loc[0, 'Idade'] = 24
## Alterando uma coluna inteira
df['Idade'] = df['Idade'] + 1
```

```
## Agrupando por 'Cidade' e calculando a média de 'Idade'
agrupado = df.groupby('Cidade')['Idade'].mean()
print(agrupado)
```

Cidade	
Curitiba	20.0
Rio de Janeiro	35.0
São Paulo	25.0

Name: Idade, dtype: float64

```
## Verificando valores faltantes
print(df.isnull().sum())

## Removendo linhas com dados faltantes
df_limpo = df.dropna()

## Preenchendo valores faltantes
df['Idade'].fillna(df['Idade'].mean())
```

Nome	0
Idade	0
Cidade	0

dtype: int64

```
0    25
1    35
2    20
Name: Idade, dtype: int64
```

1.12.2.4 Estatísticas e Operações com Dados

Pandas facilita o cálculo de estatísticas descritivas, como média, desvio padrão, contagem, entre outras.

```
## Estatísticas descritivas
print(df.describe())
```

	Idade
count	3.000000
mean	26.666667
std	7.637626
min	20.000000
25%	22.500000
50%	25.000000
75%	30.000000
max	35.000000

```
## Soma, contagem e média de colunas específicas
print(df['Idade'].sum())
print(df['Idade'].count())
print(df['Idade'].mean())
```

```
80
3
26.666666666666668
```

1.12.3 Matplotlib: Biblioteca de Visualização de Dados em Python

Matplotlib é uma biblioteca poderosa e flexível para visualização de dados em Python. Ela permite criar uma ampla gama de gráficos, desde simples gráficos de linha até visualizações complexas e altamente customizadas. Embora outras bibliotecas como Seaborn ou Plotly sejam populares para visualizações avançadas, Matplotlib continua sendo o núcleo para muitas dessas bibliotecas e é amplamente utilizado por sua versatilidade e integração com NumPy e Pandas.

A maneira mais comum de usar Matplotlib é através do submódulo pyplot, que fornece uma interface simples para criar gráficos. A estrutura básica de um gráfico com pyplot envolve definir os dados e criar o gráfico com funções que controlam o comportamento do gráfico.

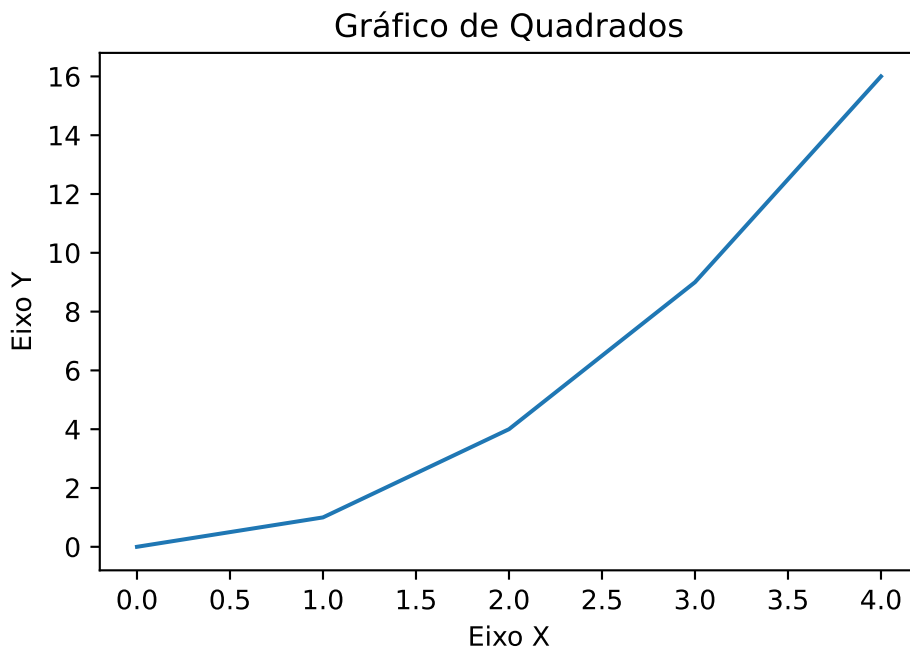
```
import matplotlib.pyplot as plt

## Dados
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]

## Criando um gráfico de linha
plt.plot(x, y)

## Adicionando título e rótulos aos eixos
plt.title('Gráfico de Quadrados')
plt.xlabel('Eixo X')
plt.ylabel('Eixo Y')

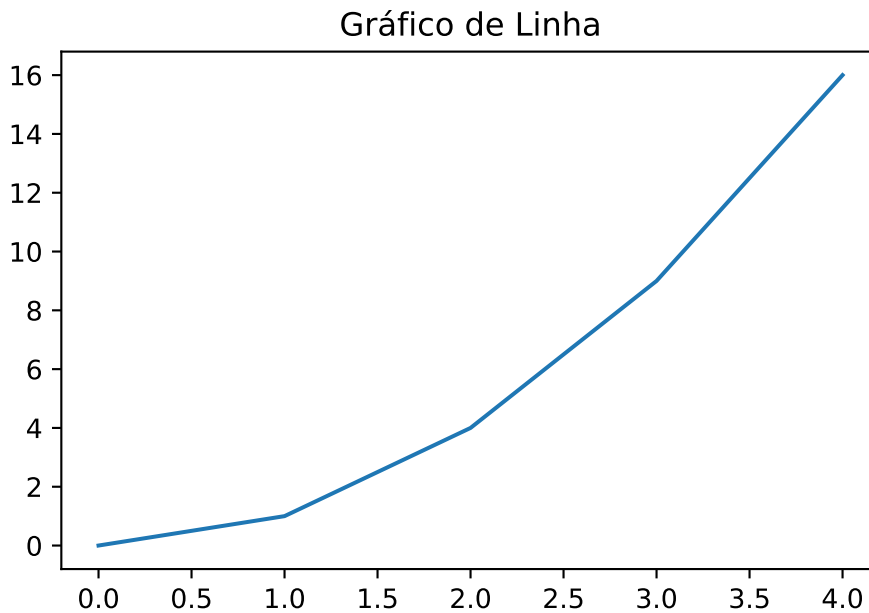
## Exibindo o gráfico
plt.show()
```



1.12.3.1 Tipos de Gráficos Comuns

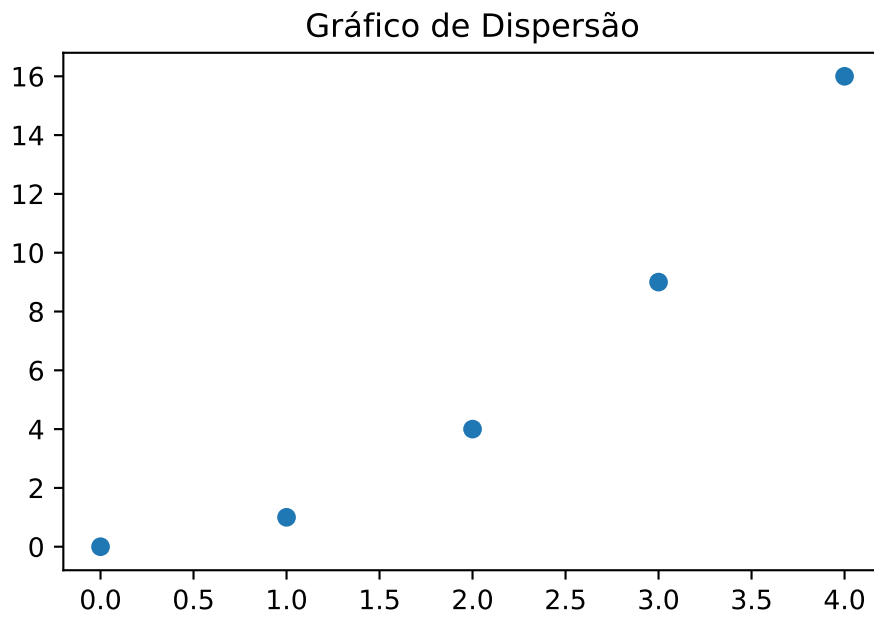
1.12.3.1.1 Linha

```
x = [0, 1, 2, 3, 4]
y = [0, 1, 4, 9, 16]
plt.plot(x, y)
plt.title('Gráfico de Linha')
plt.show()
```



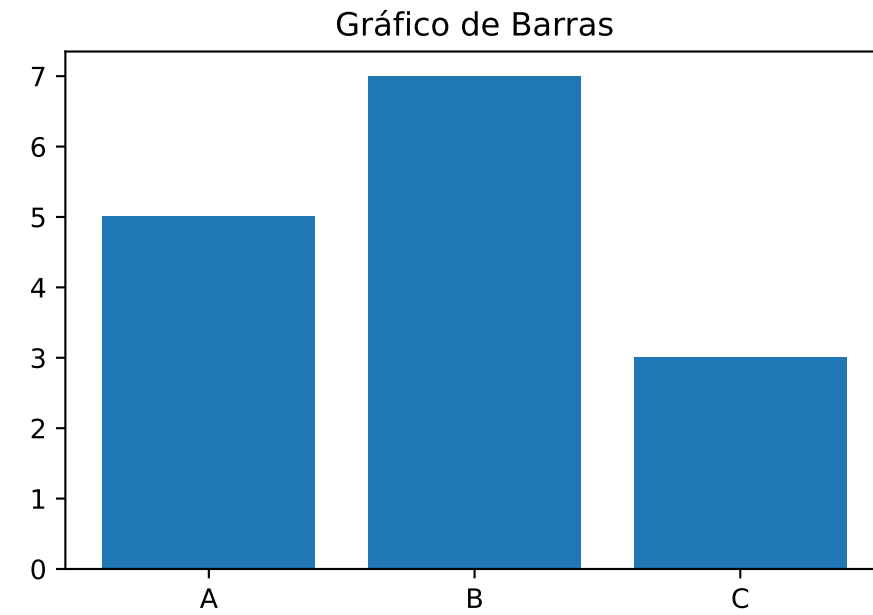
1.12.3.1.2 Dispersão

```
plt.scatter(x, y)
plt.title('Gráfico de Dispersão')
plt.show()
```



1.12.3.1.3 Barras

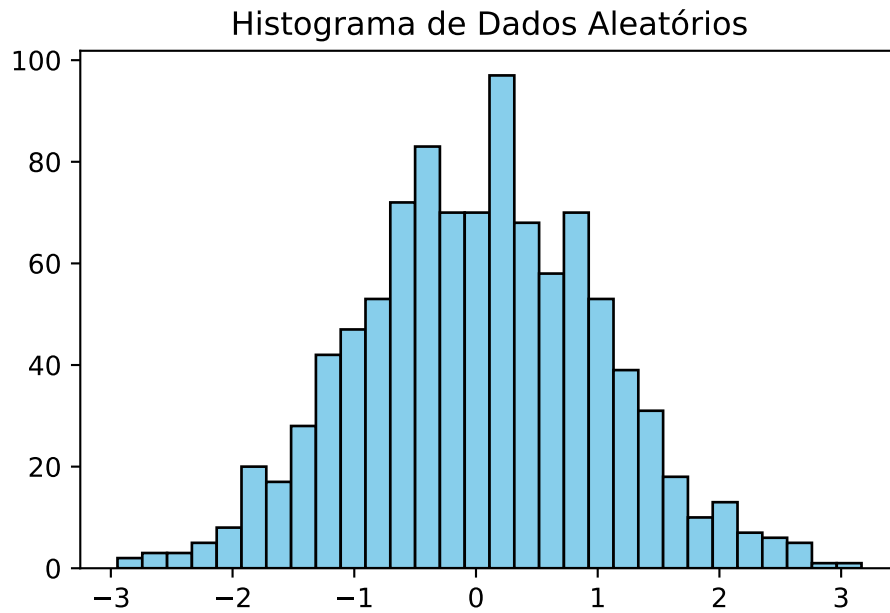
```
categorias = ['A', 'B', 'C']  
valores = [5, 7, 3]  
plt.bar(categorias, valores)  
plt.title('Gráfico de Barras')  
plt.show()
```

1.12.3.1.4 Histograma

```
import numpy as np

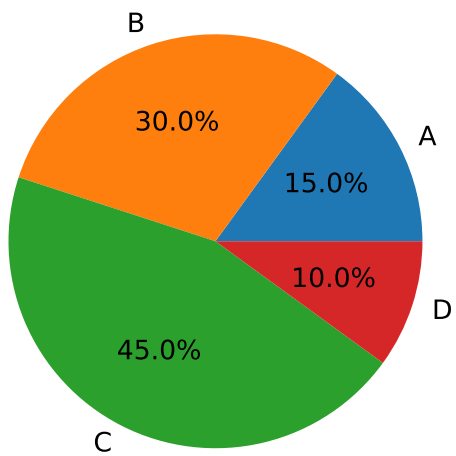
dados = np.random.randn(1000)
plt.hist(dados, bins=30, color='skyblue', edgecolor='black')
plt.title('Histograma de Dados Aleatórios')
plt.show()
```



1.12.3.1.5 Pizza

```
tamanhos = [15, 30, 45, 10]
labels = ['A', 'B', 'C', 'D']
plt.pie(tamanhos, labels=labels, autopct='%1.1f%%')
plt.title('Gráfico de Pizza')
plt.show()
```

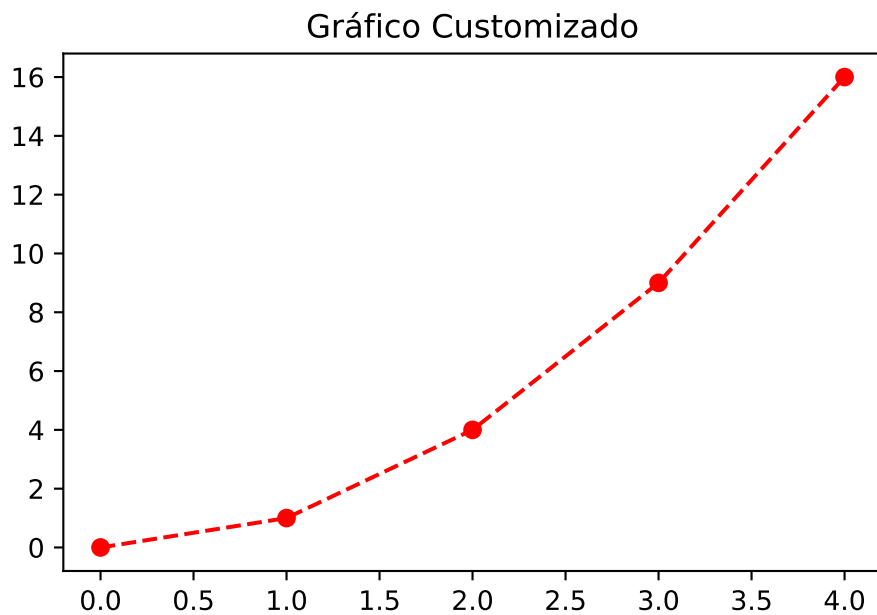
Gráfico de Pizza



1.12.3.2 Customização

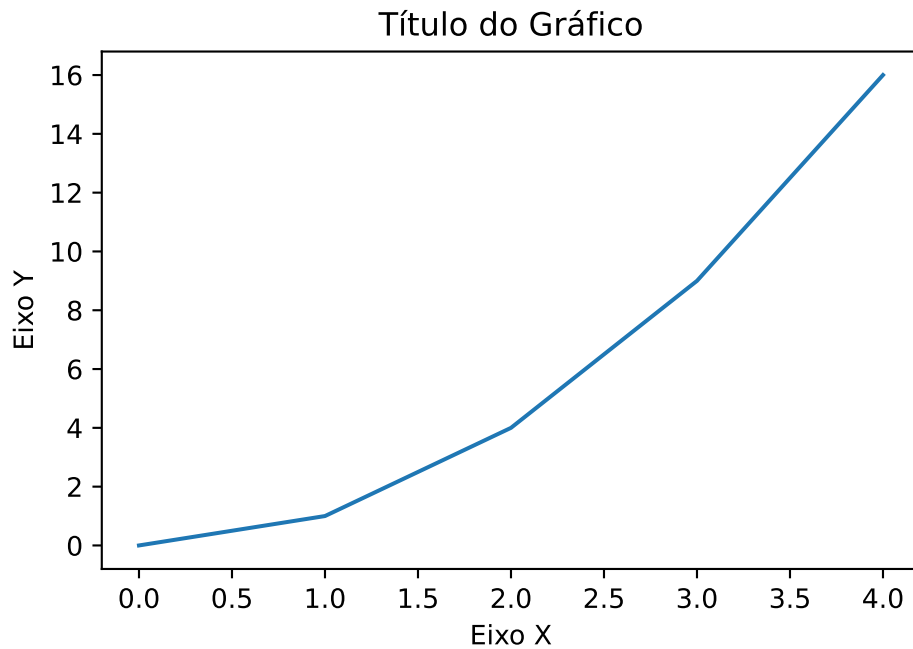
1.12.3.2.1 Alterando Cores e Estilos de Linha

```
plt.plot(x, y, color='red', linestyle='--', marker='o') ## Linha vermelha com marcador  
plt.title('Gráfico Customizado')  
plt.show()
```



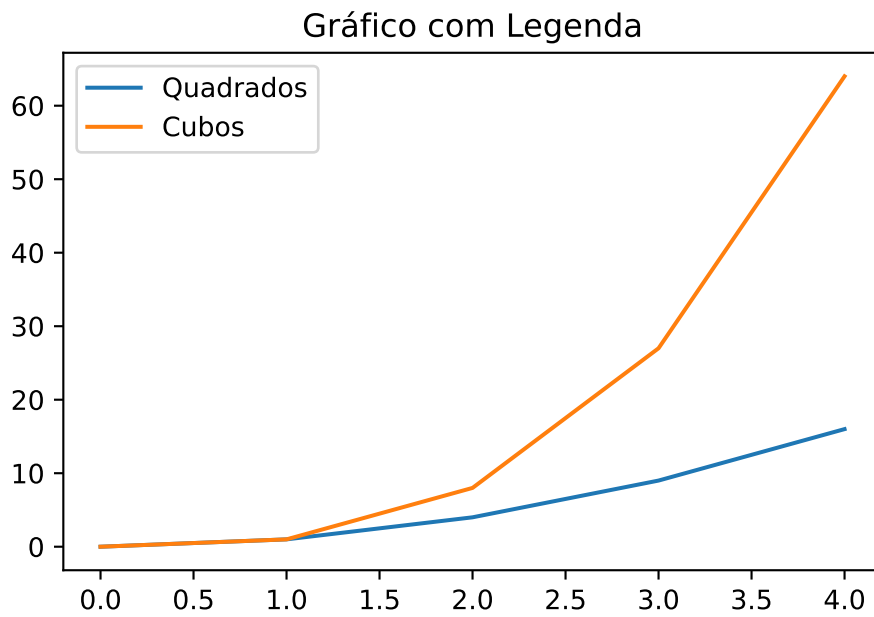
1.12.3.2.2 Adicionando Títulos e Rótulos aos Eixos

```
plt.plot(x, y)
plt.title('Título do Gráfico')
plt.xlabel('Eixo X')
plt.ylabel('Eixo Y')
plt.show()
```



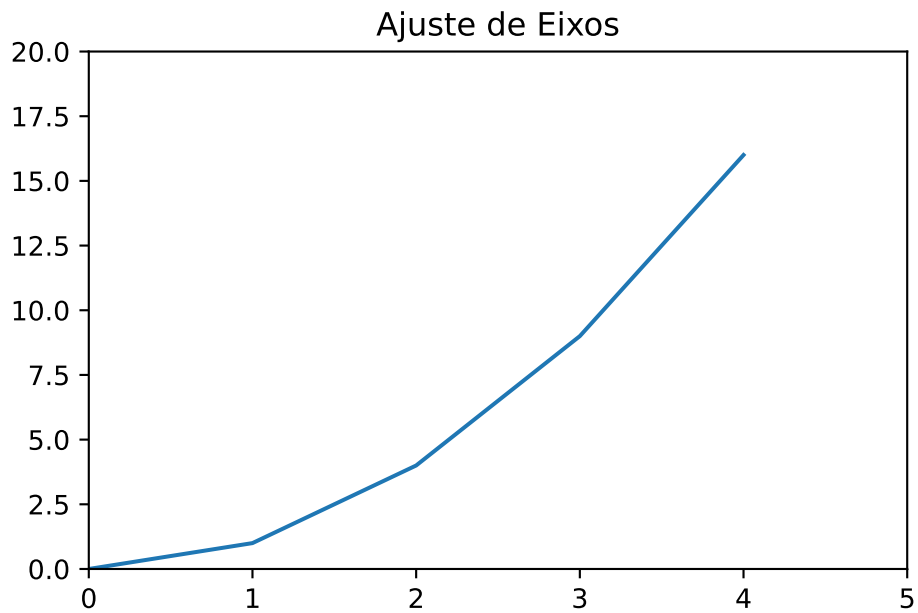
1.12.3.2.3 Legendas

```
plt.plot(x, y, label='Quadrados')
plt.plot(x, [i**3 for i in x], label='Cubos')
plt.title('Gráfico com Legenda')
plt.legend()  ## Adiciona legenda
plt.show()
```



1.12.3.2.4 Ajustando os Eixos

```
plt.plot(x, y)
plt.xlim(0, 5)
plt.ylim(0, 20)
plt.title('Ajuste de Eixos')
plt.show()
```



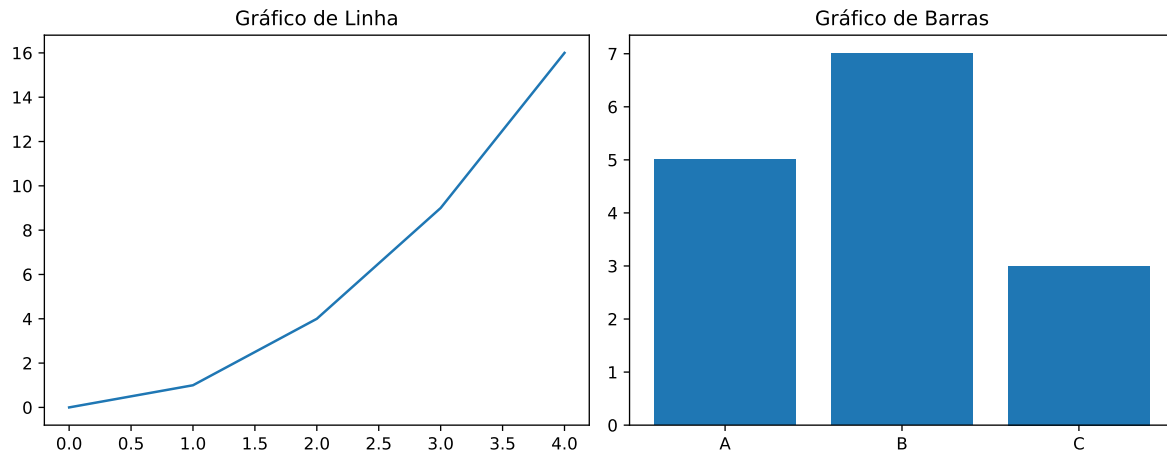
1.12.3.2.5 Subplot

```
## Criando uma figura com dois gráficos lado a lado
plt.figure(figsize=(10, 4))

plt.subplot(1, 2, 1)  ## 1 linha, 2 colunas, 1º gráfico
plt.plot(x, y)
plt.title('Gráfico de Linha')

plt.subplot(1, 2, 2)  ## 1 linha, 2 colunas, 2º gráfico
plt.bar(categorias, valores)
plt.title('Gráfico de Barras')

plt.tight_layout()
plt.show()
```

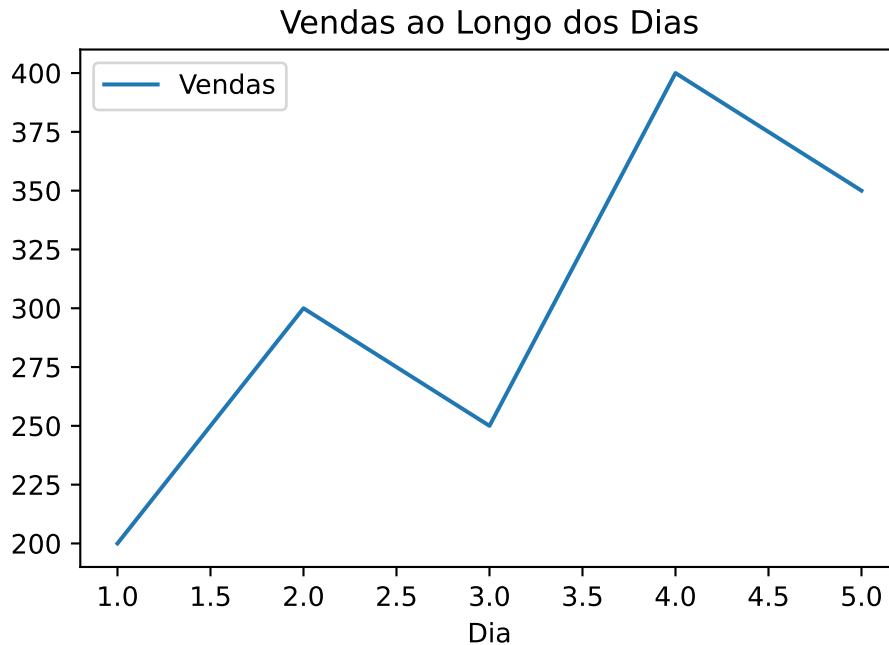


1.12.3.3 Integração com Pandas

```
import pandas as pd

## Criando um DataFrame
dados = {
    'Dia': [1, 2, 3, 4, 5],
    'Vendas': [200, 300, 250, 400, 350]
}
df = pd.DataFrame(dados)

## Plotando os dados
df.plot(x='Dia', y='Vendas', kind='line')
plt.title('Vendas ao Longo dos Dias')
plt.show()
```

1.13 Exercícios

1.13.1 Exercício 1.

Execute as seguintes operações no Python e interprete os resultados:

- Use a função `math.log()` para calcular o logaritmo natural de 10 e depois o logaritmo na base 10 de 1000.
- Descubra a função que calcula a raiz quadrada de um número utilizando a função `help()`, e aplique-a ao número 144.

1.13.2 Exercício 2.

Armazene o resultado de `2 * 7` em uma variável chamada `resultado`. Use `resultado` para calcular o valor de `resultado ** 2`.

Liste todas as variáveis no ambiente do Python usando a função `globals()`. Exclua a variável `resultado` usando `del` e verifique novamente as variáveis presentes.

1.13.3 Exercício 3.

Crie um vetor NumPy `v` contendo os números 4, 8, 15, 16, 23, 42. Use `subsetting` para selecionar: - O segundo e o quarto elemento. - Todos os elementos exceto o terceiro. - Os elementos que são maiores que 10.

Multiplique cada elemento do vetor NumPy `v` por 2 e armazene o resultado em um novo vetor `v2`.

1.13.4 Exercício 4.

Verifique se `8 > 5` e se `8 == 10`.

Crie dois vetores NumPy lógicos `a = np.array([True, False, True])` e `b = np.array([False, True, False])`. Aplique os operadores `&`, `|` e `np.logical_xor()` nesses vetores.

1.13.5 Exercício 5.

Escreva uma função `soma_quadrados` que receba dois números como argumentos e retorne a soma de seus quadrados. Teste sua função com os números 3 e 4.

Modifique a função `soma_quadrados` para que o segundo argumento tenha um valor padrão de 2. Teste a função chamando-a com apenas um argumento.

1.13.6 Exercício 6.

Escreva um laço `for` que calcule a soma dos números de 1 a 100.

Crie um laço `while` que multiplique os números de 1 a 6 e retorne o resultado.

1.13.7 Exercício 7.

Crie duas variáveis com o seu primeiro e último nome. Use a função `f-string` para juntar as duas variáveis em uma frase que diga “Meu nome completo é [nome completo]”.

Altere o separador para um traço - e junte novamente as variáveis.

1.13.8 Exercício 8.

Crie um DataFrame com os dados fictícios a seguir utilizando `pandas`:

Nome	Idade	Cidade
Ana	23	São Paulo
Pedro	34	Rio de Janeiro
João	19	Curitiba

Filtre as observações onde a idade é maior que 20.

Ordene o DataFrame pela coluna `Idade` em ordem decrescente.

1.13.9 Exercício 9.

Usando o DataFrame criado no exercício anterior, crie um gráfico de barras (`plt.bar`) que mostre a idade de cada pessoa.

No gráfico anterior, mude a cor de cada barra.

1.13.10 Exercício 10.

Crie um DataFrame com os dados de temperatura de uma semana:

Dia	Temperatura (°C)
Seg	22
Ter	24
Qua	19
Qui	21
Sex	23
Sáb	25
Dom	20

Agora, responda às perguntas abaixo:

1. Filtre todos os dias em que a temperatura foi maior que 21°C.
2. Calcule a média das temperaturas da semana.
3. Crie um gráfico de linha (`plt.plot`) que mostre a variação da temperatura ao longo da semana.

2 Breve Introdução ao R

Aqui faremos uma breve introdução ao R. Para uma introdução mais delhada, veja <https://curso-r.com/material/> e <https://r4ds.had.co.nz/>.

2.1 Variáveis Numéricas

Aqui vão alguns exemplos para começarmos a entender como o R funciona. Inicialmente, veremos como podemos usar o R como calculadora.

```
2 * 7
```

```
[1] 14
```

```
0 / 0
```

```
[1] NaN
```

```
10^1000
```

```
[1] Inf
```

```
log(1)
```

```
[1] 0
```

```
exp(1)
```

```
[1] 2.718282
```

Para entender o que uma função faz, você pode digitar o símbolo de interrogação seguido do nome da função, por exemplo:

```
?exp
```

O help contém as seguintes informações:

- **Description:** Resumo geral sobre o uso da função.
- **Usage:** Mostra como a função deve ser utilizada e quais argumentos podem ser especificados.
- **Arguments:** Explica o que é cada um dos argumentos.
- **Details:** Explica alguns detalhes sobre o uso e aplicação da função.
- **Value:** Mostra o que sai no output após usar a função (os resultados).
- **Note:** Notas sobre a função.
- **Authors:** Lista os autores da função.
- **References:** Referências para os métodos usados.
- **See also:** Outras funções relacionadas que podem ser consultadas.
- **Examples:** Exemplos do uso da função.

2.1.1 Armazenando resultados em variáveis

Podemos também armazenar resultados de contas em variáveis. Por exemplo:

```
x = 2 + 3  
x
```

```
[1] 5
```

```
y = 2 * x  
y
```

```
[1] 10
```

```
print(y)
```

```
[1] 10
```

Números grandes são impressos usando notação científica:

```
y = 2 * 1010  
print(y)
```

```
[1] 2e+10
```

Para listar quais variáveis estão declaradas no ambiente, podemos usar:

```
ls()
```

```
[1] "x" "y"
```

Para remover uma variável:

```
rm(x)
ls()
```

```
[1] "y"
```

Para remover todas as variáveis existentes:

```
rm(list = ls()) # Essa função apaga todas as variáveis existentes
gc()           # Essa função libera a memória utilizada
```

```
      used (Mb) gc trigger (Mb) max used (Mb)
Ncells 628614 33.6   1279720 68.4   1279720 68.4
Vcells 1185391  9.1    8388608 64.0   1951125 14.9
```

2.2 Variáveis Lógicas

Alguns operadores lógicos definidos no R são mostrados na tabela abaixo:

#	Operador	Descrição
1	$x < y$	x menor que y?
2	$x \leq y$	x menor ou igual a y?
3	$x > y$	x maior que y?
4	$x \geq y$	x maior ou igual a y?
5	$x == y$	x igual a y?
6	$x != y$	x diferente de y?
7	$!x$	Negativa de x
8	$x y$	x ou y são verdadeiros?

#	Operador	Descrição
9	<code>x & y</code>	x e y são verdadeiros?
10	<code>xor(x, y)</code>	Apenas um dos dois é verdadeiro?

2.2.1 Exemplos de uso de operadores lógicos

```
1 == 3
```

```
[1] FALSE
```

```
1 == 1
```

```
[1] TRUE
```

```
1 <= 3
```

```
[1] TRUE
```

```
x = 1 > 3
print(x)
```

```
[1] FALSE
```

```
x = 1; y = 3
x < y
```

```
[1] TRUE
```

```
(x == 1) & (y == 3)
```

```
[1] TRUE
```

```
(x == 1) & (y != 3)
```

```
[1] FALSE
```

```
!TRUE
```

```
[1] FALSE
```

```
x = TRUE; y = FALSE  
x | y
```

```
[1] TRUE
```

O número um tem o valor verdadeiro, e o número zero tem o valor falso:

```
x = 1; y = 0  
x | y
```

```
[1] TRUE
```

Note que ao fazermos contas com variáveis lógicas, elas são transformadas em zero e um:

```
(TRUE | FALSE) + 2
```

```
[1] 3
```

2.3 Caracteres/Strings

Para declarar cadeias de caracteres, usamos aspas no R:

```
x = "Rafael"  
y = "Izbicki"
```

Várias operações podem ser feitas com esses objetos. Um exemplo é a função **paste**:

```
paste(x, y)
```

```
[1] "Rafael Izbicki"
```



```
paste(x, y, sep = "+")
```

```
[1] "Rafael+Izbicki"
```

2.4 Vetores e Sequências

Usualmente declaramos vetores usando o operador `c` (concatenação):

```
x = c(2, 5, 7, 1)
x
```

```
[1] 2 5 7 1
```

Para acessar seus componentes, fazemos:

```
x[2]
```

```
[1] 5
```

```
# [1] 5
```

```
x = x[c(2, 3)]
x[2:3]
```

```
[1] 7 NA
```

```
x[-c(2, 3)]
```

```
[1] 5
```

As operações `x[c(2, 3)]` e `x[-c(2, 3)]` são chamadas de *subsetting*; subsetting é a seleção de um subconjunto de um objeto. Veremos mais à frente outras maneiras de fazermos isso.

Podemos mudar alguns dos valores deste vetor usando:

```
x[2:3] = 0
x
```

```
[1] 5 0 0
```

Operações aritméticas podem ser facilmente feitas para cada elemento de um vetor. Alguns exemplos:

```
x = x - 1
x
```

```
[1] 4 -1 -1
```

```
x = 2 * x
x
```

```
[1] 8 -2 -2
```

Uma função útil para criar vetores é a `seq`:

```
x = seq(from = 1, to = 100, by = 2)
x
```

```
[1] 1 3 5 7 9 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39 41 43 45 47 49
[26] 51 53 55 57 59 61 63 65 67 69 71 73 75 77 79 81 83 85 87 89 91 93 95 97 99
```

```
y = seq(from = 1, to = 100, length = 5)
y
```

```
[1] 1.00 25.75 50.50 75.25 100.00
```

Podemos também usar operadores lógicos em vetores:

```
x > 5
```

```
[1] FALSE FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[13] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[25] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[37] TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
[49] TRUE TRUE
```

Note que `x > 5` é um vetor lógico.

Podemos usar as operações lógicas em vetores lógicos:

```
x = c(TRUE, TRUE, FALSE, FALSE)
y = c(TRUE, FALSE, TRUE, FALSE)

x | y
```

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

```
x & y
```

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

Operadores lógicos também podem ser usados para fazer *subsetting*:

```
x[x > 5] = 0
x
```

```
[1] 1 1 0 0
```

```
x[!(x <= 0)] = -1
x
```

```
[1] -1 -1 0 0
```

Podemos ter vetores de caracteres:

```
x = c("a", "c", "fgh")
x[-c(2, 3)]
```

```
[1] "a"
```

```
paste(x, "add")
```

```
[1] "a add" "c add" "fgh add"
```

```
paste(x, "add", collapse = "+")
```

```
[1] "a add+c add+fgh add"
```

Algumas funções adicionais úteis:

```
rep(2, 5)
```

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

```
rep(c(1, 2), 5)
```

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

```
rep(c(1, 2), each = 5)
```

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

```
sort(c(5, 10, 0, 20))
```

```
[1] 0 5 10 20
```

```
order(c(5, 10, 0, 20))
```

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

2.5 Funções

Para declarar funções em R, fazemos:

```
minhaFuncao <- function(argumento1, argumento2) {  
  # corpo da função  
}
```

Por exemplo:

```
potencia <- function(x, y) {  
  return(x^y)  
}  
potencia(2, 3)
```

```
[1] 8
```

```
# [1] 8
```

Note que `x` e `y` podem ser vetores:

```
potencia(x = c(1, 2, 3), y = c(0, 1, 2))
```

```
[1] 1 2 9
```

Você pode inverter a ordem dos argumentos, desde que eles sejam nomeados:

```
potencia(y = c(0, 1, 2), x = c(1, 2, 3))
```

```
[1] 1 2 9
```

2.5.1 Argumentos com valores default

Os argumentos podem ter valores *default*. Por exemplo:

```
potencia <- function(x, y = rep(1, length(x))) {  
  return(x^y)  
}
```

Neste caso, se o argumento `y` não for fornecido, ele será um vetor de uns do tamanho de `x`:

```
potencia(2)
```

```
[1] 2
```

```
potencia(2, 3)
```

```
[1] 8
```

2.6 Laços

Para calcular o fatorial de um número n , podemos usar um laço `while`:

```
n <- 4
i <- 1
nFatorial <- 1
while(i <= n) {
  nFatorial <- nFatorial * i
  i <- i + 1
}
nFatorial
```

```
[1] 24
```

Ou podemos usar um laço `for`:

```
n <- 4
nFatorial <- 1
for(i in 1:n) {
  nFatorial <- nFatorial * i
}
nFatorial
```

```
[1] 24
```

Lembre-se de que laços podem ser lentos no R, especialmente se o tamanho do objeto não for previamente declarado. Vamos comparar o tempo de execução de diferentes abordagens usando a função `system.time`. Para isso, vamos calcular a soma acumulada de um vetor.

```
x <- 1:100000
```

1. Laço sem declaração de tamanho:

```

aux <- function(x) {
  somaAcumulada <- NULL
  somaAcumulada[1] <- x[1]
  for(i in 2:length(x)) {
    somaAcumulada[i] <- somaAcumulada[i - 1] + x[i]
  }
}
system.time(aux(x))[1]

```

```

user.self
0.019

```

2. Laço com declaração de tamanho:

```

aux <- function(x) {
  somaAcumulada <- rep(NA, length(x))
  somaAcumulada[1] <- x[1]
  for(i in 2:length(x)) {
    somaAcumulada[i] <- somaAcumulada[i - 1] + x[i]
  }
}
system.time(aux(x))[1]

```

```

user.self
0.009

```

3. Função nativa do R:

```

aux <- function(x) {
  cumsum(x)
}
system.time(aux(x))[1]

```

```

user.self
0.001

```

Note que, em geral, funções nativas do R são muito mais rápidas.

2.7 If/Else

O R permite o uso de condicionais `if` e `else` para controlar o fluxo de execução. A sintaxe básica é:

```
if (condição1) {  
  # código se a condição1 for verdadeira  
} else if (condição2) {  
  # código se a condição1 for falsa e condição2 verdadeira  
} else {  
  # código se todas as condições forem falsas  
}
```

Exemplo:

```
testaX <- function(x) {  
  if (!is.numeric(x)) {  
    print("x não é um número")  
  } else if (x > 0) {  
    print("x é positivo")  
  } else if (x < 0) {  
    print("x é negativo")  
  } else {  
    print("x é nulo")  
  }  
}  
testaX(5)
```

```
[1] "x é positivo"
```

```
testaX(-5)
```

```
[1] "x é negativo"
```

```
testaX("5")
```

```
[1] "x não é um número"
```


2.8 Listas

Listas são como vetores, mas podem conter componentes de diferentes tipos (inclusive outras listas).

```
minhaLista <- list("um", TRUE, 3, c("q", "u", "a", "t", "r", "o"), "cinco")
minhaLista[[3]]
```

```
[1] 3
```

Você também pode atribuir nomes aos elementos de uma lista:

```
minhaLista <- list(primeiro = "um", segundo = TRUE, terceiro = 3,
                  quarto = c("q", "u", "a", "t", "r", "o"), quinto = "cinco")
minhaLista$quinto
```

```
[1] "cinco"
```

Listas são frequentemente usadas para retornar várias saídas de uma função. Por exemplo, ao ajustar um modelo de regressão linear, os resultados são armazenados em uma lista:

```
x <- rnorm(100)
y <- 1 + 2*x + rnorm(length(x), 0.1)
ajuste <- lm(y ~ x)
typeof(ajuste)
```

```
[1] "list"
```

Os componentes da lista podem ser acessados com:

```
names(ajuste)
```

```
[1] "coefficients" "residuals"    "effects"      "rank"
[5] "fitted.values" "assign"        "qr"           "df.residual"
[9] "xlevels"       "call"          "terms"        "model"
```

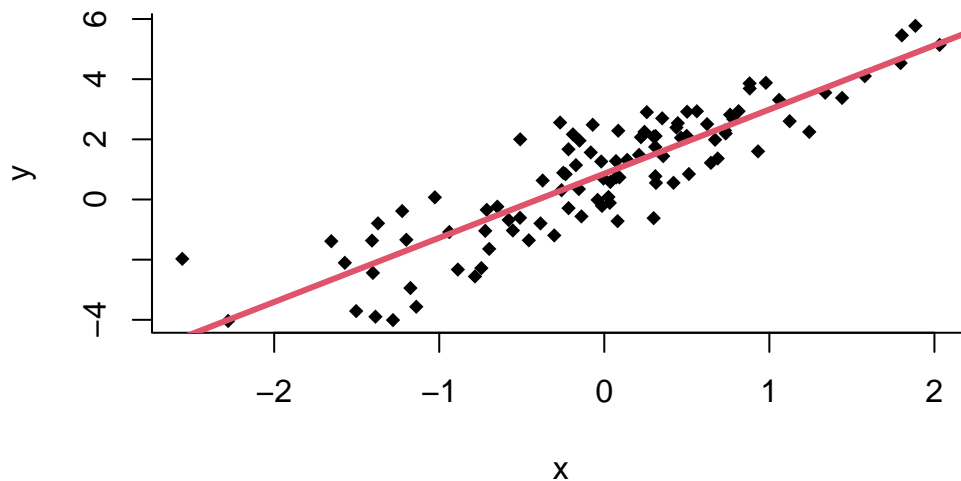
Para ver os coeficientes ajustados:

```
ajuste$coefficients
```

```
(Intercept)          x  
  0.8584361    2.1331953
```

Para adicionar a reta estimada ao gráfico:

```
plot(x, y, pch = 18, bty = "l")  
abline(ajuste, col = 2, lwd = 3)
```



Você também pode inicializar uma lista vazia:

```
minhaLista <- list()
```

2.9 Estatística Descritiva

O R é uma ferramenta poderosa para realizar análises descritivas. Vamos explorar alguns recursos para calcular estatísticas de variáveis quantitativas e qualitativas.

2.9.1 Variáveis quantitativas

```
x <- c(30, 1, 20, 5, 20, 60)  
y <- c(12, 0, 2, 15, 22, 20)  
  
mean(x)
```

```
[1] 22.66667
```

```
var(x)
```

```
[1] 448.6667
```

```
min(x)
```

```
[1] 1
```

```
which.min(x)
```

```
[1] 2
```

```
max(x)
```

```
[1] 60
```

```
which.max(x)
```

```
[1] 6
```

```
sort(x)
```

```
[1] 1 5 20 20 30 60
```

```
order(x)
```

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

```
median(x)
```

```
[1] 20
```

```
cor(x, y)
```

```
[1] 0.5229211
```

```
which(x > 10)
```

```
[1] 1 3 5 6
```

```
mean(y[x > 10])
```

```
[1] 14
```

2.9.2 Variáveis qualitativas

```
x <- c("S", "R", "P", "P", "Q", "P")
y <- c("a", "a", "b", "a", "c", "d")

x[y %in% c("a", "c")]
```

```
[1] "S" "R" "P" "Q"
```

```
table(x)
```

```
x
P Q R S
3 1 1 1
```

```
table(x, y)
```

```
      y
x     a b c d
P  1 1 0 1
Q  0 0 1 0
R  1 0 0 0
S  1 0 0 0
```

2.9.3 Subconjuntos

Podemos utilizar subsetting para trabalhar com subconjuntos de interesse. Por exemplo, suponha que `dados` é um banco de dados com informações de alunos e queremos selecionar os alunos com idade menor que 20, altura maior que 1,70 e que estejam no primeiro ou segundo ano.

```
dados[dados$idade < 20 & dados$altura > 1.70 & dados$ano %in% c("Primeiro", "Segundo"), ]
```

2.9.4 Funções Apply

As funções da família `apply` (como `apply`, `lapply`, `sapply`) são úteis para aplicar uma função a um conjunto de dados. Vamos criar um banco de dados artificial para exemplificar:

```
dados <- data.frame(altura = rnorm(100, 1.7, 0.2), salario = rnorm(100, 3000, 500))
dados$peso <- dados$altura * 35 + rnorm(100, 0, 10)
```

2.9.4.1 `apply`

Para calcular a média de cada coluna de `dados`, usamos:

```
apply(dados, 2, mean)
```

	altura	salario	peso
	1.670245	3119.763281	59.001672

Aqui, o argumento 2 indica que a operação deve ser aplicada a cada coluna. Para calcular a soma de cada linha:

```
apply(dados, 1, sum)
```

[1]	2633.851	3398.634	3792.793	2976.944	3289.344	2824.553	3493.071	2882.774
[9]	3214.105	3164.641	2758.013	2630.263	3091.097	3531.599	2720.263	2692.757
[17]	4158.441	3046.908	3481.259	3719.689	4557.890	2562.557	2905.843	3179.142
[25]	2029.865	3077.064	3156.353	2632.743	3374.917	3339.822	3378.159	3083.389
[33]	3541.555	3024.627	3454.527	3235.045	3440.762	3390.101	2906.872	4148.250
[41]	2963.246	2687.914	2960.429	3821.878	3651.480	2209.895	2434.764	2971.990
[49]	3607.466	3693.463	2907.009	3032.100	3639.341	3935.193	3680.885	2298.986
[57]	3313.046	2792.784	2811.833	3523.931	3012.899	3952.046	3872.205	2779.395

```
[65] 4213.224 2324.731 4204.417 2556.914 2968.792 2982.503 3168.006 3251.537
[73] 3342.642 3371.894 3227.199 2145.430 3221.117 3214.589 2989.637 3125.623
[81] 2501.023 3098.557 2917.466 2874.651 2348.076 3552.759 4056.966 2941.727
[89] 3134.542 3593.589 4377.328 3932.378 3626.834 2282.645 2768.262 3048.286
[97] 3677.718 2766.811 3170.251 2590.741
```

2.9.4.2 Funções anônimas

Usando funções anônimas, podemos calcular várias estatísticas descritivas ao mesmo tempo:

```
estatisticas <- apply(dados, 2, function(x) {
  listaResultados <- list()
  listaResultados$media <- mean(x)
  listaResultados$mediaAparada <- mean(x, trim = 0.1)
  listaResultados$mediana <- median(x)
  listaResultados$maximo <- max(x)
  return(listaResultados)
})

estatisticas$altura
```

```
$media
[1] 1.670245
```

```
$mediaAparada
[1] 1.675409
```

```
$mediana
[1] 1.672197
```

```
$maximo
[1] 2.054764
```

2.10 O operador %>% (pipe)

O operador pipe foi uma das grandes revoluções recentes do R, tornando o código mais legível. Esse operador está definido no pacote **magrittr**, e vários outros pacotes, como o **dplyr**, fazem uso dele (veja a próxima seção).

A ideia é simples: o operador %>% usa o resultado do lado esquerdo como o primeiro argumento da função do lado direito. Um exemplo:

```
library(magrittr)
x <- c(1, 2, 3, 4)
x %>% sum() %>% sqrt()
```

```
[1] 3.162278
```

Isso é equivalente a:

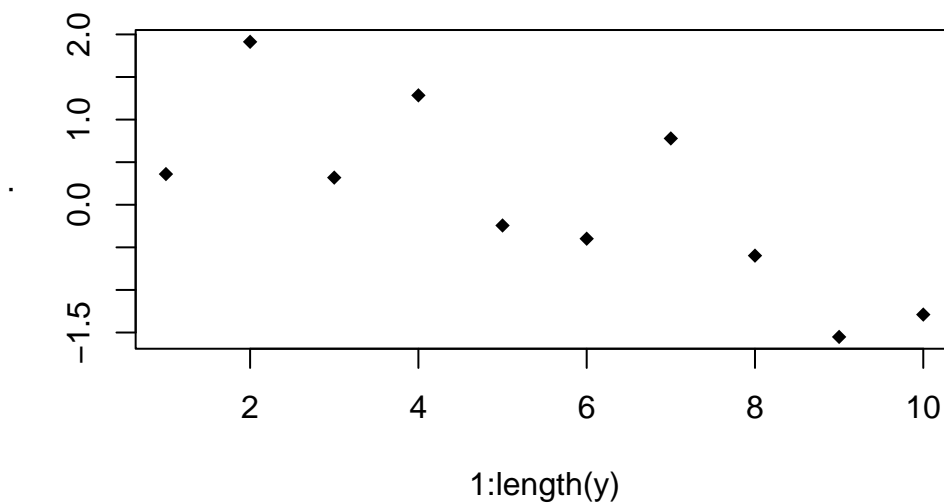
```
sqrt(sum(x))
```

```
[1] 3.162278
```

Mas a leitura com pipe é mais clara. No RStudio, você pode inserir o pipe com o atalho **ctrl + shift + m**.

O pipe envia o valor à esquerda apenas para o primeiro argumento da função à direita. Para utilizar o valor da esquerda em outro argumento, utilize o `"."`:

```
y <- rnorm(10)
y %>% plot(x = 1:length(y), y = ., pch = 18)
```



2.11 O pacote dplyr

O pacote `dplyr` é muito útil para manipulação eficiente de data frames. Vamos demonstrar alguns exemplos usando o conjunto de dados `hflights`.

```
library(dplyr)
library(hflights)
data(hflights)
```

Primeiro, converta os dados para o formato `tbl_df`, uma variação mais amigável do `data.frame`:

```
flights <- tbl_df(hflights)
```

2.11.1 Filter

A função `filter` retorna todas as linhas que satisfazem uma condição. Por exemplo, para mostrar todos os voos do dia 1º de janeiro:

```
flights %>% filter(Month == 1, DayofMonth == 1)
```

```
# A tibble: 552 x 21
   Year Month DayofMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
   <int> <int>    <int>    <int>    <int>    <int> <chr>          <int>
1  2011     1         1         6     1400     1500 AA             428
2  2011     1         1         6      728      840 AA             460
3  2011     1         1         6     1631     1736 AA            1121
4  2011     1         1         6     1756     2112 AA            1294
5  2011     1         1         6     1012     1347 AA            1700
6  2011     1         1         6     1211     1325 AA            1820
7  2011     1         1         6      557      906 AA            1994
8  2011     1         1         6     1824     2106 AS             731
9  2011     1         1         6      654     1124 B6             620
10 2011     1         1         6     1639     2110 B6             622
# i 542 more rows
# i 13 more variables: TailNum <chr>, ActualElapsedTime <int>, AirTime <int>,
#   ArrDelay <int>, DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
#   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
#   Diverted <int>
```


Para condições alternativas, podemos usar o operador | (ou) ou %in%:

```
flights %>% filter(UniqueCarrier %in% c("AA", "UA"))
```

```
# A tibble: 5,316 x 21
```

	Year	Month	DayofMonth	DayOfWeek	DepTime	ArrTime	UniqueCarrier	FlightNum
	<int>	<int>	<int>	<int>	<int>	<int>	<chr>	<int>
1	2011	1	1	6	1400	1500	AA	428
2	2011	1	2	7	1401	1501	AA	428
3	2011	1	3	1	1352	1502	AA	428
4	2011	1	4	2	1403	1513	AA	428
5	2011	1	5	3	1405	1507	AA	428
6	2011	1	6	4	1359	1503	AA	428
7	2011	1	7	5	1359	1509	AA	428
8	2011	1	8	6	1355	1454	AA	428
9	2011	1	9	7	1443	1554	AA	428
10	2011	1	10	1	1443	1553	AA	428

```
# i 5,306 more rows
```

```
# i 13 more variables: TailNum <chr>, ActualElapsedTime <int>, AirTime <int>,  
#   ArrDelay <int>, DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,  
#   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,  
#   Diverted <int>
```

2.11.2 Select

A função `select` seleciona colunas específicas de um data frame. Para selecionar as colunas `DepTime`, `ArrTime`, e `FlightNum`:

```
flights %>% select(DepTime, ArrTime, FlightNum)
```

```
# A tibble: 227,496 x 3
```

	DepTime	ArrTime	FlightNum
	<int>	<int>	<int>
1	1400	1500	428
2	1401	1501	428
3	1352	1502	428
4	1403	1513	428
5	1405	1507	428
6	1359	1503	428
7	1359	1509	428
8	1355	1454	428

```

9      1443      1554      428
10     1443      1553      428
# i 227,486 more rows

```

Para selecionar todas as colunas que contêm “Taxi” ou “Delay”:

```
flights %>% select(contains("Taxi"), contains("Delay"))
```

```

# A tibble: 227,496 x 4
  TaxiIn TaxiOut ArrDelay DepDelay
  <int>   <int>   <int>   <int>
1      7      13     -10        0
2      6       9      -9        1
3      5      17      -8       -8
4      9      22       3        3
5      9       9      -3        5
6      6      13      -7       -1
7     12      15      -1       -1
8      7      12     -16       -5
9      8      22      44       43
10     6      19      43       43
# i 227,486 more rows

```

2.11.3 Arrange

A função `arrange` ordena o data frame de acordo com algum critério. Para ordenar pelo atraso de partida (`DepDelay`):

```
flights %>% arrange(DepDelay)
```

```

# A tibble: 227,496 x 21
  Year Month DayOfMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
  <int> <int>      <int>      <int>   <int>   <int> <chr>          <int>
1  2011    12         24          6   1112    1314 00           5440
2  2011     2         14          1   1917    2027 MQ           3328
3  2011     4         10          7   2101    2206 XE           2669
4  2011     8          3          3   1741    1810 XE           2603
5  2011     1         18          2   1542    1936 CO           1688
6  2011    10          4          2   1438    1813 EV           5412
7  2011     1        26          3   2248    2343 XE           2450

```

```

8 2011      3          8          2      953      1156 CO          1882
9 2011      3         18          5     2103      2156 XE          2261
10 2011     4          3          7     1048      1307 MQ          3796
# i 227,486 more rows
# i 13 more variables: TailNum <chr>, ActualElapsedTime <int>, AirTime <int>,
#   ArrDelay <int>, DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
#   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
#   Diverted <int>

```

Para ordem decrescente:

```
flights %>% arrange(desc(DepDelay))
```

```

# A tibble: 227,496 x 21
   Year Month DayOfMonth DayOfWeek DepTime ArrTime UniqueCarrier FlightNum
   <int> <int>      <int>      <int>   <int>   <int>   <chr>          <int>
1  2011      8          1          1     156     452 CO             1
2  2011     12         12          1     650     808 AA            1740
3  2011     11          8          2     721     948 MQ            3786
4  2011      6         21          2    2334     124 UA             855
5  2011      6          9          4    2029    2243 MQ            3859
6  2011      5         20          5     858    1027 MQ            3328
7  2011      1         20          4     635     807 CO              59
8  2011      6         22          3     908    1040 CO             595
9  2011     10         25          2    2310     149 DL            1215
10 2011     12         13          2     706     824 MQ            3328
# i 227,486 more rows
# i 13 more variables: TailNum <chr>, ActualElapsedTime <int>, AirTime <int>,
#   ArrDelay <int>, DepDelay <int>, Origin <chr>, Dest <chr>, Distance <int>,
#   TaxiIn <int>, TaxiOut <int>, Cancelled <int>, CancellationCode <chr>,
#   Diverted <int>

```

2.11.4 Mutate

A função `mutate` cria novas variáveis. Para criar a variável `Speed` (velocidade) e adicioná-la ao banco:

```
flights <- flights %>% mutate(Speed = Distance / AirTime * 60)
```

2.11.5 Summarise e Group_by

A função `summarise` calcula estatísticas resumo. Para calcular o atraso médio de chegada por destino:

```
flights %>% group_by(Dest) %>% summarise(avg_delay = mean(ArrDelay, na.rm = TRUE))
```

```
# A tibble: 116 x 2
  Dest   avg_delay
  <chr>   <dbl>
1 ABQ     7.23
2 AEX     5.84
3 AGS      4
4 AMA     6.84
5 ANC    26.1
6 ASE     6.79
7 ATL     8.23
8 AUS     7.45
9 AVL     9.97
10 BFL   -13.2
# i 106 more rows
```

A função `summarise_each` aplica uma função a várias colunas ao mesmo tempo. Para calcular a média de `Cancelled` e `Diverted` por companhia aérea:

```
flights %>% group_by(UniqueCarrier) %>% summarise_each(funs(mean), Cancelled, Diverted)
```

```
# A tibble: 15 x 3
  UniqueCarrier Cancelled Diverted
  <chr>          <dbl>   <dbl>
1 AA            0.0185  0.00185
2 AS            0      0.00274
3 B6            0.0259  0.00576
4 CO            0.00678 0.00263
5 DL            0.0159  0.00303
6 EV            0.0345  0.00318
7 F9            0.00716 0
8 FL            0.00982 0.00327
9 MQ            0.0290  0.00194
10 OO           0.0139  0.00349
11 UA           0.0164  0.00241
```

12	US	0.0113	0.00147
13	WN	0.0155	0.00229
14	XE	0.0155	0.00345
15	YV	0.0127	0

Também podemos aplicar várias funções a uma mesma coluna:

```
flights %>% group_by(UniqueCarrier) %>% summarise_each(funs(mean, var), Cancelled, Diverted)
```

A tibble: 15 x 5

	UniqueCarrier	Cancelled_mean	Diverted_mean	Cancelled_var	Diverted_var
	<chr>	<dbl>	<dbl>	<dbl>	<dbl>
1	AA	0.0185	0.00185	0.0182	0.00185
2	AS	0	0.00274	0	0.00274
3	B6	0.0259	0.00576	0.0253	0.00573
4	CO	0.00678	0.00263	0.00674	0.00262
5	DL	0.0159	0.00303	0.0157	0.00302
6	EV	0.0345	0.00318	0.0333	0.00317
7	F9	0.00716	0	0.00712	0
8	FL	0.00982	0.00327	0.00973	0.00326
9	MQ	0.0290	0.00194	0.0282	0.00193
10	OO	0.0139	0.00349	0.0138	0.00347
11	UA	0.0164	0.00241	0.0161	0.00241
12	US	0.0113	0.00147	0.0111	0.00147
13	WN	0.0155	0.00229	0.0153	0.00229
14	XE	0.0155	0.00345	0.0153	0.00344
15	YV	0.0127	0	0.0127	0

Além disso, podemos usar o operador "." para aplicar funções com múltiplos argumentos:

```
flights %>% group_by(UniqueCarrier) %>% summarise_each(funs(min(., na.rm = TRUE), max(., na.rm = TRUE)), Cancelled, Diverted)
```

A tibble: 15 x 5

	UniqueCarrier	ArrDelay_min	DepDelay_min	ArrDelay_max	DepDelay_max
	<chr>	<int>	<int>	<int>	<int>
1	AA	-39	-15	978	970
2	AS	-43	-15	183	172
3	B6	-44	-14	335	310
4	CO	-55	-18	957	981
5	DL	-32	-17	701	730
6	EV	-40	-18	469	479

7	F9	-24	-15	277	275
8	FL	-30	-14	500	507
9	MQ	-38	-23	918	931
10	OO	-57	-33	380	360
11	UA	-47	-11	861	869
12	US	-42	-17	433	425
13	WN	-44	-10	499	548
14	XE	-70	-19	634	628
15	YV	-32	-11	72	54

Por fim, podemos realizar agrupamentos múltiplos:

```
flights %>% group_by(UniqueCarrier, DayOfWeek) %>% summarise_each(funs(min(., na.rm = TRUE)),
```

```
# A tibble: 105 x 6
# Groups:   UniqueCarrier [15]
  UniqueCarrier DayOfWeek ArrDelay_min DepDelay_min ArrDelay_max DepDelay_max
  <chr>          <int>      <int>      <int>      <int>      <int>
1 AA            1        -30        -14        978        970
2 AA            2        -30        -12        265        286
3 AA            3        -33        -15        179        168
4 AA            4        -38        -15        663        653
5 AA            5        -28        -13        255        234
6 AA            6        -39        -13        685        677
7 AA            7        -34        -15        507        525
8 AS            1        -30        -12        183        172
9 AS            2        -34        -12         92         68
10 AS           3        -29        -12        123        138
# i 95 more rows
```

Aqui está a seção sobre o **pacote ggplot2** e **tidyr** atualizada, mantendo a clareza e estrutura para fins didáticos:

2.12 O pacote ggplot2

Nota: Esta seção foi adaptada de curso-r.com.

O **ggplot2** é um pacote do R voltado para a criação de gráficos estatísticos. Ele é baseado na Gramática dos Gráficos (Grammar of Graphics) de Leland Wilkinson. Segundo essa gramática, um gráfico estatístico é um mapeamento dos dados por meio de atributos estéticos (cores, formas, tamanho) de formas geométricas (pontos, linhas, barras).

```
library(ggplot2)
```

2.12.1 Construindo gráficos

Vamos discutir os aspectos básicos para a construção de gráficos com o `ggplot2`, utilizando o conjunto de dados `mtcars`. Para visualizar as primeiras linhas:

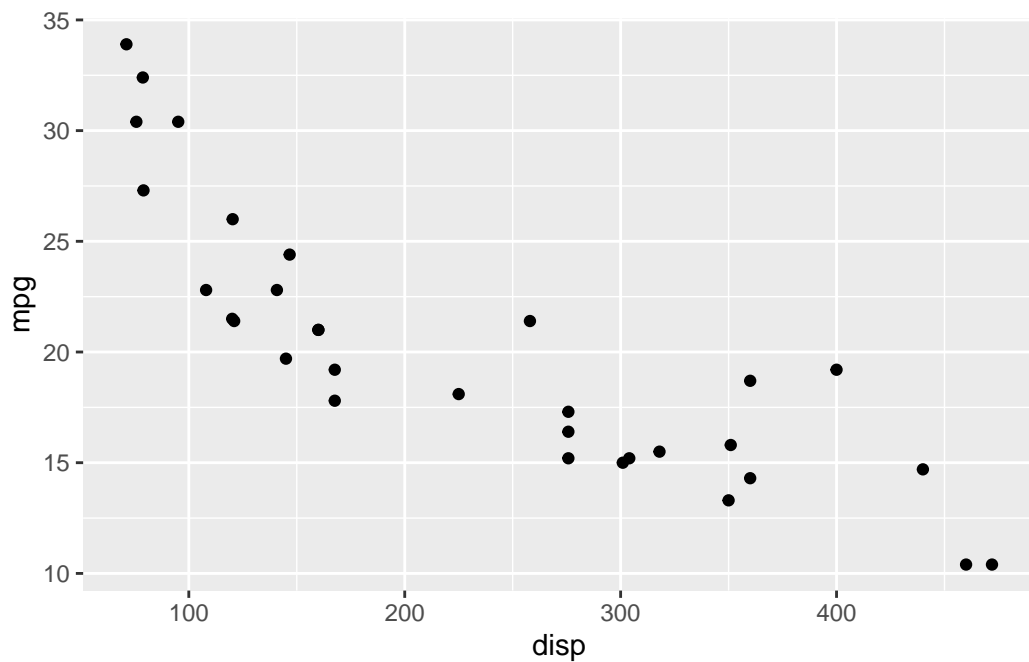
```
head(mtcars)
```

	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225	105	2.76	3.460	20.22	1	0	3	1

2.12.2 As camadas de um gráfico

Os gráficos no `ggplot2` são construídos camada por camada. A primeira camada é criada com a função `ggplot`. Um exemplo de gráfico simples:

```
ggplot(data = mtcars) +  
  geom_point(aes(x = disp, y = mpg))
```

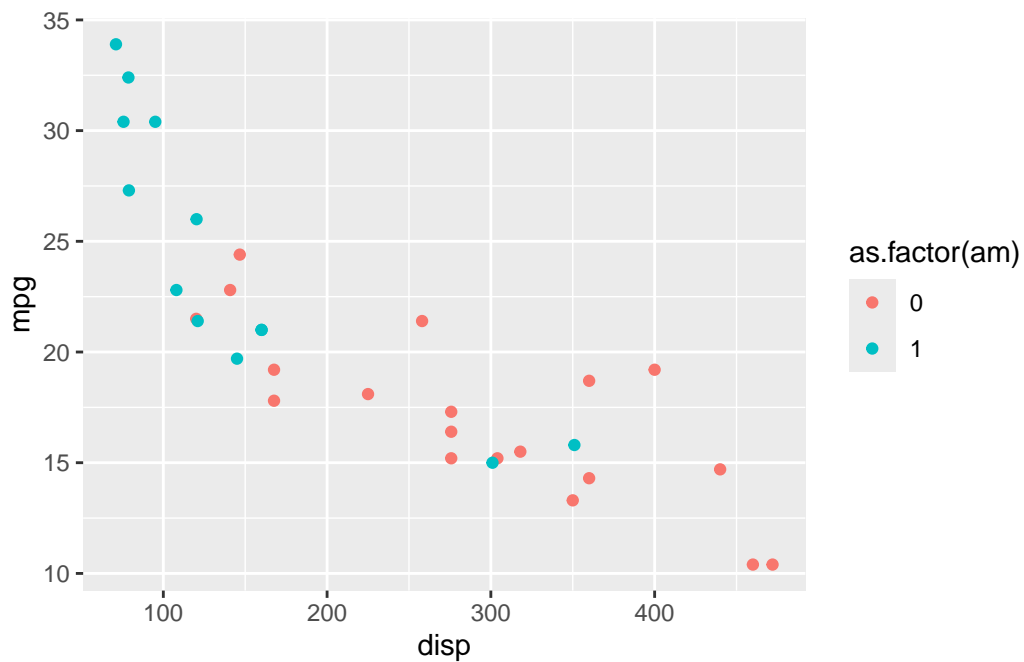


A função `aes` define o mapeamento entre as variáveis e os aspectos visuais. Neste caso, estamos criando um gráfico de dispersão (com `geom_point`) entre `disp` (cilindradas) e `mpg` (milhas por galão).

2.12.3 Aesthetics

Podemos mapear variáveis a diferentes aspectos estéticos, como cor e tamanho. Por exemplo, para mapear a variável `am` (transmissão) para a cor dos pontos:

```
ggplot(data = mtcars) +  
  geom_point(aes(x = disp, y = mpg, colour = as.factor(am)))
```

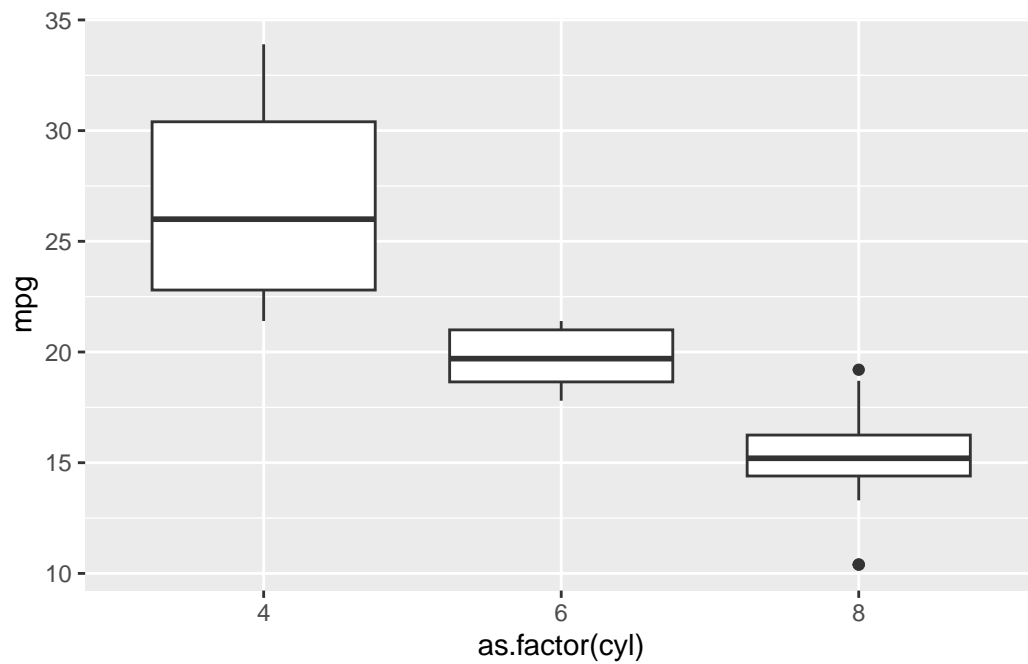
2.12.4 Geoms

Os geoms definem as formas geométricas usadas para a visualização dos dados. Além de `geom_point`, podemos usar:

- `geom_line` para linhas
- `geom_boxplot` para boxplots
- `geom_histogram` para histogramas

Exemplo de um boxplot:

```
ggplot(mtcars) +  
  geom_boxplot(aes(x = as.factor(cyl), y = mpg))
```

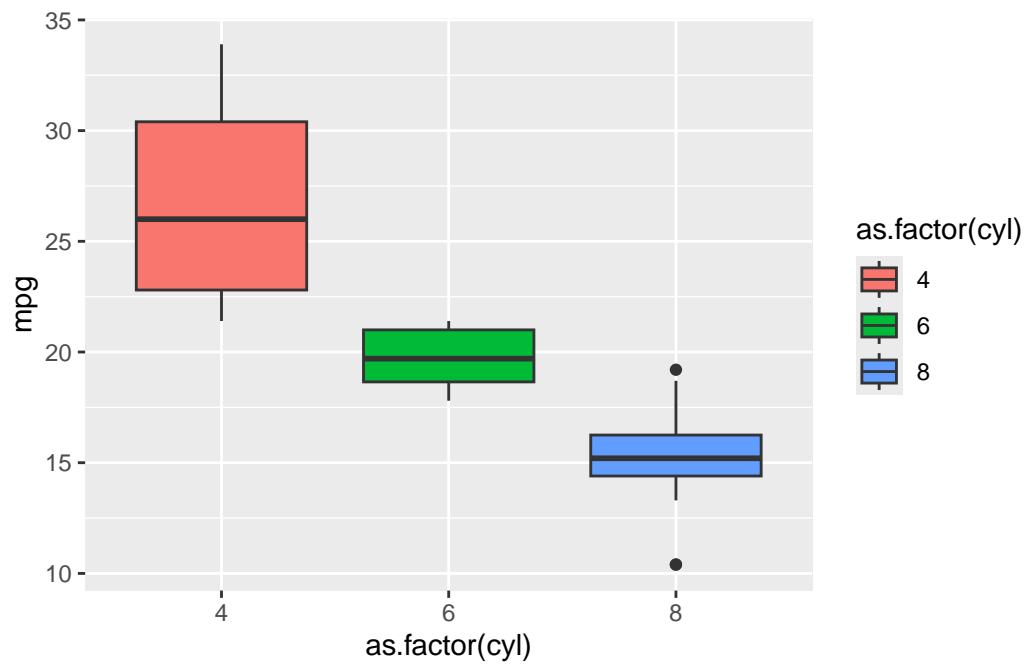


2.12.5 Personalizando os gráficos

2.12.5.1 Cores

Para mudar as cores do gráfico, podemos usar o argumento `colour` ou `fill`:

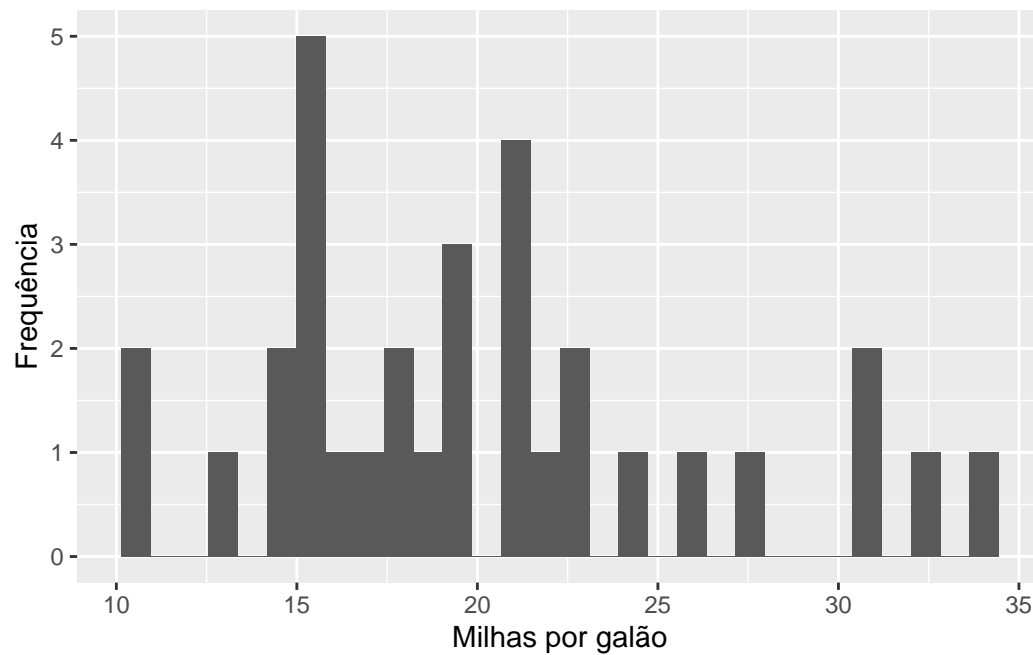
```
ggplot(mtcars) +  
  geom_boxplot(aes(x = as.factor(cyl), y = mpg, fill = as.factor(cyl)))
```



2.12.5.2 Eixos

Para alterar os rótulos dos eixos:

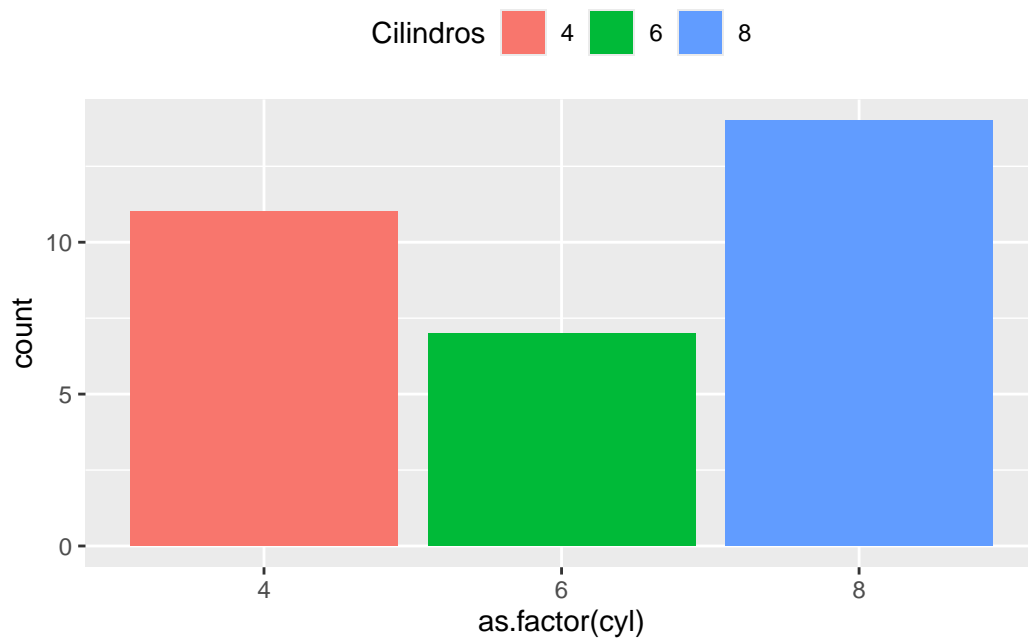
```
ggplot(mtcars) +  
  geom_histogram(aes(x = mpg)) +  
  xlab("Milhas por galão") +  
  ylab("Frequência")
```



2.12.5.3 Legendas

Podemos personalizar ou remover legendas facilmente:

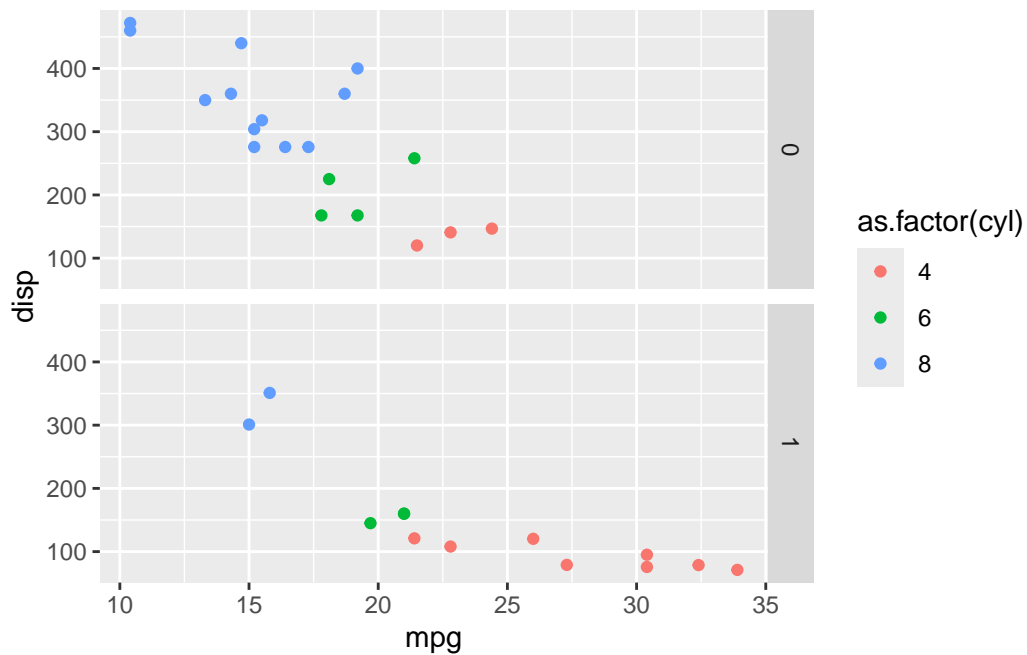
```
ggplot(mtcars) +  
  geom_bar(aes(x = as.factor(cyl), fill = as.factor(cyl))) +  
  labs(fill = "Cilindros") +  
  theme(legend.position = "top")
```



2.12.6 Facets

O `facet_grid` permite dividir o gráfico em subgráficos com base em uma variável:

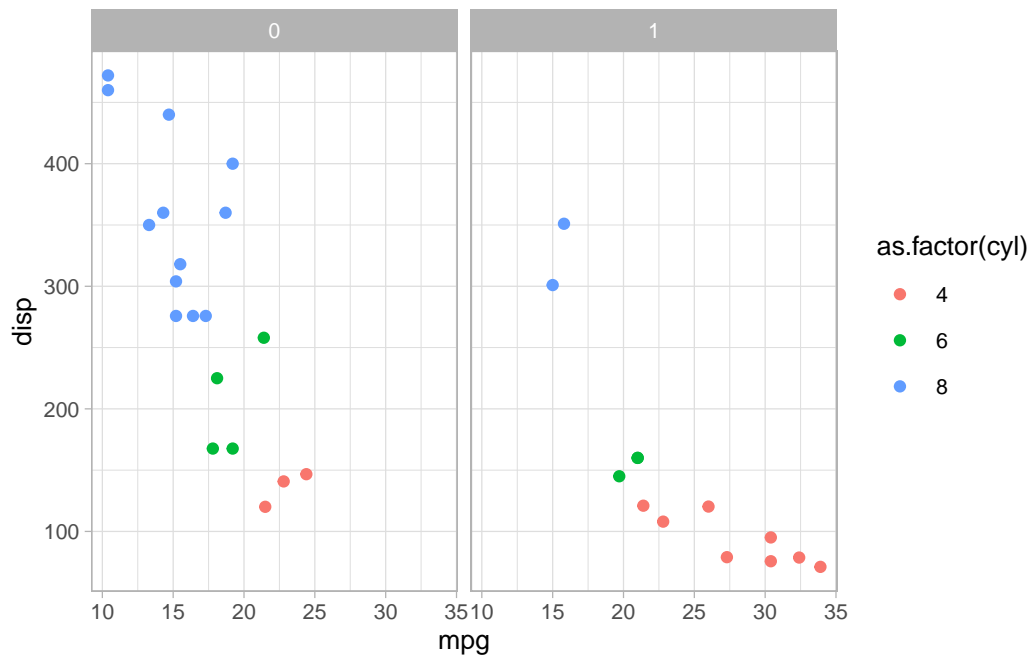
```
ggplot(mtcars) +  
  geom_point(aes(x = mpg, y = disp, colour = as.factor(cyl))) +  
  facet_grid(am ~ .)
```



2.12.7 Temas

Podemos mudar o tema de um gráfico com a função `theme_set`:

```
theme_set(theme_light(base_size = 10))
ggplot(mtcars) +
  geom_point(aes(x = mpg, y = disp, colour = as.factor(cyl))) +
  facet_grid(. ~ am)
```



2.13 O pacote tidyr

O pacote `tidyr` facilita a transformação e organização de dados no R. Para converter os dados de formato “wide” (largo) para “long” (longo), podemos utilizar a função `pivot_longer`, que substitui a antiga função `gather`. Considere os dados a seguir:

```
dados.originais <- data.frame(
  paciente = c("João", "Marcos", "Antonio"),
  pressao.antes = c(67, 80, 64),
  pressao.durante = c(54, 70, 60),
  pressao.depois = c(56, 90, 50)
)
dados.originais
```

	paciente	pressao.antes	pressao.durante	pressao.depois
1	João	67	54	56
2	Marcos	80	70	90
3	Antonio	64	60	50

Para reorganizar esses dados em um formato “long”, usando `pivot_longer`:

```
library(tidyr)
dados.novo.formato <- dados.originais %>%
  pivot_longer(cols = pressao.antes:pressao.depois,
               names_to = "instante",
               values_to = "pressao.arterial")
dados.novo.formato
```

```
# A tibble: 9 x 3
  paciente instante      pressao.arterial
  <chr>      <chr>          <dbl>
1 João     pressao.antes          67
2 João     pressao.durante        54
3 João     pressao.depois         56
4 Marcos   pressao.antes          80
5 Marcos   pressao.durante        70
6 Marcos   pressao.depois         90
7 Antonio  pressao.antes          64
8 Antonio  pressao.durante        60
9 Antonio  pressao.depois         50
```

A função `pivot_wider` é usada para converter dados de formato “long” (longo) para “wide” (largo), espalhando uma variável por várias colunas. Vamos começar com um conjunto de dados em formato “long” e então reorganizá-los para o formato “wide”.

Considere o conjunto de dados `dados.novo.formato` que organizamos anteriormente:

```
dados.novo.formato <- data.frame(
  paciente = c("João", "Marcos", "Antonio", "João", "Marcos", "Antonio", "João", "Marcos", "Antonio"),
  instante = c("pressao.antes", "pressao.antes", "pressao.antes", "pressao.durante", "pressao.durante", "pressao.durante", "pressao.depois", "pressao.depois", "pressao.depois"),
  pressao.arterial = c(67, 80, 64, 54, 70, 60, 56, 90, 50)
)
dados.novo.formato
```

```
  paciente      instante pressao.arterial
1   João  pressao.antes          67
2  Marcos  pressao.antes          80
3 Antonio  pressao.antes          64
4   João  pressao.durante          54
5  Marcos  pressao.durante          70
6 Antonio  pressao.durante          60
7   João  pressao.depois          56
```


8	Marcos	pressao.depois	90
9	Antonio	pressao.depois	50

Agora, vamos usar `pivot_wider` para transformar esses dados de volta ao formato “wide”:

```
dados.wide <- dados.novo.formato %>%
  pivot_wider(names_from = instante, values_from = pressao.arterial)
dados.wide
```

```
# A tibble: 3 x 4
  paciente pressao.antes pressao.durante pressao.depois
  <chr>          <dbl>          <dbl>          <dbl>
1 João           67            54            56
2 Marcos          80            70            90
3 Antonio         64            60            50
```

Temos os seguintes argumentos: - `names_from` especifica a coluna cujos valores serão usados para criar novos nomes de colunas (neste caso, a variável `instante`). - `values_from` especifica a coluna cujos valores serão preenchidos nas novas colunas criadas (neste caso, a variável `pressao.arterial`).

Aqui está um exemplo de como o pacote `tidyr` pode ser usado para manipular os dados antes de visualizá-los com gráficos no `ggplot2`.

2.13.1 Exemplo: Manipulação e Visualização com `pivot_longer` e `ggplot2`

O pacote `tidyr` é particularmente útil dentro do `ggplot2`. Vamos continuar com o exemplo dos dados de pressão arterial. Suponha que você queira visualizar as variações da pressão arterial dos pacientes em diferentes instantes.

Primeiro, vamos reorganizar os dados usando `pivot_longer` para colocar os dados no formato “long”, que é mais adequado para gráficos com o `ggplot2`:

```
library(tidyr)
library(ggplot2)

# Dados originais no formato "wide"
dados.originais <- data.frame(
  paciente = c("João", "Marcos", "Antonio"),
  pressao.antes = c(67, 80, 64),
  pressao.durante = c(54, 70, 60),
  pressao.depois = c(56, 90, 50)
```

```
)
dados.originais
```

	paciente	pressao.antes	pressao.durante	pressao.depois
1	João	67	54	56
2	Marcos	80	70	90
3	Antonio	64	60	50

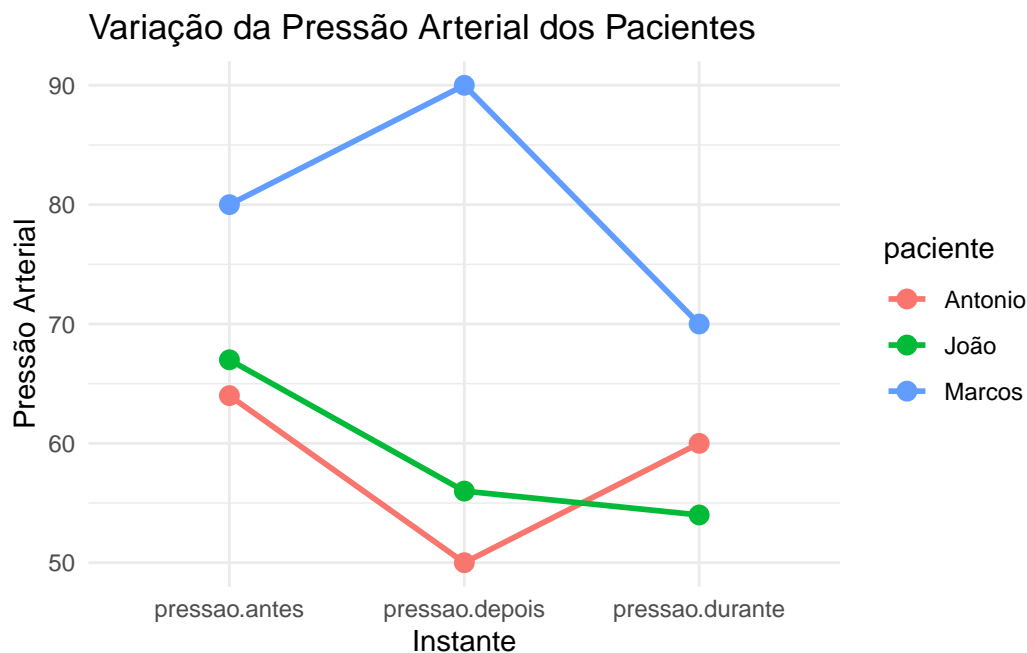
Para usá-los no ggplot, vamos os reorganizar no formato “long”:

```
dados.long <- dados.originais %>%
  pivot_longer(cols = pressao.antes:pressao.depois,
               names_to = "instante",
               values_to = "pressao.arterial")
dados.long
```

```
# A tibble: 9 x 3
  paciente instante      pressao.arterial
  <chr>      <chr>          <dbl>
1 João     pressao.antes         67
2 João     pressao.durante        54
3 João     pressao.depois         56
4 Marcos   pressao.antes         80
5 Marcos   pressao.durante        70
6 Marcos   pressao.depois         90
7 Antonio  pressao.antes         64
8 Antonio  pressao.durante        60
9 Antonio  pressao.depois         50
```

Agora, vamos criar um gráfico de linhas que mostra as mudanças na pressão arterial ao longo do tempo para cada paciente:

```
ggplot(dados.long, aes(x = instante, y = pressao.arterial, group = paciente, color = paciente)) +
  geom_line(size = 1) +
  geom_point(size = 3) +
  labs(title = "Variação da Pressão Arterial dos Pacientes",
       x = "Instante",
       y = "Pressão Arterial") +
  theme_minimal()
```



2.14 Leitura de Arquivos com readr

O `readr` é um pacote eficiente para leitura de arquivos de dados no R. Vamos explorar como utilizar o `readr` para ler e manipular arquivos CSV. Um arquivo CSV (Comma-Separated Values) é um formato comum para armazenar dados tabulares.

Para começar, carregue o pacote `readr`:

```
library(readr)
```

2.14.1 Exemplo: Leitura de um CSV com Dados Públicos

Vamos usar um banco de dados público disponível na internet. Um exemplo muito comum é o conjunto de dados de passageiros do Titanic, disponível em formato CSV. Para carregar diretamente da internet:

```
url <- "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
dados_titanic <- read_csv(url)
```

Aqui, estamos carregando os dados diretamente de um link. Após a leitura, podemos visualizar os primeiros registros:

```
head(dados_titanic)
```

```
# A tibble: 6 x 12
  PassengerId Survived Pclass Name      Sex      Age SibSp Parch Ticket  Fare Cabin
      <dbl>     <dbl> <dbl> <chr>    <chr> <dbl> <dbl> <dbl> <chr>  <dbl> <chr>
1         1         0     3 Braund~ male    22     1     0 A/5 2~  7.25 <NA>
2         2         1     1 Cuming~ fema~   38     1     0 PC 17~ 71.3  C85
3         3         1     3 Heikki~ fema~   26     0     0 STON/~  7.92 <NA>
4         4         1     1 Futrel~ fema~   35     1     0 113803 53.1  C123
5         5         0     3 Allen,~ male    35     0     0 373450  8.05 <NA>
6         6         0     3 Moran,~ male    NA     0     0 330877  8.46 <NA>
# i 1 more variable: Embarked <chr>
```

2.14.2 Manipulando os Dados Lidos

O `read_csv` detecta automaticamente os tipos de dados de cada coluna. No exemplo dos dados do Titanic, podemos realizar algumas análises rápidas, como visualizar a distribuição de sobreviventes:

```
table(dados_titanic$Survived)
```

```
0    1
549 342
```

2.14.3 Salvando o Conjunto de Dados

Se você quiser salvar o conjunto de dados em seu computador após manipulações, basta usar o `write_csv`:

```
write_csv(dados_titanic, "titanic_local.csv")
```

Aqui está o conjunto de exercícios no formato solicitado:

2.15 Exercícios

Exercício 1.

Execute as seguintes operações no R e interprete os resultados: $-5 + 9 - \frac{0}{0} - 10^5$

Utilize a função `log` para calcular o logaritmo natural de 10 e depois o logaritmo na base 10 de 1000.

Descubra a função que calcula a raiz quadrada de um número utilizando o símbolo de interrogação (?), e aplique-a ao número 144.

Exercício 2.

Armazene o resultado de 15×7 em uma variável chamada `resultado`. Use `resultado` para calcular o valor de $2 \times \text{resultado}$.

Liste todas as variáveis no ambiente do R usando a função `ls()`. Exclua a variável `resultado` e verifique novamente as variáveis presentes.

Exercício 3.

Crie um vetor `v` contendo os números 4, 8, 15, 16, 23, 42. Use subsetting para selecionar: - O segundo e o quarto elemento. - Todos os elementos exceto o terceiro. - Os elementos que são maiores que 10.

Multiplique cada elemento do vetor `v` por 2 e armazene o resultado em um novo vetor `v2`.

Exercício 4.

Verifique se 8 é maior que 5 e se 8 é igual a 10.

Crie dois vetores lógicos `a = c(TRUE, FALSE, TRUE)` e `b = c(FALSE, TRUE, FALSE)`. Aplique os operadores `&`, `|` e `xor()` nesses vetores.

Exercício 5.

Escreva uma função `soma_quadrados` que receba dois números como argumentos e retorne a soma de seus quadrados. Teste sua função com os números 3 e 4.

Modifique a função `soma_quadrados` para que o segundo argumento tenha um valor padrão de 2. Teste a função chamando-a com apenas um argumento.

Exercício 6.

Escreva um laço `for` que calcule a soma dos números de 1 a 100.

Crie um laço `while` que multiplique os números de 1 a 6 e retorne o resultado.

Exercício 7.

Crie duas variáveis com o seu primeiro e último nome. Use a função `paste()` para juntar as duas variáveis em uma frase que diga “Meu nome completo é [nome completo]”.

Altere o separador na função `paste()` para um traço - e junte novamente as variáveis.

Exercício 8.

Carregue o pacote `dplyr` e use o dataset `mtcars`. Selecione apenas as colunas `mpg`, `cyl`, e `hp`.

Filtre as observações onde `mpg` é maior que 20 e `hp` é menor que 150.

Ordene o dataset filtrado pela coluna `mpg` em ordem decrescente.

Exercício 9.

Usando o dataset `mtcars`, crie um gráfico de dispersão (`geom_point`) que mostre a relação entre `hp` (horsepower) e `mpg` (milhas por galão).

No gráfico anterior, mapeie a variável `cyl` para a cor dos pontos.

Exercício 10. Utilize a base pública de dados de voos da [nycflights13](#) para responder às perguntas abaixo.

1. Carregue o pacote `nycflights13` e explore o dataset `flights`. Visualize as primeiras 6 linhas da base.
2. Filtre todos os voos que decolaram no mês de junho e aterrissaram com atraso maior que 1 hora.
3. Agrupe os dados por companhia aérea (`carrier`) e calcule o atraso médio de chegada (`arr_delay`) para cada companhia.
4. Crie um gráfico de barras que mostre o atraso médio de chegada por companhia aérea.

3 Geração de Números Aleatórios e Aplicação em Estatística

3.1 Objetivo da Aula

Nesta aula, vamos introduzir o conceito de números pseudoaleatórios e como eles podem ser usados para resolver problemas estatísticos por meio de simulação. Vamos abordar a importância da aleatoriedade em estatísticas e em algoritmos de Monte Carlo.

3.2 Conteúdo Teórico

A geração de números aleatórios é essencial em várias áreas da estatística e da ciência de dados. Esses números são utilizados em simulações estocásticas, amostragem e para resolver problemas que envolvem incerteza. Contudo, em computadores, os números “aleatórios” gerados são na verdade pseudoaleatórios, pois seguem uma sequência previsível, gerada por um algoritmo determinístico.

Os números pseudoaleatórios são amplamente usados em algoritmos de Monte Carlo, que dependem da simulação repetida de processos aleatórios para estimar soluções para problemas matemáticos e estatísticos.

3.3 Exemplo de Problema

Vamos resolver o problema de estimar o valor de π (Pi) usando um método de Monte Carlo. A ideia é simular a área de um quarto de círculo inscrito em um quadrado. Gerando pontos aleatórios dentro do quadrado, podemos calcular a proporção desses pontos que também caem dentro do círculo e usar essa proporção para estimar o valor de π .

Como fazer isso?

4 R

```
# Definindo o número de pontos a serem gerados
n_pontos <- 1000

# Gerando pontos aleatórios (x, y) no intervalo [0, 1]
x <- runif(n_pontos, 0, 1)
y <- runif(n_pontos, 0, 1)

# Calculando a distância de cada ponto à origem
distancia <- sqrt(x^2 + y^2)

# Contando quantos pontos estão dentro do quarto de círculo (distância <= 1)
dentro_circulo <- distancia <= 1
pi_estimado <- 4 * sum(dentro_circulo) / n_pontos

# Exibindo o valor estimado de Pi
cat("Valor estimado de :", pi_estimado, "\n")
```

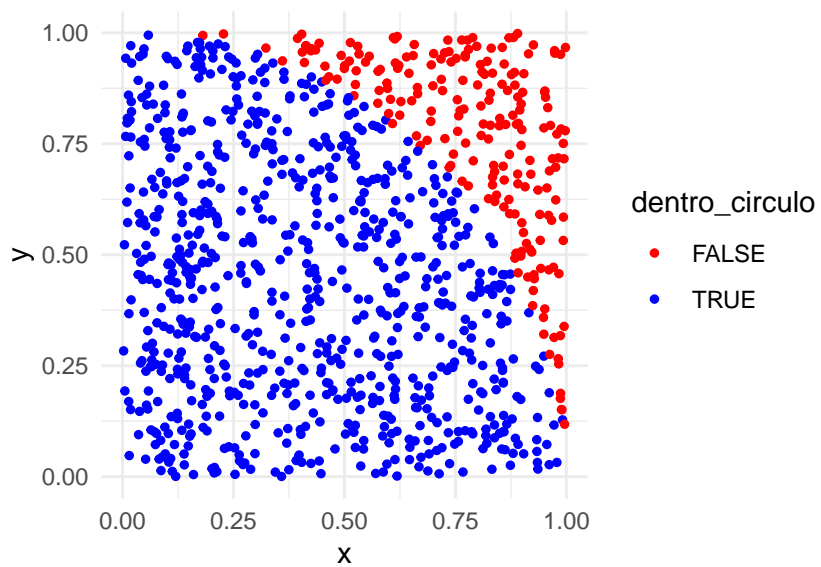
Valor estimado de : 3.16

```
# Visualizando a distribuição dos pontos
library(ggplot2)

dados <- data.frame(x = x, y = y, dentro_circulo = dentro_circulo)

ggplot(dados, aes(x = x, y = y, color = dentro_circulo)) +
  geom_point(size = 1) +
  scale_color_manual(values = c("red", "blue")) +
  ggtitle(paste0("Estimativa de usando Monte Carlo\nValor estimado: ", round(pi_estimado, 5))) +
  theme_minimal() +
  coord_equal() +
  labs(x = "x", y = "y")
```


Estimativa de π usando Monte Carlo
Valor estimado: 3.16



5 Python

```
import numpy as np
import matplotlib.pyplot as plt

# Definindo o número de pontos a serem gerados
n_pontos = 1000

# Gerando pontos aleatórios (x, y) no intervalo [0, 1]
x = np.random.uniform(0, 1, n_pontos)
y = np.random.uniform(0, 1, n_pontos)

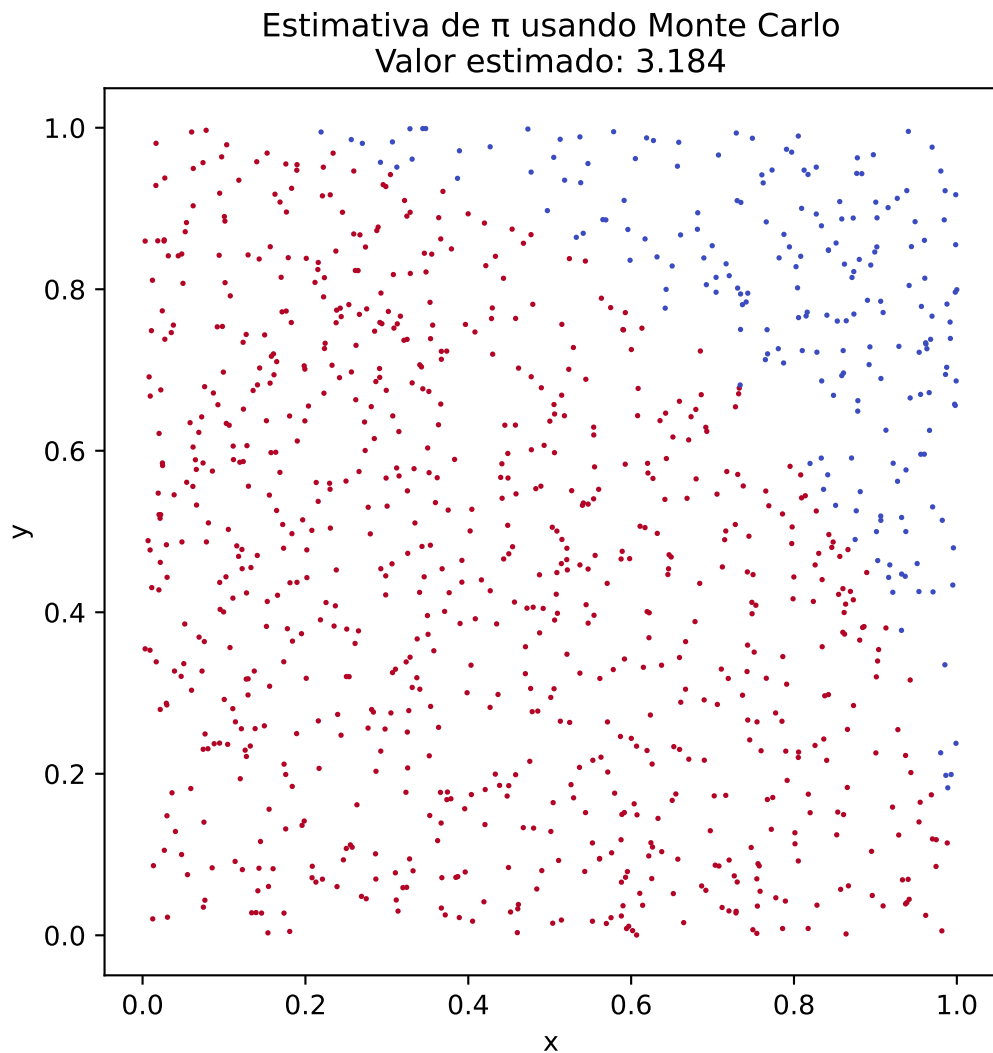
# Calculando a distância de cada ponto à origem
distancia = np.sqrt(x**2 + y**2)

# Contando quantos pontos estão dentro do quarto de círculo (distância <= 1)
dentro_circulo = distancia <= 1
pi_estimado = 4 * np.sum(dentro_circulo) / n_pontos

# Exibindo o valor estimado de Pi
print(f"Valor estimado de : {pi_estimado}")
```

Valor estimado de : 3.184

```
# Visualizando a distribuição dos pontos
plt.figure(figsize=(6,6))
plt.scatter(x, y, c=dentro_circulo, cmap='coolwarm', s=1)
plt.title(f'Estimativa de usando Monte Carlo\nValor estimado: {pi_estimado}')
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```



5.1 Exemplo: Simulação de um Jogo de Dados com Dado “Viciado”

Imagine que estamos jogando um jogo em que o dado é “viciado” e não segue uma distribuição uniforme, ou seja, alguns números têm uma chance maior de serem sorteados. Por exemplo, o número 6 pode ter uma probabilidade maior, e os outros números, menores.

Isso nos permite mostrar como alterar a probabilidade de ocorrência de eventos em uma

distribuição discreta.

6 R

```
# Definindo as faces do dado e as probabilidades
faces <- 1:6
probabilidades <- c(0.05, 0.1, 0.15, 0.2, 0.25, 0.25) # Probabilidades associadas às faces

# Verificando que a soma das probabilidades é 1
cat("Soma das probabilidades:", sum(probabilidades), "\n")
```

Soma das probabilidades: 1

```
# Simulando 10000 lançamentos de um dado viciado
n_lancamentos <- 10000
set.seed(123) # Definindo seed para reprodutibilidade
resultados <- sample(faces, size = n_lancamentos, replace = TRUE, prob = probabilidades)

# Contando as frequências de cada face
frequencias <- table(resultados) / n_lancamentos

# Exibindo os resultados da simulação
cat("Frequências de cada face após", n_lancamentos, "lançamentos:\n")
```

Frequências de cada face após 10000 lançamentos:

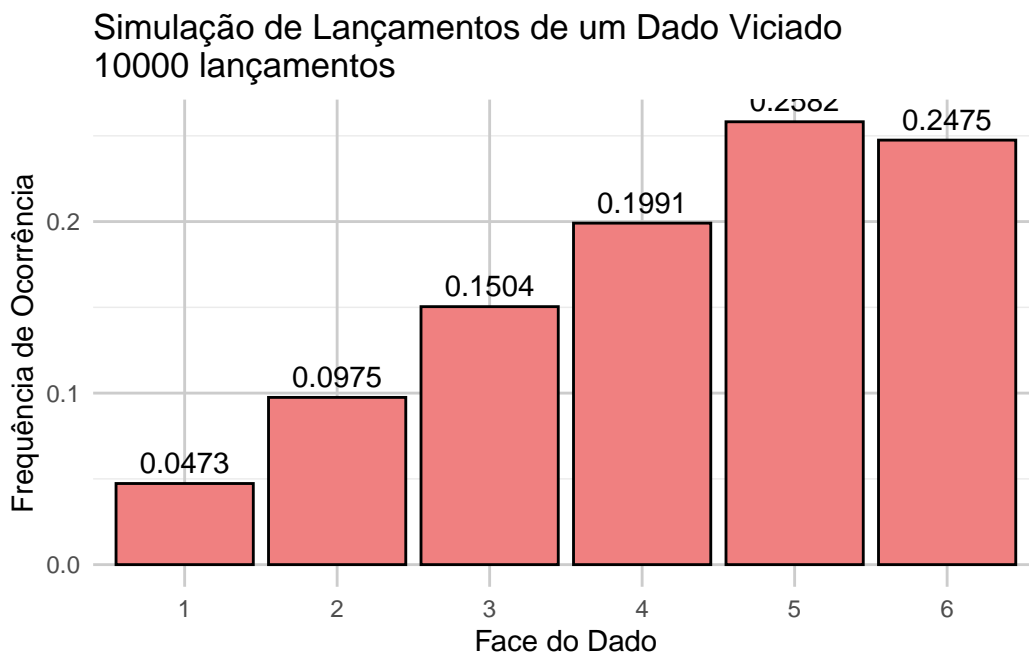
```
for (face in faces) {
  cat("Face", face, ":", frequencias[as.character(face)], "vezes\n")
}
```

```
Face 1 : 0.0473 vezes
Face 2 : 0.0975 vezes
Face 3 : 0.1504 vezes
Face 4 : 0.1991 vezes
Face 5 : 0.2582 vezes
Face 6 : 0.2475 vezes
```

```
# Visualizando os resultados em um gráfico de barras
library(ggplot2)

dados <- data.frame(faces = as.factor(faces), frequencias = as.numeric(frequencias))

ggplot(dados, aes(x = faces, y = frequencias)) +
  geom_bar(stat = "identity", fill = "lightcoral", color = "black") +
  ggtitle(paste0("Simulação de Lançamentos de um Dado Viciado\n", n_lancamentos, " lançamentos")) +
  xlab("Face do Dado") +
  ylab("Frequência de Ocorrência") +
  theme_minimal() +
  geom_text(aes(label = round(frequencias, 4)), vjust = -0.5) +
  theme(panel.grid.major = element_line(color = "grey80"))
```



7 Python

```
import numpy as np
import matplotlib.pyplot as plt

# Definindo as faces do dado e as probabilidades
faces = [1, 2, 3, 4, 5, 6]
probabilidades = [0.05, 0.1, 0.15, 0.2, 0.25, 0.25] # Probabilidades associadas às faces do

# Verificando que a soma das probabilidades é 1
print(f"Soma das probabilidades: {sum(probabilidades)}")
```

Soma das probabilidades: 1.0

```
# Simulando 10000 lançamentos de um dado viciado
n_lancamentos = 10000
resultados = np.random.choice(faces, size=n_lancamentos, p=probabilidades)

# Contando as frequências de cada face
frequencias = [np.sum(resultados == face) / n_lancamentos for face in faces]

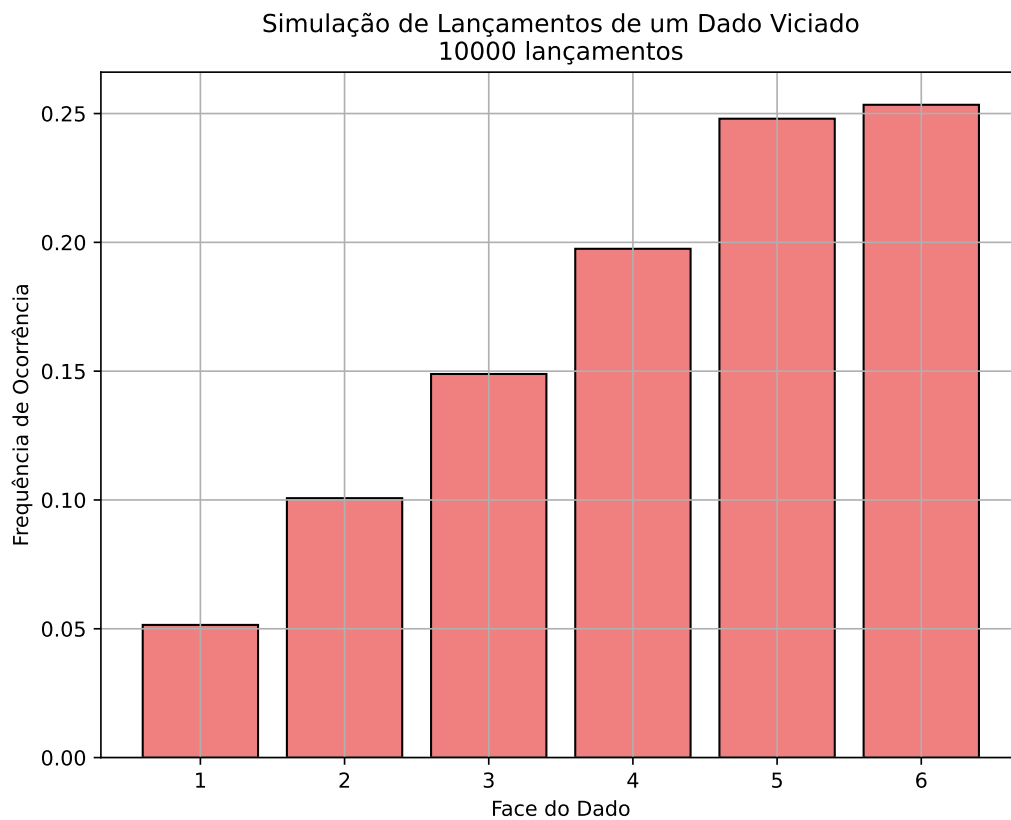
# Exibindo os resultados da simulação
print(f"Frequências de cada face após {n_lancamentos} lançamentos:")
```

Frequências de cada face após 10000 lançamentos:

```
for face, freq in zip(faces, frequencias):
    print(f"Face {face}: {freq} vezes")
```

Face 1: 0.0515 vezes
Face 2: 0.1007 vezes
Face 3: 0.1489 vezes
Face 4: 0.1975 vezes
Face 5: 0.248 vezes
Face 6: 0.2534 vezes

```
# Visualizando os resultados em um gráfico de barras
plt.figure(figsize=(8,6))
plt.bar(faces, frequencias, color='lightcoral', edgecolor='black')
plt.title(f'Simulação de Lançamentos de um Dado Viciado\n{n_lancamentos} lançamentos')
plt.xlabel('Face do Dado')
plt.ylabel('Frequência de Ocorrência')
plt.grid(True)
plt.show()
```



7.1 Pergunta: precisamos da função `np.random.choice/set.seed`?

8 R

```
# Definindo as faces do dado e as probabilidades associadas (não uniformes)
faces <- 1:6
probabilidades <- c(0.05, 0.1, 0.15, 0.2, 0.25, 0.25) # Probabilidades associadas às faces do dado

# Função para gerar uma amostra baseada em intervalos de probabilidades
gerar_amostra_por_intervalos <- function(probabilidades, faces) {
  u <- runif(1) # Gerando um número aleatório uniforme
  limite_inferior <- 0 # Limite inferior do intervalo

  # Percorrendo as probabilidades e verificando em qual intervalo o número cai
  for (i in seq_along(probabilidades)) {
    limite_superior <- limite_inferior + probabilidades[i] # Definindo o limite superior do intervalo
    if (limite_inferior <= u && u < limite_superior) {
      return(faces[i]) # Retorna a face correspondente ao intervalo
    }
    limite_inferior <- limite_superior # Atualiza o limite inferior para o próximo intervalo
  }
}

# Simulando lançamentos do dado viciado utilizando a verificação dos intervalos
n_lancamentos <- 10000
set.seed(123) # Definindo seed para reprodutibilidade
resultados <- replicate(n_lancamentos, gerar_amostra_por_intervalos(probabilidades, faces))

# Contando as frequências de cada face
frequencias <- sapply(faces, function(face) sum(resultados == face) / n_lancamentos)

# Exibindo os resultados da simulação
cat("Frequências de cada face após", n_lancamentos, "lançamentos:\n")
```

Frequências de cada face após 10000 lançamentos:

```
for (i in seq_along(faces)) {
  cat("Face", faces[i], ":", frequencias[i], "vezes\n")
}
```

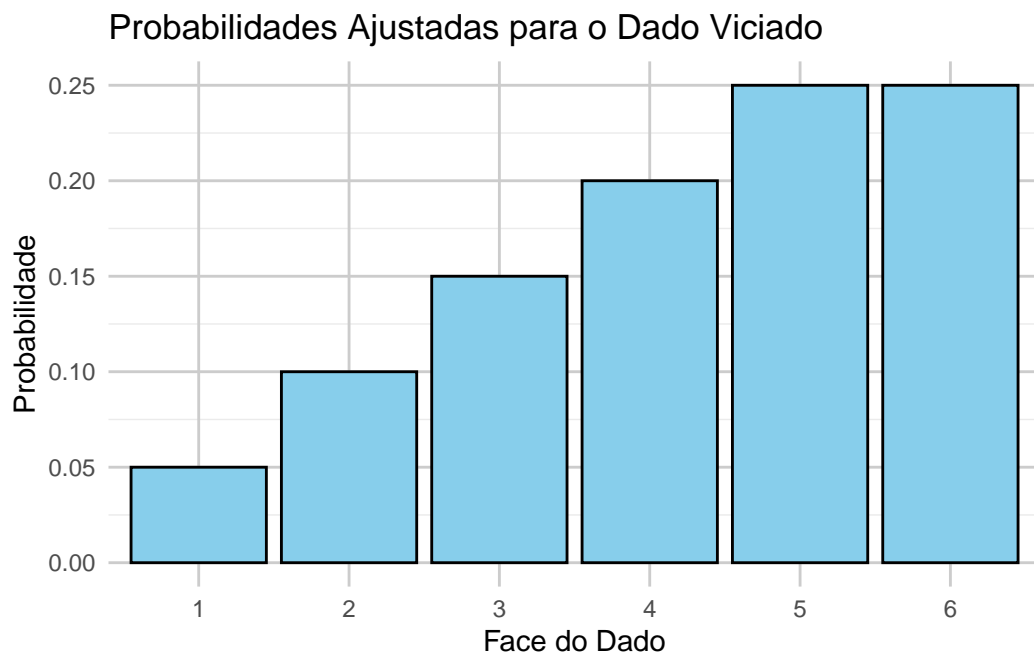
```
Face 1 : 0.0521 vezes
Face 2 : 0.0964 vezes
Face 3 : 0.1527 vezes
Face 4 : 0.2045 vezes
Face 5 : 0.2508 vezes
Face 6 : 0.2435 vezes
```

```
# Visualizando os resultados em gráficos
```

```
library(ggplot2)
```

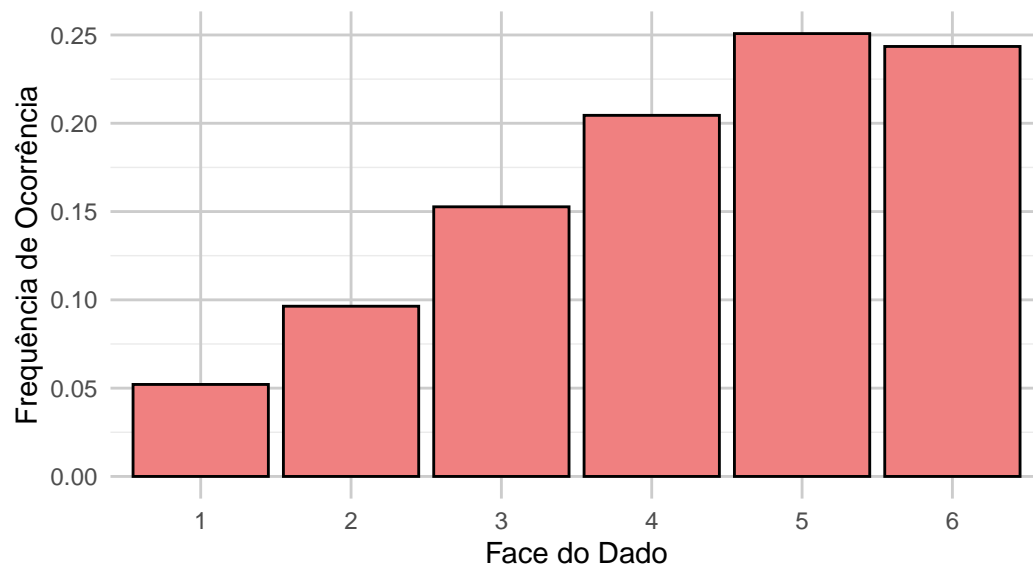
```
# Gráfico das probabilidades ajustadas
```

```
dados_probabilidades <- data.frame(faces = as.factor(faces), probabilidades = probabilidades)
ggplot(dados_probabilidades, aes(x = faces, y = probabilidades)) +
  geom_bar(stat = "identity", fill = "skyblue", color = "black") +
  ggtitle("Probabilidades Ajustadas para o Dado Viciado") +
  xlab("Face do Dado") +
  ylab("Probabilidade") +
  theme_minimal() +
  theme(panel.grid.major = element_line(color = "grey80"))
```



```
# Gráfico das frequências obtidas
dados_frequencias <- data.frame(faces = as.factor(faces), frequencias = frequencias)
ggplot(dados_frequencias, aes(x = faces, y = frequencias)) +
  geom_bar(stat = "identity", fill = "lightcoral", color = "black") +
  ggtitle(paste0("Simulação de Lançamentos de um Dado Viciado\n", n_lancamentos, " lançamentos")) +
  xlab("Face do Dado") +
  ylab("Frequência de Ocorrência") +
  theme_minimal() +
  theme(panel.grid.major = element_line(color = "grey80"))
```

Simulação de Lançamentos de um Dado Viciado
10000 lançamentos



9 Python

```
import numpy as np
import matplotlib.pyplot as plt

# Definindo as faces do dado e as probabilidades associadas (não uniformes)
faces = [1, 2, 3, 4, 5, 6]
probabilidades = [0.05, 0.1, 0.15, 0.2, 0.25, 0.25] # Probabilidades associadas às faces do

# Gerando um número aleatório e verificando em qual intervalo ele cai
def gerar_amostra_por_intervalos(probabilidades, faces):
    u = np.random.uniform(0, 1) # Gerando um número aleatório uniforme
    limite_inferior = 0 # Limite inferior do intervalo

    # Percorrendo as probabilidades e verificando em qual intervalo o número cai
    for i, p in enumerate(probabilidades):
        limite_superior = limite_inferior + p # Definindo o limite superior do intervalo
        if limite_inferior <= u < limite_superior:
            return faces[i] # Retorna a face correspondente ao intervalo
        limite_inferior = limite_superior # Atualiza o limite inferior para o próximo inter

# Simulando lançamentos do dado viciado utilizando a verificação dos intervalos
n_lancamentos = 10000
resultados = [gerar_amostra_por_intervalos(probabilidades, faces) for _ in range(n_lancamentos)]

# Contando as frequências de cada face
frequencias = [np.sum(np.array(resultados) == face) / n_lancamentos for face in faces]

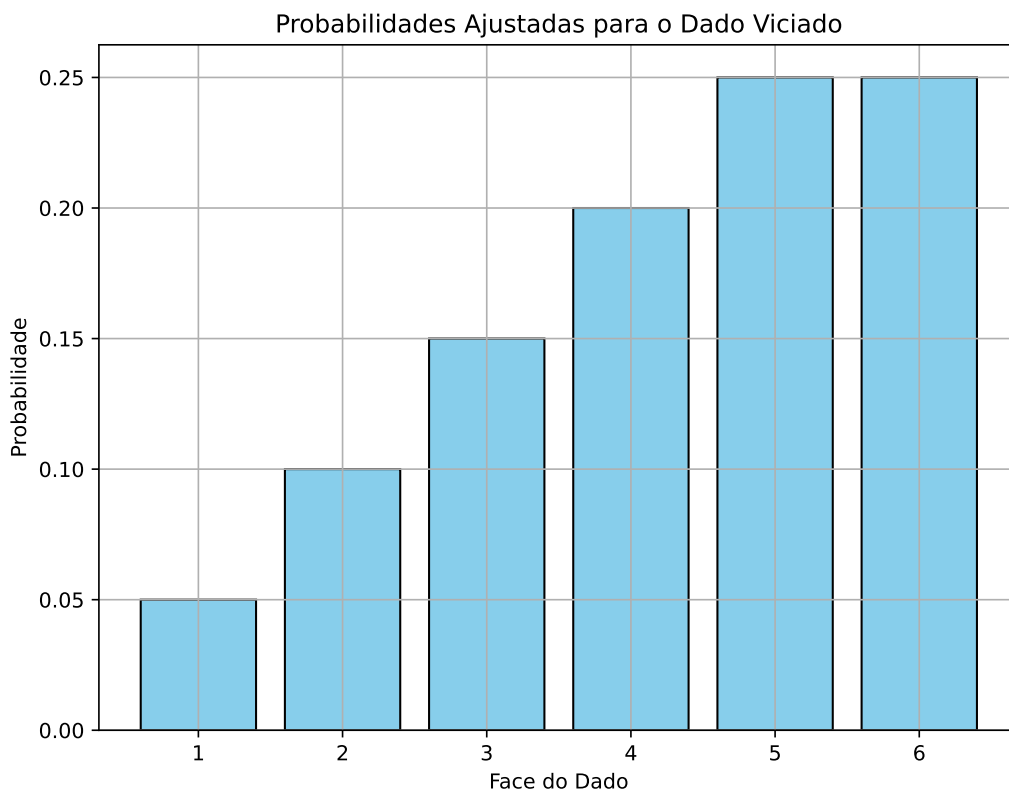
# Exibindo os resultados da simulação
print(f"Frequências de cada face após {n_lancamentos} lançamentos:")
```

Frequências de cada face após 10000 lançamentos:

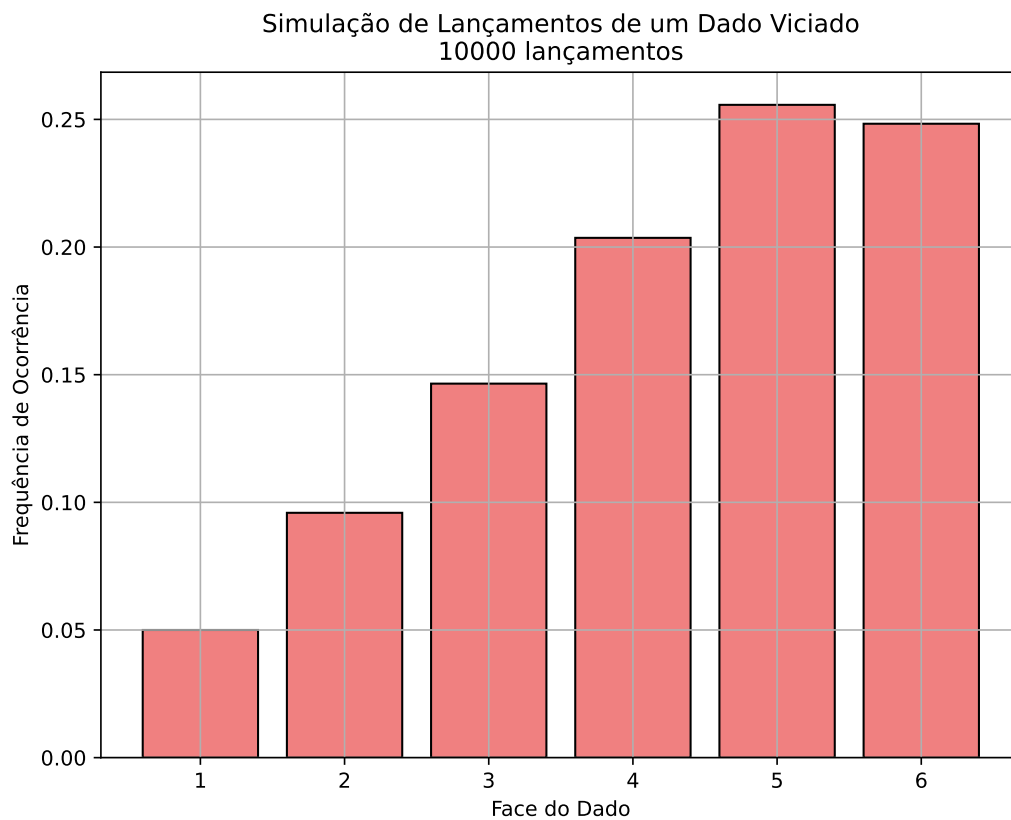
```
for face, freq in zip(faces, frequencias):
    print(f"Face {face}: {freq} vezes")
```

Face 1: 0.05 vezes
Face 2: 0.0959 vezes
Face 3: 0.1465 vezes
Face 4: 0.2036 vezes
Face 5: 0.2557 vezes
Face 6: 0.2483 vezes

```
# Gráfico das probabilidades ajustadas
plt.figure(figsize=(8,6))
plt.bar(faces, probabilidades, color='skyblue', edgecolor='black')
plt.title('Probabilidades Ajustadas para o Dado Viciado')
plt.xlabel('Face do Dado')
plt.ylabel('Probabilidade')
plt.grid(True)
plt.show()
```



```
# Gráfico das frequências obtidas
plt.figure(figsize=(8,6))
plt.bar(faces, frequencias, color='lightcoral', edgecolor='black')
plt.title(f'Simulação de Lançamentos de um Dado Viciado\n{n_lancamentos} lançamentos')
plt.xlabel('Face do Dado')
plt.ylabel('Frequência de Ocorrência')
plt.grid(True)
plt.show()
```



Ou seja, se conseguimos simular uma distribuição uniforme, conseguimos simular uma distribuição discreta. Isso vale de forma mais geral?

10 Números Pseudoaleatórios

Números aleatórios têm muitas aplicações na computação, como em simulações, amostragem estatística, criptografia e jogos de azar. No entanto, os computadores, por serem sistemas determinísticos, não podem gerar números realmente aleatórios de forma autônoma. Em vez disso, utilizam algoritmos determinísticos que geram números que parecem aleatórios, e esses números são chamados de pseudoaleatórios.

10.1 O que é um número pseudoaleatório?

Um número pseudoaleatório é gerado a partir de uma fórmula matemática que, a partir de uma semente (um valor inicial), gera uma sequência de números que tem as propriedades desejadas de uma sequência aleatória. Essa sequência parece aleatória, mas se a mesma semente for usada, a sequência será a mesma.

10.2 Geração de Números Pseudoaleatórios com o Gerador Linear Congruente (LCG)

[Link para o wikipedia](#)

O **Gerador Linear Congruente** (LCG) é um dos métodos mais antigos e simples para gerar números pseudoaleatórios. Ele segue a fórmula:

$$X_{n+1} = (a \cdot X_n + c) \mod m$$

Onde: - X_n é o número atual (ou a semente inicial), - a é o multiplicador, - c é o incremento, - m é o módulo, ou seja, o intervalo dos números gerados.

A sequência gerada pelo LCG depende diretamente dos parâmetros a , c , m e da semente inicial X_0 . Um conjunto mal escolhido de parâmetros pode resultar em uma sequência com um período curto, o que compromete a aleatoriedade da sequência.

10.2.1 O que é a Função Módulo?

A **função módulo** (também conhecida como operação de resto) retorna o **resto da divisão** de um número por outro. Em termos matemáticos, para dois números inteiros a e b , a operação módulo é representada como:

$$r = a \mod b$$

Onde: - a é o dividendo, - b é o divisor, - r é o resto da divisão de a por b .

Por exemplo, se temos $a = 17$ e $b = 5$, a divisão de 17 por 5 dá 3 com um resto de 2, então:

$$17 \mod 5 = 2$$

No contexto do **Gerador Linear Congruente (LCG)**, a função módulo é usada para garantir que os números gerados fiquem dentro de um intervalo específico, geralmente entre 0 e $m - 1$, onde m é o módulo definido no algoritmo.

11 R

```
# Exemplo de uso da função módulo em R

# Definindo os valores
a <- 17
b <- 3

# Calculando o módulo de a por b
resto <- a %% b

# Exibindo o resultado
cat("O resultado de", a, "%%", b, "é:", resto, "\n")
```

O resultado de 17 %% 3 é: 2

12 Python

```
# Exemplo de uso da função módulo em Python

# Definindo os valores
a = 17
b = 3

# Calculando o módulo de a por b
resto = a % b

# Exibindo o resultado
print(f"O resultado de {a} % {b} é: {resto}")
```

O resultado de 17 % 3 é: 2

12.1 Por que o Gerador Linear Congruente Funciona?

O Gerador Linear Congruente (LCG) é um dos métodos mais simples e eficientes para gerar números pseudoaleatórios. Sua eficácia se baseia em um bom equilíbrio entre a escolha dos parâmetros (multiplicador a , incremento c , módulo m e semente inicial X_0) e as propriedades matemáticas que garantem uma sequência suficientemente “aleatória”. Para que o LCG funcione bem, os parâmetros precisam ser cuidadosamente selecionados para garantir que a sequência gerada tenha um período longo, seja bem distribuída e evite padrões repetitivos.

12.1.1 A Fórmula do LCG

A fórmula básica do LCG é:

$$X_{n+1} = (a \cdot X_n + c) \mod m$$

Onde: - X_n é o número gerado na n -ésima iteração, - a é o multiplicador, - c é o incremento, - m é o módulo, - X_0 é a semente inicial.

O número gerado em cada iteração é o **resto da divisão** de $(a \cdot X_n + c)$ por m . Essa operação garante que os números fiquem dentro do intervalo $[0, m - 1]$. A normalização posterior geralmente transforma esses números em valores no intervalo $[0, 1)$.

12.1.2 O Papel de m

O valor de m , conhecido como módulo, define o intervalo no qual os números gerados estarão contidos. Em muitos casos, m é escolhido como uma potência de 2 (por exemplo, $m = 2^{32}$ ou $m = 2^{64}$) porque cálculos modulares com potências de 2 são mais rápidos em hardware.

A escolha de m também influencia o **período máximo** da sequência. Se todos os parâmetros forem escolhidos corretamente, o LCG pode gerar uma sequência com o período máximo, que é m . Isso significa que a sequência não repetirá nenhum número até que m números tenham sido gerados.

12.1.3 O Papel de a , c e a Condição de Coprimos

Para garantir que o gerador tenha o período máximo (ou seja, m números diferentes antes de repetir a sequência), a escolha dos parâmetros a (multiplicador), c (incremento) e m (módulo) deve satisfazer as seguintes condições baseadas em **teorias de números**:

1. **O incremento c deve ser coprimo com m :**

- Dois números são **coprimos** se o maior divisor comum deles for 1, ou seja, $\gcd(c, m) = 1$. Isso garante que, ao somar c , todos os possíveis valores de X_n possam ser atingidos antes de repetir a sequência.
- Se c não for coprimo com m , a sequência gerada pode pular certos valores, resultando em um período mais curto do que o esperado.

2. **O valor de $a - 1$ deve ser divisível por todos os fatores primos de m :**

- Se m é uma potência de 2 (por exemplo, $m = 2^k$), a escolha de a deve ser tal que $(a - 1)$ seja divisível por 2 para garantir que o período seja maximizado.

3. **Se m for divisível por 4, então $(a - 1)$ também deve ser divisível por 4:**

- Isso é necessário para garantir que todos os resíduos modulares possíveis possam ser gerados, especialmente quando m é uma potência de 2.

12.1.4 Exemplo de uma Escolha Correta de Parâmetros

Um exemplo clássico de um bom conjunto de parâmetros é:

- $m = 2^{32}$ (módulo com 32 bits),
- $a = 1664525$ (multiplicador),
- $c = 1013904223$ (incremento),
- $X_0 = 42$ (semente inicial, que pode ser qualquer valor).

Esses parâmetros foram escolhidos para garantir que o LCG tenha um longo período e uma boa distribuição dos números gerados. O módulo $m = 2^{32}$ é uma potência de 2, o que torna as operações modulares mais rápidas, e os valores de a e c satisfazem as condições matemáticas para maximizar o período.

13 R

```
# Importando o pacote necessário
library(gmp)

# Parâmetros do exemplo
m <- 2^32
a <- 1664525
c <- 1013904223

# Verificando as condições
# 1. O incremento c deve ser coprimo com m
coprimo_c_m <- gcd(c, m) == 1

# 2. a - 1 deve ser divisível por todos os fatores primos de m
a_menos_1 <- a - 1

print(a_menos_1)
```

```
[1] 1664524
```

```
# Verificando se a - 1 é divisível por 2 (único fator primo de m = 2^32)
divisivel_por_2 <- (a_menos_1 %% 2 == 0)

# 3. Se m for divisível por 4, a - 1 também deve ser divisível por 4
divisivel_por_4 <- (a_menos_1 %% 4 == 0)

list(coprimo_c_m, divisivel_por_2, divisivel_por_4)
```

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

```
[[2]]
[1] TRUE
```

```
[[3]]  
[1] TRUE
```

14 Python

```
import math

# Parâmetros do exemplo
m = 2**32
a = 1664525
c = 1013904223

# Verificando as condições
# 1. O incremento c deve ser coprimo com m
coprimo_c_m = math.gcd(c, m) == 1

# 2. a - 1 deve ser divisível por todos os fatores primos de m
a_menos_1 = a - 1

print(a_menos_1)
```

1664524

```
# Verificando se a - 1 é divisível por 2 (único fator primo de m = 2^32)
divisivel_por_2 = (a_menos_1 % 2 == 0)

# 3. Se m for divisível por 4, a - 1 também deve ser divisível por 4
divisivel_por_4 = (a_menos_1 % 4 == 0)

coprimo_c_m, divisivel_por_2, divisivel_por_4
```

(True, True, True)

14.0.1 Por que o LCG Funciona Bem?

O LCG funciona porque: - **As operações modulares** garantem que os números gerados estejam dentro de um intervalo fixo e possam cobrir todo o espaço de possíveis valores de maneira

ordenada. - **A escolha adequada dos parâmetros** garante que a sequência tenha um longo período (o maior possível dado m), evita padrões repetitivos e assegura que a sequência seja **pseudoaleatória** o suficiente para muitas aplicações, como simulações e métodos de Monte Carlo.

No entanto, o LCG pode não ser adequado para todas as aplicações, especialmente em criptografia, onde a previsibilidade é um problema. Para a maioria dos usos científicos e de simulação, ele ainda é uma escolha eficiente e simples.

14.0.2 Efeito dos Parâmetros no Gerador Linear Congruente (LCG)

Os parâmetros no Gerador Linear Congruente (LCG) têm um impacto significativo sobre a qualidade e as propriedades da sequência de números pseudoaleatórios gerados. Os parâmetros principais são:

1. Multiplicador a :

- Esse parâmetro é essencial para garantir que a sequência de números gerados tenha um bom período (o número de valores distintos antes de a sequência começar a se repetir). Se o valor de a não for bem escolhido, o período da sequência pode ser curto e a qualidade dos números gerados diminui.
- Bons valores de a são cruciais para evitar padrões repetitivos ou ciclos curtos.

2. Incremento c :

- O incremento c adiciona um valor fixo à sequência e é um dos fatores que pode garantir que todos os valores no intervalo $[0, m)$ sejam atingidos em algum momento, desde que os outros parâmetros também sejam bem escolhidos.
- Quando $c = 0$, o gerador é chamado de **multiplicativo**. Nessa forma, o LCG pode ter um comportamento menos uniforme.

3. Módulo m :

- O módulo define o intervalo dos números gerados. Comumente, m é escolhido como uma potência de 2 (por exemplo, $m = 2^{32}$) para facilitar os cálculos modulares em hardware e software.
- O valor de m também determina o período máximo da sequência de números. Com um módulo de m , o período máximo teórico que o LCG pode ter é m , mas isso depende da escolha correta dos parâmetros a e c .

4. Semente X_0 :

- A semente é o valor inicial de X_0 usado pelo LCG para iniciar a sequência. Mudar a semente resultará em uma sequência diferente, mas com o mesmo período e comportamento determinado pelos outros parâmetros.

- A semente garante que o algoritmo possa ser **reproduzido**. Se dois programas utilizarem a mesma semente com os mesmos parâmetros, ambos produzirão a mesma sequência de números.

14.0.3 Impacto dos Parâmetros:

1. Período da Sequência:

- O período é a quantidade de números gerados antes que a sequência comece a se repetir. Para obter o período máximo, os parâmetros a , c , m e a semente X_0 precisam ser cuidadosamente escolhidos.
- Se os parâmetros não forem bons, o gerador pode produzir uma sequência com um ciclo muito curto ou, pior, um conjunto pequeno de valores.

2. Distribuição dos Números:

- Embora o LCG gere números no intervalo $[0, 1)$, o quão bem distribuídos esses números estão nesse intervalo depende dos parâmetros.
- Parâmetros mal escolhidos podem causar uma distribuição não uniforme, onde certos intervalos terão mais números gerados que outros, levando a um comportamento indesejável.

3. Padrões Repetitivos:

- Se os parâmetros forem mal escolhidos, podem surgir padrões repetitivos que comprometem a aleatoriedade dos números. Esses padrões tornam o LCG inadequado para algumas aplicações, como criptografia ou simulações que exigem alta qualidade de aleatoriedade.

Por essas razões, a escolha dos parâmetros a , c , m e da semente X_0 é crítica para garantir que o LCG produza números pseudoaleatórios de alta qualidade e com um período longo.

15 R

```
# Carregando o pacote ggplot2
library(ggplot2)

# Classe para o Gerador Congruente Linear
LinearCongruentialGenerator <- setRefClass(
  "LinearCongruentialGenerator",
  fields = list(a = "numeric", c = "numeric", m = "numeric", semente = "numeric"),
  methods = list(
    initialize = function(semente, a = 1103515245, c = 12345, m = 2^32) {
      .self$a <- a
      .self$c <- c
      .self$m <- m
      .self$semente <- semente
    },
    gerar = function() {
      # Atualizando a semente
      .self$semente <- (.self$a * .self$semente + .self$c) %% .self$m
      return(.self$semente / .self$m) # Normalizando para [0, 1)
    }
  )
)

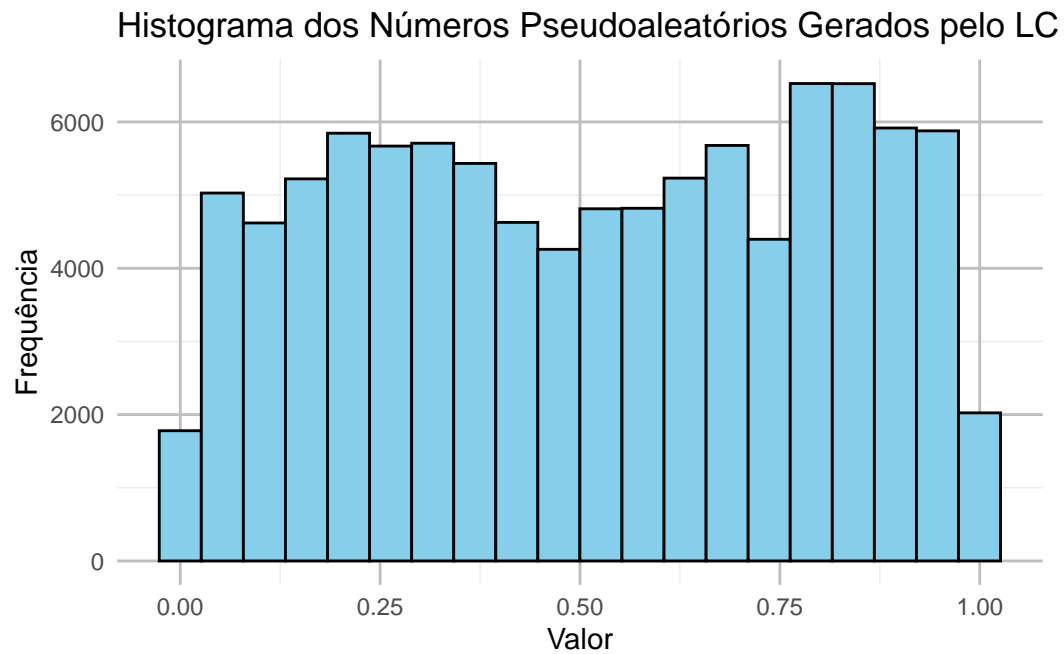
# Inicializando o gerador com uma semente
lcg <- LinearCongruentialGenerator$new(semente = 5)

# Gerando 1000 números pseudoaleatórios
numeros_gerados <- sapply(1:100000, function(x) lcg$gerar())

# Convertendo para um data.frame
dados <- data.frame(numeros_gerados = numeros_gerados)

# Plotando o histograma dos números gerados
ggplot(dados, aes(x = numeros_gerados)) +
  geom_histogram(bins = 20, fill = 'skyblue', color = 'black') +
```

```
ggtitle('Histograma dos Números Pseudoaleatórios Gerados pelo LCG') +  
xlab('Valor') +  
ylab('Frequência') +  
theme_minimal() +  
theme(panel.grid.major = element_line(color = "grey"))
```



16 Python

```
import matplotlib.pyplot as plt

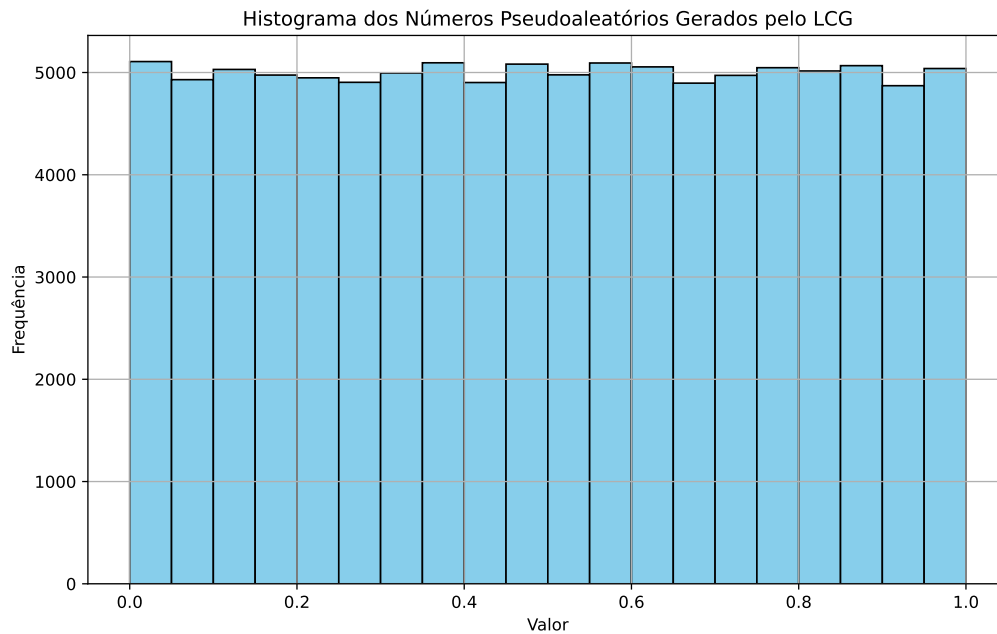
class LinearCongruentialGenerator:
    def __init__(self, semente, a=1103515245, c=12345, m=2**32):
        self.a = a
        self.c = c
        self.m = m
        self.semente = semente

    def gerar(self):
        # Atualizando a semente
        self.semente = (self.a * self.semente + self.c) % self.m
        return self.semente / self.m # Normalizando para [0, 1)

# Inicializando o gerador com uma semente
lcg = LinearCongruentialGenerator(semente=5)

# Gerando 1000 números pseudoaleatórios
numeros_gerados = [lcg.gerar() for _ in range(100000)]

# Plotando o histograma dos números gerados
plt.figure(figsize=(10, 6))
plt.hist(numeros_gerados, bins=20, color='skyblue', edgecolor='black')
plt.title('Histograma dos Números Pseudoaleatórios Gerados pelo LCG')
plt.xlabel('Valor')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()
```



16.1 Gerando números uniformes com uma moeda

Este código mostra como é possível gerar números uniformemente distribuídos no intervalo $[0, 1]$ utilizando o lançamento de uma moeda justa. A ideia central baseia-se na expansão binária, onde cada lançamento da moeda representa um bit da representação binária do número. Ao somar $2^{-(i+1)}$ para cada bit, construímos o número em base 2, garantindo uma distribuição uniforme. Isso ocorre porque cada bit tem uma probabilidade igual de ser 0 ou 1, criando uma sequência que cobre uniformemente o intervalo desejado.

17 R

```
# Carregando pacotes necessários
library(ggplot2)

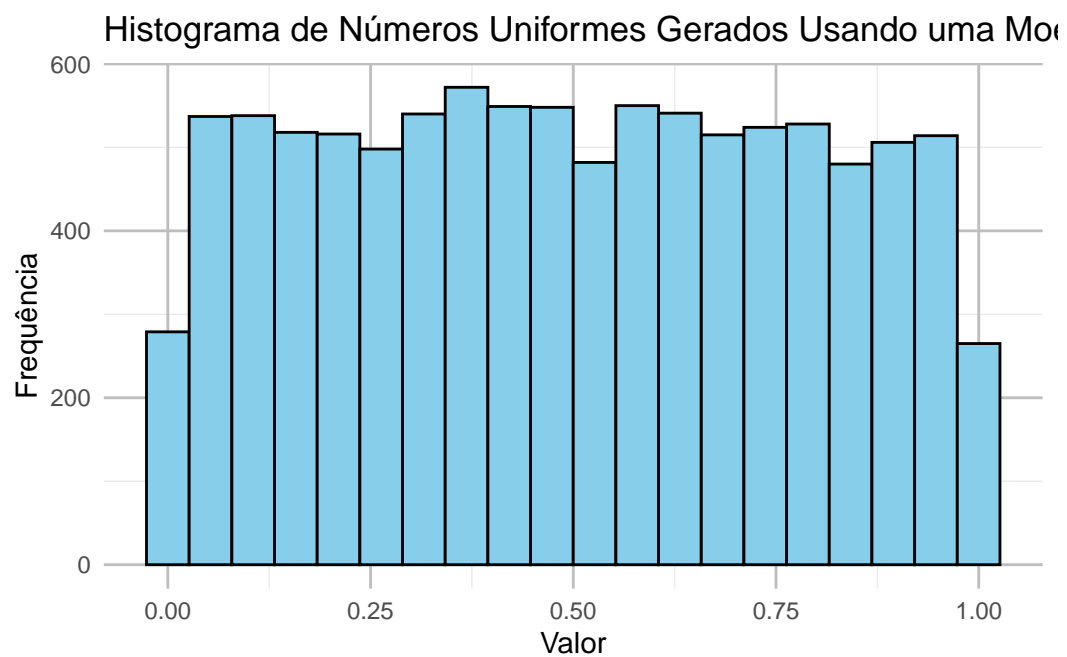
# Função para simular o lançamento de uma moeda justa
lancar_moeda <- function() {
  # Lançar moeda justa: 0 para coroa (K) e 1 para cara (C)
  sample(c(0, 1), 1)
}

# Função para gerar um número uniformemente distribuído usando uma moeda
gerar_numero_uniforme <- function(n_bits = 32) {
  numero <- 0
  for (i in 1:n_bits) {
    bit <- lancar_moeda()
    # Atualizando o número, multiplicando pela base 2
    numero <- numero + bit * (2-(i))
  }
  return(numero)
}

# Gerando 10000 números uniformemente distribuídos
numeros_uniformes <- sapply(1:10000, function(x) gerar_numero_uniforme())

# Convertendo para um data.frame
dados <- data.frame(numeros_uniformes = numeros_uniformes)

# Plotando o histograma dos números gerados
ggplot(dados, aes(x = numeros_uniformes)) +
  geom_histogram(bins = 20, fill = 'skyblue', color = 'black') +
  ggtitle('Histograma de Números Uniformes Gerados Usando uma Moeda Justa') +
  xlab('Valor') +
  ylab('Frequência') +
  theme_minimal() +
  theme(panel.grid.major = element_line(color = "grey"))
```



18 Python

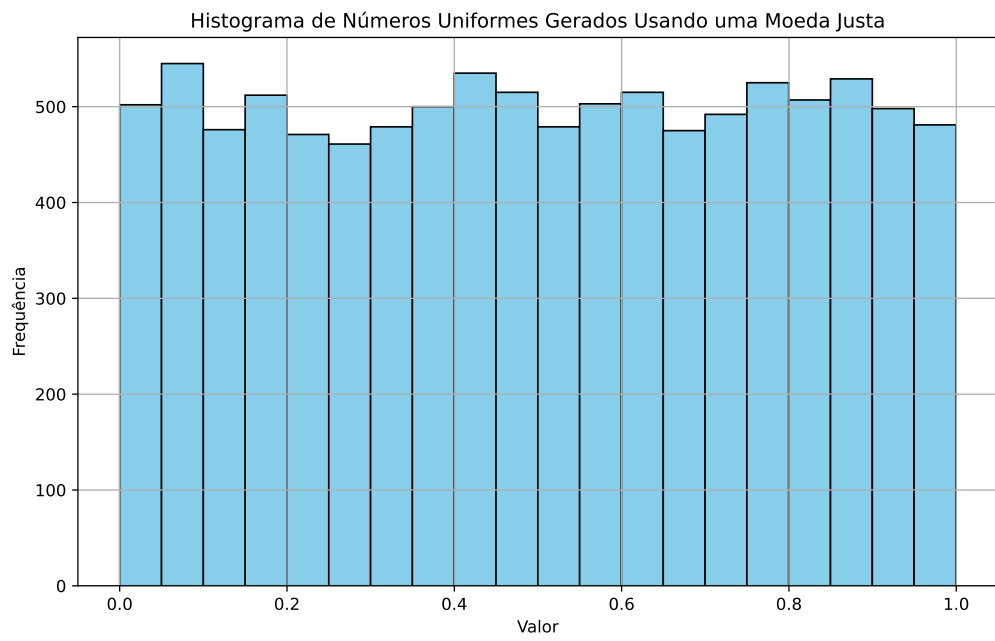
```
import random
import matplotlib.pyplot as plt

# Função para simular o lançamento de uma moeda justa
def lancar_moeda():
    # Lançar moeda justa: 0 para coroa (K) e 1 para cara (C)
    return random.choice([0, 1])

# Função para gerar um número uniformemente distribuído usando uma moeda
def gerar_numero_uniforme(n_bits=32):
    numero = 0
    for i in range(n_bits):
        bit = lancar_moeda()
        # Atualizando o número, multiplicando pela base 2
        numero += bit * (2 ** -(i + 1)) # Cada bit tem um peso de 2-(posição)
    return numero

# Gerando 1000 números uniformemente distribuídos
numeros_uniformes = [gerar_numero_uniforme() for _ in range(10000)]

# Plotando o histograma dos números gerados
plt.figure(figsize=(10, 6))
plt.hist(numeros_uniformes, bins=20, color='skyblue', edgecolor='black')
plt.title('Histograma de Números Uniformes Gerados Usando uma Moeda Justa')
plt.xlabel('Valor')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()
```



19 Técnica da Inversão para Variáveis Discretas

A **técnica da inversão** é uma maneira poderosa de gerar variáveis aleatórias (v.a.) a partir de uma distribuição arbitrária, usando números aleatórios uniformes no intervalo $[0, 1)$. A ideia básica é usar a **função de distribuição acumulada (CDF)** para mapear um número uniforme gerado entre 0 e 1 para o valor correspondente da v.a. discreta.

Passos Gerais para Gerar Variáveis Aleatórias Discretas com a Técnica da Inversão:

1. **Calcular a CDF** da variável aleatória que se deseja gerar.
2. **Gerar um número aleatório uniforme** $u \in [0, 1)$.
3. Encontrar o valor da variável aleatória cuja CDF seja maior ou igual a u .
4. Retornar esse valor como a variável aleatória gerada.

19.1 Geração de Variáveis Aleatórias Discretas Genéricas

Dado um conjunto de probabilidades $p(x_i)$, onde x_i são os valores possíveis da variável aleatória discreta, e $p(x_i)$ são suas respectivas probabilidades, a CDF $F(x)$ é calculada como:

$$F(x_i) = \sum_{j=1}^i p(x_j)$$

O algoritmo para gerar uma variável aleatória discreta genérica é:

1. Gerar um número aleatório $u \in [0, 1)$.
2. Encontrar o menor valor x_i tal que $F(x_i) \geq u$.
3. Retornar x_i .

20 R

```
# Exemplo de valores e probabilidades de uma variável aleatória discreta
valores <- c(0, 1, 2, 3, 4, 5, 6)
probabilidades <- c(0, 0.1, 0.2, 0.3, 0.25, 0.15, 0)

# Calculando a CDF
cdf <- cumsum(probabilidades)

# Gerando um número aleatório uniforme
u <- runif(1)

# Encontrando o valor correspondente na CDF
valor_gerado <- NA
for (i in seq_along(valores)) {
  if (u < cdf[i]) {
    valor_gerado <- valores[i]
    break
  }
}

# Ajustando o gráfico para corrigir a visualização da CDF e garantir que os valores estejam corretos
library(ggplot2)

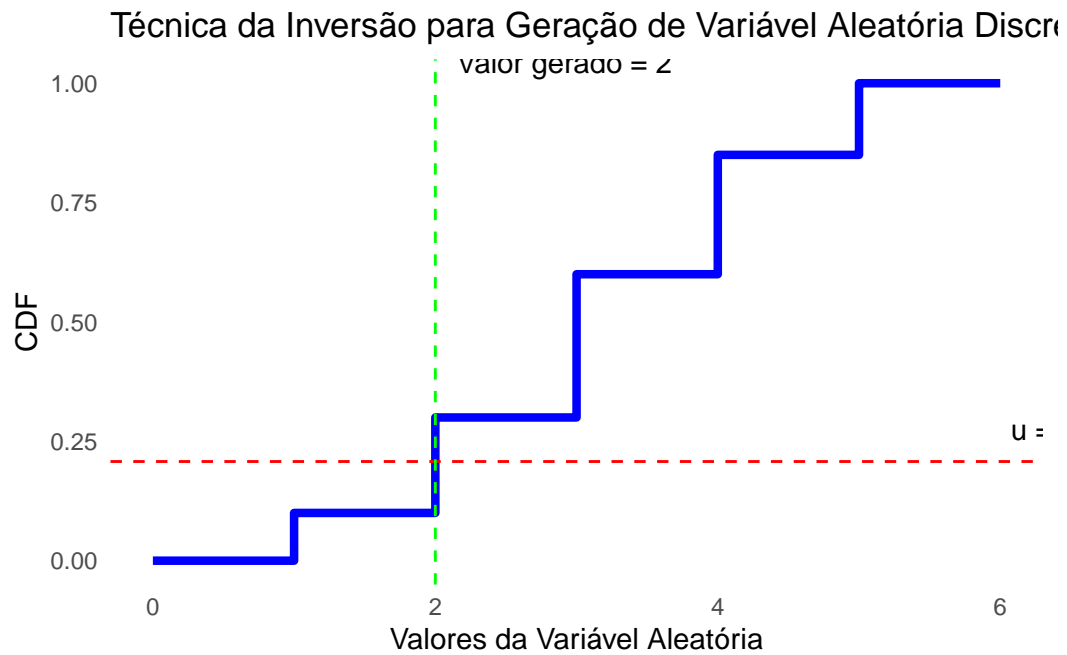
# Criando um dataframe para os valores e CDF
df <- data.frame(valores = valores, cdf = cdf)

# Gráfico da CDF com número aleatório e valor gerado
ggplot(df, aes(x = valores, y = cdf)) +
  geom_step(direction = "hv", color = "blue", size = 1.5) +
  geom_hline(yintercept = u, color = "red", linetype = "dashed") +
  geom_vline(xintercept = valor_gerado, color = "green", linetype = "dashed") +
  labs(title = "Técnica da Inversão para Geração de Variável Aleatória Discreta",
       x = "Valores da Variável Aleatória",
       y = "CDF") +
  annotate("text", x = max(valores), y = u, label = sprintf("u = %.2f", u), hjust = -0.1, vjust = "top")
```

```

annotate("text", x = valor_gerado, y = max(cdf), label = paste("Valor gerado =", valor_gerado),
theme_minimal() +
theme(panel.grid = element_blank())

```



21 Python

```
import numpy as np
import matplotlib.pyplot as plt

# Exemplo de valores e probabilidades de uma variável aleatória discreta
valores = [0, 1, 2, 3, 4, 5, 6]
probabilidades = [0, 0.1, 0.2, 0.3, 0.25, 0.15, 0]

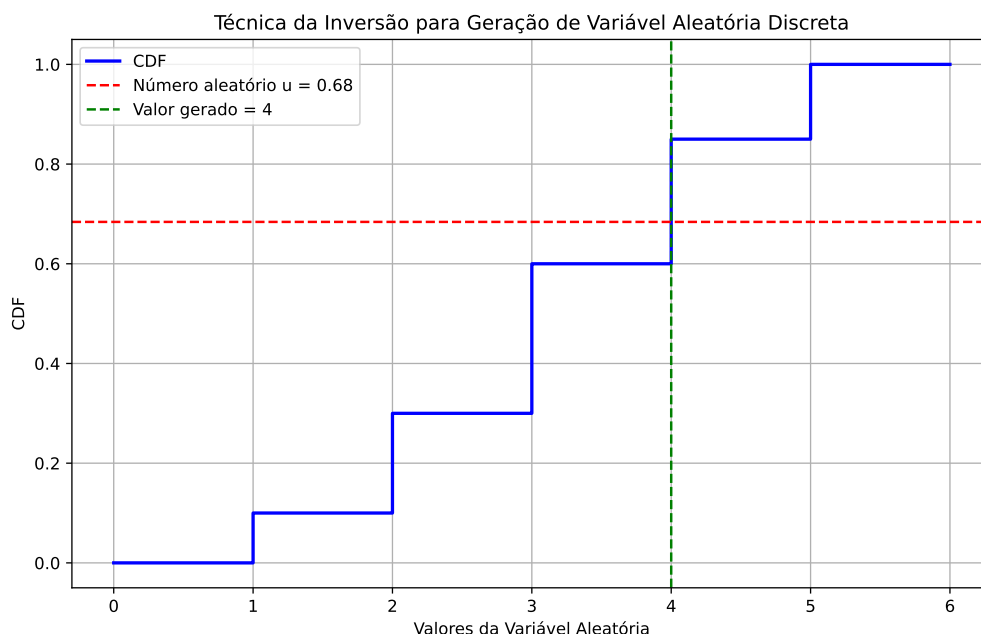
# Calculando a CDF
cdf = np.cumsum(probabilidades)

# Gerando um número aleatório uniforme
u = np.random.uniform(0, 1)

# Encontre o valor correspondente na CDF
valor_gerado = None
for i, valor in enumerate(valores):
    if u < cdf[i]:
        valor_gerado = valor
        break

# Ajustando o gráfico para corrigir a visualização da CDF e garantir que os valores estejam corretos
plt.figure(figsize=(10, 6))

# Ajustando o eixo x para que a CDF comece e termine corretamente
plt.step(valores, cdf, label='CDF', color='blue', linewidth=2, where='post')
plt.axhline(y=u, color='red', linestyle='--', label=f'Número aleatório u = {u:.2f}')
plt.axvline(x=valor_gerado, color='green', linestyle='--', label=f'Valor gerado = {valor_gerado}')
plt.title('Técnica da Inversão para Geração de Variável Aleatória Discreta')
plt.xlabel('Valores da Variável Aleatória')
plt.ylabel('CDF')
plt.legend()
plt.grid(True)
plt.show()
```



21.1 Inversa da CDF

A **inversa da CDF** (função de distribuição acumulada), também conhecida como a **função quantil** ou **função percentil**, é uma função utilizada para gerar variáveis aleatórias a partir de uma distribuição específica.

Definição: Seja $F(x)$ a função de distribuição acumulada (CDF) de uma variável aleatória X . A inversa da CDF, denotada por $F^{-1}(p)$, é definida como:

$$F^{-1}(p) = \inf\{x \in \mathbb{R} : F(x) \geq p\}, \quad \text{para } p \in [0, 1]$$

Em palavras: - A inversa da CDF $F^{-1}(p)$ mapeia um número p , que representa uma probabilidade acumulada, de volta ao valor x correspondente da variável aleatória X , tal que a probabilidade acumulada até x é igual a p . - Isso significa que, se $p = F(x)$, então $F^{-1}(p) = x$.

21.2 Geração de Variáveis Aleatórias com Distribuição Geométrica

A distribuição geométrica modela o número de tentativas até o primeiro sucesso em uma sequência de experimentos de Bernoulli. Se a probabilidade de sucesso em cada tentativa é p ,

a PMF é dada por:

21.2.1 Derivação da Inversa da CDF para a Distribuição Geométrica

A fórmula da **inversa da CDF** para a distribuição geométrica foi obtida a partir da definição da função de distribuição acumulada (CDF) da distribuição geométrica e da aplicação da técnica da inversão.

21.2.1.1 CDF da Distribuição Geométrica

A função de distribuição acumulada (CDF) da distribuição geométrica com parâmetro p (a probabilidade de sucesso) para o número de falhas $k - 1$ antes do primeiro sucesso é dada por:

$$F(k) = 1 - (1 - p)^k$$

Essa equação expressa a probabilidade acumulada de obter o primeiro sucesso em até k tentativas.

21.2.1.2 Inversa da CDF

Queremos encontrar a **inversa da CDF**, ou seja, a fórmula que, dado um valor u entre 0 e 1, nos permita calcular o valor k tal que $F(k) = u$.

Sabemos que:

$$u = F(k) = 1 - (1 - p)^k$$

Nosso objetivo é resolver essa equação para k . Vamos fazer isso passo a passo.

21.2.1.3 Isolando o termo com k

Começamos isolando o termo $(1 - p)^k$:

$$u = 1 - (1 - p)^k$$

Subtraindo 1 de ambos os lados:

$$u - 1 = -(1 - p)^k$$

Multiplicando ambos os lados por -1 :

$$1 - u = (1 - p)^k$$

21.2.1.4 Aplicando o Logaritmo

Agora aplicamos o logaritmo natural (log base e) em ambos os lados para resolver k :

$$\log(1 - u) = \log((1 - p)^k)$$

Usando a propriedade dos logaritmos que permite trazer o expoente k para frente:

$$\log(1 - u) = k \cdot \log(1 - p)$$

21.2.1.5 Isolando k

Agora, isolamos k :

$$k = \frac{\log(1 - u)}{\log(1 - p)}$$

Como k precisa ser um número inteiro (já que a distribuição geométrica conta o número de tentativas), usamos a função de arredondamento “para cima” ($\lceil \cdot \rceil$), conhecida como a **função teto**:

$$k = \lceil \frac{\log(1 - u)}{\log(1 - p)} \rceil$$

21.2.2 Conclusão

Portanto, a fórmula da inversa da CDF da distribuição geométrica é:

$$k = \lceil \frac{\log(1 - u)}{\log(1 - p)} \rceil$$

Esta fórmula nos permite gerar variáveis aleatórias com distribuição geométrica a partir de um número aleatório uniforme $u \in [0, 1)$.

22 R

```
# Função para gerar a inversa da CDF para a distribuição geométrica
inversa_cdf_geometrica <- function(p, u) {
  # Usando a fórmula inversa da CDF geométrica:  $F^{-1}(u) = \text{ceil}(\log(1 - u) / \log(1 - p))$ 
  k <- ceiling(log(1 - u) / log(1 - p))
  return(as.integer(k))
}

# Parâmetro p da distribuição geométrica
p <- 0.5

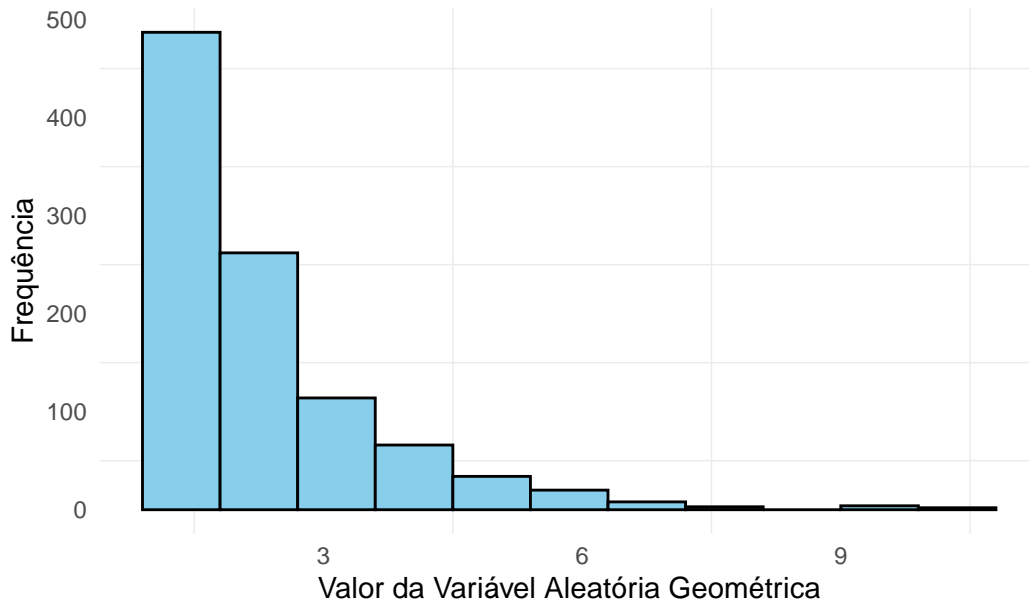
# Gerando 1000 números uniformemente distribuídos
uniformes <- runif(1000)

# Gerando a variável aleatória geométrica correspondente para cada número uniforme
geometricas <- sapply(uniformes, inversa_cdf_geometrica, p = p)

# Plotando um histograma das variáveis geométricas geradas
library(ggplot2)

df <- data.frame(geometricas = geometricas)
ggplot(df, aes(x = geometricas)) +
  geom_histogram(bins = max(geometricas) + 1, color = "black", fill = "skyblue", boundary = "black") +
  labs(title = "Histograma de Variáveis Aleatórias Geométricas Usando a Inversa da CDF",
       x = "Valor da Variável Aleatória Geométrica",
       y = "Frequência") +
  theme_minimal() +
  theme(panel.grid.major = element_blank())
```

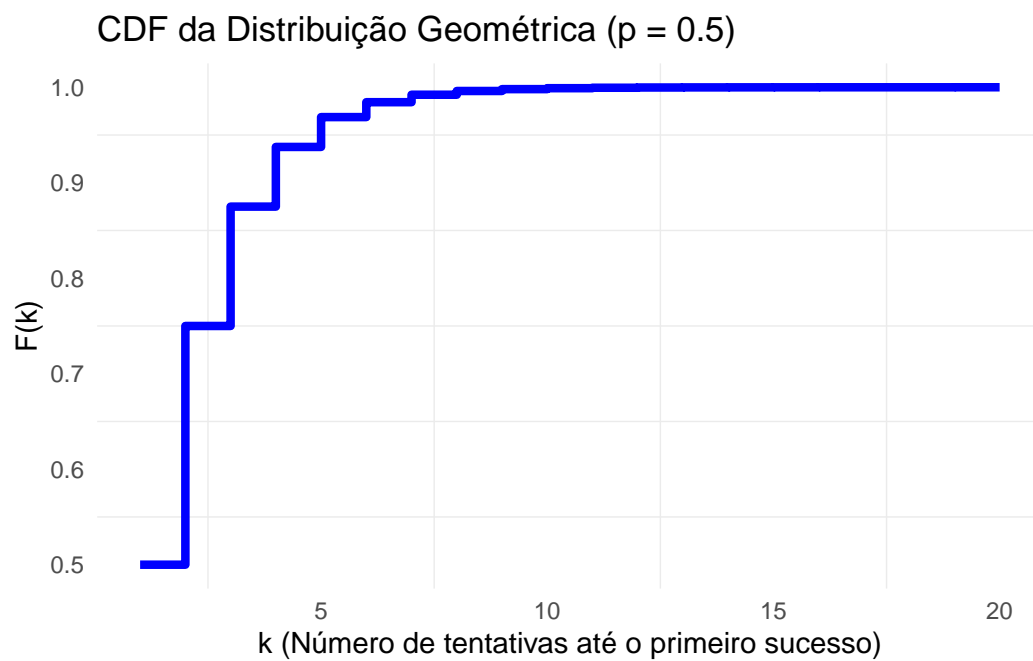
Histograma de Variáveis Aleatórias Geométricas Usando a Inv



```
# Função para calcular a CDF da distribuição geométrica
cdf_geometrica <- function(k, p) {
  return(1 - (1 - p)^k)
}

# Gerando valores de k para plotar a CDF
k_values <- 1:20
cdf_values <- sapply(k_values, cdf_geometrica, p = p)

# Plotando a CDF da distribuição geométrica
df_cdf <- data.frame(k_values = k_values, cdf_values = cdf_values)
ggplot(df_cdf, aes(x = k_values, y = cdf_values)) +
  geom_step(direction = "hv", color = "blue", size = 1.5) +
  labs(title = "CDF da Distribuição Geométrica (p = 0.5)",
       x = "k (Número de tentativas até o primeiro sucesso)",
       y = "F(k)") +
  theme_minimal() +
  theme(panel.grid.major = element_blank())
```



23 Python

```
import numpy as np
import matplotlib.pyplot as plt

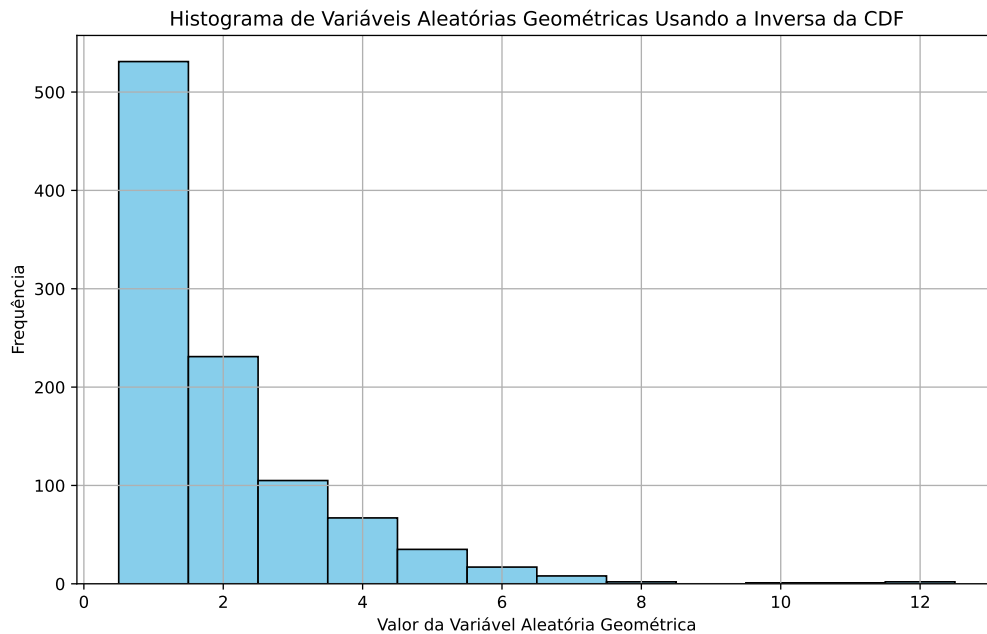
# Função para gerar a inversa da CDF para a distribuição geométrica
def inversa_cdf_geometrica(p, u):
    # Usando a fórmula inversa da CDF geométrica:  $F^{-1}(u) = \text{ceil}(\log(1 - u) / \log(1 - p))$ 
    k = np.ceil(np.log(1 - u) / np.log(1 - p))
    return int(k)

# Parâmetro p da distribuição geométrica
p = 0.5

# Gerando 100 números uniformemente distribuídos
uniformes = np.random.uniform(0, 1, 1000)

# Gerando a variável aleatória geométrica correspondente para cada número uniforme
geometricas = [inversa_cdf_geometrica(p, u) for u in uniformes]

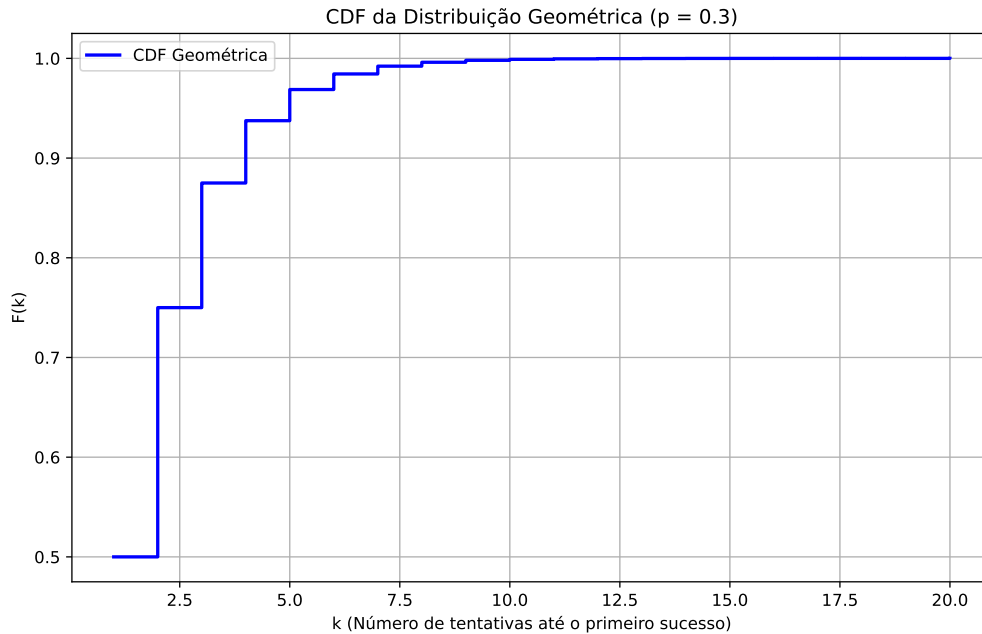
# Plotando um histograma das variáveis geométricas geradas
plt.figure(figsize=(10, 6))
plt.hist(geometricas, bins=range(1, max(geometricas) + 1), color='skyblue', edgecolor='black')
plt.title('Histograma de Variáveis Aleatórias Geométricas Usando a Inversa da CDF')
plt.xlabel('Valor da Variável Aleatória Geométrica')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()
```



```
# Calculando a CDF para a distribuição geométrica
def cdf_geometrica(k, p):
    return 1 - (1 - p)**k

# Gerando valores de k para plotar a CDF
k_values = np.arange(1, 21)
cdf_values = [cdf_geometrica(k, p) for k in k_values]

# Plotando a CDF da distribuição geométrica
plt.figure(figsize=(10, 6))
plt.step(k_values, cdf_values, where='post', color='blue', label='CDF Geométrica', linewidth=2)
plt.title('CDF da Distribuição Geométrica (p = 0.3)')
plt.xlabel('k (Número de tentativas até o primeiro sucesso)')
plt.ylabel('F(k)')
plt.grid(True)
plt.legend()
plt.show()
```



23.1 Geração de Variáveis Aleatórias com Distribuição Poisson

A distribuição de Poisson é usada para modelar o número de eventos que ocorrem em um intervalo de tempo ou espaço fixo, onde os eventos ocorrem com uma taxa constante λ e de forma independente.

A função de probabilidade de massa (PMF) da distribuição de Poisson é dada por:

$$P(X = k) = \frac{\lambda^k e^{-\lambda}}{k!}, \quad k = 0, 1, 2, \dots$$

23.1.1 Derivação da Inversa da CDF para a Distribuição Poisson Usando a Fórmula Recursiva

A **distribuição de Poisson** tem uma fórmula recursiva que pode ser usada para calcular as probabilidades de forma mais eficiente. Em vez de recalculer a probabilidade $P(X = k)$ a cada vez, podemos usar a seguinte relação recursiva:

$$P(X = k + 1) = \frac{\lambda}{k + 1} \cdot P(X = k)$$

Onde $P(X = 0) = e^{-\lambda}$.

Essa relação recursiva permite gerar variáveis aleatórias de Poisson sem precisar calcular fatoriais repetidamente, o que é mais eficiente para grandes valores de λ ou grandes números de eventos k .

23.1.2 Técnica da Inversão Usando a Fórmula Recursiva

Para gerar uma variável aleatória de Poisson usando a técnica da inversão e a fórmula recursiva, o processo é o seguinte:

1. Gerar um número aleatório uniforme $u \in [0, 1)$.
2. Calcular a CDF acumulada para valores de k , somando as probabilidades da PMF (usando a relação recursiva) até que a CDF acumulada seja maior ou igual a u .
3. O menor valor k tal que $F(k) \geq u$ será o número de eventos gerado pela distribuição Poisson.

23.1.3 Explicação:

1. **Fórmula Recursiva:** A relação recursiva $P(X = k + 1) = \frac{\lambda}{k+1} \cdot P(X = k)$ permite calcular as probabilidades de forma eficiente, atualizando a probabilidade de $P(X = k + 1)$ a partir de $P(X = k)$.
2. **Eficiência:** Esta técnica evita o cálculo repetido de fatoriais, tornando-a mais eficiente, especialmente quando se está gerando variáveis para grandes valores de λ ou quando se deseja calcular probabilidades para muitos eventos k .

Este método é amplamente utilizado por ser mais rápido e computacionalmente eficiente para a geração de variáveis aleatórias de Poisson.

24 R

```
# Função para gerar a inversa da CDF para a distribuição Poisson usando a técnica de inversão
inversa_cdf_poisson_recurativa <- function(lam, u) {
  k <- 0
  p <- exp(-lam) # P(X=0)
  F <- p # Iniciamos com a probabilidade P(X=0)

  # Continuamos somando até que F >= u
  while (u > F) {
    k <- k + 1
    p <- p * lam / k # Atualiza a probabilidade recursivamente para o próximo valor
    F <- F + p
  }

  return(k)
}

# Parâmetro lambda da distribuição Poisson
lam <- 3

# Gerando 1000 números uniformemente distribuídos
uniformes <- runif(1000)

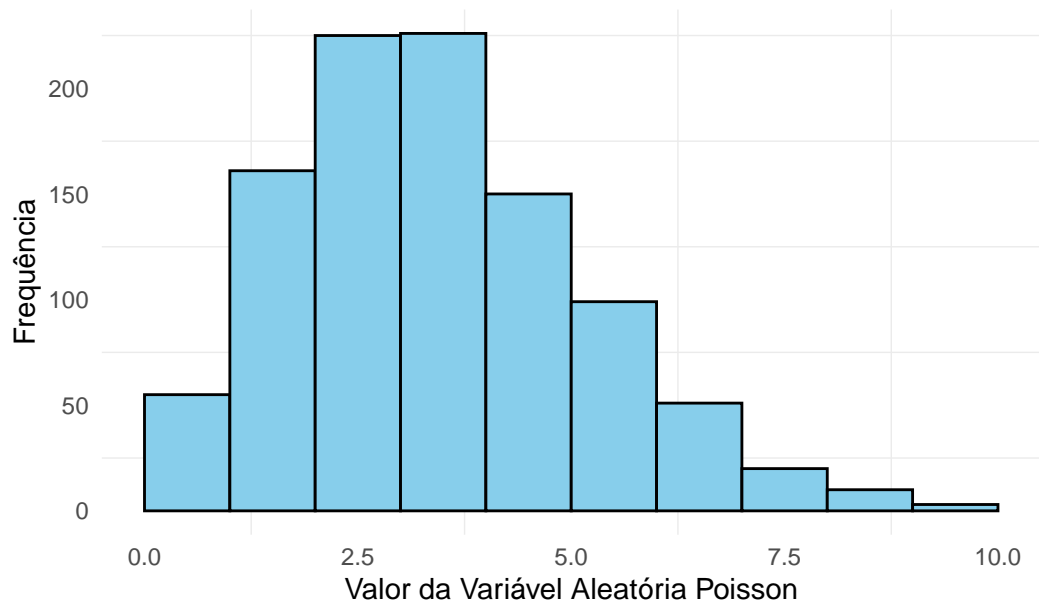
# Gerando a variável aleatória Poisson correspondente para cada número uniforme
poisson_vars <- sapply(uniformes, inversa_cdf_poisson_recurativa, lam = lam)

# Plotando o histograma das variáveis Poisson geradas
library(ggplot2)

df <- data.frame(poisson_vars = poisson_vars)
ggplot(df, aes(x = poisson_vars)) +
  geom_histogram(bins = max(poisson_vars) + 1, color = "black", fill = "skyblue", boundary =
  labs(title = "Histograma de Variáveis Aleatórias Poisson Usando a Fórmula Recursiva ( = 3)",
    x = "Valor da Variável Aleatória Poisson",
    y = "Frequência") +
```

```
theme_minimal() +
theme(panel.grid.major = element_blank())
```

Histograma de Variáveis Aleatórias Poisson Usando a Fórmula

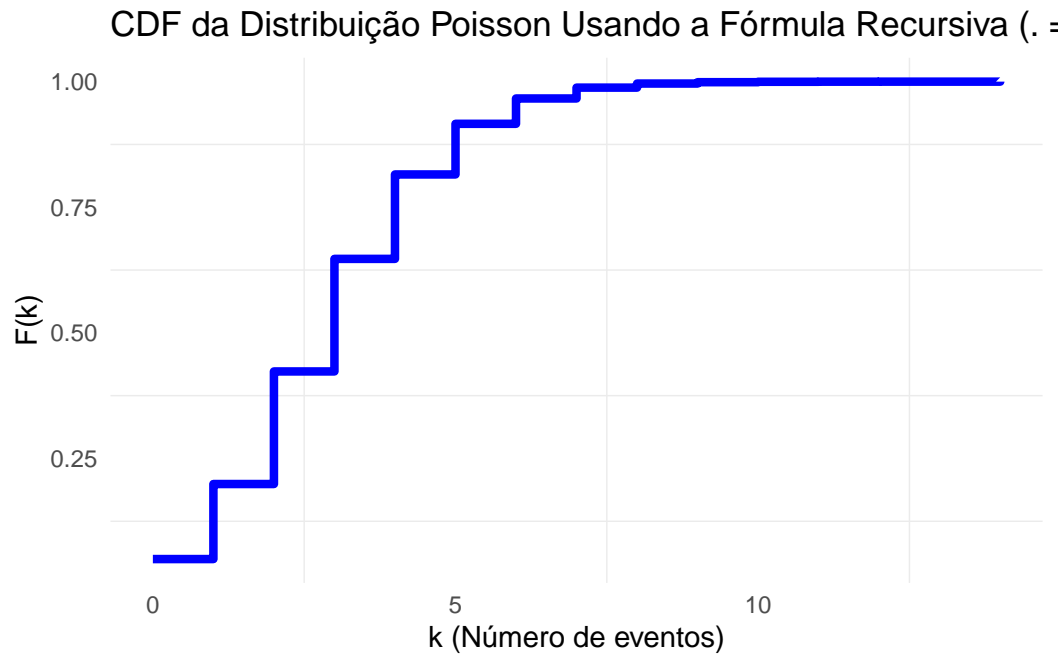


```
# Função para calcular a CDF da distribuição Poisson
cdf_poisson <- function(k, lam) {
  cdf <- 0
  p <- exp(-lam) # P(X=0)
  for (i in 0:k) {
    cdf <- cdf + p # Adiciona a probabilidade à CDF
    if (i < k) {
      p <- p * lam / (i + 1) # Atualiza a probabilidade recursivamente
    }
  }
  return(cdf)
}

# Gerando valores de k para a CDF
k_values <- 0:14
cdf_values <- sapply(k_values, cdf_poisson, lam = lam)

# Plotando a CDF da distribuição Poisson
df_cdf <- data.frame(k_values = k_values, cdf_values = cdf_values)
ggplot(df_cdf, aes(x = k_values, y = cdf_values)) +
```

```
geom_step(direction = "hv", color = "blue", size = 1.5) +
labs(title = "CDF da Distribuição Poisson Usando a Fórmula Recursiva (λ = 3)",
      x = "k (Número de eventos)",
      y = "F(k)") +
theme_minimal() +
theme(panel.grid.major = element_blank())
```



25 Python

```
import numpy as np
import matplotlib.pyplot as plt
import math

# Função para gerar a inversa da CDF para a distribuição Poisson usando a técnica de inversão
def inversa_cdf_poisson_recursiva(lam, u):
    k = 0
    p = math.exp(-lam) # P(X=0)
    F = p # Iniciamos com a probabilidade P(X=0)

    # Continuamos somando até que F >= u
    while u > F:
        k += 1
        p = p * lam / k # Atualiza a probabilidade recursivamente para o próximo valor
        F += p

    return k

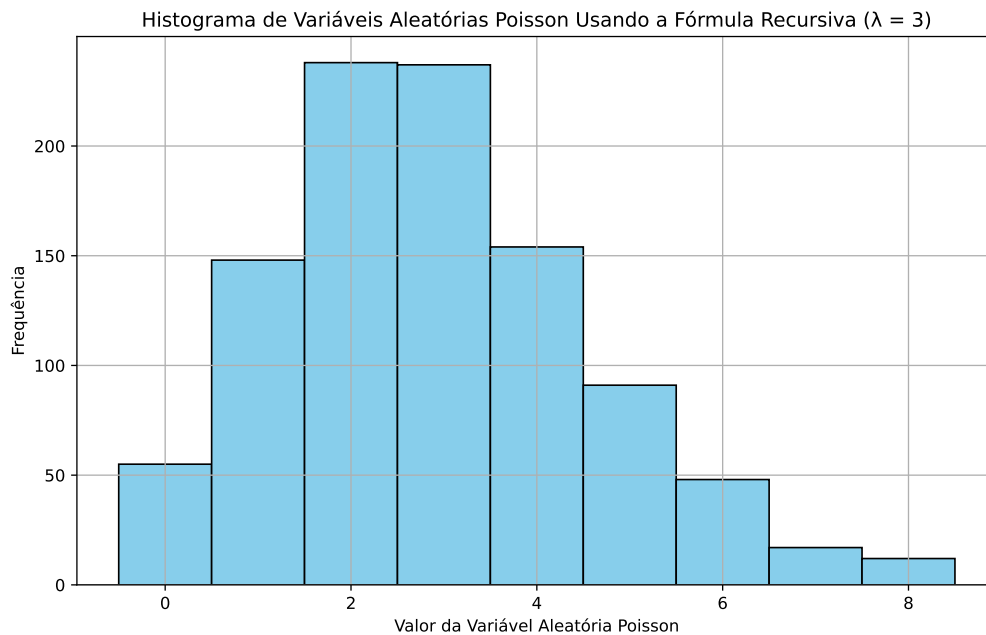
# Parâmetro lambda da distribuição Poisson
lam = 3

# Gerando 1000 números uniformemente distribuídos
uniformes = np.random.uniform(0, 1, 1000)

# Gerando a variável aleatória Poisson correspondente para cada número uniforme
poisson_vars = [inversa_cdf_poisson_recursiva(lam, u) for u in uniformes]

# Plotando o histograma das variáveis Poisson geradas
plt.figure(figsize=(10, 6))
plt.hist(poisson_vars, bins=range(0, max(poisson_vars) + 1), color='skyblue', edgecolor='black')
plt.title('Histograma de Variáveis Aleatórias Poisson Usando a Fórmula Recursiva (λ = 3)')
plt.xlabel('Valor da Variável Aleatória Poisson')
plt.ylabel('Frequência')
plt.grid(True)
```

```
plt.show()
```

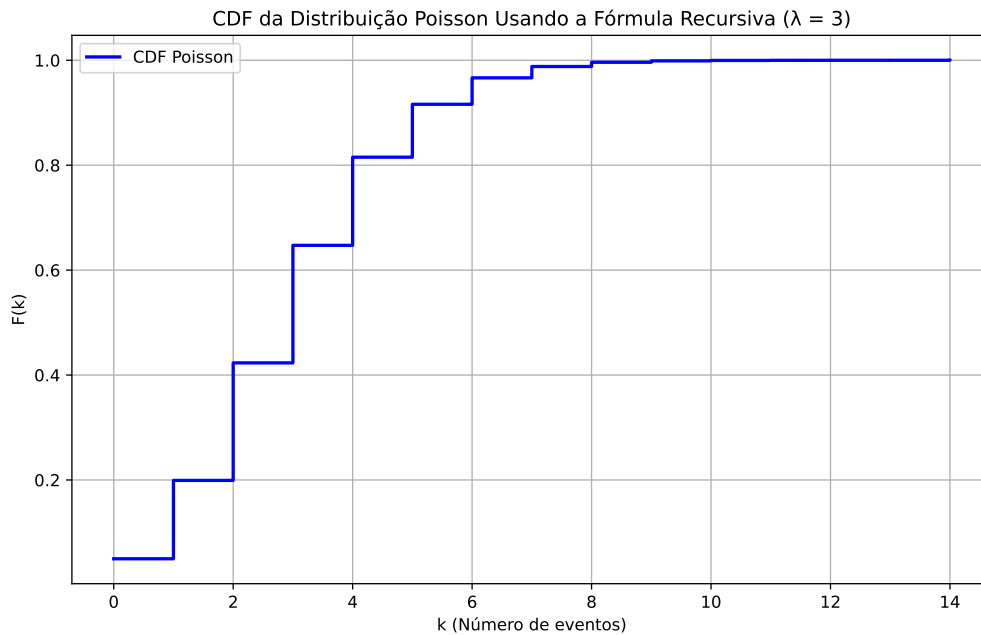


```
# Calculando a CDF da distribuição Poisson
def cdf_poisson(k, lam):
    cdf = 0
    p = math.exp(-lam) # P(X=0)
    for i in range(k+1):
        cdf += p # Adiciona a probabilidade à CDF
        if i < k: # Atualiza a probabilidade recursivamente
            p = p * lam / (i + 1)
    return cdf

# Gerando valores de k para a CDF
k_values = np.arange(0, 15)
cdf_values = [cdf_poisson(k, lam) for k in k_values]

# Plotando a CDF da distribuição Poisson
plt.figure(figsize=(10, 6))
plt.step(k_values, cdf_values, where='post', color='blue', label='CDF Poisson', linewidth=2)
plt.title('CDF da Distribuição Poisson Usando a Fórmula Recursiva (  $\lambda = 3$  )')
plt.xlabel('k (Número de eventos)')
```

```
plt.ylabel('F(k)')
plt.grid(True)
plt.legend()
plt.show()
```



25.1 Exercícios

Exercício 1. Seja X uma v.a. tal que $\mathbb{P}(X = 1) = 0.3$, $\mathbb{P}(X = 3) = 0.1$ e $\mathbb{P}(X = 4) = 0.6$.

- Escreva um pseudo-algoritmo para gerar um valor de X .
- Implemente uma função para gerar n valores de X .
- Compare a distribuição das frequências obtidas na amostra simulada com as probabilidades reais.

Exercício 2. Considere X uma v.a. tal que

$$\mathbb{P}(X = i) = \alpha \mathbb{P}(X_1 = i) + (1 - \alpha) \mathbb{P}(X_2 = i), \quad i = 0, 1, \dots$$

onde $0 \leq \alpha \leq 1$ e X_1, X_2 são v.a. discretas.

A distribuição de X é chamada de distribuição de mistura. Podemos escrever

$$X = \begin{cases} X_1, & \text{com probabilidade } \alpha \\ X_2, & \text{com probabilidade } 1 - \alpha \end{cases}$$

Pseudo-Algoritmo:

- (1) Seja $U \sim Unif(0, 1)$
- (2) Se $U \leq \alpha$, gere um valor da distribuição de X_1 . Senão, gere um valor da distribuição de X_2

Com base no pseudo-algoritmo implemente um algoritmo para gerar uma amostra de tamanho n da distribuição mistura de uma Poisson e de uma Geométrica com base nas funções implementadas nos Exercícios (2) e (3).

26 Técnica da Inversão para Variáveis Contínuas

Os valores que uma variável aleatória X pode assumir são chamados de **suporte** da distribuição de X .

Variáveis Aleatórias Contínuas são variáveis aleatórias que têm suporte em um conjunto não enumerável de valores, como intervalos na reta real, \mathbb{R} , ou $(0, \infty)$, por exemplo.

Uma variável aleatória X contínua tem função de distribuição acumulada (f.d.a.) puder ser expressa como

$$P(X \leq a) = F(a) = \int_{-\infty}^a f(x)dx, \quad \forall a \in \mathbb{R},$$

em que $f : \mathbb{R} \rightarrow [0, \infty)$ é uma função integrável, chamada de função densidade de probabilidade.

26.1 Função Inversa

Sabemos que $F : \mathbb{R} \rightarrow [0, 1]$ é estritamente crescente quando X é contínua, e, portanto, podemos definir sua função inversa $F^{-1} : [0, 1] \rightarrow \mathbb{R}$. A seguinte figura ilustra F e sua inversa.

27 R

```
# Carregando as bibliotecas necessárias
library(ggplot2)

# Definindo a função de distribuição acumulada F(x) - função logística
F <- function(x) {
  return(1 / (1 + exp(-x)))
}

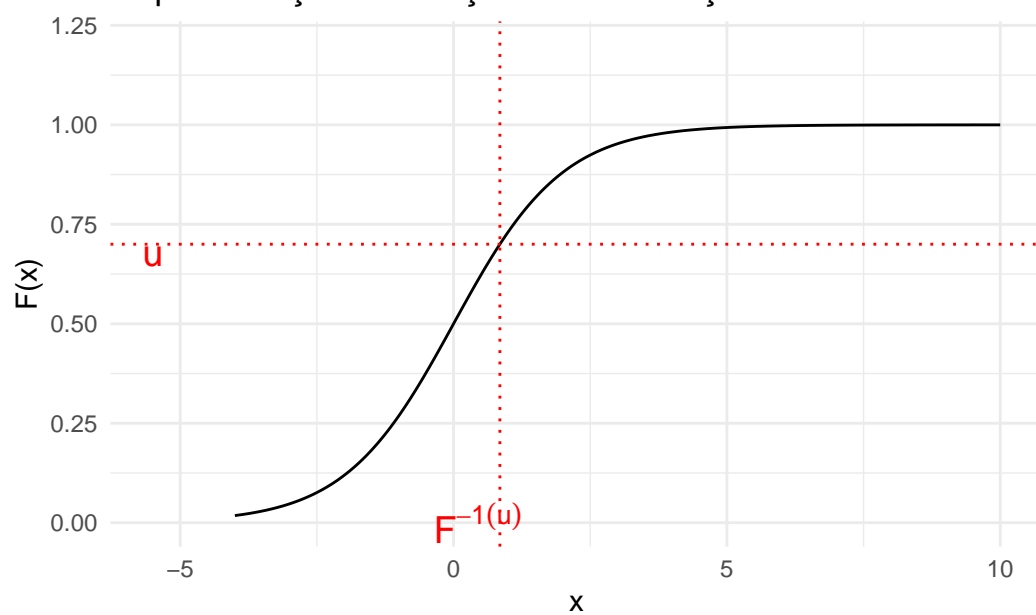
# Definindo a inversa da função de distribuição acumulada F_inv(u)
F_inv <- function(u) {
  return(-log((1 / u) - 1))
}

# Gerando valores de x e u
x <- seq(-4, 10, length.out = 400)
u <- seq(0.01, 0.99, length.out = 400)

# Definindo o valor de U para plotar as linhas
u_value <- 0.7
x_value <- F_inv(u_value)

# Criando o gráfico
ggplot(data = data.frame(x = x, F_x = F(x))) +
  geom_line(aes(x = x, y = F_x), color = "black") +
  geom_hline(yintercept = u_value, linetype = "dotted", color = "red") +
  geom_vline(xintercept = x_value, linetype = "dotted", color = "red") +
  annotate("text", x = x_value - 0.4, y = -0, label = expression(F^{-1}(u)), color = "red", size = 5) +
  annotate("text", x = -5.5, y = u_value - 0.02, label = "u", color = "red", size = 5) +
  labs(title = "Representação da Função de Distribuição Acumulada e sua Inversa",
       x = "x", y = "F(x)") +
  ylim(0, 1.2) +
  theme_minimal()
```

Representação da Função de Distribuição Acumulada e sua Inversa



28 Python

```
import numpy as np
import matplotlib.pyplot as plt

# Definindo a função de distribuição acumulada F(x)
def F(x):
    return 1 / (1 + np.exp(-x)) # Função logística como exemplo de F(x)

# Definindo a inversa da função de distribuição acumulada F_inv(u)
def F_inv(u):
    return -np.log((1 / u) - 1)

# Gerando valores de x e u para plotar
x = np.linspace(-4, 10, 400)
u = np.linspace(0.01, 0.99, 400) # U entre 0 e 1 (evitando extremos para evitar erros na in

# Plotando a função de distribuição acumulada F(x) com truncamento do eixo y no zero
plt.figure(figsize=(8, 6))
plt.plot(x, F(x), color="black")

# Adicionando linhas pontilhadas para representar U e F_inv(U)
u_value = 0.7 # Exemplo de valor de U
x_value = F_inv(u_value)

plt.hlines(u_value, min(x), x_value, linestyle='dotted', colors='red')
plt.vlines(x_value, 0, u_value, linestyle='dotted', colors='red')

# Etiquetas
plt.text(x_value-0.4, -0.05, r"$F^{-1}(u)$", fontsize=12, color='red')
plt.text(-5.5, u_value - 0.02, r"$u$", fontsize=14, color='red')

# Rótulos e estilo do gráfico
plt.title(r'Representação da Função de Distribuição Acumulada e sua Inversa', fontsize=14)
plt.xlabel(r'$x$', fontsize=12)
plt.ylabel(r'$F(x)$', fontsize=12)
```

```
plt.ylim(0, 1.2)
```

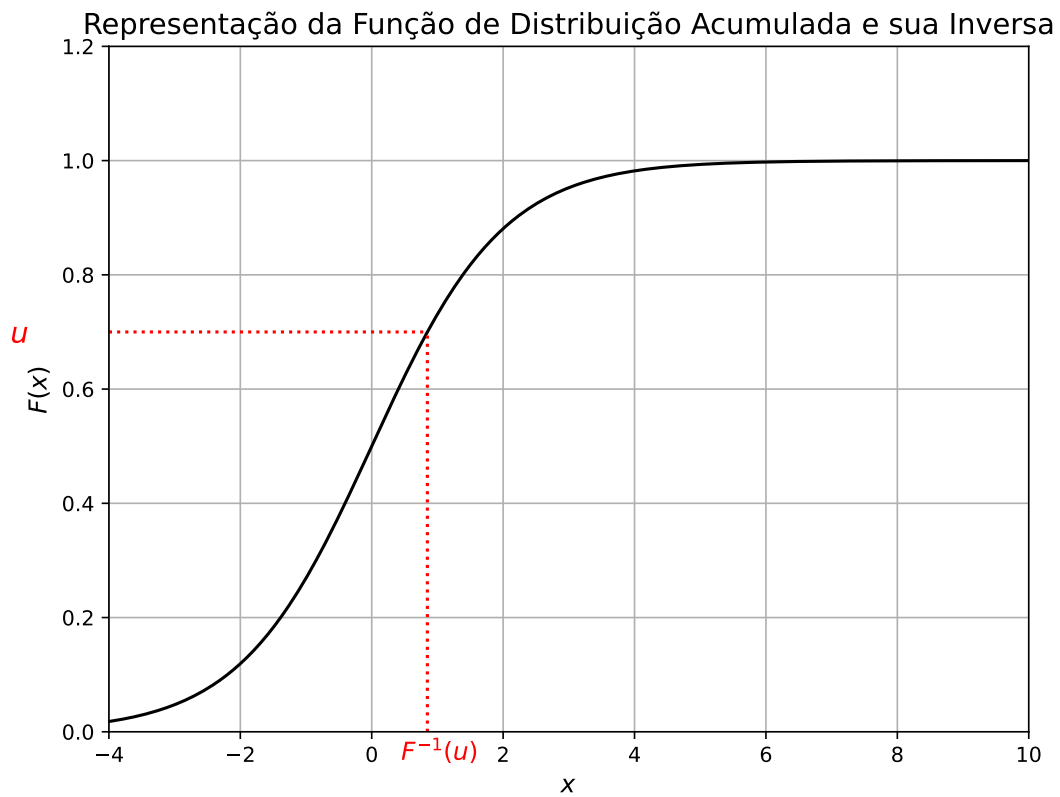
```
(0.0, 1.2)
```

```
plt.xlim(-4, 10)
```

```
(-4.0, 10.0)
```

```
plt.grid(True)
```

```
# Exibir o gráfico  
plt.show()
```



28.1 Método da Inversão

Uma maneira de gerar valores de uma variável aleatória contínua X , é o **método da inversão**, que é originado da seguinte proposição:

Proposição: Seja $U \sim \text{Unif}(0, 1)$. Para qualquer variável aleatória contínua com função de distribuição acumulada F , a variável:

$$X = F^{-1}(U)$$

tem distribuição F .

Prova:

$$\mathbb{P}(X \leq x) = \mathbb{P}(F^{-1}(U) \leq x) = \mathbb{P}(F(F^{-1}(U)) \leq F(x)) = \mathbb{P}(U \leq F(x)) = F(x).$$

Assim, o método da inversão consiste em:

1. Gerar $U \sim \text{Unif}(0, 1)$.
2. Calcular $X = F^{-1}(U)$.

28.2 Exemplo 1

Seja X uma v.a. com:

$$F(x) = x^n, \quad \text{para } 0 < x < 1.$$

A função inversa é:

$$u = F(x) = x^n \implies x = u^{1/n}.$$

Portanto, o pseudo-algoritmo para gerar X a partir do método da inversão é:

1. Gere $U \sim \text{Unif}(0, 1)$.
2. Calcule $X = U^{1/n}$.

29 R

```
# Carregar a biblioteca ggplot2
library(ggplot2)

# Definir o parâmetro n da distribuição  $F(x) = x^n$ 
n <- 3

# Gerar 1000 valores U de uma distribuição uniforme (0,1)
U <- runif(1000, min = 0, max = 1)

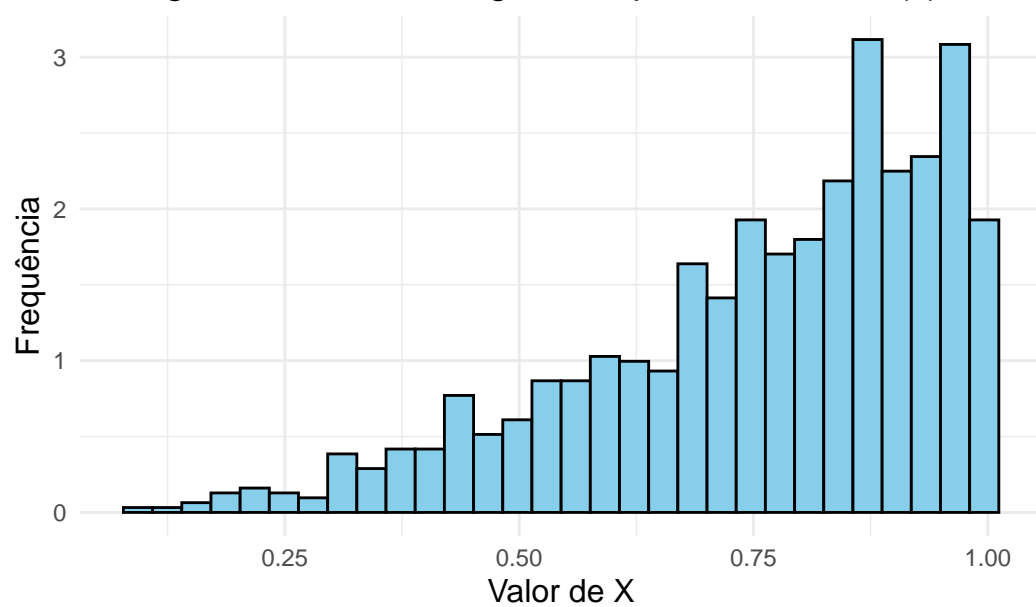
# Calcular  $X = U^{(1/n)}$ 
X <- U^(1/n)

# Criar um dataframe para o ggplot2
data <- data.frame(X = X)

# Plotar o histograma usando ggplot2
p <- ggplot(data, aes(x = X)) +
  geom_histogram(aes(y = ..density..), bins = 30, fill = 'skyblue', color = 'black') +
  labs(title = 'Histograma de variáveis geradas pela inversão:  $F(x) = x^n$ ,  $n = 3$ ',
       x = 'Valor de X', y = 'Frequência') +
  theme_minimal() +
  theme(plot.title = element_text(size = 14),
        axis.title.x = element_text(size = 12),
        axis.title.y = element_text(size = 12))

# Exibir o gráfico
print(p)
```

Histograma de variáveis geradas pela inversão: $F(x) = x^n$,



30 Python

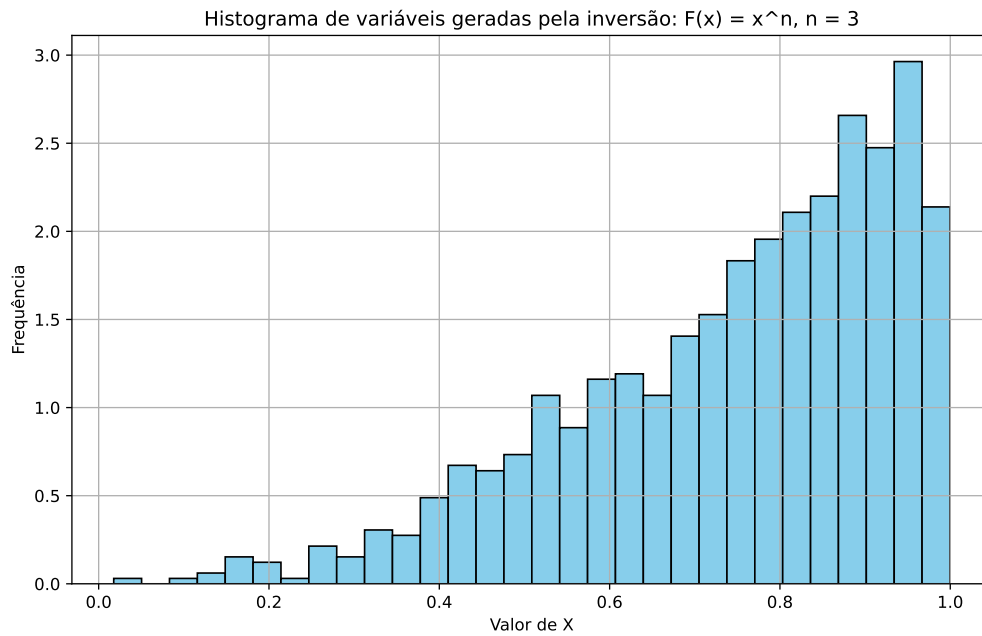
```
import numpy as np
import matplotlib.pyplot as plt

# Parâmetro n da distribuição  $F(x) = x^n$ 
n = 3

# Gerando 1000 valores U de uma distribuição uniforme (0,1)
U = np.random.uniform(0, 1, 1000)

# Calculando  $X = U^{1/n}$ 
X = U**(1/n)

# Plotando um histograma dos valores gerados
plt.figure(figsize=(10, 6))
plt.hist(X, bins=30, color='skyblue', edgecolor='black', density=True)
plt.title('Histograma de variáveis geradas pela inversão:  $F(x) = x^n$ ,  $n = 3$ ')
plt.xlabel('Valor de X')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()
```

30.0.1 Exemplo 2

Seja $X \sim \text{Exp}(\lambda)$, com:

$$F(x) = 1 - e^{-\lambda x}, \quad \text{para } x > 0.$$

A função inversa é:

$$u = F(x) = 1 - e^{-\lambda x} \implies x = -\frac{\log(1-u)}{\lambda}.$$

Um pseudo-algoritmo para gerar X é, portanto,:

1. Gere $U \sim \text{Unif}(0, 1)$.
2. Calcule $X = -\frac{\log(1-U)}{\lambda}$.

31 R

```
# Carregar a biblioteca ggplot2
library(ggplot2)

# Definir o parâmetro lambda da distribuição exponencial
lambda <- 2

# Gerar 1000 valores U de uma distribuição uniforme (0,1)
U <- runif(1000, min = 0, max = 1)

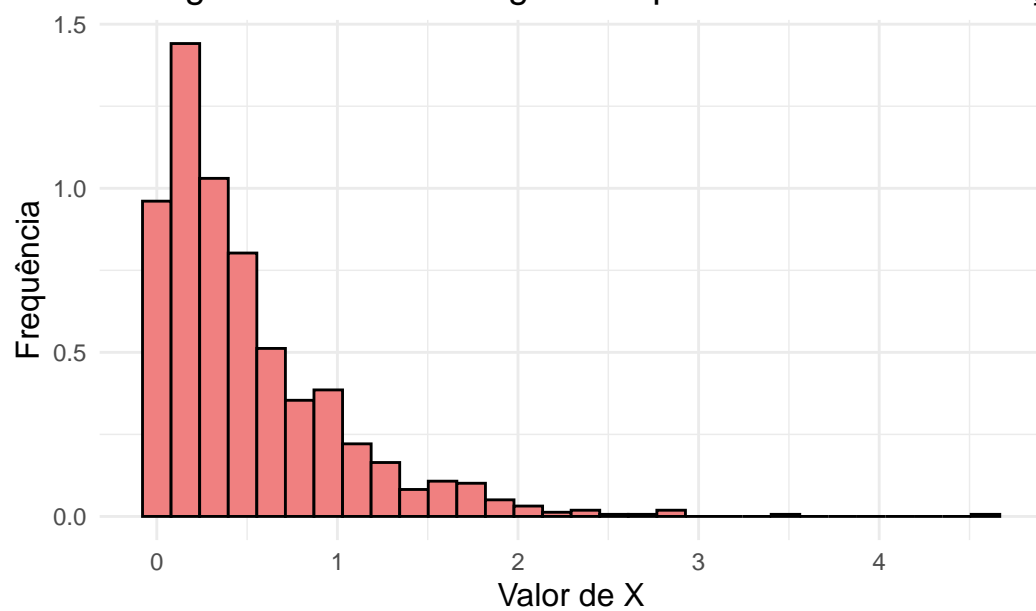
# Calcular X usando a inversa da CDF da distribuição exponencial
X <- -log(1 - U) / lambda

# Criar um dataframe para o ggplot2
data <- data.frame(X = X)

# Plotar o histograma usando ggplot2
p <- ggplot(data, aes(x = X)) +
  geom_histogram(aes(y = ..density..), bins = 30, fill = 'lightcoral', color = 'black') +
  labs(title = 'Histograma de variáveis geradas pela inversão: Distribuição Exponencial',
       x = 'Valor de X', y = 'Frequência') +
  theme_minimal() +
  theme(plot.title = element_text(size = 14),
        axis.title.x = element_text(size = 12),
        axis.title.y = element_text(size = 12))

# Exibir o gráfico
print(p)
```

Histograma de variáveis geradas pela inversão: Distribuição



32 Python

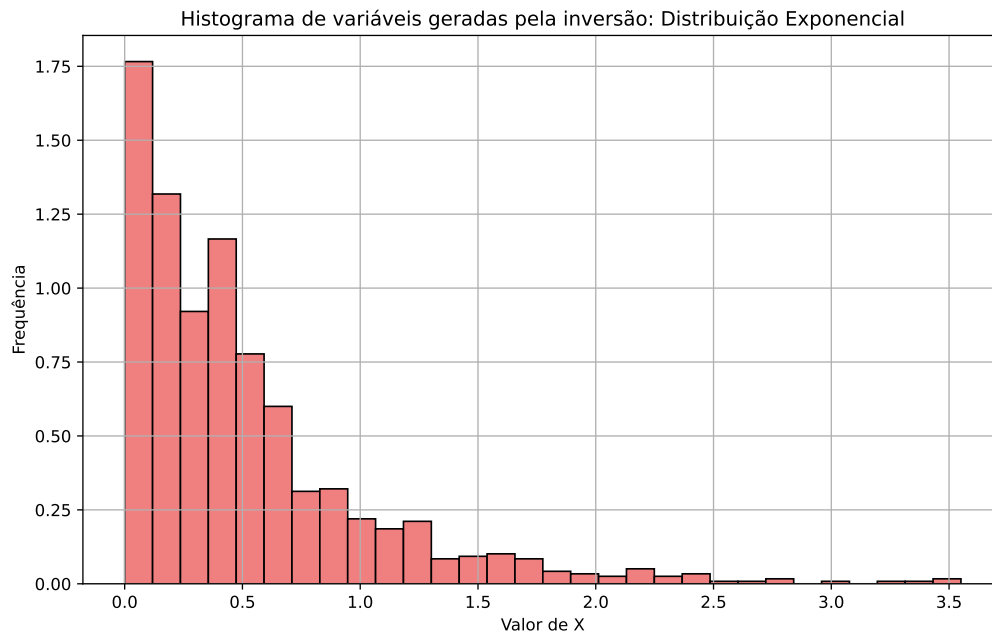
```
import numpy as np
import matplotlib.pyplot as plt

# Parâmetro lambda da distribuição exponencial
lamdb = 2

# Gerando 1000 valores U de uma distribuição uniforme (0,1)
U = np.random.uniform(0, 1, 1000)

# Calculando X usando a inversa da CDF da exponencial
X = -np.log(1 - U) / lamdb

# Plotando um histograma dos valores gerados
plt.figure(figsize=(10, 6))
plt.hist(X, bins=30, color='lightcoral', edgecolor='black', density=True)
plt.title('Histograma de variáveis geradas pela inversão: Distribuição Exponencial')
plt.xlabel('Valor de X')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()
```



32.1 Simulação de transformações de variáveis aleatórias

Agora que já sabemos uma maneira de simular uma variável aleatória X , descreveremos como gerar valores de uma transformação dessa variável, ou seja, $g(X)$. Para isso, basta aplicar a função de transformação g diretamente aos valores simulados de X . Veremos a seguinte alguns exemplos disso em funcionamento.

32.1.1 Exemplo 1: Simulando $Y \sim Unif(1, 2)$

Para gerar valores de $Y \sim Unif(1, 2)$, usamos o fato de que Y é uma simples transformação de $U \sim Unif(0, 1)$. A relação é:

$$Y = U + 1.$$

Assim, podemos usar o seguinte pseudo-algoritmo para gerar Y a partir de U :

1. Gere $U \sim Unif(0, 1)$.
2. Calcule $Y = U + 1$.

33 R

```
# Carregar biblioteca ggplot2
library(ggplot2)

# Gerar 1000 valores U de uma distribuição uniforme (0,1)
U <- runif(1000, min = 0, max = 1)

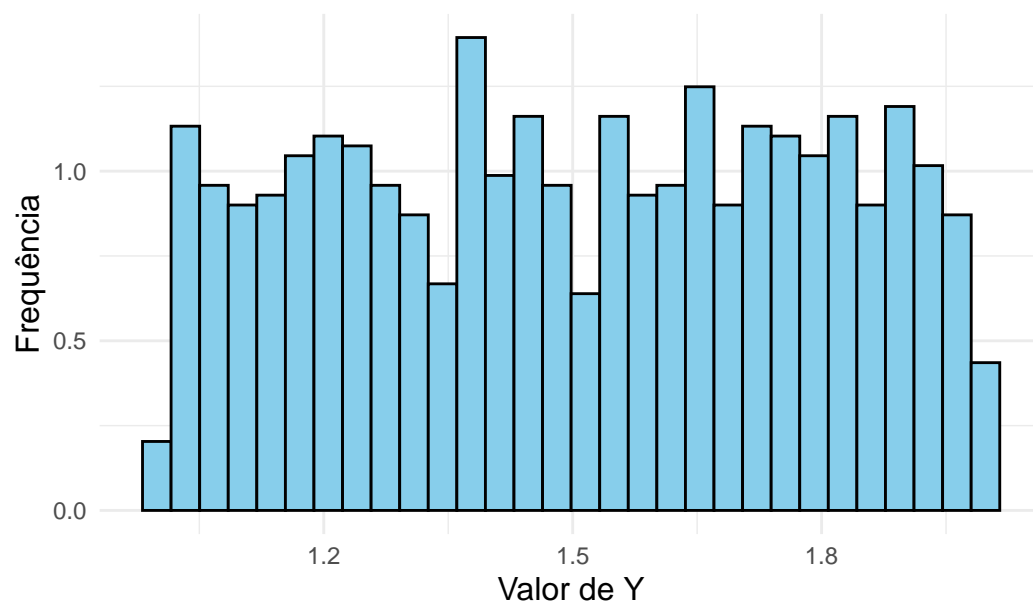
# Calcular Y = U + 1 para ter Y ~ Unif(1, 2)
Y <- U + 1

# Criar um dataframe para o ggplot2
data <- data.frame(Y = Y)

# Plotar o histograma usando ggplot2
p <- ggplot(data, aes(x = Y)) +
  geom_histogram(aes(y = ..density..), bins = 30, fill = 'skyblue', color = 'black') +
  labs(title = 'Histograma de variáveis geradas: Y ~ Unif(1, 2)',
       x = 'Valor de Y', y = 'Frequência') +
  theme_minimal() +
  theme(plot.title = element_text(size = 14),
        axis.title.x = element_text(size = 12),
        axis.title.y = element_text(size = 12))

# Exibir o gráfico
print(p)
```

Histograma de variáveis geradas: $Y \sim \text{Unif}(1, 2)$



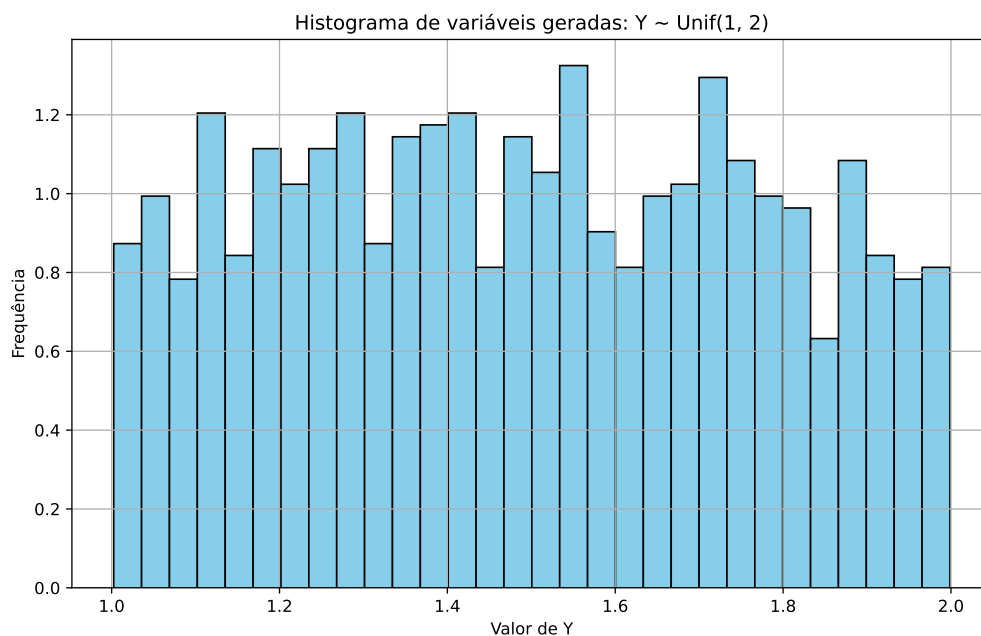
34 Python

```
import numpy as np
import matplotlib.pyplot as plt

# Gerando 1000 valores U de uma distribuição uniforme (0,1)
U = np.random.uniform(0, 1, 1000)

# Calculando Y = U + 1 para ter Y ~ Unif(1, 2)
Y = U + 1

# Plotando um histograma dos valores gerados
plt.figure(figsize=(10, 6))
plt.hist(Y, bins=30, color='skyblue', edgecolor='black', density=True)
plt.title('Histograma de variáveis geradas: Y ~ Unif(1, 2)')
plt.xlabel('Valor de Y')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()
```

34.0.1 Exemplo 2: Simulando $Y \sim \text{Gamma}(n, \lambda)$

Para gerar valores de $Y \sim \Gamma(n, \lambda)$, usamos o fato de que a soma de que Y pode ser representado como

$$Y = \sum_{i=1}^n X_i,$$

em que cada $X_i \sim \text{Exp}(\lambda)$, e X_i 's são independentes.

Assim, podemos gerar Y da seguinte forma:

1. Gere $U_1, \dots, U_n \sim \text{Unif}(0, 1)$ independentemente.
2. Calcule $X_i = -\frac{\log(1-U_i)}{\lambda}$ para $i = 1, \dots, n$.
3. Calcule $Y = X_1 + X_2 + \dots + X_n$.

35 R

```
# Carregar biblioteca ggplot2
library(ggplot2)

# Definir parâmetros
n <- 5 # número de somas
lambda <- 2 # parâmetro da distribuição exponencial

# Gerar 1000 valores U para cada uma das n somas
U <- matrix(runif(1000 * n, min = 0, max = 1), ncol = n)

# Calcular  $X_i = -\log(1 - U_i) / \lambda$  para cada  $U_i$ 
X <- -log(1 - U) / lambda

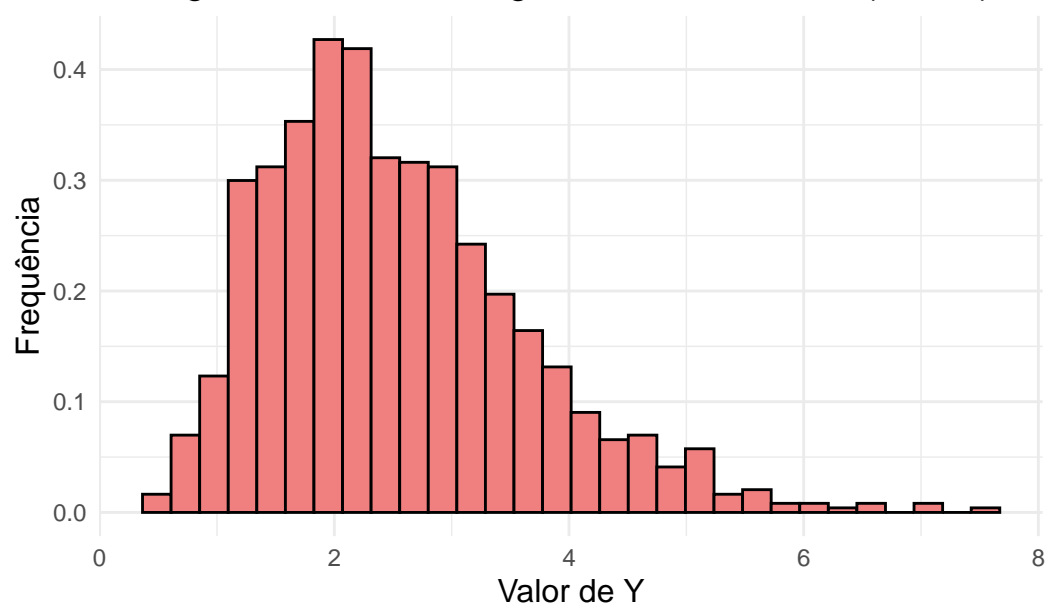
# Somar os valores de X para obter  $Y \sim \text{Gamma}(n, \lambda)$ 
Y <- rowSums(X)

# Criar um dataframe para o ggplot2
data <- data.frame(Y = Y)

# Plotar o histograma usando ggplot2
p <- ggplot(data, aes(x = Y)) +
  geom_histogram(aes(y = ..density..), bins = 30, fill = 'lightcoral', color = 'black') +
  labs(title = paste('Histograma de variáveis geradas: Y ~ Gamma(', n, ', ', lambda, ')'),
       x = 'Valor de Y', y = 'Frequência') +
  theme_minimal() +
  theme(plot.title = element_text(size = 14),
        axis.title.x = element_text(size = 12),
        axis.title.y = element_text(size = 12))

# Exibir o gráfico
print(p)
```

Histograma de variáveis geradas: $Y \sim \text{Gamma}(5, 2)$



36 Python

```
import numpy as np
import matplotlib.pyplot as plt

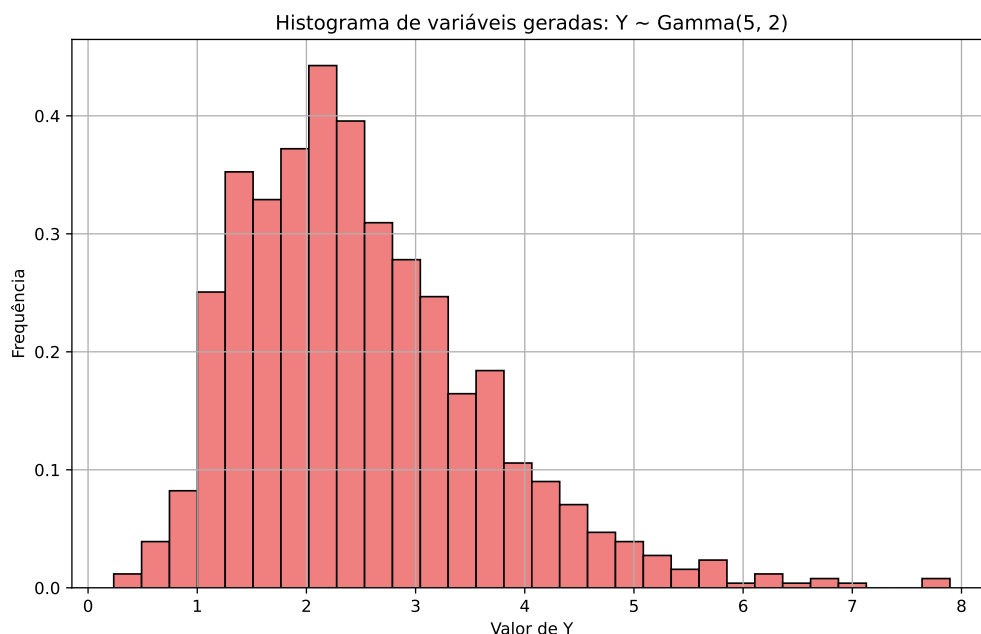
# Definindo parâmetros
n = 5 # número de somas
lamdb = 2 # parâmetro da distribuição exponencial

# Gerando 1000 valores U para cada uma das n somas
U = np.random.uniform(0, 1, (1000, n))

# Calculando  $X_i = -\log(1 - U_i) / \lambda$  para cada  $U_i$ 
X = -np.log(1 - U) / lamdb

# Somando os valores de X para obter  $Y \sim \text{Gamma}(n, \lambda)$ 
Y = np.sum(X, axis=1)

# Plotando um histograma dos valores gerados
plt.figure(figsize=(10, 6))
plt.hist(Y, bins=30, color='lightcoral', edgecolor='black', density=True)
plt.title(f'Histograma de variáveis geradas:  $Y \sim \text{Gamma}(\{n\}, \{\lambda\})$ ')
plt.xlabel('Valor de Y')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()
```



36.1 Exercícios

Exercício 1.

Utilizando o método da inversão, simule $U \sim U(1, 3)$.

Exercício 2.

- Implemente uma função para gerar uma amostra de tamanho n da distribuição Exponencial de parâmetro λ .
- Compare a distribuição empírica dos valores simulados com a densidade da Exponencial $f(x) = \lambda e^{-\lambda x}, x > 0$.

Exercício 3.

- Implemente uma função para gerar uma amostra de tamanho n da distribuição $Gama(a, b)$, para a sendo um valor inteiro.
- Compare a distribuição empírica dos valores simulados com a densidade da Gama $f(x) = \frac{b^a}{\Gamma(a)} x^{a-1} e^{-bx}, x > 0$.

Exercício 4. Seja X uma v.a. com função densidade dada por:

$$f(x) = \frac{1}{8}x, \quad 0 < x < 4.$$

- a) Escreva um pseudo-algoritmo para simular um único valor da variável X pelo método da inversão.
- b) Compare a distribuição empírica dos valores simulados com a densidade de X .

37 Método da Rejeição para Variáveis Discretas

Fill in

38 Método da Rejeição para Variáveis Contínuas

A amostragem por rejeição é uma técnica útil para gerar variáveis aleatórias a partir de uma distribuição complexa, utilizando uma distribuição proposta mais simples. A ideia básica é simular de uma distribuição fácil de amostrar e, em seguida, rejeitar ou aceitar essas amostras com base na distribuição alvo.

38.1 Algoritmo

O método assume que conhecemos uma distribuição $g(x)$, fácil de simular, que cubra o suporte da distribuição alvo $f(x)$. Essa distribuição é chamada de **distribuição proposta**. Além disso ele assume que existe c tal que $\frac{f(x)}{g(x)} \leq c$ para todo x , e que conseguimos calcular c . O método de aceitação e rejeição segue os seguintes passos:

1. Gere um valor Y da distribuição proposta $g(x)$.
2. Gere um valor $U \sim \text{Uniform}(0, 1)$.
3. Aceite Y como uma amostra de $f(x)$ se $U \leq \frac{f(Y)}{c \cdot g(Y)}$, caso contrário, rejeite Y e repita o processo.

A figura a seguir ilustra esse processo quando g é uma distribuição uniforme entre 0 e 1.

39 R

```
library(ggplot2)

# Função densidade alvo f(x)
f <- function(x) {
  20 * x * (1 - x)^3
}

# Função densidade proposta g(x) (distribuição uniforme)
g <- function(x) {
  rep(1, length(x)) # g(x) é uniforme no intervalo (0, 1)
}

# Constante c (máximo de f(x))
opt_result <- optimize(f = function(x) -f(x), interval = c(0, 1))
c <- f(opt_result$minimum)

# Gerar valores de x
x <- seq(0, 1, length.out = 1000)

# Valor de Y para aceitar/rejeitar (escolhido aleatoriamente)
Y <- 0.6

# DataFrame para o gráfico
df <- data.frame(x = x, fx = f(x), gx = c * g(x))

# Plotando as funções f(x) e a linha horizontal c*g(x)
ggplot(df, aes(x = x)) +
  geom_line(aes(y = fx, color = "f(x)"), size = 1) +
  geom_hline(aes(yintercept = c * g(x), color = "c * g(x)"), size = 1.5) +

  # Destacar a área de aceitação e rejeição
  geom_text(aes(x = Y + 0.025, y = 0.05, label = "Y"), color = "black", size = 5) +

  geom_segment(aes(x = Y, xend = Y, y = 0, yend = f(Y)), color = "green", size = 1.5) +
```

```

geom_segment(aes(x = Y, xend = Y, y = f(Y), yend = c * g(Y)), color = "red", size = 1.5) +

# Mover rótulos de Aceitar/Rejeitar para a esquerda da linha Y
geom_text(aes(x = Y + 0.06, y = f(Y) / 2, label = "Aceitar x"), color = "green", size = 4)
geom_text(aes(x = Y + 0.06, y = f(Y) + (c * g(Y) - f(Y)) / 2, label = "Rejeitar x"), color = "red", size = 4)

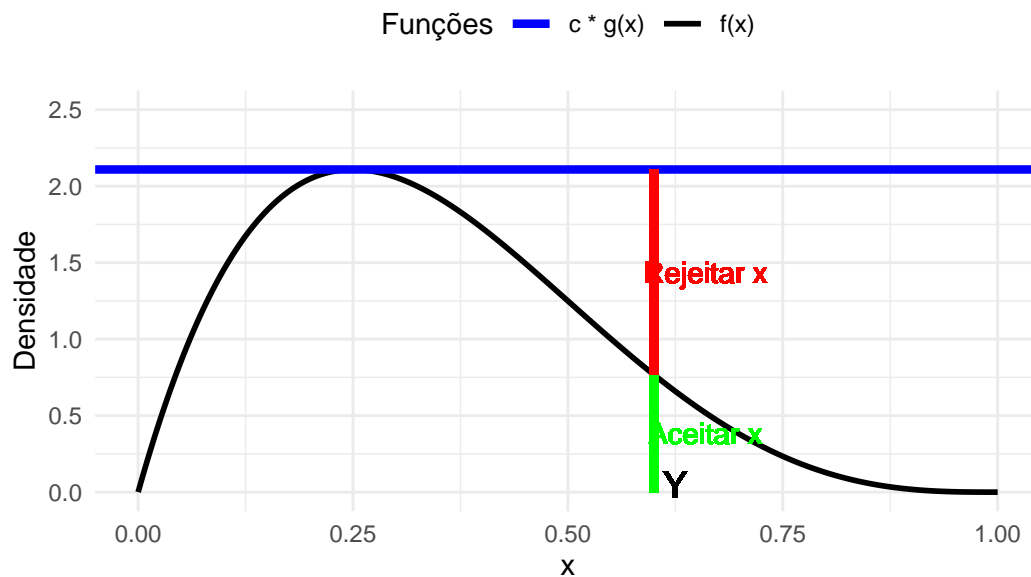
# Limites e rótulos
xlim(0, 1) +
ylim(0, 2.5) +
labs(x = "x", y = "Densidade", title = "Amostragem por Rejeição para g uniforme") +

# Legenda
scale_color_manual(values = c("f(x)" = "black", "c * g(x)" = "blue"), name = "Funções") +

theme_minimal() +
theme(legend.position = "top")

```

Amostragem por Rejeição para g uniforme



40 Python

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import minimize_scalar

# Função densidade alvo f(x)
def f(x):
    return 20 * x * (1 - x)**3

# Função densidade proposta g(x) (distribuição uniforme)
def g(x):
    return 1 # g(x) é uniforme no intervalo (0, 1)

# Constante c
result = minimize_scalar(lambda x: -f(x), bounds=(0, 1), method='bounded')
c = f(result.x)

# Gerar valores de x
x = np.linspace(0, 1, 1000)

# Valor de Y para aceitar/rejeitar (escolhido aleatoriamente)
Y = 0.6
U = 0.8
accept_threshold = f(Y) / (c * g(Y))

# Plotando as funções f(x) e a linha horizontal c*g(x)
# Ajustando o código para truncar o eixo y no zero
plt.plot(x, f(x), label='f(x)', color='black')
plt.hlines(c * g(x), 0, 1, color='blue', label='c*g(x)', linewidth=2) # Linha horizontal c*g(x)

# Destacar a área de aceitação e rejeição
plt.vlines(Y, 0, f(Y), colors='green', label='Aceitar $x$', linewidth=2)
plt.vlines(Y, f(Y), c * g(Y), colors='red', label='Rejeitar $x$', linewidth=2)

# Adicionar rótulo de Y na posição da linha vertical
```

```
plt.text(Y+0.025, 0.05, 'Y', color='black', fontsize=12, horizontalalignment='center')

# Adicionar rótulos de Aceitar/Rejeitar
plt.text(Y + 0.01, f(Y) / 2, 'Aceitar x', color='green', fontsize=10, verticalalignment='center')
plt.text(Y + 0.01, f(Y) + (c * g(Y) - f(Y)) / 2, 'Rejeitar x', color='red', fontsize=10, verticalalignment='center')

# Limites e rótulos
plt.xlim(0, 1)
```

(0.0, 1.0)

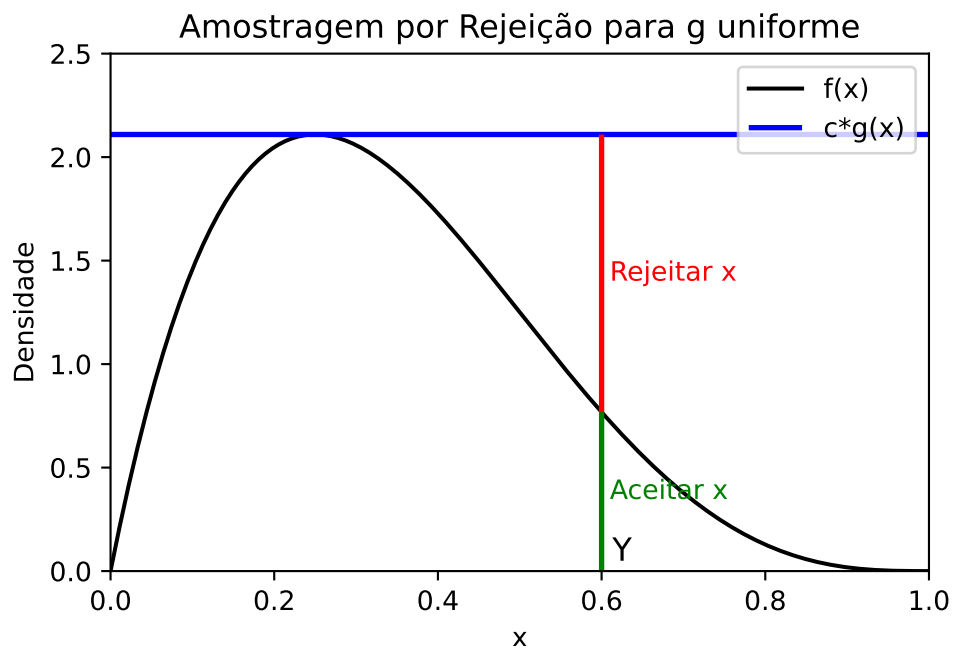
```
plt.ylim(0, 2.5) # Truncando o eixo y no zero
```

(0.0, 2.5)

```
plt.xlabel('x')
plt.ylabel('Densidade')

# Adicionar a legenda para f(x) e c*g(x)
plt.legend(['f(x)', 'c*g(x)'])

# Mostrar gráfico
plt.title('Amostragem por Rejeição para g uniforme')
plt.show()
```



40.1 Exemplo

Vamos demonstrar a amostragem por rejeição gerando amostras de uma distribuição alvo $f(x)$, utilizando a distribuição uniforme como distribuição proposta.

A função densidade alvo que utilizaremos é:

$$f(x) = 20x(1-x)^3 \quad \text{para } 0 < x < 1$$

E a distribuição proposta será $g(x) = 1$ para $0 < x < 1$, que é uma distribuição uniforme.

Implementamos o seguinte algoritmo:

41 R

```
library(ggplot2)

# Função densidade alvo f(x)
f <- function(x) {
  20 * x * (1 - x)^3
}

# Amostragem por rejeição
amostragem_rejeicao <- function(f, c, n_amstras) {
  amostras <- numeric(0)
  while (length(amostras) < n_amstras) {
    # Gere Y ~ g(x) (uniforme entre 0 e 1)
    Y <- runif(1, 0, 1)

    # Gere U ~ Uniforme(0, 1)
    U <- runif(1, 0, 1)

    # Verifica se aceitamos Y
    if (U <= f(Y) / (c)) {
      amostras <- c(amostras, Y)
    }
  }
  return(amostras)
}

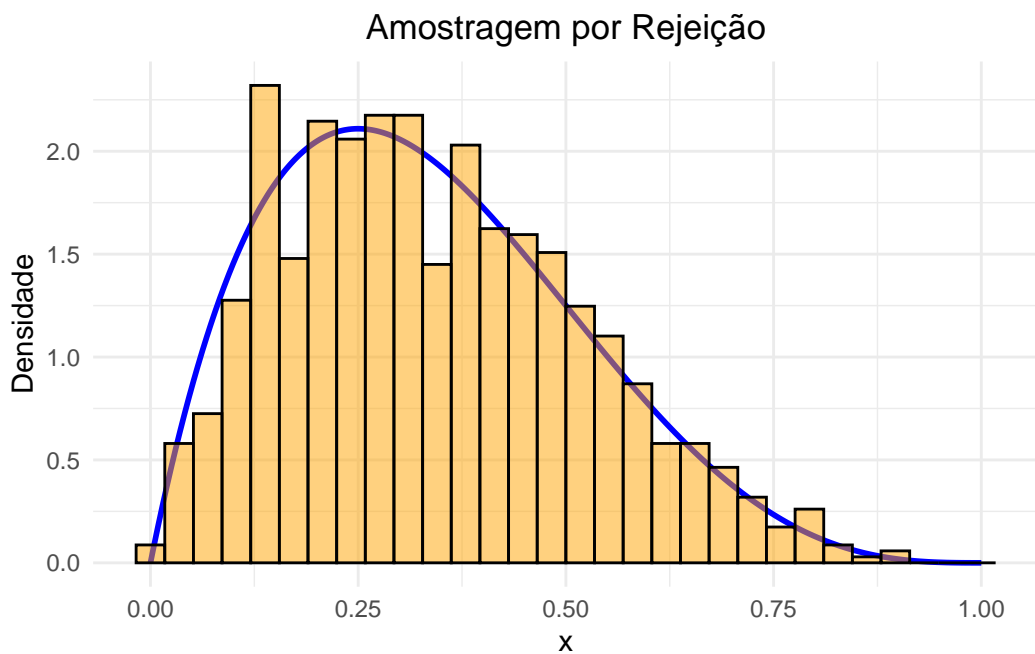
# A constante c é o limite superior de f(x)
c <- 135 / 64

# Gere 1000 amostras usando o método de aceitação e rejeição
set.seed(123)
amostras <- amostragem_rejeicao(f, c, 1000)

# Criação do data frame para o ggplot2
x_vals <- seq(0, 1, length.out = 1000)
```

```
f_vals <- f(x_vals)
df <- data.frame(x = x_vals, y = f_vals)

# Gráfico com ggplot2
ggplot() +
  geom_line(data = df, aes(x = x, y = y), color = 'blue', size = 1, linetype = 'solid') +
  geom_histogram(aes(x = amostras, y = ..density..), bins = 30, fill = 'orange', alpha = 0.5) +
  ggtitle("Amostragem por Rejeição") +
  xlab("x") +
  ylab("Densidade") +
  theme_minimal() +
  theme(plot.title = element_text(hjust = 0.5))
```



42 Python

```
import numpy as np
import matplotlib.pyplot as plt

# Função densidade alvo f(x)
def f(x):
    return 20 * x * (1 - x)**3

# Função densidade proposta g(x) (distribuição uniforme)
def g(x):
    return 1 # g(x) é uniforme no intervalo (0, 1)

# Amostragem por rejeição
def amostragem_rejeicao(f, g, c, n_amstras):
    amostras = []
    while len(amostras) < n_amstras:
        # Gere Y ~ g(x)
        Y = np.random.uniform(0, 1)

        # Gere U ~ Uniforme(0, 1)
        U = np.random.uniform(0, 1)

        # Verifica se aceitamos Y
        if U <= f(Y) / (c * g(Y)):
            amostras.append(Y)

    return np.array(amostras)

# A constante c é o limite superior de f(x)/g(x)
c = 135 / 64 # Pré-calculado

# Gere 1000 amostras usando o método de aceitação e rejeição
amostras = amostragem_rejeicao(f, g, c, 1000)

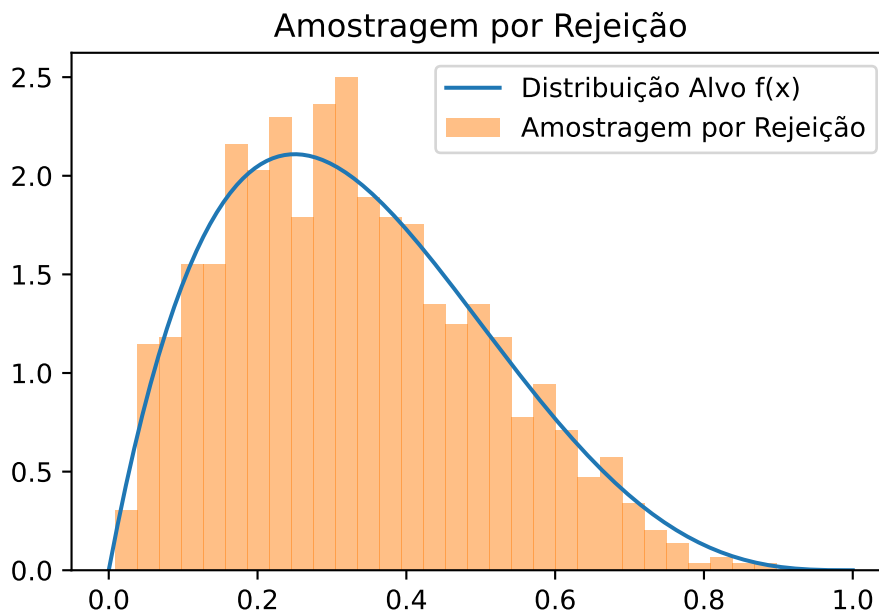
# Plotando os resultados
```



```

x = np.linspace(0, 1, 1000)
plt.plot(x, f(x), label='Distribuição Alvo f(x)')
plt.hist(amostras, bins=30, density=True, alpha=0.5, label='Amostragem por Rejeição')
plt.legend()
plt.title("Amostragem por Rejeição")
plt.show()

```



42.1 Resultados Teóricos

42.1.1 Correção do Método de Aceitação e Rejeição

Teorema:

A variável aleatória Y gerada pelo método da rejeição tem densidade $f(x)$.

Prova:

Considere a função de distribuição acumulada (CDF) de X , a variável gerada pelo método:

$$P(X \leq x) = P(Y \leq x \mid \text{aceitar } Y)$$

Podemos escrever essa probabilidade como:

$$P(X \leq x) = \frac{P(Y \leq x, U \leq \frac{f(Y)}{c \cdot g(Y)})}{P(U \leq \frac{f(Y)}{c \cdot g(Y)})}$$

Usando a probabilidade condicional, o numerador pode ser expresso como:

$$P(Y \leq x, U \leq \frac{f(Y)}{c \cdot g(Y)}) = \int_{-\infty}^x \int_0^{\frac{f(y)}{c \cdot g(y)}} g(y) du dy$$

Resolvendo a integral em u :

$$P(Y \leq x, U \leq \frac{f(Y)}{c \cdot g(Y)}) = \int_{-\infty}^x \frac{f(y)}{c \cdot g(y)} g(y) dy = \frac{1}{c} \int_{-\infty}^x f(y) dy$$

De maneira semelhante, o denominador é dado por:

$$P\left(U \leq \frac{f(Y)}{c \cdot g(Y)}\right) = \int_{-\infty}^{\infty} \frac{f(y)}{c \cdot g(y)} g(y) dy = \frac{1}{c} \int_{-\infty}^{\infty} f(y) dy$$

Como $\int_{-\infty}^{\infty} f(y) dy = 1$ (pois $f(x)$ é uma função densidade), o denominador resulta em $\frac{1}{c}$. Substituindo essas expressões na equação original, obtemos:

$$P(X \leq x) = \frac{\frac{1}{c} \int_{-\infty}^x f(y) dy}{\frac{1}{c}} = \int_{-\infty}^x f(y) dy$$

Portanto, a variável X gerada pelo método de aceitação e rejeição tem função de distribuição acumulada $\int_{-\infty}^x f(y) dy$, o que implica que X tem densidade $f(x)$. Assim, o método gera amostras corretamente distribuídas de acordo com $f(x)$, como desejado.

42.1.2 Eficiência computacional

Teorema:

A variável aleatória gerada pelo método de aceitação e rejeição tem função de densidade $f(x)$. O número de passos que o algoritmo necessita para gerar X tem distribuição geométrica com média c , onde c é a constante de normalização que define o limite superior da razão $\frac{f(x)}{g(x)}$.

Prova:

Seja Y a variável aleatória gerada pela distribuição proposta $g(x)$, e $U \sim \text{Unif}(0, 1)$ uma variável aleatória uniforme. O método de aceitação e rejeição aceita Y como amostra de $f(x)$ se $U \leq \frac{f(Y)}{cg(Y)}$, caso contrário, o valor é rejeitado e o processo é repetido.

A probabilidade de aceitar uma amostra Y , dado que $Y \leq x$, é dada por:

$$P(Y \leq x \text{ e aceitar}) = P\left(Y \leq x, U \leq \frac{f(Y)}{cg(Y)}\right)$$

Podemos reescrever essa probabilidade como uma integral:

$$P(Y \leq x \text{ e aceitar}) = \int_{-\infty}^x \int_0^{\frac{f(y)}{cg(y)}} g(y) du dy$$

Resolvendo a integral em u , temos:

$$P(Y \leq x \text{ e aceitar}) = \int_{-\infty}^x \frac{f(y)}{cg(y)} g(y) dy = \frac{1}{c} \int_{-\infty}^x f(y) dy$$

A probabilidade de aceitar qualquer valor Y é dada por:

$$P(\text{aceitar}) = P\left(U \leq \frac{f(Y)}{cg(Y)}\right) = \int_{-\infty}^{\infty} \int_0^{\frac{f(y)}{cg(y)}} g(y) du dy$$

Simplificando:

$$P(\text{aceitar}) = \int_{-\infty}^{\infty} \frac{f(y)}{cg(y)} g(y) dy = \frac{1}{c} \int_{-\infty}^{\infty} f(y) dy = \frac{1}{c}$$

Assim, a cada passo, a probabilidade de aceitar um valor é $\frac{1}{c}$, o que implica que o número de passos necessários para aceitar uma amostra tem distribuição geométrica com média c .

Por fim, sabemos que a função de distribuição acumulada da variável X , dado que ela foi aceita, é:

$$P(X \leq x) = P(Y \leq x \mid \text{aceitou}) = \frac{P(Y \leq x, \text{aceitou})}{P(\text{aceitar})}$$

Substituindo as expressões:

$$P(X \leq x) = \frac{\frac{1}{c} \int_{-\infty}^x f(y) dy}{\frac{1}{c}} = \int_{-\infty}^x f(y) dy$$

Portanto, a variável X gerada pelo método de aceitação e rejeição tem função de densidade $f(x)$, como desejado.

42.2 Exercícios

Exercício 1. Seja $X \sim \text{Gama}(3/2, 1)$ com função densidade dada por

$$f(x) = \frac{1}{\Gamma(3/2)} x^{1/2} e^{-x}, \text{ para } x > 0.$$

- a) Escreva um pseudo-algoritmo para simular um valor da distribuição de X usando o método da aceitação e rejeição usando como distribuição proposta a distribuição exponencial de parâmetro $2/3$.

Observação: Note que $\mathbb{E}[X] = 3/2$ e $\mathbb{E}[Y] = 3/2$, por isso o parâmetro da distribuição exponencial proposta foi tomado como $2/3$.

- b) Crie uma função para gerar um valor de X .
- c) Qual foi o número de passos necessários para gerar um valor de X ? Compare com o valor real.
- d) Crie uma função para gerar uma amostra de tamanho n de X .
- e) Qual foi o número médio de passos necessários para gerar a amostra de tamanho n ?
- f) Compare a distribuição empírica dos valores simulados com a distribuição real de X .

43 Transformações e Misturas

44 R

45 Python

45.1 Exercícios

Exercício 1.

46 Método de Box-Muller

47 R

48 Python

48.1 Exercícios

Exercício 1.

49 Método de Monte Carlo

Os **Métodos de Monte Carlo** (MMC) são uma técnica poderosa que utiliza a simulação de valores aleatórios para calcular aproximações de valores esperados e integrais que não podem ser resolvidos analiticamente. Neste contexto, aprendemos a utilizar variáveis aleatórias simuladas para realizar o cálculo aproximado de integrais.

Seja X uma v.a. discreta com função de probabilidade $p(x)$ ou uma v.a. contínua com função densidade $f(x)$. O Método de Monte Carlo propõe um estimador para

$$\theta = \mathbb{E}[g(X)] = \begin{cases} \int_{-\infty}^{\infty} g(x)f(x) dx & \text{se } X \text{ é contínua} \\ \sum_x g(x)p(x) & \text{se } X \text{ é discreta} \end{cases}$$

Especificamente, ele funciona da seguinte forma. Seja X_1, \dots, X_n uma sequência de variáveis aleatórias i.i.d. com a mesma distribuição que X . Então:

$$\hat{\theta}_n = \frac{1}{n} \sum_{i=1}^n g(X_i)$$

é o estimador de Monte Carlo para θ .

Pseudo-Algoritmo para aproximar θ :

1. Gere X_1, \dots, X_n valores independentes da v.a. X .
2. Calcule:

$$\hat{\theta}_n = \frac{1}{n} \sum_{i=1}^n g(X_i)$$

Porque Funciona?

O método de Monte Carlo funciona devido à Lei Forte dos Grandes Números, que garante que a média amostral convergirá quase certamente para a esperança verdadeira conforme o número de amostras tende ao infinito:

$$\hat{\theta}_n = \frac{1}{n} \sum_{i=1}^n g(X_i) \longrightarrow \theta \text{ (quase certamente)}$$

Assim, quando n cresce, a estimativa de Monte Carlo converge para o verdadeiro valor de θ .

Importante: Freqüentemente n é denotado por B na literatura de Monte Carlo, de forma a não haver confusão com o tamanho de amostras não geradas no computador.

49.1 Exemplo 1: Estimativa de uma Integral

Queremos obter uma estimativa para

$$\theta = \int_0^1 e^{-x} dx$$

Para isso, basta observar que se $U \sim Unif(0,1)$, então $\theta = \mathbb{E}[e^{-U}]$. De fato, se $U \in Unif(0,1)$,

$$\theta = \int_0^1 e^{-x} dx = \int_0^1 e^{-u} f(u) du.$$

Assim, o estimador desta integral via MMC pode ser obtido da seguinte forma:

50 R

```
n <- 100
u <- runif(n, min = 0, max = 1)

# Estimativa de theta usando Monte Carlo
theta_hat <- mean(exp(-u))
cat("Estimativa de theta_hat:", theta_hat, "\n")
```

Estimativa de theta_hat: 0.6262142

```
# Valor real da integral
valor_real <- 1 - exp(-1)
cat("Valor real:", valor_real, "\n")
```

Valor real: 0.6321206

51 Python

```
import numpy as np

# Número de amostras
n = 100

# Geração de amostras uniformes
u = np.random.uniform(0, 1, n)

# Estimativa de theta usando Monte Carlo
theta_hat = np.mean(np.exp(-u))
print("Estimativa de theta_hat:", theta_hat)
```

Estimativa de theta_hat: 0.6634994440601483

```
# Valor real da integral
valor_real = 1 - np.exp(-1)
print("Valor real:", valor_real)
```

Valor real: 0.6321205588285577

Vamos verificar como o valor de n influencia na aproximação:

52 R

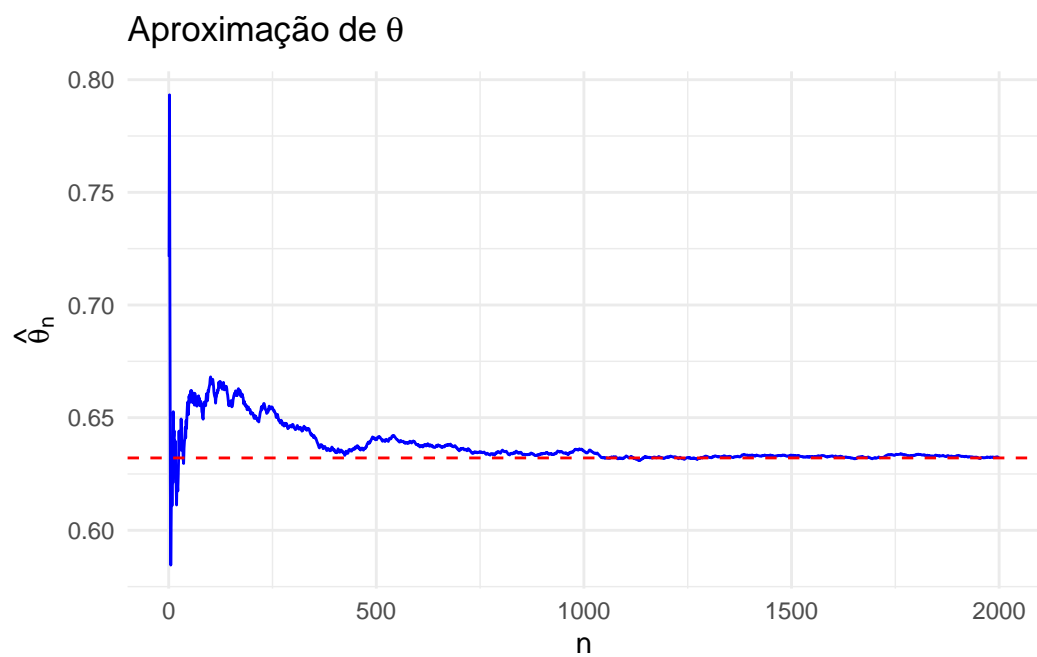
```
library(ggplot2)

set.seed(58)
N <- 1:2000
theta_hat <- numeric(length(N))
u <- numeric(length(N)) # Pré-alocação do vetor

# Preenchendo o vetor u e calculando theta_hat
for (i in seq_along(N)) {
  u[i] <- runif(1)
  theta_hat[i] <- mean(exp(-u[1:i]))
}

# Criando o dataframe para o ggplot
df <- data.frame(N = N, theta_hat = theta_hat)

# Plotar o gráfico usando ggplot2
ggplot(df, aes(x = N, y = theta_hat)) +
  geom_line(color = "blue") +
  geom_hline(yintercept = 1 - exp(-1), color = "red", linetype = "dashed") +
  labs(x = "n", y = expression(hat(theta)[n]), title = expression(Aproximação de theta)) +
  theme_minimal()
```



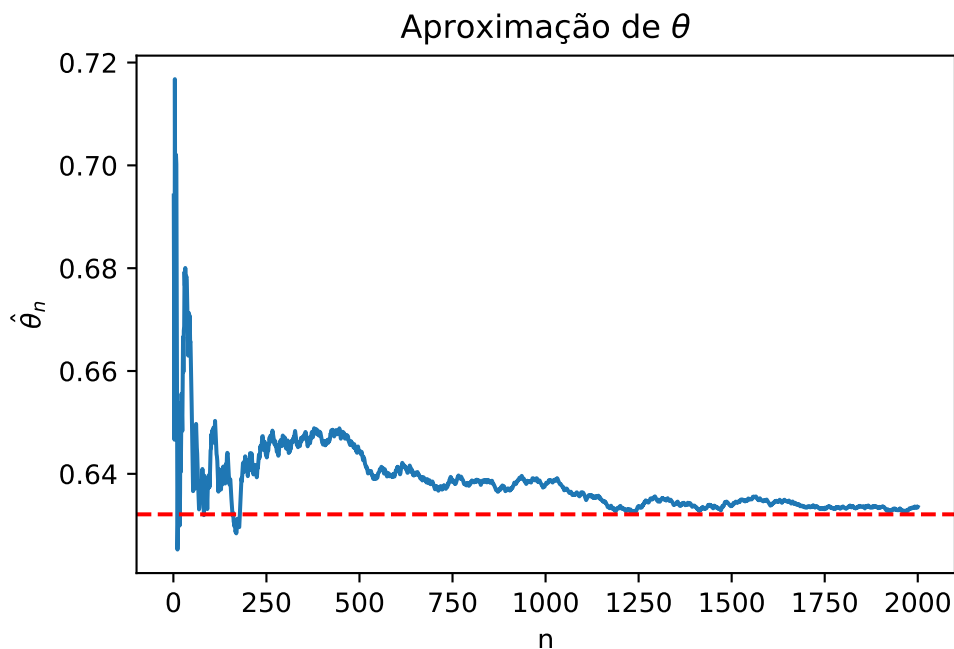
53 Python

```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(58)
N = np.arange(1, 2001, 1)
theta_hat = np.zeros(len(N))
u = np.array([])

for i in range(len(N)):
    u = np.append(u, np.random.uniform(0, 1))
    theta_hat[i] = np.mean(np.exp(-u))

plt.plot(N, theta_hat)
plt.axhline(y=1 - np.exp(-1), color='red', linestyle='--')
plt.xlabel('n')
plt.ylabel(r'$\hat{\theta}_n$')
plt.title(r'Aproximação de $\theta$')
plt.show()
```



Esse gráfico mostra a evolução da estimativa à medida que o número de variáveis aleatórias n aumenta, comparando com o valor real da integral.

53.1 Exemplo 2: Aproximando uma Probabilidade

Seja $X \sim \text{Gama}(2, 3)$. Queremos aproximar o valor de $\mathbb{P}(X \geq 0.4)$ usando o método de Monte Carlo. Note que

$$\theta := \mathbb{P}(X \geq 0.4) = \int g(x)f(x)dx,$$

em que $g(x) = I(x \geq 0.4)$ e $f(x)$ é a densidade da $\text{Gama}(2, 3)$. Assim, podemos aproximar a probabilidade gerada via MC.

Neste caso, o algoritmo corresponde a gerar $X_i \sim \text{Gama}(2, 3)$, definir e definir uma variável indicadora Y_i que vale 1 quando $X_i \geq 0.4$ e 0 caso contrário. A estimativa de Monte Carlo é dada pela média dos Y_i 's:

54 R

```
set.seed(58)
library(ggplot2)

# Número de amostras
N <- 50000

# Geração de amostras da distribuição Gamma
x <- rgamma(N, shape = 2, rate = 3) # Usando rate = 1/scale

# Cálculo de  $P(X \geq 0.4)$ 
y <- as.integer(x >= 0.4)

# Estimativa via Monte Carlo
valor_aproximado <- mean(y)
cat("Valor aproximado via Monte Carlo:", valor_aproximado, "\n")
```

Valor aproximado via Monte Carlo: 0.66282

```
# Valor real usando a função de distribuição acumulada (CDF)
valor_real <- 1 - pgamma(0.4, shape = 2, rate = 3)
cat("Valor real (CDF):", valor_real, "\n")
```

Valor real (CDF): 0.6626273

55 Python

```
import numpy as np
from scipy.stats import gamma

# Número de amostras
N = 50000

# Geração de amostras da distribuição Gamma
x = gamma.rvs(2, scale=1/3, size=N)

# Cálculo de  $P(X \geq 0.4)$ 
y = (x >= 0.4).astype(int)

# Estimativa via Monte Carlo
valor_aproximado = np.mean(y)
print("Valor aproximado via Monte Carlo:", valor_aproximado)
```

Valor aproximado via Monte Carlo: 0.66098

```
# Valor real usando a função de distribuição acumulada (CDF)
valor_real = 1 - gamma.cdf(0.4, 2, scale=1/3)
print("Valor real (CDF):", valor_real)
```

Valor real (CDF): 0.6626272662068446

55.1 Exemplo 3: Aproximando o valor de π

Neste exemplo, queremos aproximar o valor de π utilizando o método de Monte Carlo. A ideia é gerar pontos aleatórios em um quadrado e contar quantos caem dentro de um círculo inscrito no quadrado. Vamos seguir o raciocínio a partir da geometria básica.

55.1.0.1 Geometria

- Considere um quadrado com lado 2 centrado na origem, ou seja, o quadrado vai de $(-1, -1)$ até $(1, 1)$.
- Dentro deste quadrado, inscreva um círculo de raio 1, também centrado na origem.
- A área do quadrado é 4 (já que $2 \times 2 = 4$) e a área do círculo é $\pi \cdot r^2 = \pi \cdot 1^2 = \pi$.

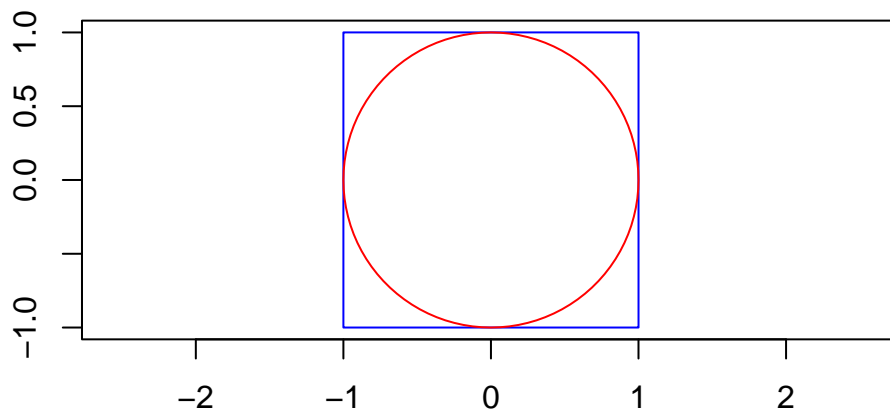
A razão entre a área do círculo e a área do quadrado é dada por:

$$\frac{\text{Área do círculo}}{\text{Área do quadrado}} = \frac{\pi}{4}$$

56 R

```
require(plotrix)
require(grid)

plot(c(-1, 1), c(-1,1), type = "n", asp=1,xlab="",yla="")
rect( -1, -1, 1, 1,border="blue")
draw.circle( 0, 0, 1 ,border="red")
```



57 Python

```
import matplotlib.pyplot as plt
from matplotlib.patches import Rectangle, Circle

# Configura o gráfico
fig, ax = plt.subplots()
ax.set_aspect('equal') # Define o aspecto como 1:1 (quadrado)
ax.set_xlim(-1, 1)
```

(-1.0, 1.0)

```
ax.set_ylim(-1, 1)
```

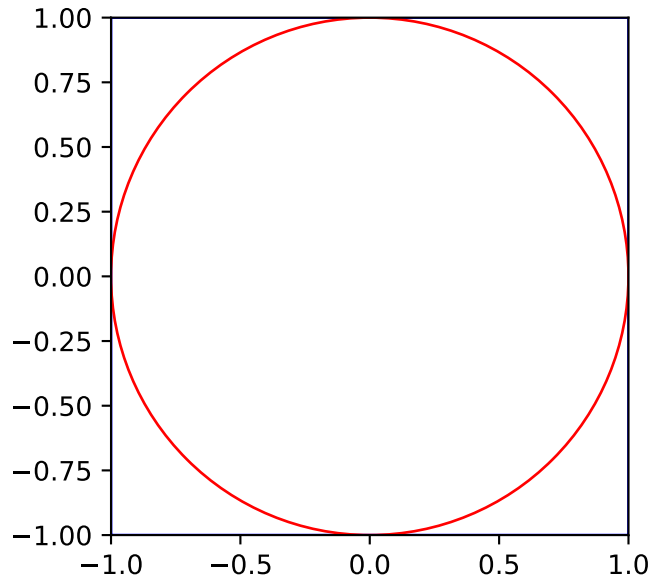
(-1.0, 1.0)

```
ax.set_xlabel('')
ax.set_ylabel('')

# Desenha o retângulo
rect = Rectangle((-1, -1), 2, 2, edgecolor='blue', facecolor='none')
ax.add_patch(rect)

# Desenha o círculo
circle = Circle((0, 0), 1, edgecolor='red', facecolor='none')
ax.add_patch(circle)

# Mostra o gráfico
plt.show()
```



Para estimar π usando Monte Carlo, procedemos da seguinte forma:

1. Geramos pontos aleatórios (x, y) no quadrado $[-1, 1] \times [-1, 1]$.
2. Verificamos se cada ponto está dentro do círculo, o que ocorre se $x^2 + y^2 \leq 1$.
3. A fração de pontos que caem dentro do círculo aproxima a razão $\frac{\pi}{4}$.
4. Multiplicamos essa fração por 4 para obter uma estimativa de π .

57.0.0.1 Matemática do Estimador

Formalmente, se $(X, Y) \sim \text{Unif}(-1, 1) \times \text{Unif}(-1, 1)$ e $g(x, y) = I((x, y) \text{ está no círculo})$, temos que

$$\theta := \int g(x, y) f(x, y) dx dy = \frac{\text{Área do círculo}}{\text{Área do quadrado}} = \frac{\pi}{4}$$

Assim, se (X_i, Y_i) é um ponto gerado uniformemente dentro do quadrado, o estimador de Monte Carlo para $\frac{\pi}{4}$ é dado por:

$$\hat{\theta}_n = \frac{1}{n} \sum_{i=1}^n g(x_i, y_i) = \frac{1}{n} \sum_{i=1}^n z_i,$$

em que $z_i = 1$ se o ponto i está dentro do círculo (i.e., se $x_i^2 + y_i^2 \leq 1$) e $z_i = 0$ caso contrário.

Multiplicando por 4, obtemos a estimativa de π :

$$\hat{\pi}_n = 4 \cdot \hat{\theta}_n = 4 \cdot \frac{1}{n} \sum_{i=1}^n z_i$$

De acordo com a **Lei Forte dos Grandes Números**, sabemos que:

$$\hat{\pi}_n \longrightarrow \pi \quad (\text{quase certamente, quando } n \rightarrow \infty).$$

Ou seja, à medida que o número de pontos simulados n aumenta, a estimativa $\hat{\pi}_n$ convergirá para o valor verdadeiro de π .

Implementação

Os trechos de código fornecidos em R e Python simulam esse processo, gerando n pontos e calculando a aproximação de π com base nos pontos que caem dentro do círculo. Além disso, são gerados gráficos que mostram como a estimativa de π melhora conforme o número de simulações aumenta, evidenciando a convergência mencionada.

Agora que todos os detalhes matemáticos do exemplo estão claros, o código para simulação pode ser executado para observar a aproximação prática de π .

58 R

```
set.seed(459)
library(ggplot2)

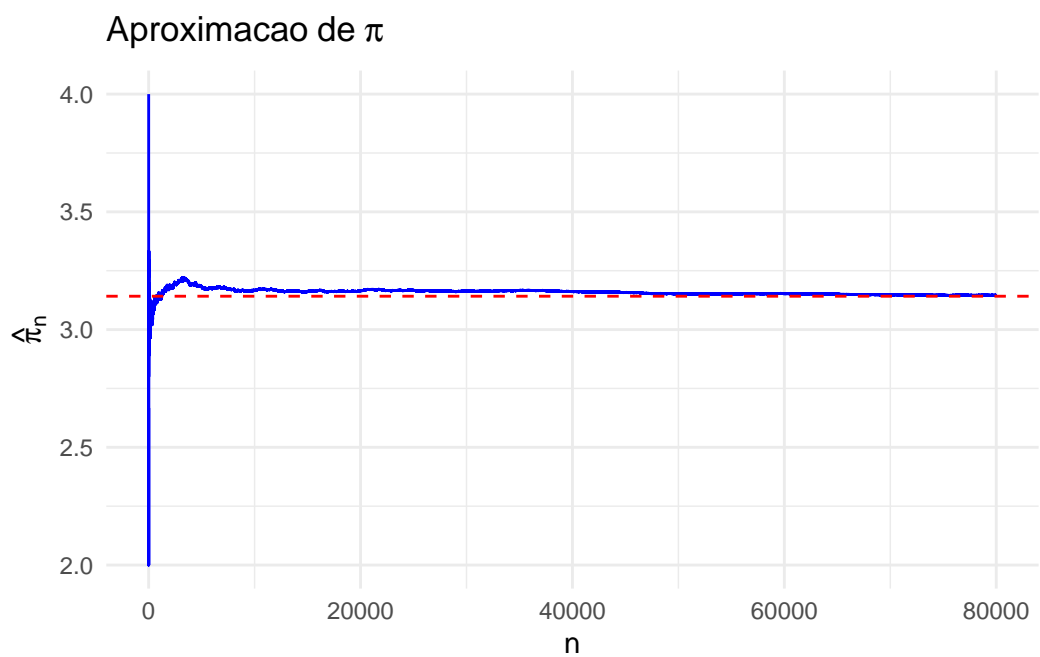
N <- 1:80000
z <- numeric(length(N))

# Loop para gerar os pontos e verificar se estão dentro do círculo
for (i in seq_along(N)) {
  x <- 2 * runif(1) - 1
  y <- 2 * runif(1) - 1
  z[i] <- (x^2 + y^2 <= 1)
}

# Cálculo da estimativa de pi
theta_hat <- cumsum(z) / N
pi_hat <- theta_hat * 4

# Criando o dataframe para o ggplot
df <- data.frame(N = N, pi_hat = pi_hat)

# Plotar o gráfico usando ggplot2
ggplot(df, aes(x = N, y = pi_hat)) +
  geom_line(color = "blue") +
  geom_hline(yintercept = pi, color = "red", linetype = "dashed") +
  labs(x = "n", y = expression(hat(pi)[n]), title = expression(Aproximacao-de~pi)) +
  theme_minimal()
```



59 Python

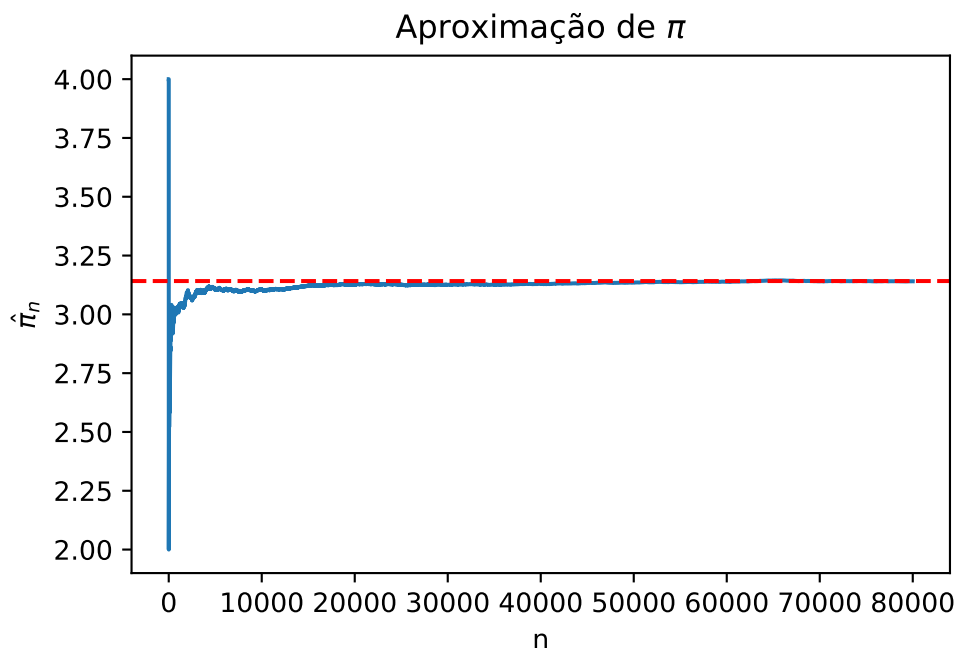
```
import numpy as np
import matplotlib.pyplot as plt

np.random.seed(459)
N = np.arange(1, 80001, 1)
z = np.zeros(len(N))

for i in range(len(N)):
    x = 2 * np.random.uniform(0, 1) - 1
    y = 2 * np.random.uniform(0, 1) - 1
    z[i] = (x**2 + y**2 <= 1)

theta_hat = np.cumsum(z) / N
pi_hat = theta_hat * 4

plt.plot(N, pi_hat)
plt.axhline(y=np.pi, color='red', linestyle='--')
plt.xlabel('n')
plt.ylabel(r'$\hat{\pi}_n$')
plt.title(r'Aproximação de $\pi$')
plt.show()
```



Esse gráfico mostra como a estimativa de π melhora conforme o número de pontos simulados n aumenta.

59.1 Intervalos de Confiança para Estimativas de Monte Carlo

O Método de Monte Carlo possibilita quantificar a incerteza em nossas estimativas. Para isso, podemos utilizar **intervalos de confiança**.

Suponha que temos uma sequência de amostras X_1, X_2, \dots, X_n de uma variável aleatória X , e calculamos a média amostral $\hat{\theta}_n = \frac{1}{n} \sum_{i=1}^n g(X_i)$. Pelo **Teorema Central do Limite**, sabemos que, para n suficientemente grande, a média amostral $\hat{\theta}_n$ é aproximadamente normalmente distribuída:

$$\hat{\theta}_n \sim N\left(\theta, \frac{\sigma^2}{n}\right)$$

onde $\sigma^2 = \text{Var}[g(X)]$ é a variância de $g(X)$. Essa aproximação permite a construção de um intervalo de confiança para θ .

O intervalo de confiança aproximado para θ com nível de confiança $(1 - \alpha) \times 100\%$ é dado por:

$$\hat{\theta}_n \pm z_{\alpha/2} \frac{\hat{\sigma}}{\sqrt{n}}$$

onde $z_{\alpha/2}$ é o quantil da distribuição normal padrão associado à probabilidade $\alpha/2$ e $\hat{\sigma}$ é a estimativa da variância de $g(X)$, obtida a partir das amostras.

Pseudo-Algoritmo para construir um intervalo de confiança:

1. Gere X_1, \dots, X_n amostras i.i.d. da v.a. X .
2. Calcule $\hat{\theta}_n = \frac{1}{n} \sum_{i=1}^n g(X_i)$.
3. Estime a variância de $g(X)$ como:

$$\hat{\sigma}^2 = \frac{1}{n} \sum_{i=1}^n (g(X_i) - \hat{\theta}_n)^2$$

4. Calcule o intervalo de confiança:

$$\hat{\theta}_n \pm z_{\alpha/2} \frac{\hat{\sigma}}{\sqrt{n}}$$

59.1.1 Exemplo: Intervalo de Confiança para a Estimativa de π

Neste exemplo, construímos um intervalo de confiança para a estimativa de π usando o método de Monte Carlo, seguindo os passos acima.

60 R

```
set.seed(0)
N <- 10000 # número de amostras
z <- numeric(N)

# Loop para gerar os pontos e verificar se estão dentro do círculo
for (i in 1:N) {
  x <- 2 * runif(1) - 1
  y <- 2 * runif(1) - 1
  z[i] <- (x^2 + y^2 <= 1)
}

# Estimativa de pi
theta_hat <- mean(z)
pi_hat <- theta_hat * 4
cat("Estimativa de pi:", pi_hat, "\n")
```

Estimativa de pi: 3.1308

```
# Cálculo da variância e intervalo de confiança
sigma_hat <- sqrt(var(z) / N)
alpha <- 0.05 # Nível de significância
z_alpha2 <- qnorm(1 - alpha / 2)

# Intervalo de confiança
IC <- c(theta_hat - z_alpha2 * sigma_hat, theta_hat + z_alpha2 * sigma_hat) * 4
cat("Intervalo de confiança para pi:", IC, "\n")
```

Intervalo de confiança para pi: 3.098466 3.163134

61 Python

```
import numpy as np
from scipy.stats import norm

# Semente para reprodução
np.random.seed(0)
N = 10000 # número de amostras
z = np.zeros(N)

# Loop para gerar os pontos e verificar se estão dentro do círculo
for i in range(N):
    x = 2 * np.random.uniform(0, 1) - 1
    y = 2 * np.random.uniform(0, 1) - 1
    z[i] = (x**2 + y**2 <= 1)

# Estimativa de pi
theta_hat = np.mean(z)
pi_hat = theta_hat * 4
print("Estimativa de pi:", pi_hat)
```

Estimativa de pi: 3.1228

```
# Cálculo da variância e intervalo de confiança
sigma_hat = np.sqrt(np.var(z) / N)
alpha = 0.05 # Nível de significância
z_alpha2 = norm.ppf(1 - alpha / 2)

# Intervalo de confiança
IC = (theta_hat - z_alpha2 * sigma_hat, theta_hat + z_alpha2 * sigma_hat)
IC = [i * 4 for i in IC]

print("Intervalo de confiança para pi:", IC)
```

Intervalo de confiança para pi: [np.float64(3.090360848359833), np.float64(3.155239151640166)]

Neste caso, o intervalo de confiança construído em torno da estimativa $\hat{\pi}_n$ oferece uma ideia de quão próxima nossa estimativa está do valor verdadeiro de π , com uma certa confiança.

61.1 Exemplo 4: Problema das Figurinhas

Um colecionador está juntando figurinhas para completar um álbum da Copa. Qual é a probabilidade de que, ao comprar n pacotes, pelo menos um deles contenha duas ou mais figurinhas iguais? Durante o curso de probabilidade, você aprenderá como calcular essa probabilidade. Outra abordagem possível é o uso do método de Monte Carlo. Nesse caso, simulamos a compra de n pacotes, cada um com 5 figurinhas, de um total de 640 figurinhas disponíveis no álbum. Repetimos essa simulação $B = 1000$ vezes e contamos quantas vezes ocorreu pelo menos uma repetição de figurinhas em algum dos pacotes. A proporção de simulações com repetições é a estimativa de Monte Carlo da probabilidade que estamos interessados em calcular.

62 R

```
nFigurinhas <- 640
B <- 1000
nPacotes <- 1:50
aoMenosUmaRepetidaNoMesmoPacote <- matrix(NA, length(nPacotes), B)
for(ii in 1:length(nPacotes)) {
  for(jj in 1:B) {
    figurinhas <- sample(1:nFigurinhas, nPacotes[ii] * 5, replace = TRUE)
    figurinhas <- matrix(figurinhas, nPacotes[ii], 5)
    aoMenosUmaRepetidaNoMesmoPacote[ii,jj] <- sum(apply(figurinhas, 1, function(xx) length(u
  })
}

probCoincidencia <- apply(aoMenosUmaRepetidaNoMesmoPacote, 1, mean)
```

63 Python

```
import numpy as np
import random

nFigurinhas = 640
B = 1000
nPacotes = np.arange(1, 51) # Intervalo de 1 a 50 (Python é exclusivo no final)
aoMenosUmaRepetidaNoMesmoPacote = np.empty((len(nPacotes), B))

for ii in range(len(nPacotes)):
    for jj in range(B):
        figurinhas = [random.choices(range(1, nFigurinhas + 1), k=5) for _ in range(nPacotes)]
        figurinhas = np.array(figurinhas)
        aoMenosUmaRepetidaNoMesmoPacote[ii, jj] = np.sum([len(set(pacote)) != len(pacote) for _ in range(nPacotes)])

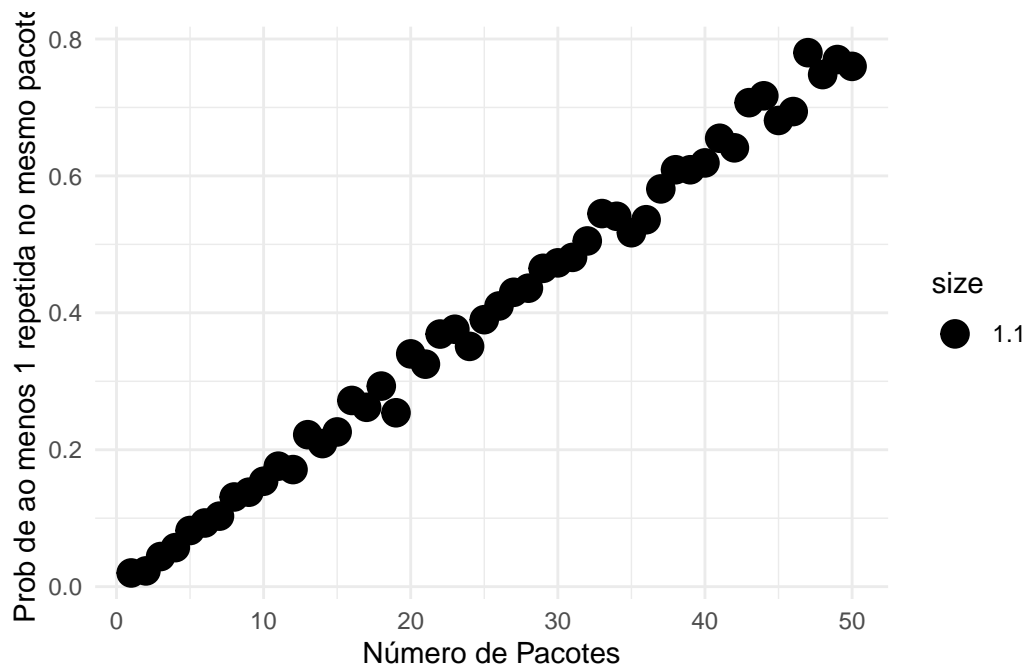
probCoincidencia = np.mean(aoMenosUmaRepetidaNoMesmoPacote, axis=1)
```

Agora podemos visualizar os resultados obtidos por meio do gráfico a seguir:

64 R

```
dados <- data.frame(probCoincidencia = probCoincidencia, nPacotes = nPacotes)

library(ggplot2)
ggplot(dados, aes(x = nPacotes, y = probCoincidencia)) +
  geom_point(aes(size = 1.1)) +
  xlab("Número de Pacotes") +
  ylab("Prob de ao menos 1 repetida no mesmo pacote") +
  theme(legend.position = "none")+
  theme_minimal()
```

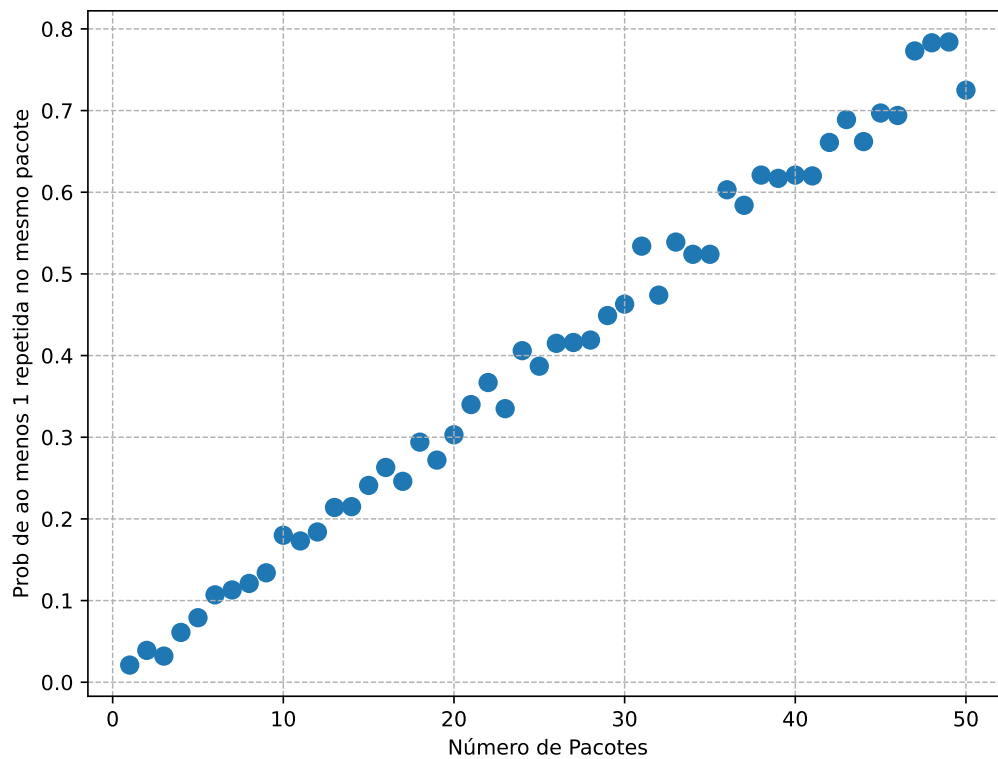


65 Python

```
import pandas as pd
import matplotlib.pyplot as plt

# Criação do DataFrame
dados = pd.DataFrame({'nPacotes': nPacotes, 'probCoincidencia': probCoincidencia})

# Criação do gráfico
plt.figure(figsize=(8, 6))
plt.scatter(dados['nPacotes'], dados['probCoincidencia'], s=70) # 's' ajusta o tamanho dos p
plt.xlabel('Número de Pacotes')
plt.ylabel('Prob de ao menos 1 repetida no mesmo pacote')
plt.grid(True, which='both', linestyle='--', linewidth=0.7)
plt.gca().set_facecolor('white') # Para seguir o tema minimalista
plt.show()
```



Simulação para completar o álbum

Uma segunda pergunta que também podemos responder via simulação é: quantos pacotes necessitamos em média para completar o álbum? A solução estimada pode ser obtida com o código abaixo:

66 R

```
numeroPacotes <- rep(NA, B)
for(jj in 1:B) {
  albumCompleto <- FALSE
  nPacotes <- 0
  figurinhas <- NULL
  while(!albumCompleto) {
    figurinhas <- c(figurinhas, sample(1:nFigurinhas, 5, replace = TRUE))
    if(length(unique(figurinhas)) == nFigurinhas) {
      albumCompleto <- TRUE
    }
    nPacotes <- nPacotes + 1
  }
  numeroPacotes[jj] <- nPacotes
}
```

67 Python

```
import numpy as np
import random

nFigurinhas = 640
B = 1000
numeroPacotes = np.empty(B)

for jj in range(B):
    albumCompleto = False
    nPacotes = 0
    figurinhas = []

    while not albumCompleto:
        figurinhas.extend(random.choices(range(1, nFigurinhas + 1), k=5))
        if len(set(figurinhas)) == nFigurinhas:
            albumCompleto = True
            nPacotes += 1

    numeroPacotes[jj] = nPacotes
```

O número médio de pacotes necessários para completar o álbum é:

68 R

```
mean(numeroPacotes)
```

```
[1] 901.629
```

69 Python

```
mean_numeroPacotes = np.mean(numeroPacotes)
mean_numeroPacotes
```

```
np.float64(902.261)
```

Com essa simulação, também podemos estimar as seguintes probabilidades:

70 R

```
cat("Probabilidade de precisar de mais de 800 pacotes: ", mean(numeroPacotes > 800) * 100, "%")
```

Probabilidade de precisar de mais de 800 pacotes: 69.3%

```
cat("Probabilidade de precisar de mais de 1000 pacotes: ", mean(numeroPacotes > 1000) * 100, "%")
```

Probabilidade de precisar de mais de 1000 pacotes: 23.2%

71 Python

```
prob_mais_800 = np.mean(numeroPacotes > 800) * 100
prob_mais_1000 = np.mean(numeroPacotes > 1000) * 100

print(f"Probabilidade de precisar de mais de 800 pacotes: {prob_mais_800}%")
```

Probabilidade de precisar de mais de 800 pacotes: 70.5%

```
print(f"Probabilidade de precisar de mais de 1000 pacotes: {prob_mais_1000}%")
```

Probabilidade de precisar de mais de 1000 pacotes: 21.4%

71.1 Exercícios

Exercício 1.

Utilize o método de Monte Carlo para aproximar a integral

$$I = \int_0^{10} \sin(x^2) dx$$

Forneça um intervalo de confiança para avaliar a precisão da estimativa.

Exercício 1.

Utilize o método de Monte Carlo para aproximar a integral

$$I = \int_1^{\infty} \frac{1}{x^3} dx$$

Forneça um intervalo de confiança para avaliar a precisão da estimativa.

Exercício 2.

Seja $X \sim \text{Exp}(\lambda)$ com $\lambda = 2$. Estime a esperança $\mathbb{E}[X^2]$ utilizando o método de Monte Carlo. Forneça um intervalo de confiança para avaliar a precisão da estimativa. Compare com o valor real dessa quantidade.

Exercício 3.

Considere que X tem distribuição Normal(0,1) e Y tem distribuição Gamma(1,1). Estime $P(X \times Y > 3)$. Forneça um intervalo de confiança para avaliar a precisão da estimativa.

72 Técnicas de Redução de Variância

73 R

74 Python

74.1 Exercícios

Exercício 1.

75 Amostragem por Importância

O Método de Monte Carlo, embora funcione, nem sempre é eficiente. Veremos a seguir um exemplo disso.

75.1 Exemplo 1: Estimativa de $\mathbb{P}(Z > 4.5)$

Neste exemplo, vamos estimar a probabilidade $\mathbb{P}(Z > 4.5)$ onde $Z \sim N(0, 1)$. Este é um evento raro, com probabilidade pequena. O valor exato dessa probabilidade é:

$$\log(\mathbb{P}(Z > 4.5)) = -12.59242$$

Aplicaremos o método de Monte Carlo simulando $n = 100.000$ valores de $Z \sim N(0, 1)$ e calculando a proporção dos valores que são maiores que 4.5.

76 R

```
set.seed(0)
# Valor exato de  $P(Z > 4.5)$ 
valor_exato <- log(pnorm(4.5, 0, 1, lower.tail = FALSE))
cat("Valor exato (log(P(Z > 4.5))):", valor_exato, "\n")
```

Valor exato (log(P(Z > 4.5))): -12.59242

```
# Simulação Monte Carlo
n <- 100000
z <- rnorm(n, 0, 1)

# Estimativa Monte Carlo de  $P(Z > 4.5)$ 
p_estimada <- mean(1 * (z > 4.5))
cat("Estimativa Monte Carlo de P(Z > 4.5):", p_estimada, "\n")
```

Estimativa Monte Carlo de $P(Z > 4.5)$: 0

```
# Logaritmo da estimativa Monte Carlo
log_p_estimada <- log(p_estimada)
cat("Logaritmo da estimativa Monte Carlo:", log_p_estimada, "\n")
```

Logaritmo da estimativa Monte Carlo: -Inf

77 Python

```
import numpy as np
from scipy.stats import norm

# Valor exato de  $P(Z > 4.5)$ 
valor_exato = np.log(norm.sf(4.5)) # sf is the survival function, equivalent to (1 - cdf)
print(f"Valor exato ( $\log(P(Z > 4.5))$ ): {valor_exato}")
```

Valor exato ($\log(P(Z > 4.5))$): -12.592419735713081

```
# Simulação Monte Carlo
n = 100000
z = np.random.normal(0, 1, n)

# Estimativa Monte Carlo de  $P(Z > 4.5)$ 
p_estimada = np.mean(z > 4.5)
print(f"Estimativa Monte Carlo de  $P(Z > 4.5)$ : {p_estimada}")
```

Estimativa Monte Carlo de $P(Z > 4.5)$: 0.0

```
# Logaritmo da estimativa Monte Carlo
log_p_estimada = np.log(p_estimada)
print(f"Logaritmo da estimativa Monte Carlo: {log_p_estimada}")
```

Logaritmo da estimativa Monte Carlo: -inf

O método de Monte Carlo exige uma grande quantidade de amostras para fornecer uma estimativa precisa devido à raridade do evento $Z > 4.5$. A abordagem pode se tornar ineficiente para eventos raros.

77.1 Amostragem por Importância

A Amostragem por Importância é uma técnica que nos permite simular valores Y_1, \dots, Y_n a partir de uma densidade auxiliar $g(y)$.

A ideia chave é que, se X tem densidade $f(x)$, a esperança $\mathbb{E}[h(X)]$ pode ser reescrita como:

$$\mathbb{E}[h(X)] = \int h(x)f(x)dx = \int h(x)\frac{f(x)}{g(x)}g(x)dx = \mathbb{E}\left[h(Y)\frac{f(Y)}{g(Y)}\right],$$

em que Y tem densidade g . Isso vale desde que g possua suporte tão ou mais amplo que $f(x)$. Assim, podemos estimar $\mathbb{E}[h(X)]$ usando n valores simulados de Y independentemente pela fórmula:

$$\hat{\theta}_n = \frac{1}{n} \sum_{i=1}^n \frac{f(y_i)}{g(y_i)} h(y_i)$$

Note que os pesos $w_i = \frac{f(y_i)}{g(y_i)}$ ajustam a importância de cada amostra, tornando a simulação mais eficiente.

77.1.1 Revistando o Exemplo 1 via Amostragem por Importância

Para melhorar a eficiência no Exemplo 1, vamos agora usar **Amostragem por Importância**. Em vez de simular diretamente de $Z \sim N(0, 1)$, simulamos de uma distribuição exponencial truncada $Exp(1)$ a partir de 4.5. A densidade de importância é:

$$g(y) = e^{-(y-4.5)}, \quad y > 4.5$$

O estimador de Monte Carlo ponderado é, portanto,

$$\hat{\theta}_n = \frac{1}{n} \sum_{i=1}^n \frac{f(y_i)}{g(y_i)} I(y_i > 4.5) = \frac{1}{n} \sum_{i=1}^n \frac{f(y_i)}{g(y_i)},$$

onde $f(y_i)$ é a densidade da normal $N(0, 1)$. Aqui está o código equivalente em Python no formato solicitado:

78 R

```
n <- 100
# Gerar n valores da distribuição exponencial truncada
y <- rexp(n) + 4.5

# Estimativa por importância
estimativa_importancia <- mean(dnorm(y) / exp(-(y - 4.5)))
cat("Estimativa por amostragem por importância:", estimativa_importancia, "\n")
```

Estimativa por amostragem por importância: 3.723979e-06

```
# Logaritmo da estimativa por importância
log_estimativa_importancia <- log(estimativa_importancia)
cat("Logaritmo da estimativa por amostragem por importância:", log_estimativa_importancia, "\n")
```

Logaritmo da estimativa por amostragem por importância: -12.50072

79 Python

```
import numpy as np
from scipy.stats import expon, norm

n = 100
# Gerar n valores da distribuição exponencial truncada
y = expon.rvs(scale=1, size=n) + 4.5

# Estimativa por importância
estimativa_importancia = np.mean(norm.pdf(y) / np.exp(-(y - 4.5)))
print(f"Estimativa por amostragem por importância: {estimativa_importancia}")
```

Estimativa por amostragem por importância: 3.6627982519945895e-06

```
# Logaritmo da estimativa por importância
log_estimativa_importancia = np.log(estimativa_importancia)
print(f"Logaritmo da estimativa por amostragem por importância: {log_estimativa_importancia}")
```

Logaritmo da estimativa por amostragem por importância: -12.517283152944959

Com essa técnica, concentramos as simulações na região relevante, melhorando a eficiência da estimativa para eventos raros.

79.1 Exemplo 2: Estimativa de $\mathbb{E}[\sqrt{X}]$ com Amostragem por Importância

Agora, vamos estimar $\theta = \mathbb{E}[\sqrt{X}]$ onde $X \sim \text{Exp}(1)$. Sabemos que o valor exato desta integral é $\sqrt{\pi}/2$.

79.1.1 Aproximação via Monte Carlo

Vamos simular $X \sim \text{Exp}(1)$ e calcular a média de \sqrt{X} diretamente: Aqui está o código equivalente em Python no formato solicitado:

80 R

```
# Valor exato da integral
real <- sqrt(pi) / 2
cat("Valor exato:", real, "\n")
```

Valor exato: 0.8862269

```
# Número de amostras
n <- 10000

# Geração de amostras da distribuição exponencial
x <- rexp(n, 1)

# Estimativa Monte Carlo para E[sqrt(X)]
estimativa_mc <- mean(sqrt(x))
cat("Estimativa Monte Carlo de E[sqrt(X)]:", estimativa_mc, "\n")
```

Estimativa Monte Carlo de E[sqrt(X)]: 0.8842487

```
# Variância da estimativa
variancia_mc <- var(sqrt(x))
cat("Variância da estimativa:", variancia_mc, "\n")
```

Variância da estimativa: 0.2075346

81 Python

```
import numpy as np
from scipy.stats import expon

# Valor exato da integral
real = np.sqrt(np.pi) / 2
print(f"Valor exato: {real}")
```

Valor exato: 0.8862269254527579

```
# Número de amostras
n = 10000

# Geração de amostras da distribuição exponencial
x = expon.rvs(scale=1, size=n)

# Estimativa Monte Carlo para  $E[\sqrt{X}]$ 
estimativa_mc = np.mean(np.sqrt(x))
print(f"Estimativa Monte Carlo de  $E[\sqrt{X}]$ : {estimativa_mc}")
```

Estimativa Monte Carlo de $E[\sqrt{X}]$: 0.8858603621560999

```
# Variância da estimativa
variancia_mc = np.var(np.sqrt(x), ddof=1)
print(f"Variância da estimativa: {variancia_mc}")
```

Variância da estimativa: 0.2088529703798996

81.0.1 Aproximação via Amostragem por Importância

Agora, vamos usar a **Amostragem por Importância** com duas distribuições diferentes para melhorar a eficiência.

1. Usando $Y = |Z|$, onde $Z \sim N(0, 1)$:

A densidade de importância combinada é:

$$g(y) = \text{dnorm}(y) + \text{dnorm}(-y)$$

```
# Gerando amostras de uma distribuição normal absoluta
z <- abs(rnorm(n))

# Função de densidade combinada g(x)
g <- function(x) {
  return(dnorm(x) + dnorm(-x))
}

# Estimativa por amostragem por importância
estimativa_importancia <- mean((sqrt(z)) * dexp(z, 1) / g(z))
cat("Estimativa por amostragem por importância:", estimativa_importancia, "\n")
```

Estimativa por amostragem por importância: 1.038467

```
# Variância da estimativa por amostragem por importância
variancia_importancia <- var((sqrt(z)) * dexp(z, 1) / g(z))
cat("Variância da estimativa por amostragem por importância:", variancia_importancia, "\n")
```

Variância da estimativa por amostragem por importância: 367.4869

2. Usando $Y = |Z|$, onde $Z \sim \text{Cauchy}(0, 1)$:

A densidade de importância combinada agora é:

$$g_2(y) = \text{dcauchy}(y) + \text{dcauchy}(-y)$$

82 R

```
# Gerando amostras de uma distribuição Cauchy absoluta
z <- abs(rcauchy(n))

# Função de densidade combinada g2(x) para a Cauchy
g2 <- function(x) {
  return(dcauchy(x) + dcauchy(-x))
}

# Estimativa por amostragem por importância usando Cauchy
estimativa_importancia_cauchy <- mean((sqrt(z)) * dexp(z, 1) / g2(z))
cat("Estimativa por amostragem por importância (Cauchy):", estimativa_importancia_cauchy, "\n")
```

Estimativa por amostragem por importância (Cauchy): 0.885894

```
# Variância da estimativa por amostragem por importância usando Cauchy
variancia_importancia_cauchy <- var((sqrt(z)) * dexp(z, 1) / g2(z))
cat("Variância da estimativa por amostragem por importância (Cauchy):", variancia_importancia_cauchy, "\n")
```

Variância da estimativa por amostragem por importância (Cauchy): 0.1985572

83 Python

```
import numpy as np
from scipy.stats import cauchy, expon

# Gerando amostras de uma distribuição Cauchy absoluta
z = np.abs(cauchy.rvs(size=n))

# Função de densidade combinada  $g_2(x)$  para a Cauchy
def g2(x):
    return cauchy.pdf(x) + cauchy.pdf(-x)

# Estimativa por amostragem por importância usando Cauchy
estimativa_importancia_cauchy = np.mean(np.sqrt(z) * expon.pdf(z, scale=1) / g2(z))
print(f"Estimativa por amostragem por importância (Cauchy): {estimativa_importancia_cauchy}")
```

Estimativa por amostragem por importância (Cauchy): 0.8887968817462532

```
# Variância da estimativa por amostragem por importância usando Cauchy
variancia_importancia_cauchy = np.var(np.sqrt(z) * expon.pdf(z, scale=1) / g2(z), ddof=1)
print(f"Variância da estimativa por amostragem por importância (Cauchy): {variancia_importancia_cauchy}")
```

Variância da estimativa por amostragem por importância (Cauchy): 0.1950708899110345

Podemos visualizar as funções de densidade $h(x)f(x)$ e as funções de importância $g(x)$ e $g_2(x)$ para entender melhor a escolha das distribuições de importância .

Gráfico de $h(x)f(x)$ e as funções de importância $g(x)$ e $g_2(x)$: Aqui está o código convertido para Python usando a biblioteca `matplotlib` e `seaborn` no formato solicitado:

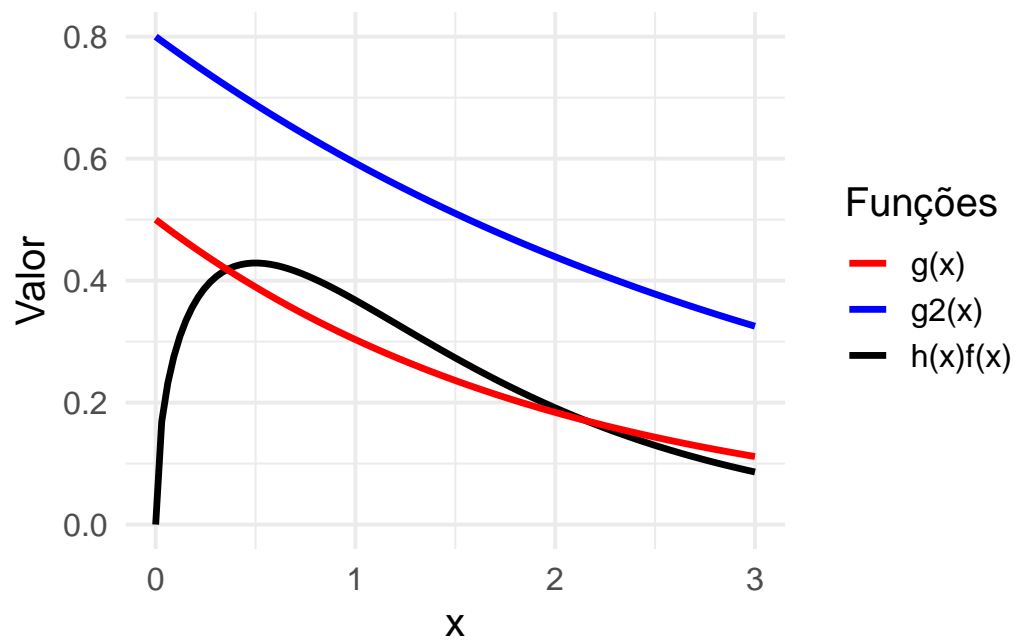
84 R

```
library(ggplot2)

# Funções
h_f <- function(x) sqrt(x) * dexp(x, 1)
g <- function(x) 0.5 * exp(-0.5 * x) # Exemplo para g(x)
g2 <- function(x) 0.8 * exp(-0.3 * x) # Exemplo para g2(x)

# Dados
x_vals <- seq(0, 3, length.out = 100)
data <- data.frame(
  x = x_vals,
  h_f = h_f(x_vals),
  g = g(x_vals),
  g2 = g2(x_vals)
)

# Gráfico
ggplot(data, aes(x)) +
  geom_line(aes(y = h_f, color = "h(x)f(x)"), size = 1.2) +
  geom_line(aes(y = g, color = "g(x)"), size = 1.2) +
  geom_line(aes(y = g2, color = "g2(x)"), size = 1.2) +
  scale_color_manual(values = c("h(x)f(x)" = "black", "g(x)" = "red", "g2(x)" = "blue")) +
  labs(y = "Valor", color = "Funções") +
  theme_minimal(base_size = 15)
```



85 Python

```
import numpy as np
import matplotlib.pyplot as plt

# Funções
def h_f(x):
    return np.sqrt(x) * np.exp(-x)

def g(x):
    return 0.5 * np.exp(-0.5 * x)

def g2(x):
    return 0.8 * np.exp(-0.3 * x)

# Dados
x_vals = np.linspace(0, 3, 100)
h_f_vals = h_f(x_vals)
g_vals = g(x_vals)
g2_vals = g2(x_vals)

# Gráfico
plt.plot(x_vals, h_f_vals, label="h(x)f(x)", color="black", linewidth=1.2)
plt.plot(x_vals, g_vals, label="g(x)", color="red", linewidth=1.2)
plt.plot(x_vals, g2_vals, label="g2(x)", color="blue", linewidth=1.2)

plt.xlabel("x")
plt.ylabel("Valor")
plt.legend(title="Funções")
plt.grid(True)
plt.tight_layout()
plt.show()
```

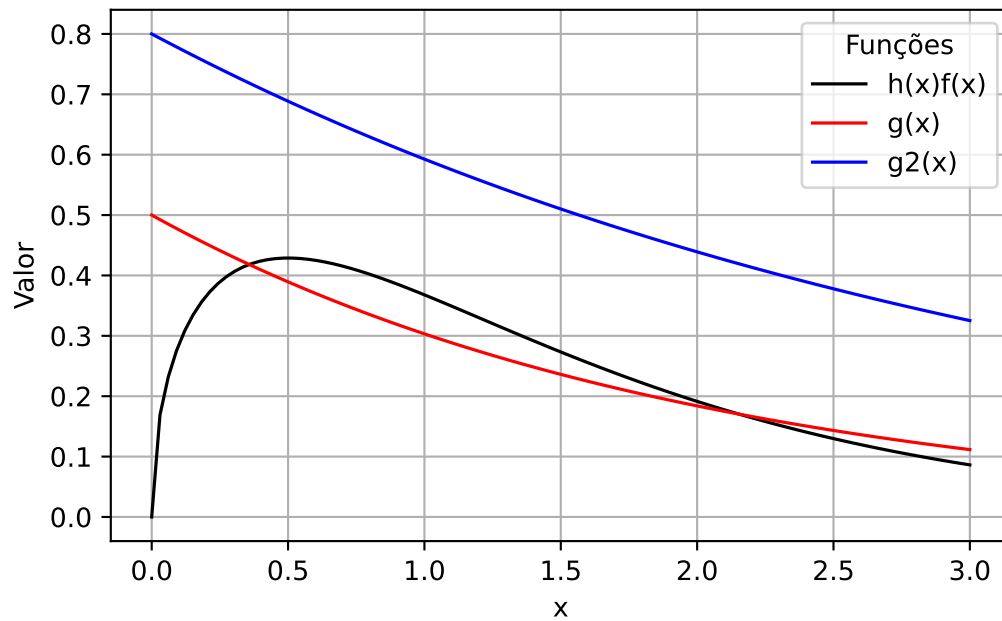


Gráfico de $h(x)f(x)/g(x)$ e $h(x)f(x)/g2(x)$: Aqui está o código equivalente em Python usando `matplotlib` no formato solicitado:

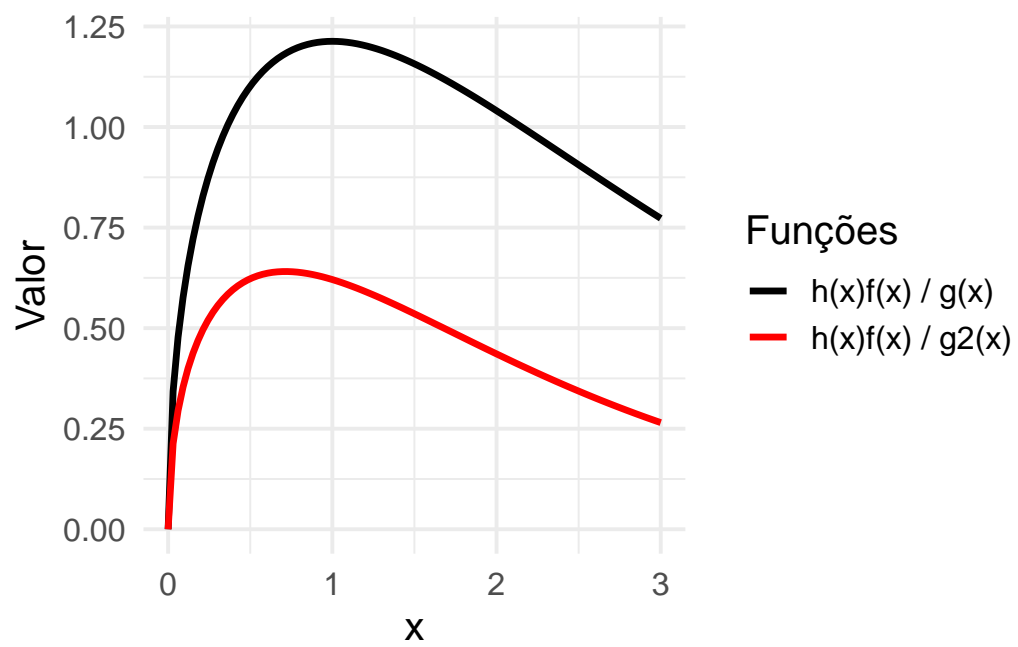
86 R

```
library(ggplot2)

# Funções
h_f <- function(x) sqrt(x) * dexp(x, 1)
g <- function(x) 0.5 * exp(-0.5 * x) # Exemplo para g(x)
g2 <- function(x) 0.8 * exp(-0.3 * x) # Exemplo para g2(x)

# Dados
x_vals <- seq(0, 3, length.out = 100)
data <- data.frame(
  x = x_vals,
  h_f_over_g = h_f(x_vals) / g(x_vals),
  h_f_over_g2 = h_f(x_vals) / g2(x_vals)
)

# Gráfico
ggplot(data, aes(x)) +
  geom_line(aes(y = h_f_over_g, color = "h(x)f(x) / g(x)", size = 1.2) +
  geom_line(aes(y = h_f_over_g2, color = "h(x)f(x) / g2(x)", size = 1.2) +
  scale_color_manual(values = c("h(x)f(x) / g(x)" = "black", "h(x)f(x) / g2(x)" = "red")) +
  labs(y = "Valor", color = "Funções") +
  theme_minimal(base_size = 15)
```

87 Python

```
import numpy as np
import matplotlib.pyplot as plt

# Funções
def h_f(x):
    return np.sqrt(x) * np.exp(-x)

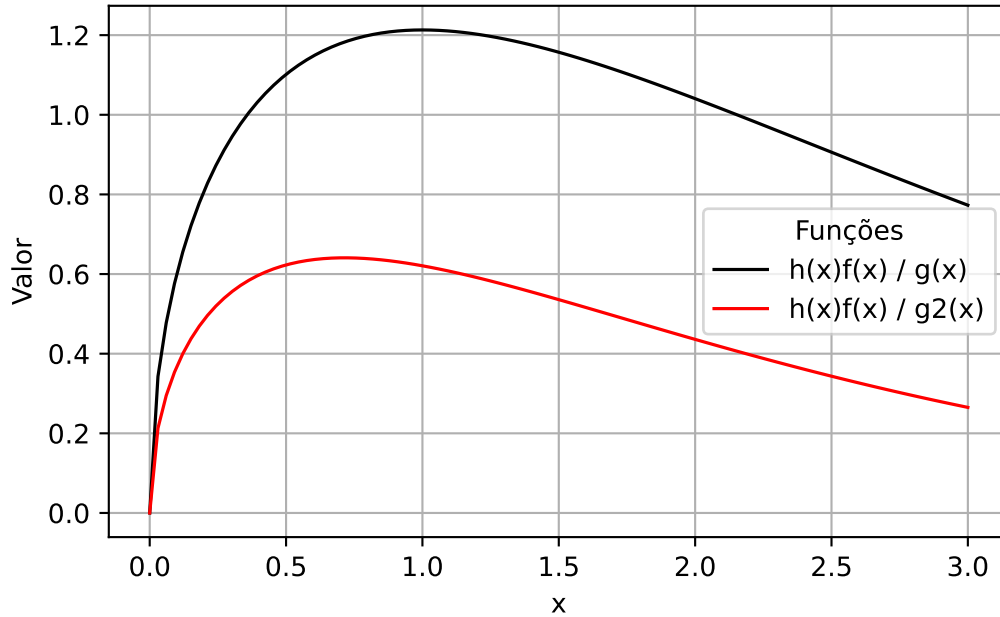
def g(x):
    return 0.5 * np.exp(-0.5 * x)

def g2(x):
    return 0.8 * np.exp(-0.3 * x)

# Dados
x_vals = np.linspace(0, 3, 100)
h_f_over_g = h_f(x_vals) / g(x_vals)
h_f_over_g2 = h_f(x_vals) / g2(x_vals)

# Gráfico
plt.plot(x_vals, h_f_over_g, label="h(x)f(x) / g(x)", color="black", linewidth=1.2)
plt.plot(x_vals, h_f_over_g2, label="h(x)f(x) / g2(x)", color="red", linewidth=1.2)

plt.xlabel("x")
plt.ylabel("Valor")
plt.legend(title="Funções")
plt.grid(True)
plt.tight_layout()
plt.show()
```



Esses gráficos mostram como as diferentes escolhas de distribuições de importância afetam a função ponderada, ajudando a reduzir a variância do estimador.

87.1 Exercícios

Exercício 1. Usando a **Amostragem por Importância**, estime a probabilidade de que uma variável aleatória $Z \sim N(0, 1)$ seja maior que 5.0. Compare os resultados obtidos com a amostragem direta usando Monte Carlo simples e discuta a eficiência da Amostragem por Importância em relação ao método direto.

Exercício 2. Estime o valor da integral $\int_0^2 \sqrt{x}e^{-x}dx$ utilizando a **Amostragem por Importância** com duas distribuições de importância diferentes. Justifique a escolha das distribuições e compare a eficiência de cada uma.

Exercício 3. Utilize a **Amostragem por Importância** para estimar $\mathbb{E}[\log(1+X)]$, onde $X \sim \text{Exp}(1)$. Escolha uma densidade auxiliar apropriada para melhorar a eficiência da estimativa e justifique sua escolha. Compare os resultados com a aproximação via Monte Carlo simples.

References

Ross, Sheldon M. 2006. *Simulation, Fourth Edition*. USA: Academic Press, Inc.