Curso de Programação Estatística

Andressa Cerqueira, Rafael Izbicki, Thiago Rodrigo Ramos

2024-10-03

Table of contents

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).

Autores: Andressa Cerqueira, Rafael Izbicki, Thiago Rodrigo Ramos

1 Geração de Números Aleatórios e Aplicação em Problemas Estatísticos

1.1 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.

1.2 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.

1.3 Exemplo de Problema

Vamos resolver o problema de estimar o valor de (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 Pi.

Como fazer isso?

2 R

```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  : 3.184
```

```r
# 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")
```

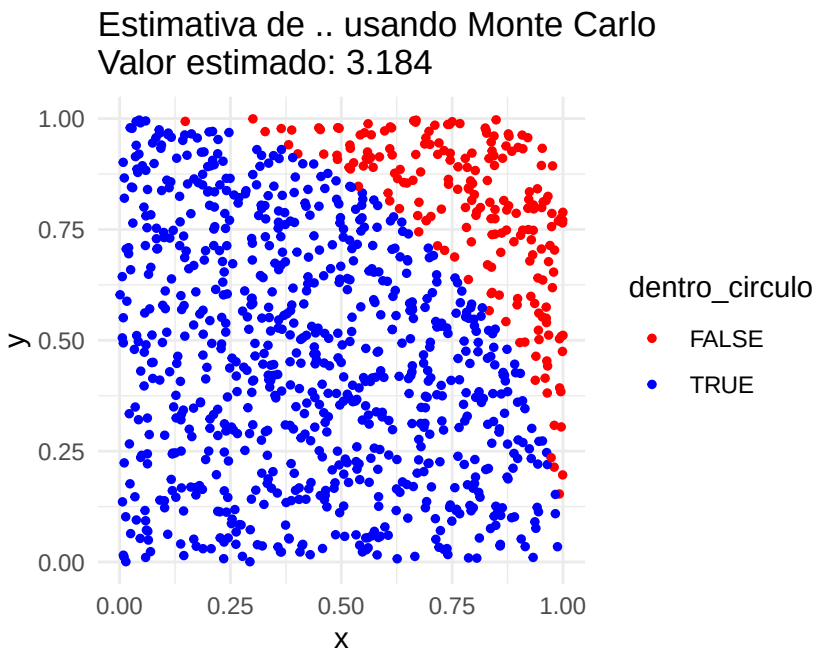
```
Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
```

```
conversion failure on 'Estimativa de  usando Monte Carlo' in 'mbcsToSbcs': dot
substituted for <cf>
```

```
Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'Estimativa de  usando Monte Carlo' in 'mbcsToSbcs': dot
substituted for <80>
```

```
Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'Estimativa de  usando Monte Carlo' in 'mbcsToSbcs': dot
substituted for <cf>
```

```
Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'Estimativa de  usando Monte Carlo' in 'mbcsToSbcs': dot
substituted for <80>
```

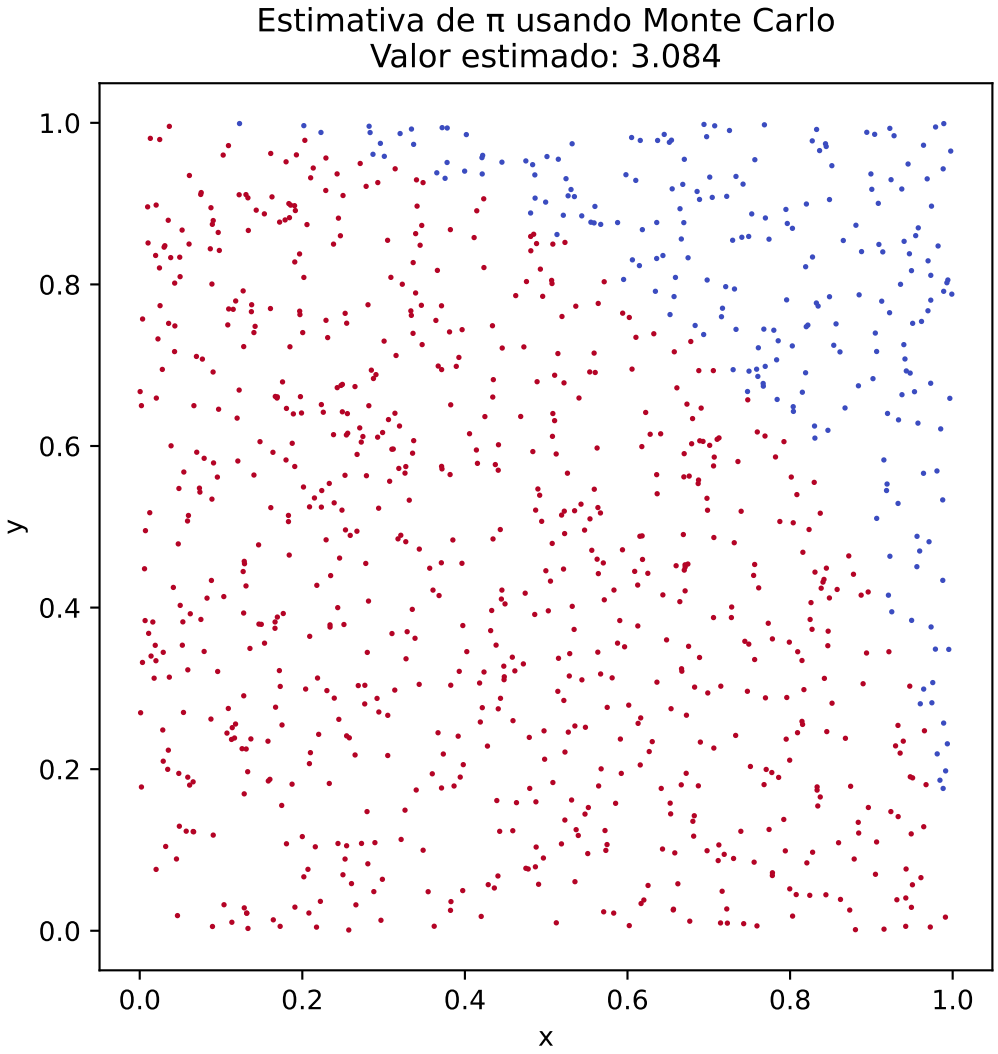


3 Python

```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  : 3.084
```

```python
# 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()
```



3.1 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.

4 R

```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 d

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

```
Soma das probabilidades: 1
```

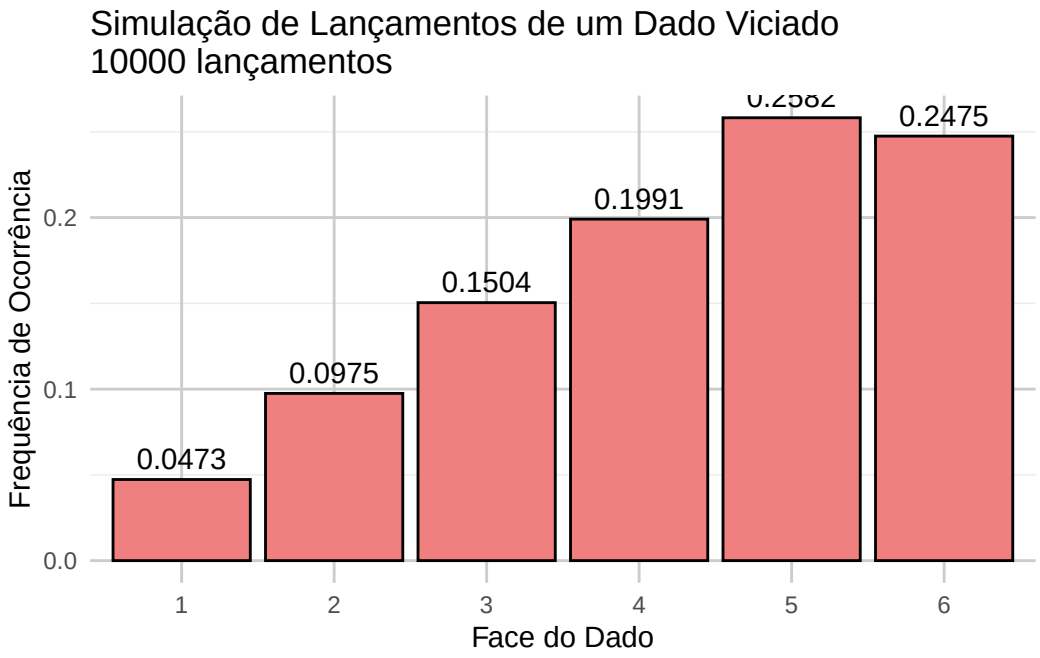
```r
# Simulando 10000 lançamentos de um dado viciado
n_lancamentos <- 10000
set.seed(123)  # Definindo seed para reprodutibilidade
resultados <- sample(faces, size = n_lancamentos, replace = TRUE, prob = probabilidades)

# 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:
```

```r
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
```

```r
# 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, " lançamento
  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"))
```



5 Python

```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
```

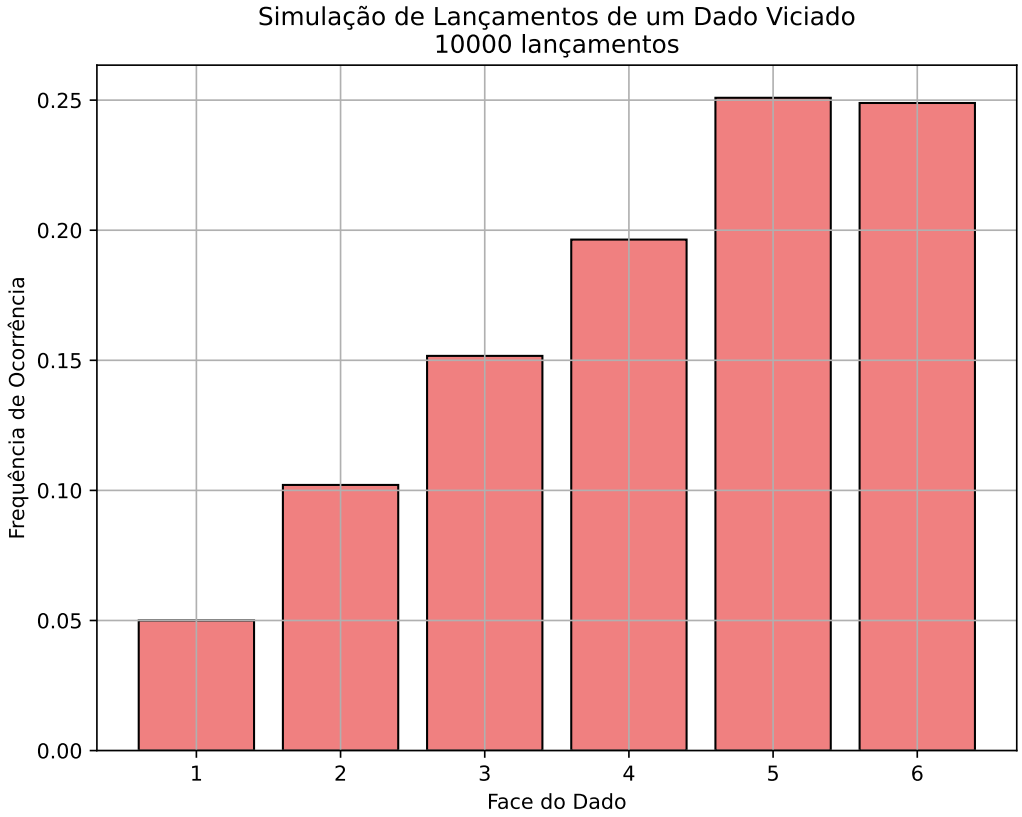
```python
# Simulando 10000 lançamentos de um dado viciado
n_lancamentos = 10000
resultados = np.random.choice(faces, size=n_lancamentos, p=probabilidades)

# 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:
```

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

```
Face 1: 0.05 vezes
Face 2: 0.1021 vezes
Face 3: 0.1517 vezes
Face 4: 0.1964 vezes
Face 5: 0.2509 vezes
Face 6: 0.2489 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()
```



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

6 R

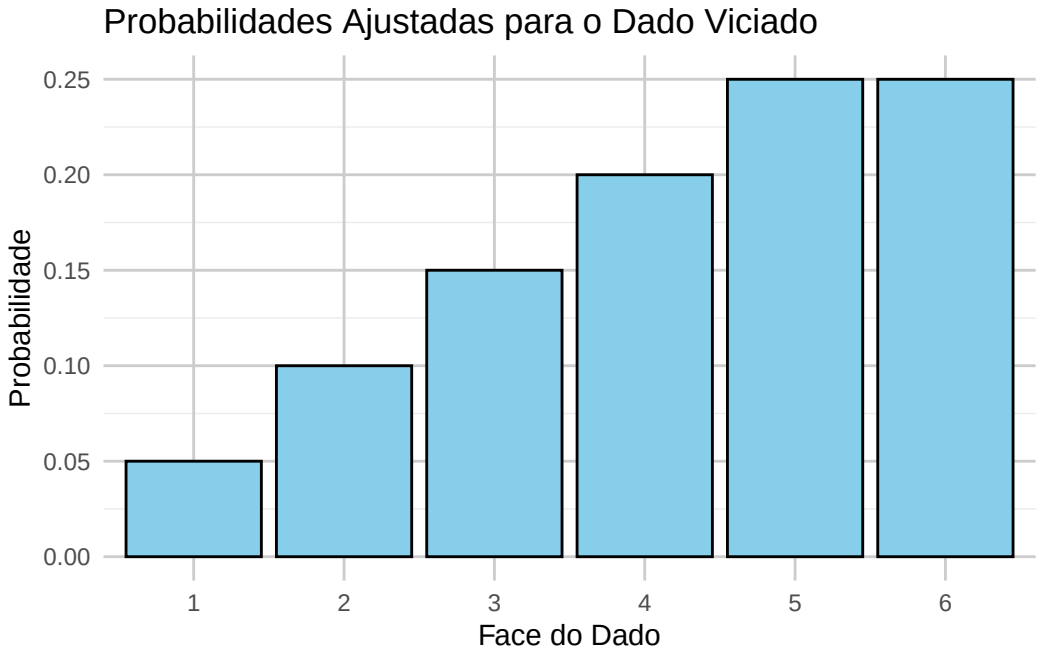
```r
# Definindo as faces do dado e as probabilidades associadas (não uniformes)
faces <- 1:6
probabilidades <- c(0.05, 0.1, 0.15, 0.2, 0.25, 0.25)  # Probabilidades associadas às faces d

# 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
    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:
```

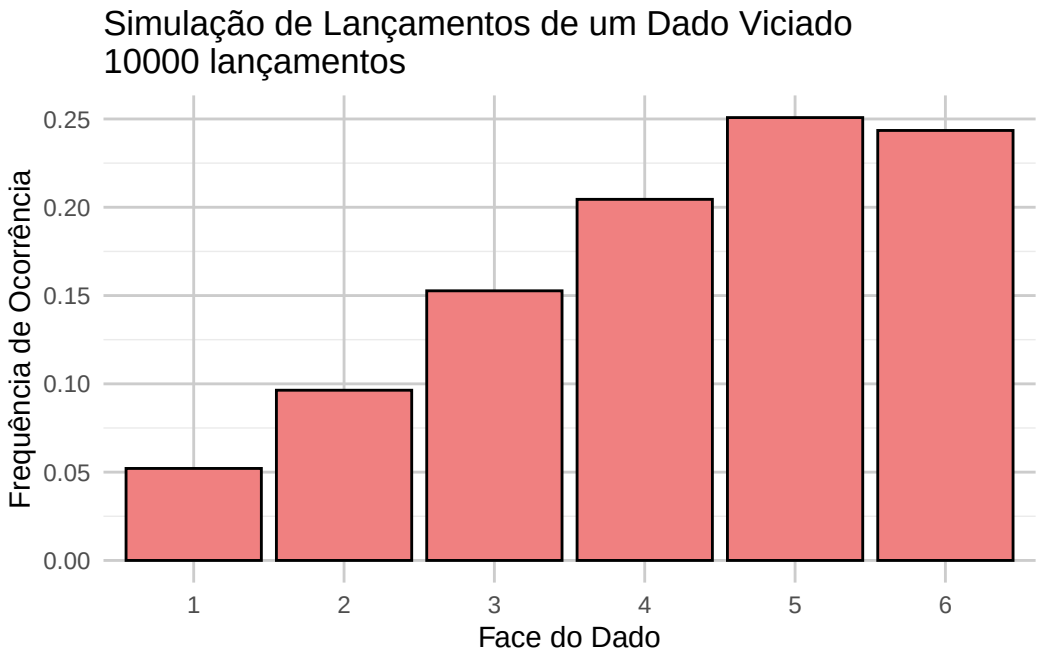
```r
for (i in seq_along(faces)) {
  cat("Face", faces[i], ":", frequencias[i], "vezes\n")
}
```

```
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
```

```r
# 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"))
```




```r
# 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, " lançament
  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
Frequência de Ocorrência
0.25
0.20
0.15
0.10
0.05
0.00
1
2
3
4
5
6
Face do Dado

7 Python

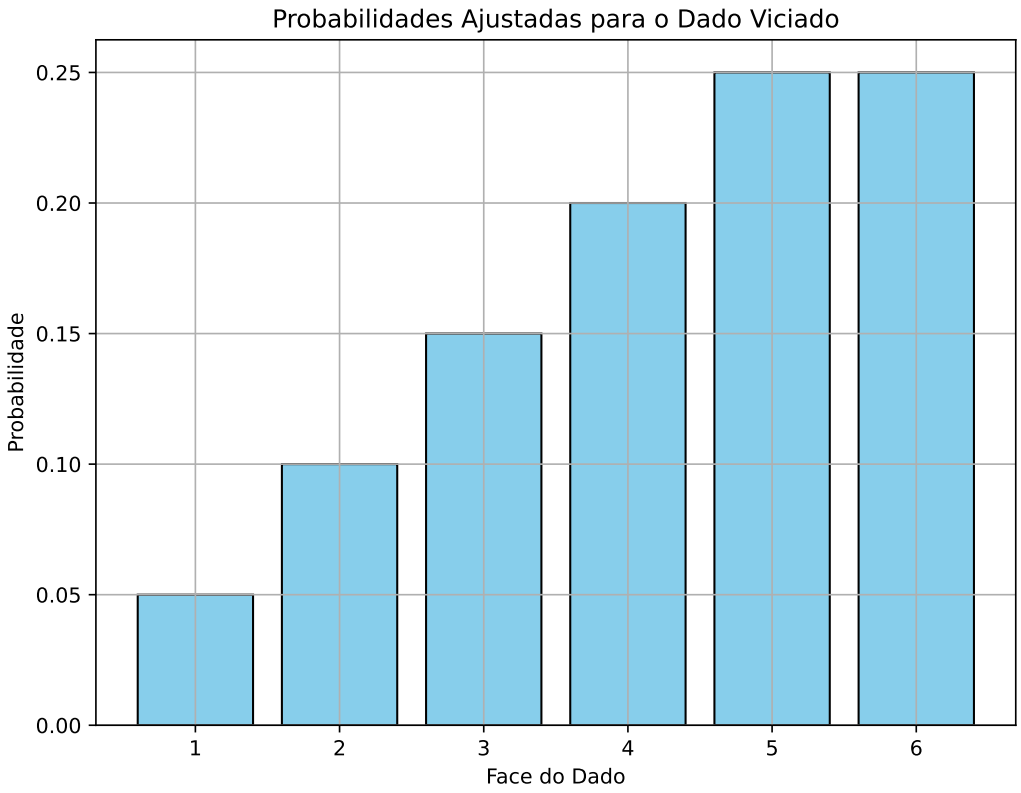
```python
import numpy as np
import matplotlib.pyplot as plt

# 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 interv

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

# 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:
```

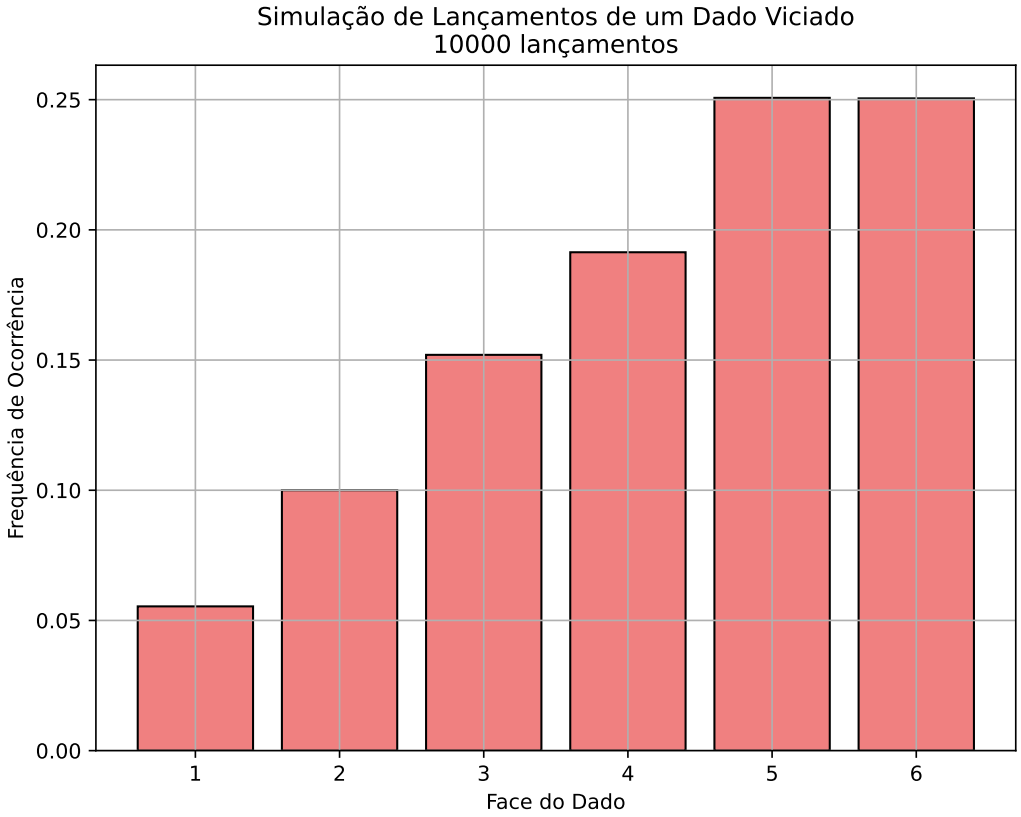
```python
for face, freq in zip(faces, frequencias):
    print(f"Face {face}: {freq} vezes")
```

```
Face 1: 0.0554 vezes
Face 2: 0.1 vezes
Face 3: 0.152 vezes
Face 4: 0.1914 vezes
Face 5: 0.2507 vezes
Face 6: 0.2505 vezes
```

```python
# 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?

8 Geração de 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.

8.1 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.

8.2 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.

8.2.1 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.

9 R

```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
```

10 Python

```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
```

10.1 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.

10.1.1 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 $[0, 1)$.

10.1.2 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.

10.1.3 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.

10.1.4 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.

11 R

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

```
Attaching package: 'gmp'

The following objects are masked from 'package:base':

    %*%, apply, crossprod, matrix, tcrossprod
```

```r
# 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
```

```r
# 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
```

12 Python

```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
```

```python
# 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)
```

12.0.1 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.

12.0.2 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.

12.0.3 Impacto dos Parâmetros:

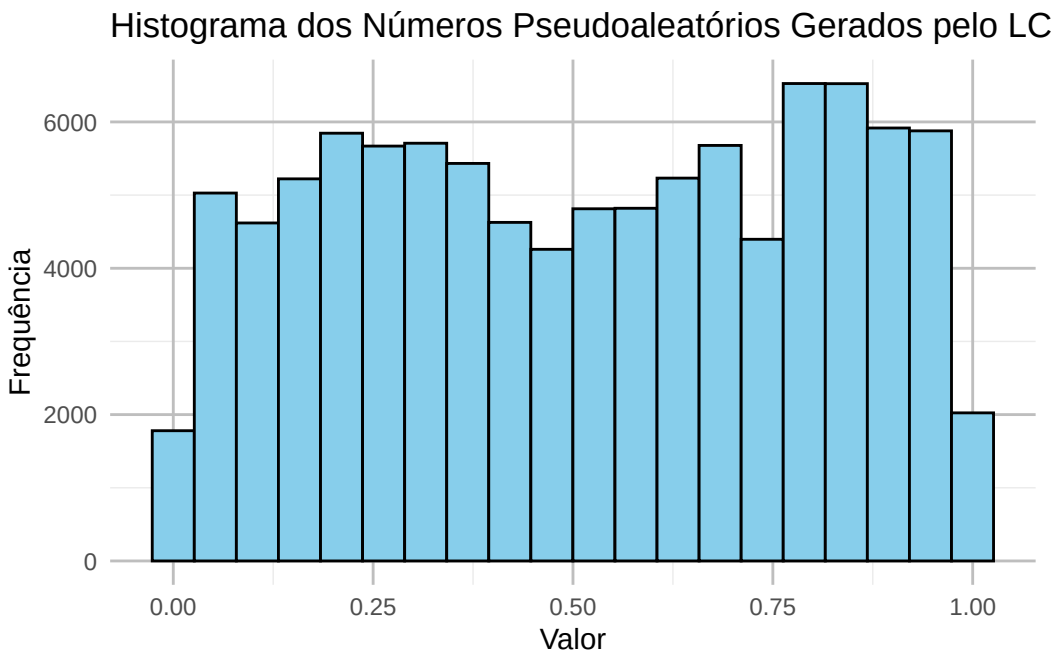
1. **Período da Sequência**:
   - O período é a quantidade de números gerados antes que a sequência comece a se repetir. Para obter o período máximo, os parâmetros a, c, m e a semente X_0 precisam ser cuidadosamente escolhidos.
   - Se os parâmetros não forem bons, o gerador pode produzir uma sequência 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.

13 R

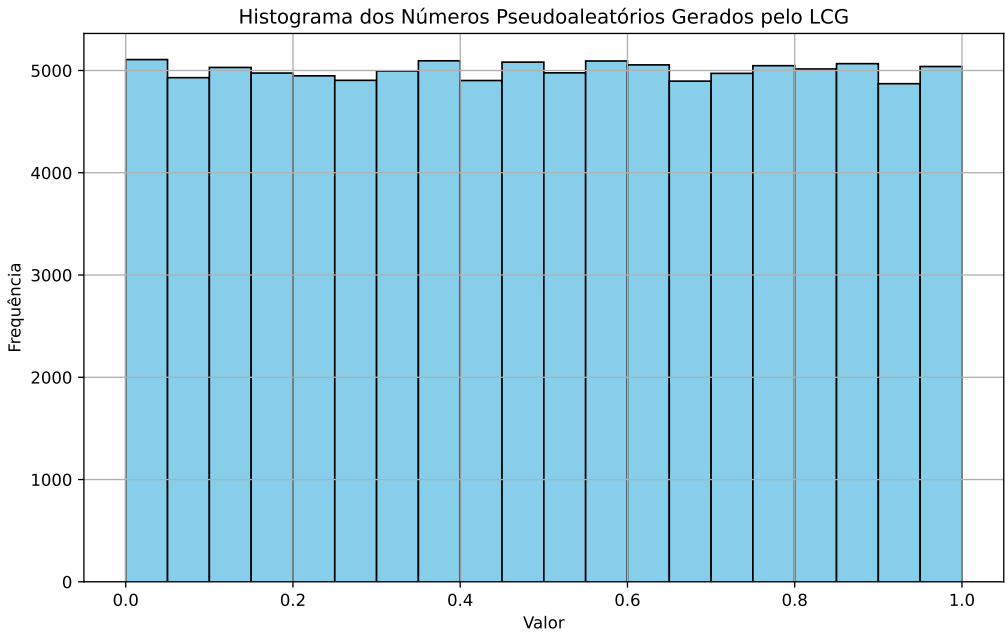
```r
# Carregando o pacote ggplot2
library(ggplot2)

# 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"))
```



14 Python

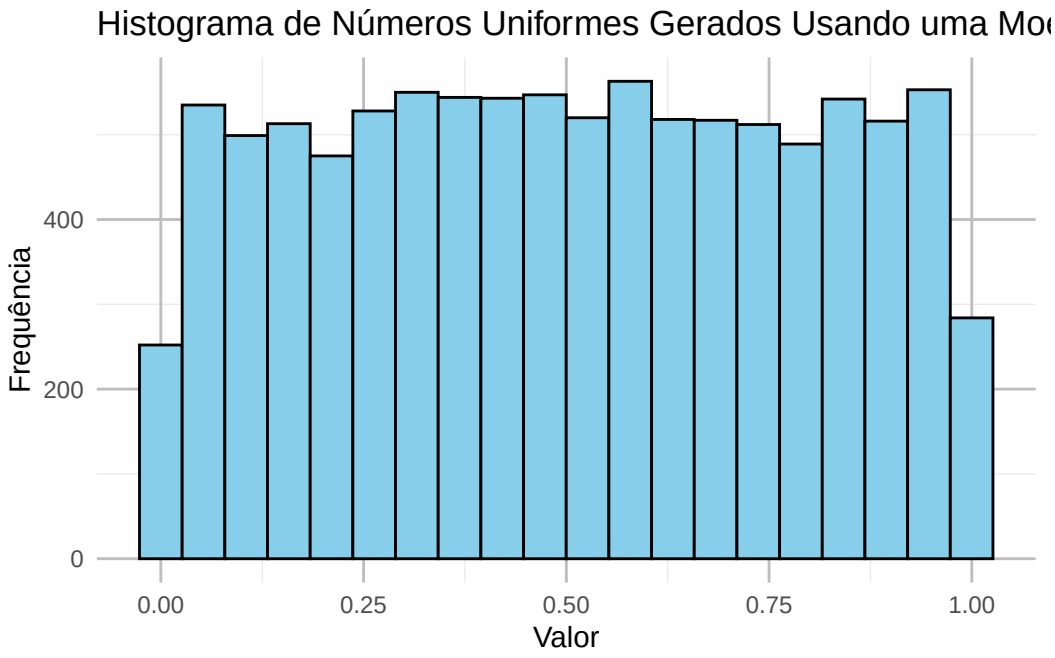
```python
import matplotlib.pyplot as plt

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()
```



14.1 Gerando números uniformes com uma moeda

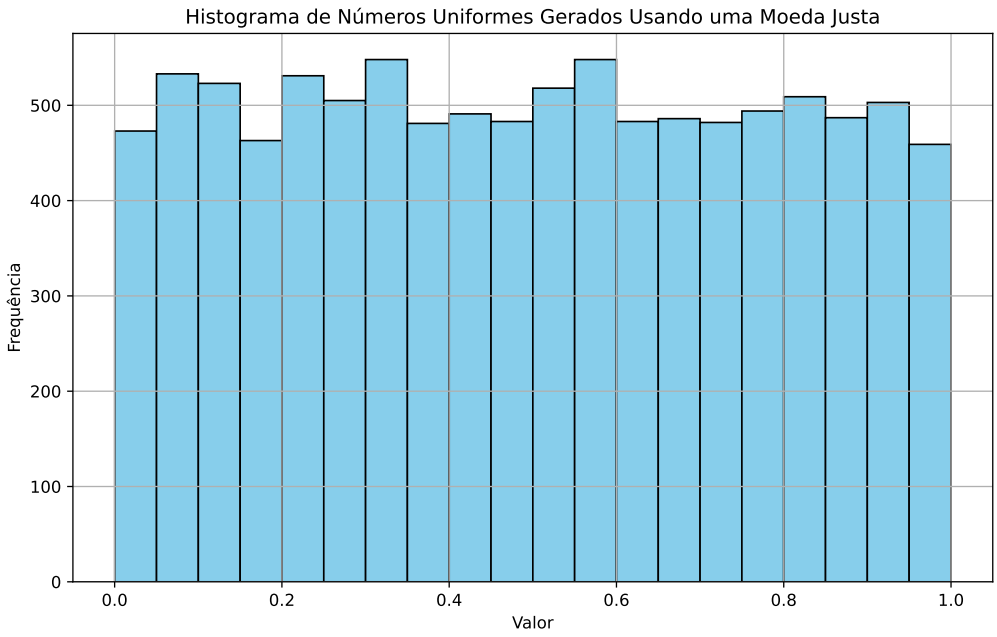
15 R

```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"))
```


Histograma de Números Uniformes Gerados Usando uma Mo
Frequência
400
200
0
0.00
0.25
0.50
0.75
1.00
Valor

16 Python

```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()
```


Histograma de Números Uniformes Gerados Usando uma Moeda Justa
Frequência
Valor
0
100
200
300
400
500
0.0
0.2
0.4
0.6
0.8
1.0

17 Geração de Variáveis Aleatórias Discretas Usando a Técnica da Inversão

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.

17.1 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:

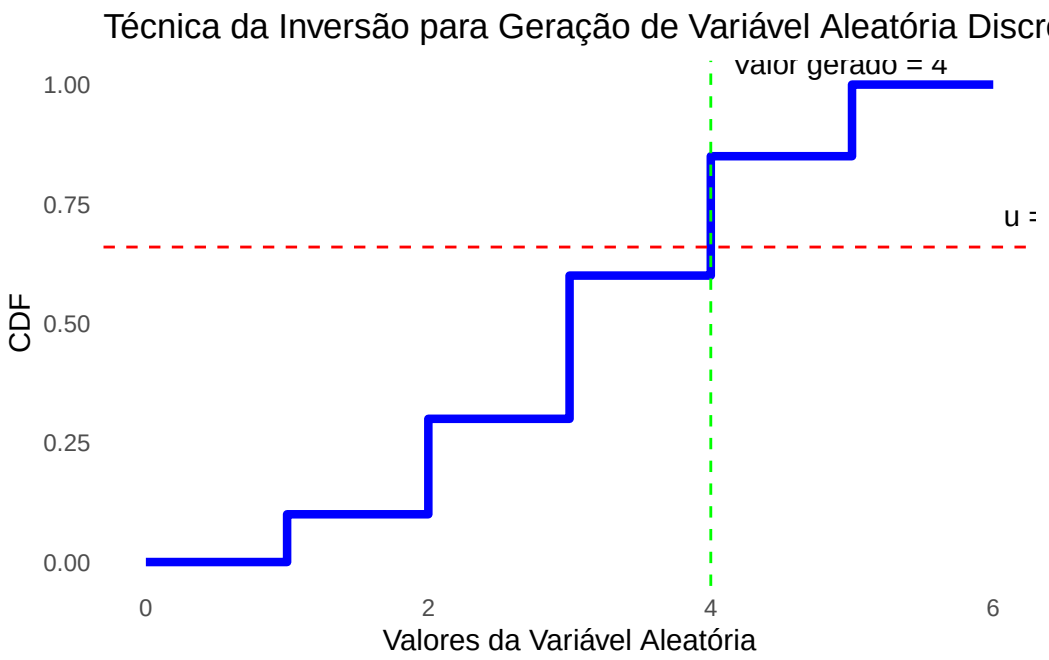
1. Gerar um número aleatório $u \in [0, 1)$.
2. Encontrar o menor valor x_i tal que $F(x_i) \geq u$.
3. Retornar x_i.

18 R

```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 c
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, vju
```

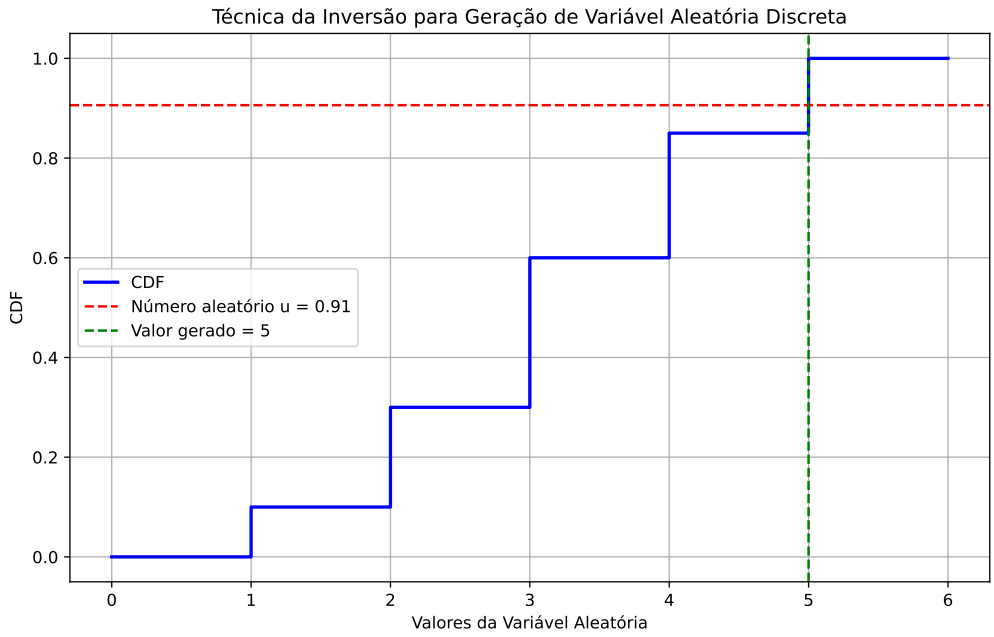
```
  annotate("text", x = valor_gerado, y = max(cdf), label = paste("Valor gerado =", valor_gera
  theme_minimal() +
  theme(panel.grid = element_blank())
```

```
Warning: Using `size` aesthetic for lines was deprecated in ggplot2 3.4.0.
i Please use `linewidth` instead.
```



19 Python

```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 c
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_gera
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()
```



19.1 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 $F^{-1}(p) = x$.

19.2 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:

19.2.1 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.

19.2.1.1 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.

19.2.1.2 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.

19.2.1.3 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:

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

19.2.1.4 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:

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

19.2.1.5 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**:

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

19.2.2 Conclusão

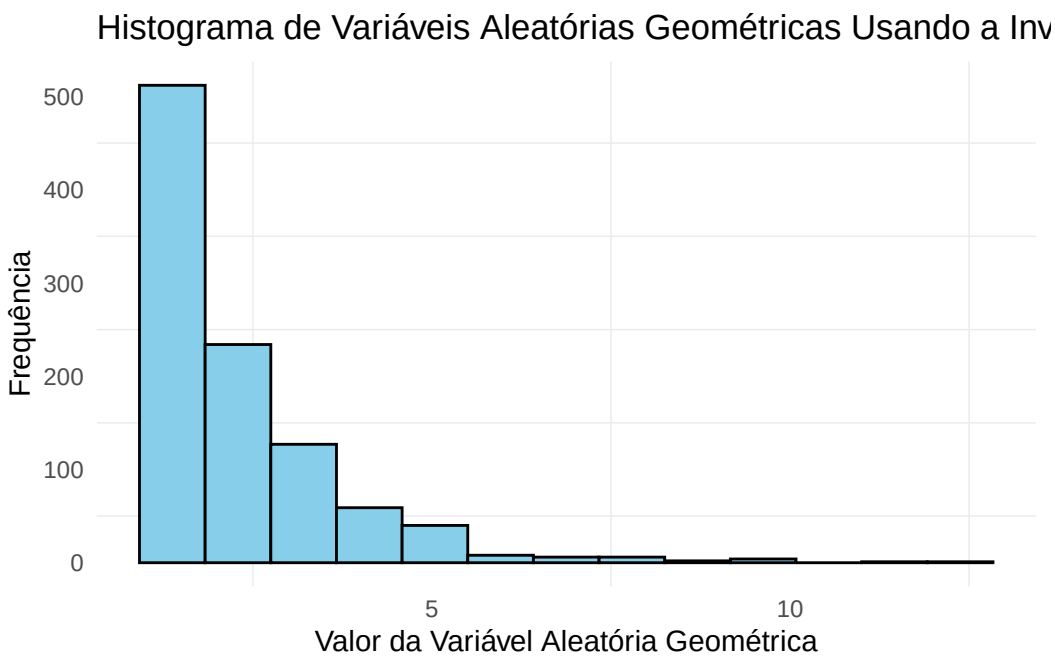
Portanto, a fórmula da inversa da CDF da distribuição geométrica é:

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

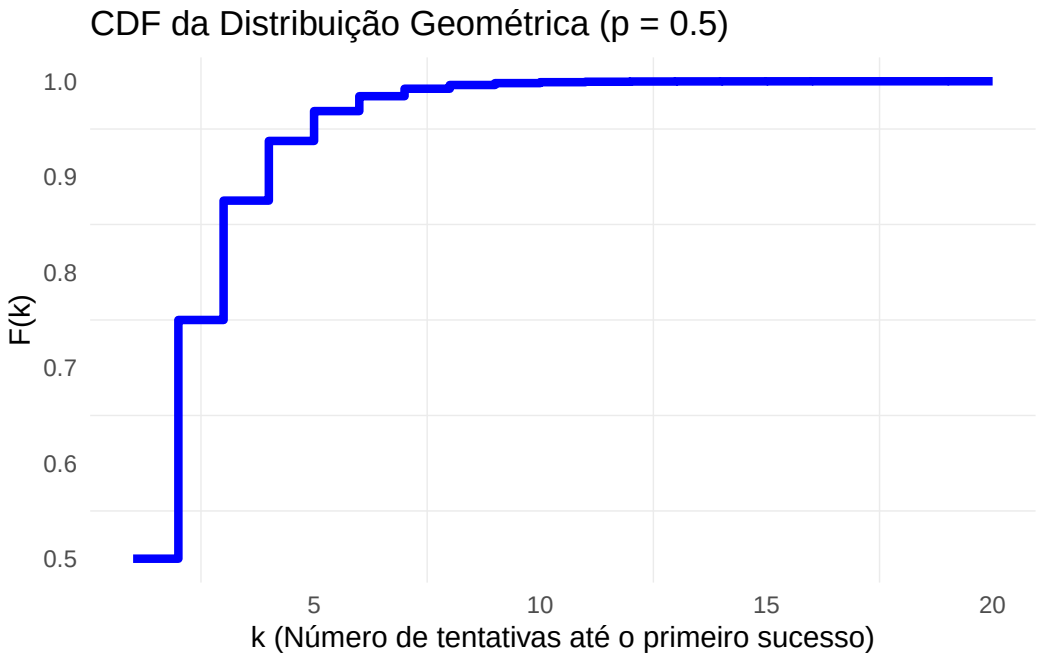
Esta fórmula nos permite gerar variáveis aleatórias com distribuição geométrica a partir de um número aleatório uniforme $u \in [0, 1)$.

20 R

```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) = 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 = 0
  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())
```

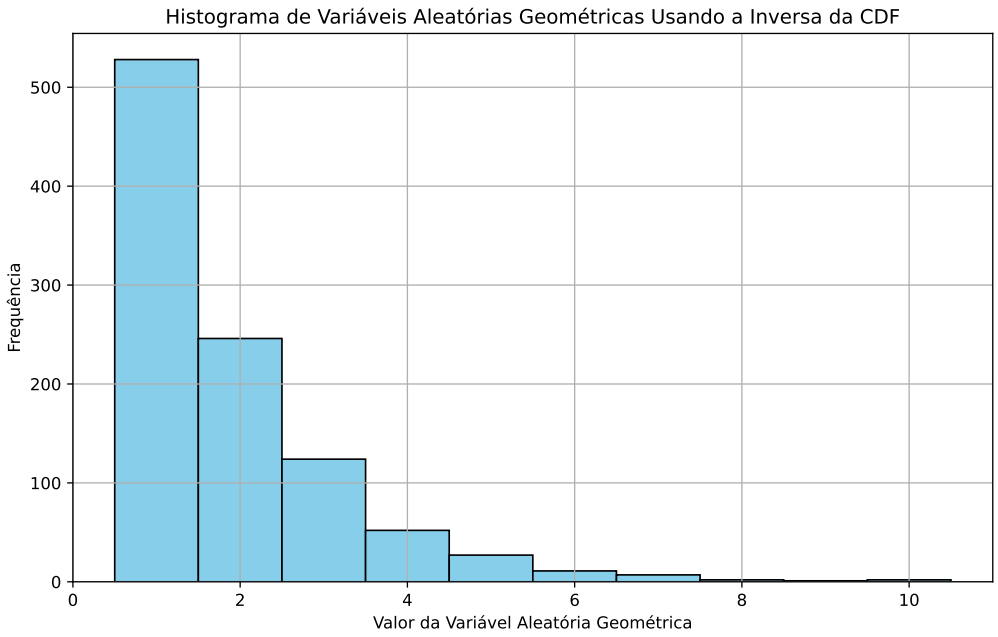



```r
# Função para calcular a CDF da distribuição geométrica
cdf_geometrica <- function(k, p) {
  return(1 - (1 - p)^k)
}

# 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())
```

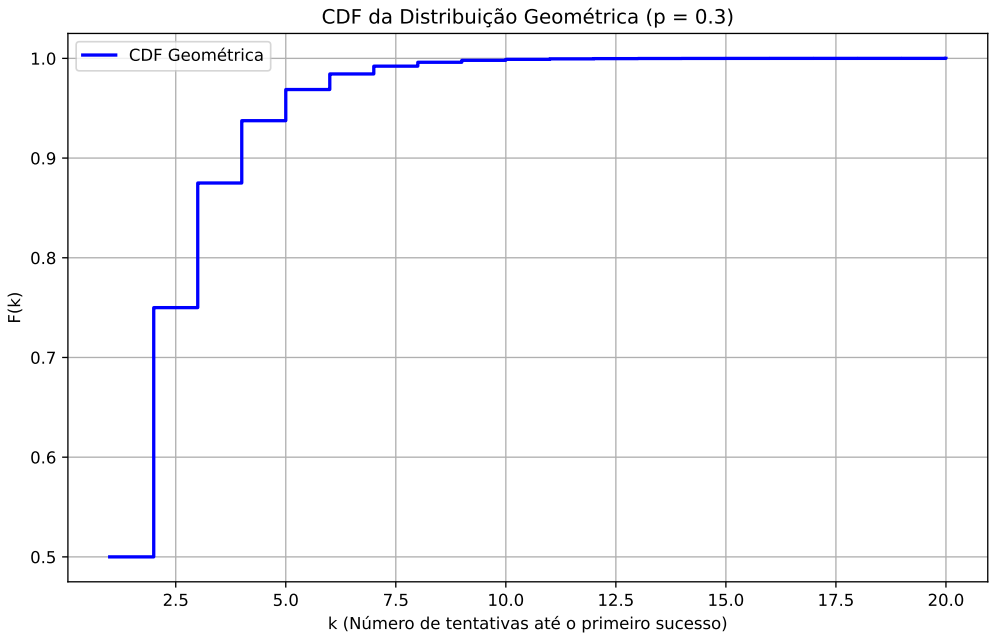

CDF da Distribuição Geométrica (p = 0.5)
1.0
0.9
0.8
0.7
0.6
0.5
F(k)
5
10
15
20
k (Número de tentativas até o primeiro sucesso)

21 Python

```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) = 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=
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()
```



21.1 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:

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

21.1.1 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 recalcular 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.

21.1.2 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.

21.1.3 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.

22 R

```r
# Função para gerar a inversa da CDF para a distribuição Poisson usando a técnica de inversão
inversa_cdf_poisson_recursiva <- 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_recursiva, 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") +
```

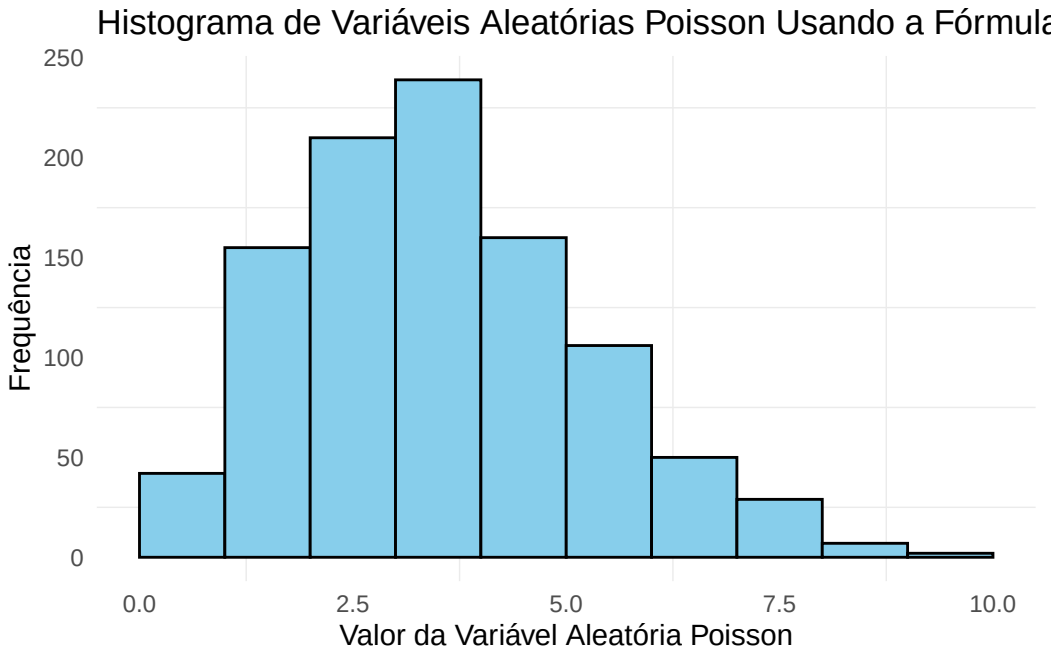
```
  theme_minimal() +
  theme(panel.grid.major = element_blank())
```

```
Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'Histograma de Variáveis Aleatórias Poisson Usando a
Fórmula Recursiva (  = 3)' in 'mbcsToSbcs': dot substituted for <ce>
```

```
Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'Histograma de Variáveis Aleatórias Poisson Usando a
Fórmula Recursiva (  = 3)' in 'mbcsToSbcs': dot substituted for <bb>
```

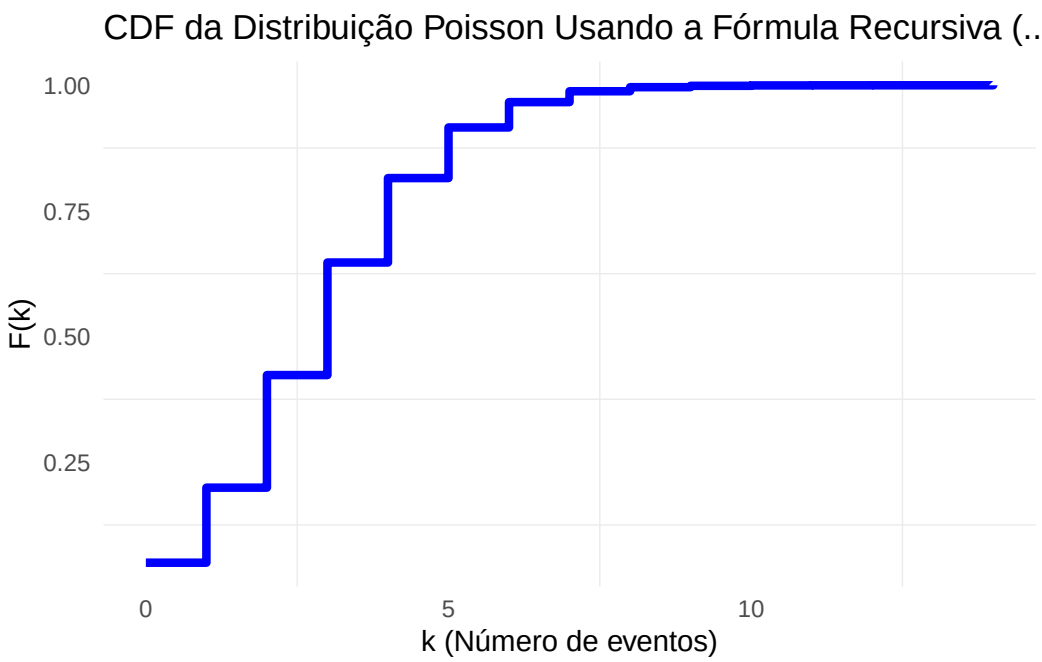
```
Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'Histograma de Variáveis Aleatórias Poisson Usando a
Fórmula Recursiva (  = 3)' in 'mbcsToSbcs': dot substituted for <ce>
```

```
Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'Histograma de Variáveis Aleatórias Poisson Usando a
Fórmula Recursiva (  = 3)' in 'mbcsToSbcs': dot substituted for <bb>
```



```r
# 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 (  = 3)",
       x = "k (Número de eventos)",
       y = "F(k)") +
  theme_minimal() +
  theme(panel.grid.major = element_blank())
```

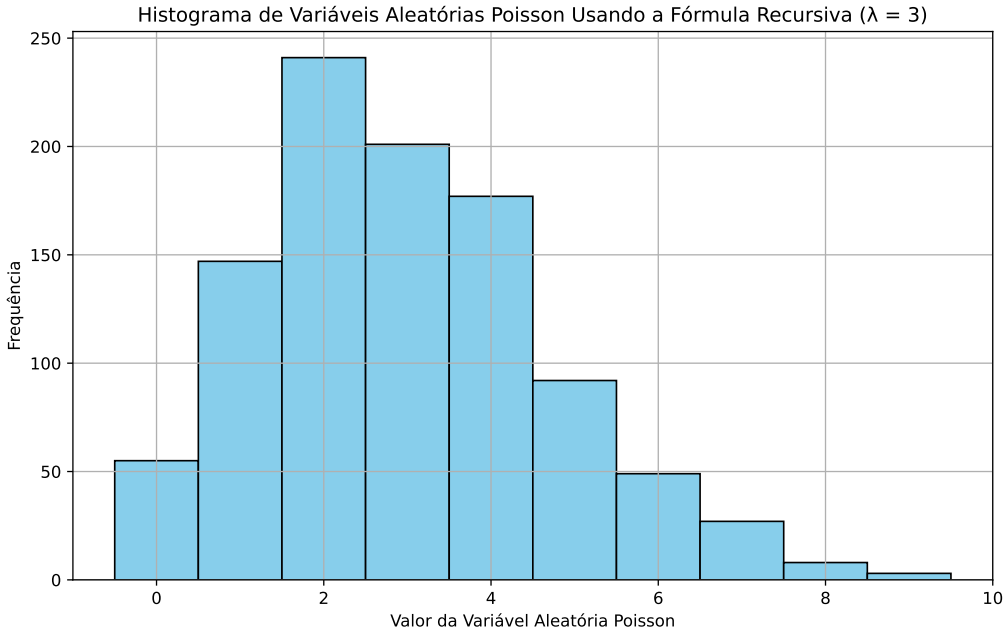
```
Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'CDF da Distribuição Poisson Usando a Fórmula Recursiva
(  = 3)' in 'mbcsToSbcs': dot substituted for <ce>

Warning in grid.Call(C_textBounds, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'CDF da Distribuição Poisson Usando a Fórmula Recursiva
(  = 3)' in 'mbcsToSbcs': dot substituted for <bb>

Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'CDF da Distribuição Poisson Usando a Fórmula Recursiva
(  = 3)' in 'mbcsToSbcs': dot substituted for <ce>

Warning in grid.Call.graphics(C_text, as.graphicsAnnot(x$label), x$x, x$y, :
conversion failure on 'CDF da Distribuição Poisson Usando a Fórmula Recursiva
(  = 3)' in 'mbcsToSbcs': dot substituted for <bb>
```


CDF da Distribuição Poisson Usando a Fórmula Recursiva (..
1.00
0.75
0.50
0.25
F(k)
0
5
10
k (Número de eventos)

23 Python

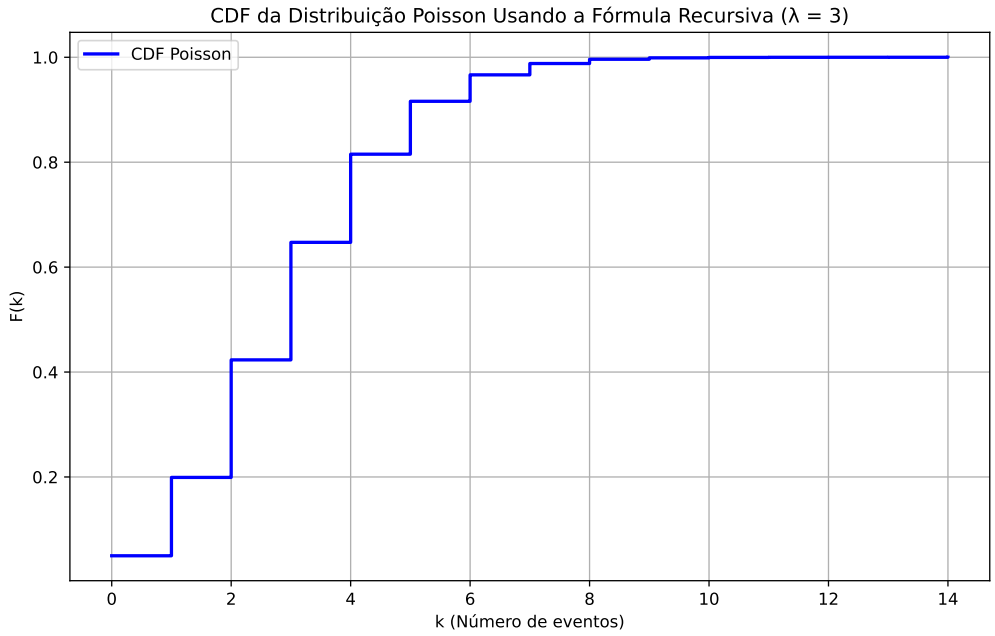
```python
import numpy as np
import matplotlib.pyplot as plt
import math

# Função para gerar a inversa da CDF para a distribuição Poisson usando a técnica de inversã
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='bla
plt.title('Histograma de Variáveis Aleatórias Poisson Usando a Fórmula Recursiva (  = 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 (  = 3)')
plt.xlabel('k (Número de eventos)')
```

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



23.1 Exercícios

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

(a) Escreva um pseudo-algoritmo para gerar um valor de X.

(b) Implemente uma função para gerar n valores de X.

(c) Compare a distribuição das frequências obtidas na amostra simulada com as probabilidades reais.

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

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

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-algorimo 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).

24 Geração de Variáveis Aleatórias Contínuas Usando a Técnica da Inversão e Transformações

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

As **Variáveis Aleatórias Contínuas** 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 é considerada contínua se sua função de distribuição acumulada (f.d.a.) puder ser expressa como:

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

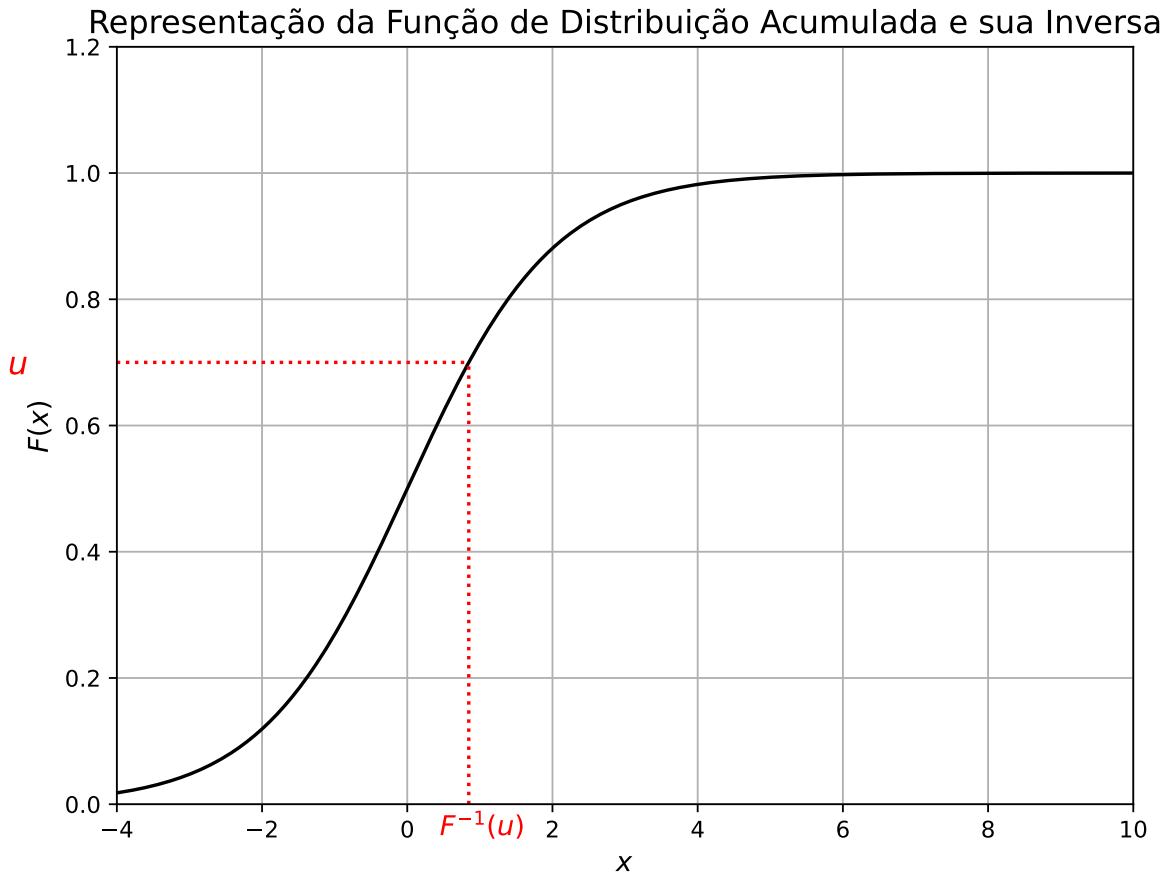
para uma função integrável $f : \mathbb{R} \to [0, \infty)$.

24.1 Função Inversa

Sabemos que $F : \mathbb{R} \to [0, 1]$, e, portanto, podemos definir a função inversa $F^{-1} : [0, 1] \to \mathbb{R}$. A seguir, mostramos como isso funciona.

```
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 inv

# 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, linestyles='dotted', colors='red')
plt.vlines(x_value, 0, u_value, linestyles='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)
plt.xlim(-4, 10)
plt.grid(True)

# Exibir o gráfico
plt.show()
```



24.2 Método da Inversão

Para gerar valores de uma variável aleatória contínua X, usamos o **método da inversão**, que segue a 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:

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

O método da inversão consiste em:

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

24.3 Exemplo 1

Seja X uma v.a. com:

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

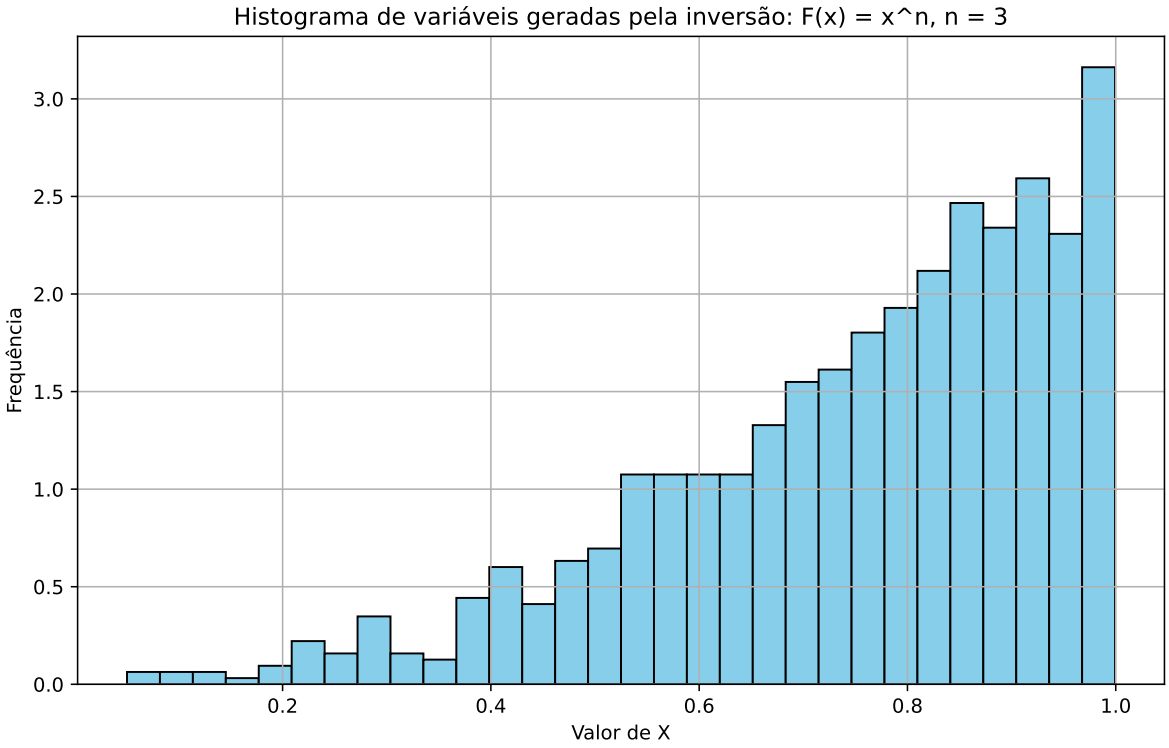
A função inversa é:

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

Portanto, o pseudo-algoritmo para gerar X é:

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

```
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()
```



24.3.1 Exemplo 2

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

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

A função inversa é:

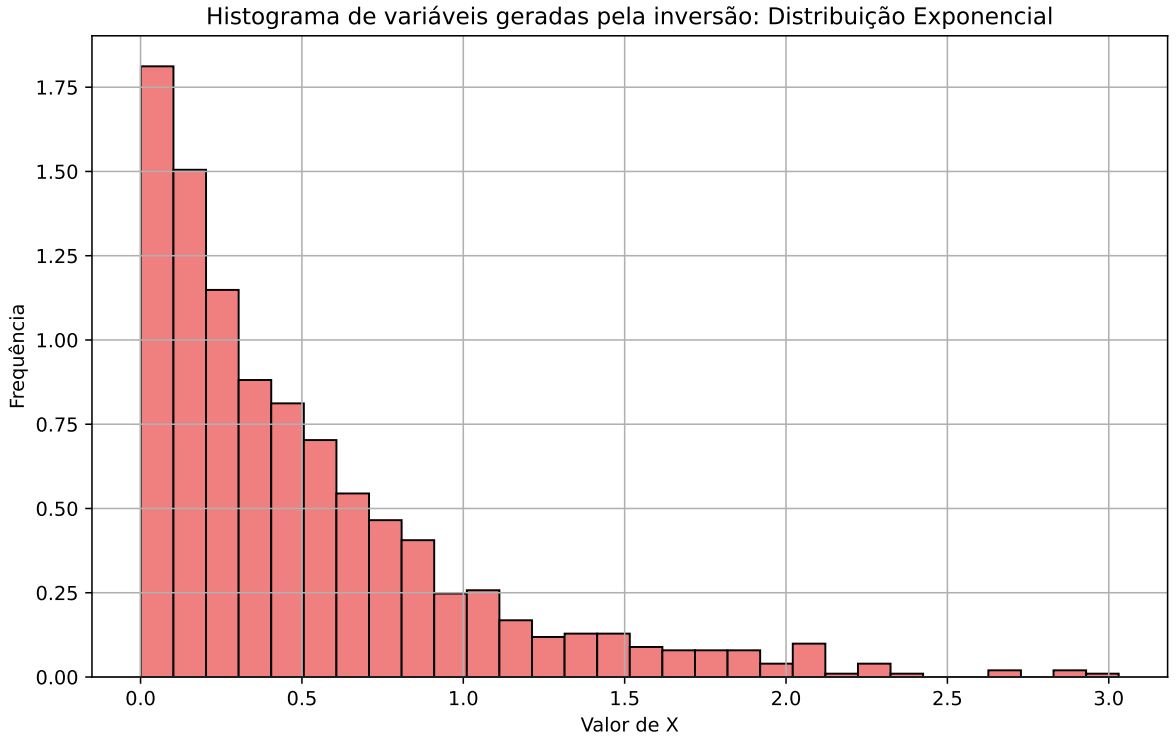
$$u = F(x) = 1 - e^{-\lambda x} \implies x = -\frac{\log(1-u)}{\lambda}.$$

O pseudo-algoritmo para gerar X é:

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

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

# Parâmetro lambda da distribuição exponencial
lambd = 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) / lambd

# 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()
```



24.4 Simulação de transformações de variáveis aleatórias

Agora que já sabemos como simular uma variável aleatória X, o próximo passo é gerar valores de uma transformação dessa variável, ou seja, $g(X)$.

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

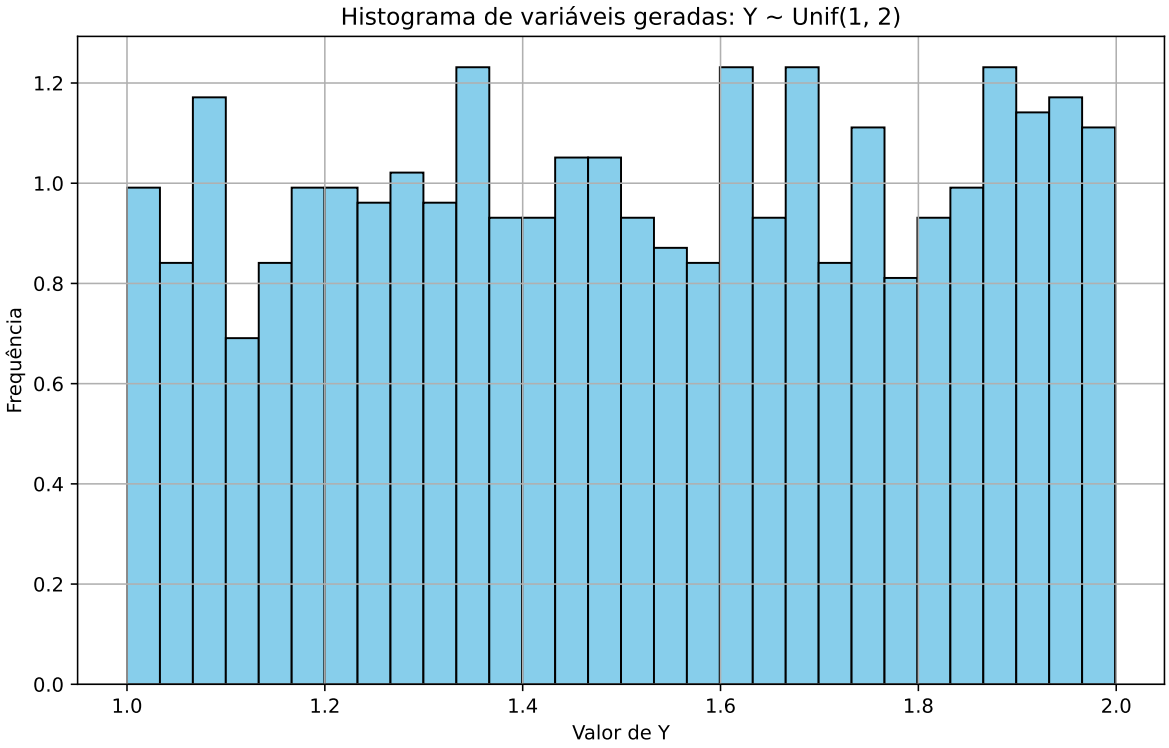
Para gerar valores de $Y \sim Unif(1, 2)$, usamos o fato de que Y é uma simples transformação de $U \sim Unif(0, 1)$. A relação é:

$$Y = U + 1.$$

O pseudo-algoritmo é:

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

```
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()
```



24.4.2 Exemplo 2: Simulando $Y \sim Gamma(n, \lambda)$

Para gerar valores de $Y \sim \Gamma(n, \lambda)$, somamos n variáveis aleatórias $X_1, \ldots, X_n$, onde cada $X_i \sim Exp(\lambda)$. A relação é:

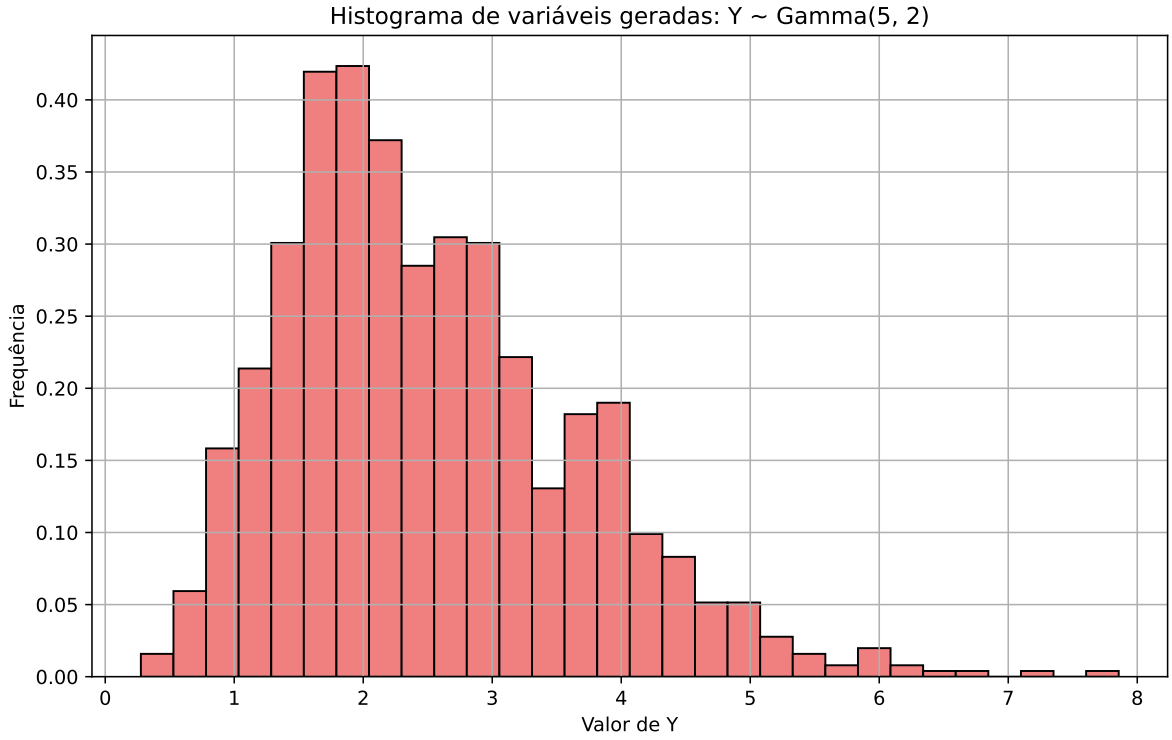
$$Y = X_1 + X_2 + \cdots + X_n.$$

O pseudo-algoritmo é:

1. Gere $U_1, \ldots, U_n \sim Unif(0, 1)$ independentemente.
2. Calcule $X_i = -\frac{\log(1-U_i)}{\lambda}$ para $i = 1, \ldots, n$.
3. Calcule $Y = X_1 + X_2 + \cdots + X_n$.

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



# Definindo parâmetros
n = 5  # número de somas
lambd = 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) / lambd

# Somando os valores de X para obter Y ~ 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 ~ Gamma({n}, {lambd})')
plt.xlabel('Valor de Y')
plt.ylabel('Frequência')
plt.grid(True)
plt.show()
```



References

Ross, Sheldon M. 2006. *Simulation, Fourth Edition.* USA: Academic Press, Inc.