Estatística Computacional em Python

Daniel Furtado Ferreira

2024-08-11

Table of contents

Prefácio

O Livro Estatística Computacional em Python é um resumo e adaptação do Livro Estatística Computacional em Java, publicado pela Editora UFLA. Este Livro tem por objetivo primeiro o aprimoramento do autor em Quarto e, segundo, o aprendizado de Python.

Adaptaremos inicialmente a apostila Estatística Computacional em R para abrigar os códigos em Python. Posteriormente, ampliaremos o conteúdo desta primeira Edição do Livro.

Para aprender mais sobre Livros do Quarto visite o site https://quarto.org/docs/books.

Nestas notas de aula tivemos a intenção de abordar o tema de estatística computacional que é tão importante para a comunidade científica e principalmente para os estudantes dos cursos de pós-graduação em estatística. Podemos afirmar sem medo de errar que a estatística computacional se tornou e é hoje em dia uma das principais áreas da estatística. Além do mais, os conhecimentos desta área podem ser e, frequentemente, são utilizados em outras áreas da estatística, da engenharia e da física. A inferência Bayesiana é um destes exemplos típicos em que geralmente utilizamos uma abordagem computacional. Quando pensamos nestas notas de aulas tivemos muitas dúvidas do que tratar e como abordar cada tópico escolhido. Assim, optamos por escrever algo que propiciasse ao leitor ir além de um simples receituário, mas que, no entanto, não o fizesse perder em um emaranhado de demonstrações. Por outro lado buscamos apresentar os modelos e os métodos de uma forma bastante abrangente e não restritiva.

Uma outra motivação que nos conduziu e nos encorajou a desenvolver este projeto, foi a nossa experiência pessoal em pesquisas com a estatística computacional. Também fizemos isso pensando no benefício pessoal, não podemos negar, que isso nos traria ao entrarmos em contato direto com a vasta publicação existente neste ramo da estatística. Não temos, todavia, a intenção de estudarmos todos os assuntos e nem mesmo pretendemos para um determinado tópico esgotar todas as possibilidades. Pelo contrário, esperamos que estas notas sejam uma introdução a estatística computacional e que sirvam de motivação para que os alunos dos cursos de graduação em estatística possam se adentrar ainda mais nessa área.

Estas notas são baseadas em um livro que escrevemos sobre a estatística computacional utilizando a linguagem Java. A adaptação para o Python de algumas das rotinas implementadas neste livro foi uma tarefa bastante prazerosa e reveladora. Aproveitamos esta oportunidade para desvendar um pouco dos inúmeros segredos que este poderoso programa possui e descobrir um pouco sobre seu enorme potencial e também, por que não dizer, de suas fraquezas. Nessa primeira versão não esperamos perfeição, mas estamos completamente cientes que muitas falhas devem existir e esperamos contar com a colaboração dos leitores para saná-las. Estamos iniciando a primeira versão não revisada e não ampliada, utilizando ainda o Quarto e trocando o R para o Python. Essas notas serão constantemente atualizadas na internet. Esperamos que este manuscrito venha contribuir para o crescimento profissional dos estudantes de nosso programa de pós-graduação em Estatística e Experimentação Agropecuária, além de estudantes de outros programas da UFLA ou de outras instituições. Dessa forma nosso objetivo terá sido atingido.

Chapter 1

Introdução ao Python

O programa Python foi escolhido para ministrar este curso por uma série de razões. Além de ser um programa livre, no sentido de possuir livre distribuição e código fonte aberto, pode ser utilizado nas plataformas `Windows` e `Linux`. Além do mais, o Python possui grande versatilidade no sentido de possuir inúmeros pacotes já prontos e nos possibilitar criar novas rotinas e funções. O `PyPi` é o repositório oficial do Python onde todos os pacotes são armazenados. Você pode pensar nele como um Github para os pacotes do Python. O Python foi criado pelo holandês Guido van Rossum para ser uma linguagem de programação simples e legível, além de ser muito produtiva. O Python evoluiu e se tornou em uma linguagem muito atrativa e uma das principais escolhas para aplicações de desenvolvimento web, análise de dados e inteligência artificial, entre outras. Por ser genuinamente um programa orientado por objeto nos possibilita programar com muita eficiência e versatilidade, embora apresente algumas mudanças em sua implementação em relação a outras linguagens orientadas por objetos. Outro aspecto que é bastante atrativo no Python refere-se ao fato de o mesmo receber contribuições de pesquisadores de todo o mundo na forma de pacotes. Essa é uma característica que faz com que haja grande desenvolvimento do programa em relativamente curtos espaços de tempo e que nos possibilita encontrar soluções para quase todos os problemas com os quais nos deparamos em situações reais. Para os problemas que não conseguimos encontrar soluções, o ambiente de programação Python nos possibilita criar nossas próprias soluções.

Nestas notas de aulas pretendemos apresentar os conceitos básicos da estatística computacional de uma forma bastante simples. Inicialmente obteremos nossas próprias soluções para um determinado método ou técnica e em um segundo momento mostraremos que podemos ter a mesma solução pronta do Python quando esta estiver disponível. Particularmente neste capítulo vamos apresentar algumas características do ambiente e da linguagem para implementarmos nossas soluções. Nosso curso não pretende dar soluções avançadas e de eficiência máxima para os problemas que abordaremos, mas propiciar aos alunos um primeiro contato com a linguagem Python e com os problemas básicos da estatística computacional.

A desvantagem é que o Python não é um programa fácil de aprender. Alguns esforços iniciais são necessários até que consigamos obter algum benefício. Não temos a intenção de apresentar neste curso os recursos do Python para análises de modelos lineares de posto completo ou incompleto, de modelos não-lineares, de modelos lineares generalizados ou de gráficos. Eventualmente poderemos utilizar algumas destas funções como um passo intermediário da solução do problema que estaremos focando. Este material será construído com uma breve e simplificada abordagem teórica do tópico e associará exemplificações práticas dos recursos de programação Python para resolver algum problema formulado, em casos particulares da teoria estudada.

Este material é apenas uma primeira versão que deverá ter muitos defeitos. Assim, o leitor que encontrá-los ou tiver uma melhor solução para o problema poderá contribuir enviando um e-mail para danielff@ufla.br.

Visite minha homepage https://des.ufla.br/~danielff/.

1.1 Introdução aos Comandos e Objetos do Python

No Python os objetos podem ser de diferentes tipos ou estruturas, tais como os números (`int`, `float` e complexos), `boolean`, `string`, `list`, `tuple`, `set`, `dictionary`, `functions` (objeto que encapsula códigos), `dataframes` e muitos outros. Vamos descrever de forma sucinta e gradativa alguns destes objetos e comandos básicos do Python.

As instruções do Python podem ser escritas em um editor de texto e digitadas no terminal do programa. Para usarmos o Python, inicialmente instalamos uma distribuição do Programa. Recomendamos a versão atual (no momento do lançamento do livro) que pode ser baixada no site https://www.python.org/ do link https://www.python.org/ftp/python/3.13.0/python-3.13.0-amd64.exe. Para digitarmos os códigos, recomendamos que seja baixado o program livre **Positron** no site https://posit.co/ usando o link https://github.com/posit-dev/positron/releases/download/2024.11.0-140/Positron-2024.11.0-140-Setup.exe.

Veja um print do Positron:

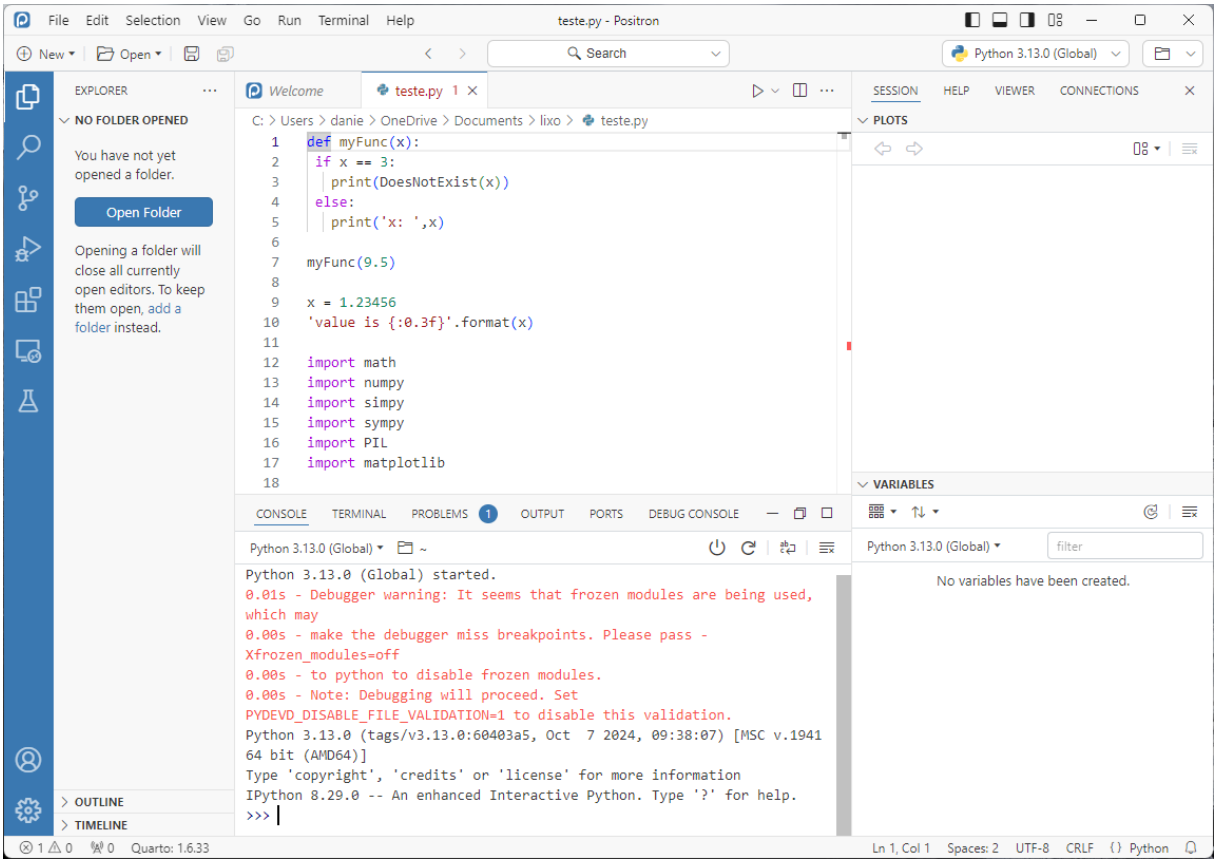


Figure 1.1: Positron: para códigos em Python ou R

Uma vez instalado, podemos digitar os códigos e com **Ctrl Enter** executamos os códigos linha por linha ou um bloco de linhas marcadas. É importante instalarmos algumas bibliotecas básicas, caso elas já não estejam instaladas. A seguir, temos um código Python para esse propósito.

```
pip install numpy
pip install sympy
pip install PIL
```

```
pip install jupyter
pip install matplotlib
```

Em seguida devemos importar as libraries que precisarmos. No `script` a seguir consideramos a importação de todas as libraries. A library `math` não precisa ser instalada, pois já vem com as distribuições do Python.

```
import math
import numpy
import sympy
import PIL
import jupyter
import matplotlib
```

1.1.1 Operações Aritméticas Básicas

Podemos usar o Python como uma calculadora, temos o seguinte programa, em que cada linha física do editor tem um comando Python específico. Podemos separar em uma mesma linha vários comandos com ponto e vírgula.

```
# Programa ilustrativo de operações elementares
# em Python
1 + 2 + 3     # soma dos 3 primeiros inteiros
3**10 - 1 + 8
6 / 5 + 0.5**4
```

```
6
```

```
59056
```

```
1.2625
```

Podemos observar que o símbolo `#` é usado para inserirmos comentários no código Python e o operador `/` faz divisão usando operadores reais. Para divisão de inteiros, podemos usar `//`, assim, `6 // 5` retorna 1 e `6 / 5` retorna `1,2`. O resto da divisão por inteiro é obtido pelo operador `%`. Assim, `6 % 5` retorna `1`. Temos também que o operador `**` é a função potência, ou seja, por exemplo, 3^{10} é `3**10` em Python.

O Python também pode realizar operações com números complexos, que no caso, são representados por $a + bj$, em que a é a parte real do número e bj, a parte imaginária, sendo $j = \sqrt{-1}$. O j é representado por i nos livros de matemática e de outras áreas. O programa a seguir ilustra uma operação com números complexos dada por $(6 - 4i)^2$. Assim, temos

```
(6-4j)**2
```

```
(20-48j)
```

Apresentamos a seguir um `script` que faz uso da library `math`, cujo primeiro comando foi para importá-la, o penúltimo para obter o valor de π e o último comando calculou $\sqrt{2}$. Também fizemos mais uma operação com números complexos.

```
import math
2 + 4 + 5.6
2 / 3 - 4
(3-4j)*(3+4j)
math.pi
math.sqrt(2)
```

```
11.6
```

```
-3.3333333333333335
```

```
(25+0j)
```

```
3.141592653589793
```

```
1.4142135623730951
```

As bibliotecas `numpy` e `sympy` são para diversos cálculos matemáticos, sendo que a última efetua cálculos simbólicos.

```
import sympy
import numpy
numpy.set_printoptions(legacy='1.25')
sympy.sin(sympy.pi/5)
numpy.sin(numpy.pi/5)
type(1.5 + 2.1j) # tipo do objeto
```

$$\sqrt{\frac{5}{8} - \frac{\sqrt{5}}{8}}$$

```
0.5877852522924731
```

```
complex
```

Podemos realizar uma operação matemática básica como algumas das anteriormente apresentadas ou até mesmo, mais complexa e armazenar o valor em uma variável, digamos `x`. Essa variável pode ser usada para outras operações matemáticas e até simbólicas, se usarmo o `sympy`. Veja Knuth [1984] para discussão sobre programação simbólica. O `script` a seguir ilustra alguns casos deste procedimento.

```
x = sympy.symbols('y')
z = (1+x**2)**2
sympy.simplify(z)
y = 2.3 + 6.7**2
r = y**2 + 1 / 2**3
print('r =', r, 'y = ', y, 'e', 'z =', z)
```

$$\left(y^2 + 1\right)^2$$

```
r = 2227.0211 y =  47.19 e z = (y**2 + 1)**2
```

Assim, atribuímos dados às variáveis Python. O Python diferencia maiúsculas de minúsculas e nomes como `X` e `x` são diferentes. No Python as variáveis são ponteiros (pointers). Comandos como $x = 2$ cria um objeto x e atribuí (armazena) o valor 2 nele em outras linguagens, mas no Python, há um objeto inteiro 2 e x é um ponteiro, apontando para ele. Veja as consequências disso a seguir, sendo que o comando `\n` realiza uma quebra de linha. Não há maiores implicações em objetos escalares como este, mas quando se trata, por exemplo, de listas, nosso próximo objeto, a questão já é bem diferente.

```
x = z = b = 1
b = 7
print('x is pointing to', x,
      '\nz is pointing to', z, '\nb is pointing to', b)
```

```
x is pointing to 1
z is pointing to 1
b is pointing to 7
```

Para lidarmos com funções de números complexos a library `cmath` deve ser importada e as funções trigonométricas de números complexos podem ser usadas com base nesta biblioteca e não na library `math`,

que é designada para números reais (`float`). Veja o `script` ilustrativo a seguir.

```
import cmath
(cmath.cos(0.1 - 0.4j) + cmath.sin(0.2 + 0.6j))**2.5
```

```
(1.0143106793360148+2.4164569923716543j)
```

1.2 Variáveis Booleanas

Variáveis booleanas em Python recebem os valores `True` e `False` apenas. Várias operações de comparações como `==`, `>=`, `<=`, `&` (and), `|` (`or`) e `!` (negação - `not`) podem ser usadas para este tipo de objeto, as variáveis booleanas. Veja um simples exemplo disso. Posteriormente, voltaremos a falar destes operadores de variáveis booleanas.

```
x = True
x
y = False
y
x != y # ou
z =  not(y)
x == z
```

```
True
```

```
False
```

```
True
```

```
True
```

1.3 Strings

As strings (variáveis texto) são um importante tipo de objeto Python. Uma vez que temos um objeto definido, os métodos e funções estão disponíveis para serem usados. As strings são denotadas por `str` em Python. Veja alguns exemplos, em que as strings foram atribuídas ou não a objetos (variáveis).

```
'Esta é uma string'
mensagem = 'Universidade Federal '
type(mensagem)
mensagem
UFLA = mensagem + 'de Lavras, MG.'
UFLA
```

```
'Esta é uma string'
```

```
str
```

```
'Universidade Federal '
```

```
'Universidade Federal de Lavras, MG.'
```

Alguns métodos que podemos usar com as strings são ilustrados a seguir, entre muitos outros, mostrando o poder da linguagem orientada por objetos.

```
UFLA.capitalize()
UFLA.lower()
UFLA.upper()
UFLA
```

```
'Universidade federal de lavras, mg.'
```

```
'universidade federal de lavras, mg.'
```

```
'UNIVERSIDADE FEDERAL DE LAVRAS, MG.'
```

```
'Universidade Federal de Lavras, MG.'
```

Estes métodos atuam no objeto, mas, como ficou claro no exemplo anterior, não mudam o conteúdo do objeto. Podemos realizar operações com strings, como ilustrado a seguir. O método `str.format` possibilita formatar `strings`, como, por exemplo, incluir `substrings` nos campos marcados com `{}`.

```
UFLA * 2
nome = 'Nome: {}, Sobrenome: {}'
nome.format('Daniel', 'Furtado Ferreira')
```

```
'Universidade Federal de Lavras, MG.Universidade Federal de Lavras, MG.'
```

```
'Nome: Daniel, Sobrenome: Furtado Ferreira'
```

Podemos usar o Python para interagir com o usuário, solicitando a entrada de dados (`strings` no caso) com o comando `input`. Veja os exemplos a seguir.

```
nome = input('entre com seu primeiro nome: ')
print(nome + ' foi aprovado!')
x = int(input('entre com um valor inteiro: '))
  # transforma o str em inteiro: int
x # se o número de entrada não for int, resulta em erro
```

Se o usuário entrar com `Daniel`, o resultado será `Daniel foi aprovado!`. Esta versão de `Markdown` ainda não suporta interatividade com o usuário. Portanto, o comando `input` não foi avaliado na saída deste `script`. No segundo comando, se o usuário entrar com um número não inteiro, haverá uma mensagem de erro do Python. Existem opções para lidar com erros deste tipo e de outras causas também.

1.4 Listas, Tuplas, Conjuntos e Dicionários

Vamos abordar cada um destes objetos separadamente. Vamos começar pelas listas.

1.4.1 Listas

As listas, `lists` são os primeiros blocos de construção para lidarmos como manipulação de dados. As listas são vetores cujo primeiro elemento inicia-se no `0`, mas cujos elementos de cada célula pode ser diferentes tipos mistos, desde inteiros, booleanos, reais, complexos, strings, caracteres, conjuntos, tuplas e outras listas. As listas fazem parte do quarteto `list`, `tuple`, `set` e `dictionary`. A biblioteca `numpy` fornece ferramentas adicionais para lidarmos com grande coleções de dados.

```
x = [1, 2, 3, 4]
x
y = [1, 'Estat', 3.5, 4+5j]
y
type(x)
type(y)
y[3]
```

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

```
[1, 'Estat', 3.5, (4+5j)]
```

```
list
```

```
list
```

```
(4+5j)
```

A variável `x` é uma lista de inteiros com 4 elementos, que são indexados por 0, 1, 2, 3. Assim, $x[1]$ aponta para o valor 2 e $x[0]$ para o valor 1. A variável `y` também é uma lista com 4 elementos de diferentes tipo, sendo $y[0]$ um inteiro, $y[1]$ uma `string` $y[2]$ um `float` e $y[3]$, um número complexo. Para criar a lista, simplesmente utilizamos as chaves `[]`, com cada elemento da lista separado por uma vírgula. É possível criar uma lista com elementos com valores repetidos e eles serão identificados como sendo diferentes, pois a lista respeita as ordens de entradas dos valores e preserva a ordem. Podemos verificar se um elemento pertence a lista com o comando `in`, como mostra o `script` a seguir, entre outros exemplos.

```
[2, 7] == [7, 2]
[5, 7] == [5, 7, 7]
x
2 in x
7 in x
y
4+5j in y
'Daniel' in y
```

```
False
```

```
False
```

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

```
True
```

```
False
```

```
[1, 'Estat', 3.5, (4+5j)]
```

```
True
```

```
False
```

Podemos, como foi feito com as `strings` realizar algumas operações aritméticas com as listas, como mostra o exemplo do seguinte `script`.

```
x+y
x+[[0,1],'teste',[1,0]]
y*2
y[0] # primeiro elemento da lista
y[-1] # último elemento da lista
# numeração -1,-2,-3,-4 para o índice
# acessa as posições, 3,2,1,0,
# respectivamente da lista y
```

```
[1, 2, 3, 4, 1, 'Estat', 3.5, (4+5j)]
```

```
[1, 2, 3, 4, [0, 1], 'teste', [1, 0]]
```

```
[1, 'Estat', 3.5, (4+5j), 1, 'Estat', 3.5, (4+5j)]
```

```
1
```

```
(4+5j)
```

Vejamos agora o problema dos ponteiros, por meio do exemplo do `script` apresentado na sequência.

```
x = z = b = [1,2,3]
b[1] = 7
print('x is pointing to', x,
      '\nz is pointing to', z, '\nb is pointing to', b)
# todos os objetos foram alterados e não só b
# pois eles apontam para a mesma lista [1,2,3]
```

```
x is pointing to [1, 7, 3]
z is pointing to [1, 7, 3]
b is pointing to [1, 7, 3]
```

Observamos que se `x`, `z` e `b` apontarem para o mesmo objeto, então se alterarmos o valor `b[1]` de `2` para `7`, então todos os três objetos serão alterados na posição `1`, que corresponde ao segundo valor da lista, pois ela se inicia na posição `0`. Entretanto, se em vez de `b[1] = 7` tivéssemos usado a atribuição `b = [7,9]`, então os vetores `x` e `z` não seriam alterados, com a nova atribuição do vetor `b`. Nos exemplos anteriores, vimos também que os elementos de uma lista são acessados pelo seu índice que varia de `0` a `n-1`, sendo `n` o seu tamanho. Assim, a lista `x=[1,2,3,4]` tem seus elementos `x[0]` igual a `1`, `x[1]` igual a `2`, `x[2]`igual a `3` e `x[3]` igual a `4`. Também podemos variar o índice de `-1` a `-n`, sendo que `-1` significa a última posição da lista, ou seja, a posição `n-1`, `-2` corresponde a posição `n-2` e assim por diante até `-n`, que corresponde a posição `0` da lista.

As listas são objetos e como tais podemos utilizar alguns métodos associados a eles. As listas são mutáveis e dinâmicas (podemos alterar seus elementos), são ordenadas (cada elemento da lista possui uma ordem definida na sua criação) e permitem elementos repetidos. Para usarmos um método ou uma função deveremos considerar a diferença entre eles. Embora todos métodos sejam funções em Python, nem toda função é um método. As funções recebem os objetos como entradas e não os modifica e os métodos agem nos objetos. A seguir apresentamos uma relação de alguns métodos ou funções associados às listas:

- `sort()`: ordena a lista em ordem crescente.
- `append()`: adiciona um elemento ao final da lista.
- `extend`: adiciona múltiplos elementos à lista.
- `index()`: usado para encontrar o índice de um elemento na lista.
- `max(list)`: retorna o valor máximo de uma lista.
- `min(list)`: retorna o valor mínimo de uma lista.
- `list(tuple)`: transforma uma `tuple` numa lista.
- `len(list)`: retorna o tamanho da lista (número de elementos).
- `filter(fun,list)`: filtra uma lista usando uma função `fun` Python.

Vamos ilustrar alguns destes métodos com exemplos particulares. Vamos considerar uma lista e aplicarmos o método `sort()` para ordenarmos os seus valores. Neste exemplo a seguir, vamos ver a diferença de um método e de uma função, observando como o método modifica o objeto que o chamou. Neste caso, a chamada de um método é dada pelo nome da lista (objeto) seguida de um ponto e do nome do método: `lista.met()`.

```
x = [7.4, 5.8, 9.3, 3.2]
x # objeto x original
x.sort()
x # objeto x modificado pelo método sort() ordenado
```

```
[7.4, 5.8, 9.3, 3.2]
```

```
[3.2, 5.8, 7.4, 9.3]
```

Para o método `append` temos o seguinte `script`, que acrescentou o mês de Abril ao final de uma lista com os três primeiros meses do ano.

```
mes = ['Janeiro', 'Fevereiro', 'Março']
mes.append('Abril')
print(mes)
```

```
['Janeiro', 'Fevereiro', 'Março', 'Abril']
```

O método `extend` é aplicado na lista `x` anterior e acrescenta mais dois elementos ao final da mesma.

```
x
x.extend([1.6, 11.6])
x
x.sort() # ordena x, pois não estava mais em ordem
x
```

```
[3.2, 5.8, 7.4, 9.3]
```

```
[3.2, 5.8, 7.4, 9.3, 1.6, 11.6]
```

```
[1.6, 3.2, 5.8, 7.4, 9.3, 11.6]
```

O `index()` é um método para encontra o índice de um elemento na lista. Se o elemento procurado não estiver na lista, o Python mostrará uma mensagem de erro.

```
i = x.index(7.4)
i # lembre-se que a lista começa no 0 e não no 1
```

```
3
```

Já as funções `len`, `max()` e `min()` atuam no objeto, passado como entrada da função, mas não o modificam. Veja o exemplo na lista `x` dos exemplos anteriores o efeito destas duas funções.

```
len(x)     # tamanho da lista x
len(mes)   # tamanho da lista mes
b = max(x)
a = min(x)
a
b
mes
x # x e mes não modificados
```

```
6
```

```
4
```

```
1.6
```

```
11.6
```

```
['Janeiro', 'Fevereiro', 'Março', 'Abril']
```

```
[1.6, 3.2, 5.8, 7.4, 9.3, 11.6]
```

Para ilustrar a uso da função `filter()` vamos considerar uma função que retorna `True` ou `False` para uma certa condição de interesse. Por exemplo, se quiséssemos saber quais números dos seis elementos da lista `x` possui resto da divisão por `2` menor que `1,5`. Esse resultado é obtido com a comparação `a % 2 <= 1.5`, que irá retornar verdadeiro ou falso para o número representado por `a`. Só que devemos fazer isso para todos os elementos da lista `x` ou de outra lista qualquer. Devemos criar uma função para receber

cada elemento da lista e verificar a condição, retornando `True` ou `False` e passar pela função `filter()` para realizar a iteração nos elementos da lista `x` ou na lista de interesse. Vamos criar um primeira função no exemplo a seguir e em seguida aplicar a função `filter()`. O tipo de objeto retornado desta função é `filter`, logo, tem de ser transformado em lista antes de imprimir.

```
def resto(a):
    if ((a % 2) <= 1.5):
        return True
    else:
        return False
# aplicar a função filter
x_filtrado = filter(resto, x)
print(list(x_filtrado))
```

```
[3.2, 7.4, 9.3]
```

Podemos usar a função `list(objeto)` para construir uma lista a partir deste objeto, como ocorreu com o objeto `x_filtrado`do `script` anterior. Então a função `list()` é um construtor de listas. Observe que o operador `%` retorna o resto da divisão por inteiro, que no caso, foi por `2`. A função resto retorna verdadeiro ou falso, de acordo com a condição do resto da divisão de `a` por `2`. Para definir uma função necessariamente usamos o comando `def` seguido pelo nome da função (`resto`) com o argumento (`a`). Depois vem o corpo da função, que deve ter indentação obrigatória. Não há separação com `{}` ou `[]` ou outros caracteres para separar o corpo da função. Esta separação é feita apenas com uso das indentações apropriadas. Falaremos posteriormente de função com mais detalhes.

Podemos aproveitar algumas funções prontas das listas para calcularmos algumas quantidades de interesse, como, por exemplo, a soma dos seus elementos. Para isso, poderíamos usar uma estrutura de repetição, como o `for` que veremos posteriormente e na medida que o *loop* se adianta, vamos atualizando a soma. Porém podemos usar a função `sum()`. Um *loop* `for` é executado como ***bytecode*** Python interpretado, enquanto a função `sum()` é escrita puramente na linguagem C, portanto, bem mais rápida e eficiente. Veja o código a seguir para ilustrarmos a soma de todos os valores do vetor `x`. Falaremos das estruturas condicionais e de repetições posteriormente.

```
soma = sum(x)
soma
# média
soma / len(x)
```

```
38.9
```

```
6.483333333333333
```

Outros métodos como o `count()` (conta o número de ocorrências de um dado valor), `reverse()` (ordena a lista em ordem reversa a ordem original), `clear()` (limpa todos os dados da lista), `copy()` (copia todos os dados da lista), `insert()` (insere um elemento em uma posição específica da lista) e `pop()` (remove um elemento em uma posição específica) como apresentado no scrip a seguir.

```
L = [1, 2, 3, 5, 7, 2, 4.5]
L.count(2)
L.count(2.1)
L1 = L.copy()
L.reverse()
L
L.clear()
L
L1
```

```
L1.insert(1, 3) # insere o valor 3 na posição 1
L1
L1.pop(1) # retira o elemento 3 da posição 1
L1
```

```
2
```

```
0
```

```
[4.5, 2, 7, 5, 3, 2, 1]
```

```
[]
```

```
[1, 2, 3, 5, 7, 2, 4.5]
```

```
[1, 3, 2, 3, 5, 7, 2, 4.5]
```

```
3
```

```
[1, 2, 3, 5, 7, 2, 4.5]
```

Podemos construir uma matriz, haja vista que Python não possui um objeto matricial, usando uma lista. Se criarmos uma lista de `n`componentes, com cada um dos componentes tendo `m` componentes, teremos uma matriz $n \times m$. Vejamos no `script` a seguir a construção da seguinte matriz:

$$A = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}. \tag{1.1}$$

O `script` correspondente a matriz definida em (Equation 1.1) é:

```
A = [[1,4],[2,5],[3,6]]
print('A = ',A)
```

```
A =  [[1, 4], [2, 5], [3, 6]]
```

Vamos apresentar também alguns detalhes extras para acessarmos os valores de uma lista. Podemos acessar o valor de uma lista `x` indicando a posição do elemento que queremos acessar da seguinte forma: `x[i]`, em que `i` representa um valor inteiro entre `0` e `len(x)`, digamos `n`. Para acessarmos um subconjunto de uma lista `L` de tamanho `n`, podemos usar o seguinte comando `L[m:s]` que irá acessar os elementos da posição `m` até a posição `s-1` e não `s`. Isso corresponde aos elementos `m+1`, `m+2`, `...`, `s`. Assim não devemos confundir o elemento com sua posição, pois o vetor inicia-se em `0` (elemento `1`) e não em `1`. Se os índices são negativos, então a lista será acessada de trás para frente, sendo que `-1` corresponde a posição `n-1`, `-2` a posição `n-2` e assim sucessivamente até `-n`, que corresponde a posição `0`. Veja alguns exemplos no `script` a seguir.

```
L = [1, 2, 3, 4, 5, 6, 7]
L
```

```
[1, 2, 3, 4, 5, 6, 7]
```

Acessando posições em particulares, para impressão ou para atribuição:

```
L[0]
L[1] = 8
L
```

```
1
```

```
[1, 8, 3, 4, 5, 6, 7]
```

Acessando, posições com os índices negativos.

```
L[-1]  # posição n-1
L[-len(L)] # posição 0
```

```
7
```

```
1
```

Para blocos de elementos temos que o comando `L{m:s:r]` acessa os elementos nas posições `m`, `m+1+r`, `m+1+2r`, `...` até a `(s-1)`-ésima posição ou até a posição mais próxima de `s-1` possível. Muito cuidado deve ser tomado, pois o limite, superior não indica onde o subconjunto termina entre os índices válidos de uma lista e sim,que ela termina na posição destacada subtraída de `1`.

```
L[1:4]
L[1:23] # passa do limite len(L)
L[0:5:2]
L[-1:-8:-1]
L[4:]
L[:6]
```

```
[8, 3, 4]
```

```
[8, 3, 4, 5, 6, 7]
```

```
[1, 3, 5]
```

```
[7, 6, 5, 4, 3, 8, 1]
```

```
[5, 6, 7]
```

```
[1, 8, 3, 4, 5, 6]
```

Se omitirmos os limites inferior ou superior da sequência, então a lista selecionada será iniciada no índice `0` (valor inicial) ou terminará no último índice (valor final da lista), como nos dois últimos exemplos apresentados.

1.4.2 Tuplas

As tuplas são objetos Python muito parecidos com as listas. Vários métodos e funções que se aplicam às listas também se aplicam às tuplas. Ao contrário das listas, as tuplas são objetos imutáveis, ou seja, uma vez criadas elas não podem ser modificadas. Assim, se criarmos uma tupla por `t = (1,2,3)` **não** poderemos atribuir valor, por exemplo, deste jeito `t[1] = 9`. Elas podem conter mais de um valor idêntico e são ordenadas, como as listas. A forma de criar a tupla em relação à lista é o uso dos parênteses no lugar dos colchetes.

```
t = (1, 2, 'DFF', 3)
'DFF' in t
t[2]
t[0]
t[:3]
len(t)
```

```
True
```

```
'DFF'
```

```
1
```

```
(1, 2, 'DFF')
```

```
4
```

Os métodos `count()` e `index` podem ser usados nas tuplas, como ilustrado a seguir. O método `index()` tem a seguinte sintaxe, sendo que os dois últimos argumentos são opcionais: `tuple.index(element, start, end)`.

```
t.count(1) # número de ocorrência de 1
t.index('DFF') # índice da posição de 'DFF'
```

```
1
```

```
2
```

A tupla pode ter qualquer tipo como sendo seus elementos, incluindo uma tupla ou uma lista.

```
t1 = ((1,2),'r', [2,3,4])
t1
print('Componente lista da tupla ',t1[2])
print('Elemento 0 do componente lista da tupla ',t1[2][0])
t1[2].append(5)
print('Modificando o componente lista da tupla ',t1)
t2 = [1,2,3]
sum(t2)
```

```
((1, 2), 'r', [2, 3, 4])
```

```
Componente lista da tupla  [2, 3, 4]
Elemento 0 do componente lista da tupla  2
Modificando o componente lista da tupla  ((1, 2), 'r', [2, 3, 4, 5])
```

```
6
```

A questão é: por que devemos usar tuplas, se elas não podem ser modificadas? A resposta para isso vem do fato de que a manipulação de dados via tuplas que são imutáveis é muito mais rápida do que nas listas. Apesar da tupla apontar para a mesma identificação da memória, fomos capazes de modificar um de seus elementos, que era a lista na sua segunda posição. Isso não mudou a identificação na memória para a qual a tupla `t1` apontava.

Existem muitos métodos ou funções que funcionam com as tuplas como o `len(t)` e o `count()` como ilustrado a seguir.

```
t = (1,1,2,3,3,3,4,5)
t.count(3)
len(t)
```

```
3
```

```
8
```

A função `any(t)` retorna `True` se há algum item `True` na tupla e retorna `False`, caso contrário. Neste caso, a tupla ou qualquer outro tipo apropriado poderá ter elementos `0` e `1` ou booleanos. Pode ser aplicada nas listas, conjuntos e nos dicionários.

```
t = (True, False, False,False,True)
any(t)
```

```
True
```

Podemos usar ainda as funções `min()`, `max()`, `sum()`e `sorted()`, como ilustrado a seguir. A função `sorted()` ordena a tupla e retorna uma **lista** ordenada como resultado. Veja que o método `lista.sort()`

altera o objeto `lista` e não pode ser aplicado na tupla, pelo fato de a tupla ser imutável.

```
t = (2.3,4.5,1.2,1.1,9.7,5.3)
min(t)
max(t)
sum(t)
sorted(t)
t
```

```
1.1
```

```
9.7
```

```
24.099999999999998
```

```
[1.1, 1.2, 2.3, 4.5, 5.3, 9.7]
```

```
(2.3, 4.5, 1.2, 1.1, 9.7, 5.3)
```

1.4.3 Conjuntos

Os conjuntos `set` em Python tem uma conotação muito próxima com a definição de conjuntos da matemática. Esses são conjuntos que a ordem ou duplicação de seus elementos não mudam o conjunto. Assim, são imutáveis, não ordenados e não pode ter mais de um elemento idêntico em suas ocorrências. Podemos usar o construtor (função) `set()` para criar um conjunto ou usarmos as chaves`{}` para digitar seus elementos separados por vírgula.

```
phi = set()
print('Conjunto vazio: ', phi)
# precisa ser uma lista ou tupla de argumento
A = set(['A','D','B','C','E'])
A
B = {1,2,3.4,5,6,6}
'A' in A
B # repare que o elemento
  #repetido 6 aparece 1 vez apenas
B[0]
```

```
Conjunto vazio:  set()
```

```
{'A', 'B', 'C', 'D', 'E'}
```

```
True
```

```
{1, 2, 3.4, 5, 6}
```

```
TypeError: 'set' object is not subscriptable
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[35], line 10
      8 B # repare que o elemento
      9   #repetido 6 aparece 1 vez apenas
---> 10 B[0]
TypeError: 'set' object is not subscriptable
```

Não podemos acessar um elemento de um conjunto por `B[0]`, por exemplo, o que ocasiona um erro, como pode ser visto no resultado do `script` anterior. A ordem não é importante. Vejamos a comparação do conjunto `A` anterior com o novo conjunto `C` criado a seguir.

```
C = set(['A','B','C','D','E'])
C
A == C
```

```
{'A', 'B', 'C', 'D', 'E'}
```

```
True
```

Algumas operações matemáticas com conjuntos estão disponíveis em Python, como união, interseção, diferença $(A^c \cap B)$ e diferença simétrica $((A^c \cap B) \cup (A \cap B^c))$, como ilustrados no exemplo a seguir.

```
A = {1,2,3,4,5,6}
B = {4,5,7,8,9,10}
A.union(B)
A.intersection(B)
A.difference(B) # esta em A, mas não em B
B.difference(A) # está em B, mas não em A
A.symmetric_difference(B) # está só em A ou só em B
A & B # intersecção
A | B # união
A - B # diferença
B - A # diferença
A^B   # diferença simétrica
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
{4, 5}
```

```
{1, 2, 3, 6}
```

```
{7, 8, 9, 10}
```

```
{1, 2, 3, 6, 7, 8, 9, 10}
```

```
{4, 5}
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
```

```
{1, 2, 3, 6}
```

```
{7, 8, 9, 10}
```

```
{1, 2, 3, 6, 7, 8, 9, 10}
```

1.4.4 Dicionários

Os dicionários `dictionary()` são objetos mutáveis e iteráveis. Os seus elementos vem sempre aos pares, sendo o primeiro valor uma chave e o segundo elemento, o valor da chave. Tanto a chave e seu valor são objetos Python. A chave é imutável, mas seus valores associados são objetos mutáveis ou não.

```
D = {1: [1,2,3.4,5], 2: 3.7, 3: {1,2,3}}
D[1]
D[2]
type(D[3])
type(D[1])
```

```
[1, 2, 3.4, 5]
```

```
3.7
```

```
set
```

```
list
```

Temos um dicionário, com as chaves `1`, `2` e `3`. Para a chave `1` temos uma lista como seu valor, para a chave `2`, temos um valor `float` e para a chave `3`, criamos um objeto do tipo conjunto. A seguir, acrescentamos uma chave, nomeada `4` com um valor booleano associado. O método `keys()` recupera as chaves do objeto dicionário `D`, transforma numa lista e imprime a lista e seu primeiro elemento `C[0]`.

```
D[4] = True
print(D)
C = list(D.keys())
print(C)
C[0]
```

```
{1: [1, 2, 3.4, 5], 2: 3.7, 3: {1, 2, 3}, 4: True}
[1, 2, 3, 4]
```

```
1
```

Do mesmo modo,podemos obter os valores das chaves facilmente, como ilustrado a seguir. Posteriormente, veremos como poderemos utilizar uma estrutura de repetição `for` para percorrer os elementos do dicionário e armazenar os valores em uma lista ou processá-los um a um.

```
D.values()
list(D.values())
D[C[0]] # acessando o valor com chave 1, C[0]
C[0] in D # verificando se chave 1 pertence a D
5 in D    # verificar se a chave 5 pertence a D
```

```
dict_values([[1, 2, 3.4, 5], 3.7, {1, 2, 3}, True])
```

```
[[1, 2, 3.4, 5], 3.7, {1, 2, 3}, True]
```

```
[1, 2, 3.4, 5]
```

```
True
```

```
False
```

Podemos deletar o conteúdo de uma chave, usando a função `del` ou usando o método `pop()`. E podemos atualizar o dicionário, criando novas chaves e valores, com o método `update`.

```
del D[3] # elimina a chave 3
D
D.pop(4) # elimina a chave 4
D
D.update({5: 'sou novo', 6:'eu também'})
D
```

```
{1: [1, 2, 3.4, 5], 2: 3.7, 4: True}
```

```
True
```

```
{1: [1, 2, 3.4, 5], 2: 3.7}
```

```
{1: [1, 2, 3.4, 5], 2: 3.7, 5: 'sou novo', 6: 'eu também'}
```

Podemos criar um objeto dicionário, criando duas tuplas, digamos `c` e `v`, com as chaves e com os valores das chaves (de mesmo tamanho). Em seguida emparelhamos os elementos com o comando `zip(c, v)`. Finalmente, usamos a função `dict` para criar o dicionário dos valores emparelhados.

```
c = (1,2,3,4) # chaves
v = ([1,2],True,4.5,('r','s'))
y = dict(zip(c,v))
y
y.get(5,'Chave não existe')
# get() não gera erro em chave inexistente
# mas, y[5]  geraria erro
y.get(1)
y[1]# como 1 existe, é equivalente ao get()
```

```
{1: [1, 2], 2: True, 3: 4.5, 4: ('r', 's')}
```

```
'Chave não existe'
```

```
[1, 2]
```

```
[1, 2]
```

Assim, tendo acesso a um valor da lista, podemos usar os métodos e funções apropriadas para lidarmos com eles. Mais detalhes destes objetos, aparecerão oportunamente, quando avançarmos em mais características da programação em Python.

1.5 Matrizes e Arranjos

As matrizes em Python, como dissemos e mostramos anteriormente, podem ser criadas pelas listas. Assim, vamos criar a seguir uma matriz 2×2 usando list para ilustrarmos o procedimento de criação. Se a dimensão for uma só, as listas são arranjos de uma dimensão, conhecidas por vetores.

```
A = [[4,1],[1,1]] # matriz 2 x 2
print('A = ',A)
A[1][1]           # retorna o valor A[2,2]
A[1][1] = 2       # altera o seu valor
A
```

```
A =  [[4, 1], [1, 1]]
```

```
1
```

```
[[4, 1], [1, 2]]
```

Para lidarmos com funções vetoriais (`arrays`) ou matriciais, podemos, entre outras possibilidades usar a biblioteca (pacote) `numpy`. Nosso primeiro passo é importar o pacote `numpy` com o apelido, `np`, que é o mais usado, para facilitar a chamada de seus métodos e funções. Isso só pode ser feito, se já tivermos instalado o pacote `numpy`.

```
del numpy    # eliminar a última importação
import numpy as np
```

Em seguida, criamos uma matriz com o uso da função `array()`. Vamos usar nossa lista `A` anterior, para fazer isso.

```
B = np.array(A)
B
```

```
array([[4, 1],
       [1, 2]])
```

Podemos criar também a partir de tuplas, em vez de listas, a matriz `numpy`. Além disso, existem funções próprias do pacote para criarmos matrizes, como, por exemplo, a matriz de zeros 2×4 a seguir. Também existem funções para criarmos `arrays` (vetores) unidimensionais, como o método `arange()` e o `linspace()`. O primeiro cria um vetor indo de `n` até o máximo `m` (inteiros) sem incluí-lo de 1 em 1, ou do mínimo `n` até o máximo `m` (excluindo o máximo) de r em r: `arange(n,m)` ou `arange(n,m,r)`. Já o `linspace(n,m,s)` inicia em `n`, finaliza em `m`, mas com passo igual a `(m - n) / (s - 1)`.

```
C = np.zeros((2,4))
print(C)
np.arange(2,7) # vetor com elementos 2,3,..,6
np.arange(2,7,0.5) # de 2 até 7, de 0.5 em 0,5 (exceto o 7)
np.linspace(2,6,6)
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

```
array([2, 3, 4, 5, 6])
```

```
array([2. , 2.5, 3. , 3.5, 4. , 4.5, 5. , 5.5, 6. , 6.5])
```

```
array([2. , 2.8, 3.6, 4.4, 5.2, 6. ])
```

Vamos ilustrar alguns cálculos simples com vetores.

```
x = np.arange(2,3,0.2)
y = np.arange(3,6,0.7)
print('', x,'\n', y)
x + y # adição dos vetores
x * y # produto elementwise
y ** x # potenciação elementwise
```

```
 [2.  2.2 2.4 2.6 2.8]
 [3.  3.7 4.4 5.1 5.8]
```

```
array([5. , 5.9, 6.8, 7.7, 8.6])
```

```
array([ 6.  ,  8.14, 10.56, 13.26, 16.24])
```

```
array([  9.        ,  17.78458735,  35.01760371,  69.13253113,
       137.27719037])
```

A multiplicação de matrizes por sua vez pode ser feita com a função `np.dot`, que tem uma versão na forma de método também, não modificando o objeto que o acionou.

```
C = np.array([[2,1],[1,2]])
B
C
np.dot(B,C)
B.dot(C)
B
```

```
array([[4, 1],
       [1, 2]])
```

```
array([[2, 1],
       [1, 2]])
```

```
array([[9, 6],
       [4, 5]])
```

```
array([[9, 6],
       [4, 5]])

array([[4, 1],
       [1, 2]])
```

As inversas podem ser obtidas com a função `np.linalg.inv()` e as inversas generalizadas de Moore-Penrose pelo método `numpy.linalg.pinv()`. O determinante pode ser obtido por `np.linalg.det()` e os autovalores e autovetores (decomposição espectral) por `np.linalg.eig()`. É importante observar que os autovalores do `np.linalg.eig()` não necessariamente estará ordenado do maior para o menor, como é convencionalmente adotado em diferentes outros programas. Este método é bem geral e pode ser usado em matrizes reais ou complexas quadradas. Alternativamente, para matrizes simétricas o `numpy` possui o método `np.linalg.eigh()`, que retorna os autovalores em ordem crescente. Veja o exemplo a seguir.

```
np.linalg.inv(B)
np.linalg.pinv(B) # igual a inversa (posto completo)
np.linalg.det(B)
L, P = np.linalg.eig(B)
print('Autovalores: ',L,'\nAutovetores: ',P)
L1, P1 = np.linalg.eigh(B)
print('Autovalores: ',L1,'\nAutovetores: ',P1)
```

```
array([[ 0.28571429, -0.14285714],
       [-0.14285714,  0.57142857]])

array([[ 0.28571429, -0.14285714],
       [-0.14285714,  0.57142857]])

6.999999999999999

Autovalores:  [4.41421356 1.58578644]
Autovetores:  [[ 0.92387953 -0.38268343]
 [ 0.38268343  0.92387953]]
Autovalores:  [1.58578644 4.41421356]
Autovetores:  [[ 0.38268343 -0.92387953]
 [-0.92387953 -0.38268343]]
```

Também podemos usar a biblioteca `scipy` com os métodos `sp.linalg.eig()` e `sp.linalg.eigh()`, que fazem exatamente como os seus similares da biblioteca `numpy`. Para isto devemos importar, depois de instalada, a biblioteca usando `import scipy as sp`.

Para matrizes `n x m`, podemos usar uma decomposição matricial muito útil para a estatística, em métodos como componentes principais, AMMI, biplot, entre outros. Esse método é chamado de decomposição do valor singular. Nele obtemos a decomposição de uma matriz $\mathbf{A}$ `(m x n)` da seguinte forma: $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{V}^\top$, em que o método `np.linalg.svd()` retorna $\mathbf{U}$ uma matriz $m \times k$ ortonormal por colunas, $\mathbf{V}$ uma matriz $n \times k$ ortonormal por coluna e o vetor correspondente à diagonal de $\mathbf{\Lambda}$, que é uma matriz diagonal $k \times k$ de elementos reais positivos, se usarmos a opção `full_matrices=False`. As matrizes $\mathbf{U}$ e $\mathbf{V}$ são as matrizes dos vetores singulares e a matriz $\mathbf{\Lambda}$ é a matriz dos valores singulares, sendo k o posto de $\mathbf{A}$ que é $k = \min(n,m)$. O `script` a seguir ilustra a obtenção da decomposição singular de uma matriz 3×2.

```
A = np.array([[1,4],[2,7],[9, 3]])
U,L,Vt = np.linalg.svd(A, full_matrices=False)
print('Vetores singulares à esquerda: ',U,
'\nVetor dos valores singulares: ',L,
'\nVetores singulares à direita (transposto): ',Vt)
```

```
len(L) # posto de A
np.diag(L) # constrói a matriz Lambda
```

```
Vetores singulares à esquerda:  [[-0.30248631 -0.39963983]
 [-0.54614812 -0.67137377]
 [-0.78116852  0.62413562]]
Vetor dos valores singulares:  [11.19813546  5.88232625]
Vetores singulares à direita (transposto):  [[-0.75238412 -0.65872463]
 [ 0.65872463 -0.75238412]]

2

array([[11.19813546,  0.        ],
       [ 0.        ,  5.88232625]])
```

Para verificarmos que a decomposição realmente é adequada, temos o seguinte `script`:

```
np.dot(np.dot(U,np.diag(L)),Vt)
U.dot(np.diag(L)).dot(Vt) # alternativo
U @ np.diag(L) @ Vt       # alternativo
```

```
array([[1., 4.],
       [2., 7.],
       [9., 3.]])

array([[1., 4.],
       [2., 7.],
       [9., 3.]])

array([[1., 4.],
       [2., 7.],
       [9., 3.]])
```

Muitas outras funções existem no pacote `numpy` para lidarmos ou operarmos matrizes. Se necessitarmos de alguma outra função para alguma operação matricial, vamos apresentá-la nestas ocasiões.

1.6 Arquivos de Dados

Usaremos pouco esta estrutura de dados neste material, embora vamos apresentar a biblioteca `pandas` para lidarmos com os `DataFrames`. Iremos apresentar a leitura de um arquivo particular com duas variáveis `X1` e `X2`, gravado como arquivo texto separado por espaço entre as colunas (variáveis). O caminho onde este arquivo se encontra em meu computador é: `g:/Meu Drive/daniel/Cursos/Estatistica computacional/Apostila/` e seu nome é `dados.txt`. Devemos inicialmente instalar a biblioteca `pandas` com o comando: `pip install pandas`. Posteriormente, carregamos o pacote pandas, como apresentado no `script` a seguir. Para mudar o diretório, precisamos importar o pacote `os` e usar `os.getcwd()` para obter o diretório de trabalho atual e para alterá-lo `os.chdir('[path]')`.

```
import pandas as pd
import os
apath = os.getcwd() # caminho do projeto
os.chdir('g:/Meu Drive/daniel/Cursos/Estatistica computacional/Apostila/')
os.getcwd()
```

```
'g:\\Meu Drive\\daniel\\Cursos\\Estatistica computacional\\Apostila'
```

Para lermos um arquivo deste diretório em um objeto `DataFrame` podemos usar a função `pd.read_csv('dados.txt',r'\s+')`, cujo símbolo `r'\s+'` significa que o arquivo é separado por espaços, podendo variar o número de espaços entre colunas de registro (linha) para registro, ou seja, se o número de espaços não está padronizado entre as colunas para as diferentes linhas do arquivo de dados.

```
dados = pd.read_csv('dados.txt',sep=r'\s+')
dados
```

	X1	X2
0	13.4	14
1	14.6	15
2	13.5	19
3	15.0	23
4	14.6	17
5	14.0	20
6	16.4	21
7	14.8	16
8	15.2	27
9	15.5	34
10	15.2	26
11	16.9	28
12	14.8	24
13	16.2	26
14	14.7	23
15	14.7	9
16	16.5	18
17	15.4	28
18	15.1	17
19	14.2	14

Os `DataFrames` são estruturas de dados tabular, sendo que cada coluna possui constitui de uma sequência de valores do mesmo tipo (booleano, float, strings, etc.), em que as diferentes colunas podem ser e potencialmente são de diferentes tipos. Os `DataFrames` possuem uma coluna adicional chamada `index` que no exemplo anterior, do objeto `dados`, variou de `0` a `19`, pois nosso `DataFrame` possui `20` linhas e duas variáveis `X1` e `X2`, que no arquivo `dados.txt` estavam identificadas na primeira linha física do arquivo que foi lido pelo método `read_csv()`. O `index` mapeia as linhas do `DataFrame` com os `labels` mencionados.

Vamos mostrar como construir um `DataFrame` diretamente a partir de um objeto `dict` para o construtor `DataFrame()` do `pandas`. Vamos criar um `DataFrame` de um delineamento inteiramente casualizado, com `2`tratamentos e `3` repetições de cada um, com os respectivas produtividades avaliadas nas `6` parcelas experimentais.

```
dic = {'rep': [1,2,3,1,2,3],
       'trat': [1,1,1,2,2,2],
       'prod': [3.4,2.3,5.6,5.7,6.3,7.1]}
arqd = pd.DataFrame(dic)
arqd
```

	rep	trat	prod
0	1	1	3.4
1	2	1	2.3

	rep	trat	prod
2	3	1	5.6
3	1	2	5.7
4	2	2	6.3
5	3	2	7.1

Podemos acrescentar uma nova coluna em nosso `DataFrame`, ou eliminarmos uma já existente, criado um novo `DataFrame` para receber o resultado, como mostrado a seguir, com a opção `columns`, para qualquer ordem das chaves (nomes das colunas).

```
arqd
arqd1 = pd.DataFrame(arqd,columns=['trat','prod'])
arqd1 # selecionando 2 variáveis no DataFrame
arqd['alt'] = [1.5,1.3,1.4,1.2,1.1,1.6] #criando variável altura
arqd
```

	rep	trat	prod
0	1	1	3.4
1	2	1	2.3
2	3	1	5.6
3	1	2	5.7
4	2	2	6.3
5	3	2	7.1

	trat	prod
0	1	3.4
1	1	2.3
2	1	5.6
3	2	5.7
4	2	6.3
5	2	7.1

	rep	trat	prod	alt
0	1	1	3.4	1.5
1	2	1	2.3	1.3
2	3	1	5.6	1.4
3	1	2	5.7	1.2
4	2	2	6.3	1.1
5	3	2	7.1	1.6

Para acessarmos as chaves (colunas), os índices e os valores do `DataFrame` podemos usar os seguintes códigos ilustrativos. Observamos que a instrução `arqd.prod` para acessar a coluna `prod`, resulta em erro, pois `prod` é uma palavra reservada (produto). Devemos usar a alternativa anterior para esta chave.

```
arqd.columns
arqd.index
```

```
arqd.values
arqd['prod'][0] # produção do índice 0
arqd['prod'] # toda a coluna de produção
arqd.prod # cuidado, pois prod é palavra reservada: erro
arqd.rep
```

```
Index(['rep', 'trat', 'prod', 'alt'], dtype='object')
```

```
RangeIndex(start=0, stop=6, step=1)
```

```
array([[1. , 1. , 3.4, 1.5],
       [2. , 1. , 2.3, 1.3],
       [3. , 1. , 5.6, 1.4],
       [1. , 2. , 5.7, 1.2],
       [2. , 2. , 6.3, 1.1],
       [3. , 2. , 7.1, 1.6]])
```

```
3.4
```

```
0    3.4
1    2.3
2    5.6
3    5.7
4    6.3
5    7.1
Name: prod, dtype: float64
```

```
<bound method DataFrame.prod of    rep  trat  prod  alt
0    1     1   3.4  1.5
1    2     1   2.3  1.3
2    3     1   5.6  1.4
3    1     2   5.7  1.2
4    2     2   6.3  1.1
5    3     2   7.1  1.6>
```

```
0    1
1    2
2    3
3    1
4    2
5    3
Name: rep, dtype: int64
```

Para extrairmos uma linha inteira usamos o método `loc` do `DataFrame` associado ao índice da linha (registro), como ilustrado no `scripta` seguir.

```
L  = arqd.loc[1] # segunda linha do DataFrame
print(L)
L2 = arqd.loc[[0,5]] # as linhas 1 e 6 de arqd
L2                   # com os índices 0 e 5
```

```
rep     2.0
trat    1.0
prod    2.3
alt     1.3
Name: 1, dtype: float64
```

	rep	trat	prod	alt
0	1	1	3.4	1.5
5	3	2	7.1	1.6

Para selecionar um bloco de registros (linhas) indo de `inicio` ao `fim` (excluindo o limite final) usamos `arqd[inicio:fim]`. Como ilustrado a seguir, onde extraímos do índice `0` até o índice `3`, ou seja, as três primeiras linhas com os índices `0`, `1` e `2` do `arqd`.

```
arqd[0:3]
```

	rep	trat	prod	alt
0	1	1	3.4	1.5
1	2	1	2.3	1.3
2	3	1	5.6	1.4

Valores perdidos podem fazer parte do `DataFrame` e neste caso, eles assumem o valor `NaN`, do inglês `not a number`. Podemos deletar uma coluna com o comando del, da seguinte forma.

```
del arqd['alt']
arqd
```

	rep	trat	prod
0	1	1	3.4
1	2	1	2.3
2	3	1	5.6
3	1	2	5.7
4	2	2	6.3
5	3	2	7.1

Podemos filtrar impondo condições específicas ao `DataFrame`. Por exemplo, se estivermos interessado no `DataFrame` resultante dos elementos em que a produção é maior ou igual a `5`, teremos o seguinte resultado. O resultado filtrado mostra os registros nas mesmas posições originais e o seu `DataFrame` original `arqd` permanece inalterado.

```
arqd[arqd['prod'] >= 5.0]
arqd
```

	rep	trat	prod
2	3	1	5.6
3	1	2	5.7
4	2	2	6.3
5	3	2	7.1

	rep	trat	prod
0	1	1	3.4
1	2	1	2.3

	rep	trat	prod
2	3	1	5.6
3	1	2	5.7
4	2	2	6.3
5	3	2	7.1

Para gravarmos um `DataFrame` podemos escolher o diretório (pasta) e o nome do arquivo e gravarmos em um arquivo `csv` (arquivo separado por vírgula) com o comando `arqd.to_csv('nome.csv,index=False,header=True)` para não salvar o índice e salvar o cabeçalho. Vamos recuperar em nosso código, o path original com o comando `os.chdir(apath)`, em que `apath` foi obtido quando iniciamos o assunto sobre `DataFrame` e refere-se ao diretório deste projeto. Escolhemos o nome `dic.csv` para o arquivo. Podemos ler o arquivo novamente, conforme mostramos nos primeiros passos da abordagem dos `DataFrames`. Depois de gravado, repetimos sua leitura e o colocamos no objeto `dic`, em que devemos atentar para o separador de colunas, que neste caso é a vírgula.

```
os.chdir(apath)
arqd.to_csv('dic.csv',index=False,header=True)
dic = pd.read_csv('dic.csv',sep=r',')
dic
```

	rep	trat	prod
0	1	1	3.4
1	2	1	2.3
2	3	1	5.6
3	1	2	5.7
4	2	2	6.3
5	3	2	7.1

Em futuras edições, mostraremos mais detalhes dos `DataFrames`. Nos capítulos posteriores, caso venhamos a precisar de um `DataFrame` e de algumas de suas propriedades, então iremos adicionar os conteúdos necessários nesta ocasião. Falaremos agora das estruturas condicionais e as estruturas de repetições.

1.7 Estruturas de Controle de Programação

O Python é um ambiente de programação em que programas contêm os módulos, os módulos contém instruções, as instruções contém comandos e as expressões criam e processam os objetos. Estas instruções são as atribuições tipo `a=b`, a chamada de métodos e funções como `print(a)`, `if/elif/else` para seleção de ações, o `for/else` para realizar iterações, o `while/else` para `loops`em geral, `break` e `continue`para controle de `loops`, `def` para definições de funções, `yeld` para gerador de funções, entre outros. As instruções são organizadas em expressões em grupos de comandos, que diferentemente de outras linguagens são organizados pelas indentações (recuo do parágrafo). Assim, o corpo ou grupo de comando de uma instrução específica, ficará reunida se elas tiverem indentadas em relação a instrução principal e com o mesmo nível de indentação. Em outra linguagens os grupos de comandos são reunidos pelas chaves: `{grupos de comandos}`. O fim de uma linha termina a instrução, que também pode ser finalizada por um ponto e vírgula. Uma instrução pode continuar em uma nova linha se a linha terminar com o \ ou com o uso dos parênteses.

O nome das variáveis em Python em Python devem iniciar com `underscore` ou letra, seguido por números ou letras ou underscore. O python diferencia as maiúsculas das minúsculas, assim `Y` é diferente de `y`. Os nomes devem evitar as palavras reservadas do Python, como False, None, True, class, and, if, elif, yield,

while, break, global, nor, try, return, break,in, etc. Por convenção, as classes em Python começam com maiúsculo e os módulos e nomes de variáveis por minúsculo.

As estruturas condicionais, `if/elif/else` são estruturas em Python para selecionar ações. Esta instrução pode conter outras instruções do mesmo tipo ou diferentes instruções em seus grupos de comando. O formato geral é dado por

```
if condição1:
  instruções1
elif condição2: # opcional elifs
  instruções2
else:           # opcional else
    instruções3
```

A instrução `elif` significa `else if` e é opcional. Se a `condição1` for verificada, é executado o bloco denominado `instruções1`, que estão indentados em relação ao `if`. Caso a condição seja falsa, é testada a `condição2` e se ela for verdadeira, são executados as instruções denotadas por `instruções2`, que pode ser uma simples instrução ou várias instruções, indentadas em relação ao `elif`. Finalmente, se a `condição2` for falsa são executadas as `instruções3`. Cada linha das instruções `if`, `elif` ou `else` são seguidas por um ponto e vírgula. Veja o exemplo simples a seguir.

```
x = 5
if x > 6:
  print('Recebe mais que 6 salários.')
  print('Você está entre os 20% mais ricos!')
else:
  print('Você recebe 6 salários ou menos.')
  print('Você representa 80% da população!')
```

```
Você recebe 6 salários ou menos.
Você representa 80% da população!
```

Para um modelo probabilístico temos o seguinte modelo para a função de distribuição:

$$F_X(x) = \begin{cases} 0 & \text{se } x < 0 \\ x^2 & \text{se } 0 \leq x \leq 1 \\ 1 & \text{se } x > 1. \end{cases}$$

Para este modelo, temos o seguinte script, no qual decidimos qual parte do programa rodar, conforme os valores de `x` são atribuídos.

```
x = 0.8
if x < 0:
  F = 0
elif 0 <= x <= 1:
  F = x**2
else:
  F = 1
F
```

```
0.6400000000000001
```

As estruturas de repetição do Python são o `for` e o `while/else`. Existe ainda um terceiro tipo de procedimento em Python para realizarmos iterações. A estrutura geral de um código Python para o `for` è apresentado no `scripta` seguir.

```
for i in:
  instruçoes1
else:            # opcional instrução else
  instruções2
```

Os objetos do comando `for` são os objetos iteradores ou iteráveis, que são aqueles que contém um número contáveis de valores. As listas, tuplas, dicionários e conjuntos ão todos objetos iteráveis. Vamos ilustrar com um simples exemplo a seguir, para calcularmos a soma, o produtório e a média de uma lista de valores.

```
x = [2.3, 4.1, 1.5, 2.3, 4.7]
soma = 0
prod = 1
n = len(x)
for y in x:
  soma = soma  + y
  prod = prod * y
media = soma / n
print('A soma é: ',soma)
print('O produtório é: ',prod)
print('A média é: ',media)
```

```
A soma é:  14.899999999999999
O produtório é:  152.90744999999995
A média é:  2.9799999999999995
```

Para realizarmos iterações em um dicionário, temos o seguinte exemplo:

```
D = {1: 1.3, 2: 3.1, 3: 1.7}
for i in D: # iterar nas chaves
  print(i, '=', D[i])
print('Agora iterando nas chaves e valores:')
for (i, valor) in D.items():
  print(i, '=', valor)  # iterar em chave e valor
```

```
1 = 1.3
2 = 3.1
3 = 1.7
Agora iterando nas chaves e valores:
1 = 1.3
2 = 3.1
3 = 1.7
```

Finalmente, um exemplo em um conjunto:

```
A = {1,2,3,4,5}
for i in A:
  print('elemento: ',i)
```

```
elemento:  1
elemento:  2
elemento:  3
elemento:  4
elemento:  5
```

Podemos criar um sequência de valores com o comando `range(n)` que vai de `0` a `6`. Assim, também

podemos usar o `for` nessa sequência, como ilustrado no exemplo a seguir.

```
x = [2.3, 4.1, 1.5, 2.3, 4.7]
soma = 0
n = len(x)
for i in range(n):
  soma = soma  + x[i]
media = soma / n
print('A soma é: ',soma)
print('A média é: ',media)
```

```
A soma é:  14.899999999999999
A média é:  2.9799999999999995
```

O `while` é uma outra estrutura de repetição, em que o bloco de comandos indentados irão ser executados até que uma condição seja satisfeita. A estrutura geral é dada a seguir.

```
while condição:
  instruçoes1
else:            # opcional instrução else
  instruções2
```

O exemplo a seguir, ilustra o cálculo do total e da média de uma lista.

```
x = [2.3, 4.1, 1.5, 2.3, 4.7]
soma = 0
n = len(x)
i = 0
while i < n:
  soma = soma  + x[i]
  i = i + 1
media = soma / n
print('A soma é: ',soma)
print('A média é: ',media)
```

```
A soma é:  14.899999999999999
A média é:  2.9799999999999995
```

Podemos usar os comandos `break` e `continue` dentro do while (ou do for). O `break` é usado após uma segunda condição ser verificada no bloco de comandos do interior da estrutura de repetição e pula a execução do programa para a primeira linha de instrução após o bloco do `loop`, ou seja, encerra o `loop`. O `continue` executa a primeira linha testando a condição primária do `while` ou tomando o próximo valor do iterador no `for`, ou seja, vai para o início do `loop`. Veja o exemplo a seguir.

```
y = 35 # experimente outro número inteiro > 1
x = y // 2
while x > 1:
  if y % x == 0:
    print(y, 'tem fator ', x)
    break
  x = x - 1
else:
  print(y,' é primo')
```

```
35 tem fator  7
```

Podemos utilizar a função `filter`, como já ilustramos anteriormente neste material, para realizarmos

iterações em nosso código. Não daremos mais detalhes disso, por enquanto.

1.8 Funções

As funções em todas as linguagens é uma poderosa ferramenta de programação, que nos permite quebrar um grande problema em pequenas tarefas (as funções), facilitando assim a resolução do problema como um todo. Dizemos que é a estratégia de dividir para conquistar. As funções em geral recebem um objeto e o processa de acordo com as regras definidas em seu bloco de comando. Desta forma a linguagem ganha grande poder, conveniência e elegância. O aprendizado em escrever funções úteis é uma das muitas maneiras de fazer com que o uso do Python seja confortável e produtivo. A sintaxe geral de uma função é dada por:

```
def nome(arg1, arg2,...,argn):
  instruções
```

As instruções significam um bloco de comandos (indentados) e podem ou não ter o comando de `return objeto`, que pode acontecer em qualquer parte do bloco de comandos. A função pode não ter este comando de `return`, se ela modificar um arquivo apenas gravando um novo resultado ou se imprimir uma mensagem quando chamada. Os argumentos ou parâmetros são passado para a função e a sua chamada deve obedecer estritamente a ordem em que eles aparecem, a menos que a chamada seja com chave, ou seja, do tipo ‘`arg1 = 2.3`, por exemplo. Neste caso, os argumentos podem ser colocados em qualquer ordem. Os argumentos de uma função podem conter valores `default`, ou seja, na declaração do nome da função podemos ter algo do tipo: `def nome(x, theta = 0.5)`. O argumento `theta=0.5` pode ser omitido na chamada da função, que será atribuído seu valor `0,5` padrão.

Vamos apresentar uma função simples para testar a hipótese $H_0 : \mu = \mu_0$ a partir de uma amostra simples de uma distribuição normal. Dois argumentos serão utilizados: o vetor (lista) de dados x de tamanho n e o valor real hipotético μ_0. A função calculará o valor da estatística t_c do teste por:

$$t_c = \frac{\bar{X} - \mu_0}{\frac{S}{\sqrt{n}}}. \tag{1.2}$$

A função resultante, em Python, é apresentada a seguir. Neste exemplo uma amostra de tamanho $n = 8$ foi utilizada para obter o valor da estatística para testar a hipótese $H_0 : \mu = 3{,}0$ (se a amostra era proveniente do povo na’vu). Podemos observar que o resultado final da função é igual ao do último comando executado, ou seja o valor da estatística e do valor-p, por meio de um objeto do tipo dicionário. Esta função utiliza no seu escopo três funções do Python (funções básicas do `numpy`), ainda não apresentadas. As duas primeiras, `var()` e `mean()` retornam a variância e a média do vetor utilizado como argumento, respectivamente, e a terceira, `pt()`, retorna a probabilidade acumulada da distribuição t de Student para o primeiro argumento da função com ν graus de liberdade, que é o seu segundo argumento.

```
import scipy as sp # para calcular probabilidade da t
def t_test(x, mu0):
  n = len(x)
  s2 = np.var(x,ddof=1) # ddof=1, divisor n-1 para s2
  xb = np.mean(x)
  t = {'tc':0,'p.val':0}
  t['tc'] = (xb-mu0) / (s2 / n)**0.5
  t['p.val'] = 2*(1-sp.stats.t.cdf(abs(t['tc']),n-1))
  return t
y = [1.76,1.81,1.74,1.71,1.79,1.75]
t = t_test(y, 3.0) # altura de avatares
```

```
print('tc = ',t['tc'])
print('p.val = ',t['p.val'])
```

```
tc =  -84.89699641330068
p.val =  4.297311617662558e-09
```

Podemos reescreve esta função para colocarmos um valor `default` para o argumento `mu0`. Se escolhêssemos o valor `0` e em alguma chamada da função, esse argumento fosse omitido, seria feito o teste da hipótese $H_0 : \mu = 0$ por padrão.

```
def t_test(x, mu0 = 0):
  n = len(x)
  s2 = np.var(x,ddof=1) # ddof=1, divisor n-1 para s2
  xb = np.mean(x)
  t = {'tc':0,'p.val':0,'S2': s2, 'xbar': xb}
  t['tc'] = (xb-mu0) / (s2 / n)**0.5
  t['p.val'] = 2*(1-sp.stats.t.cdf(abs(t['tc']),n-1))
  return t
y = [1.76,1.81,1.74,1.71,1.79,1.75]
t = t_test(y) # teste H0: mu0=0
print('tc = ',t['tc'])
print('p.val = ',t['p.val'])
print(list(t.items())[2:4])
```

```
tc =  120.49896265113645
p.val =  7.465636997494585e-10
[('S2', 0.001280000000000002), ('xbar', 1.76)]
```

Vamos realizar um test para a correlação em populações normais bivariadas. Assim, dado o par de variáveis vetoriais `x` e `y` tomados em `n` indivíduos, temos a hipótese nula

$$H_0 : \rho = 0$$

e a estatística do teste sob H_0 dada por

$$t_c = \frac{r\sqrt{n-2}}{\sqrt{1-r^2}},$$

em que r é o coeficiente de correlação amostral entre X e Y; e n é o tamanho da amostra. Sob H_0, essa estatística segue a distribuição t de *Student* com $\nu = n - 2$ graus de liberdade.

A função deve receber os vetores **`x`** e **`y`** e retornar o resultado do teste: estatística e valor-p. Pode-se utilizar a função `cor` do Python `scipy` para obter a correlação entre **`x`** e **`y`**.

```
def cor_test(x, y):
  n = len(x)
  if n != len(y):
    print('Listas devem ter o mesmo tamanho!')
    return
  r = np.corrcoef(x,y)[0,1]
  t = {'tc':0,'p.val':0,'r': r}
  t['tc'] = r * (n-2)**0.5 / (1 - r**2)**0.5
```

```
  t['p.val'] =2*(1-sp.stats.t.cdf(abs(t['tc']),n-2))
  return t
x = [1, 2, 3.1, 4.2]
y = [2.1, 3.9, 6.1, 8.3]
t = cor_test(x, y)
print('tc = ',t['tc'])
print('p.val = ',t['p.val'])
print('r = ',t['r'])
```

```
tc =  58.469836352468235
p.val =  0.0002923787083537466
r =  0.9997076212916461
```

Finalmente, vamos obter uma função para obtermos potências reais de matrizes quadradas simétricas positivas definidas. Para isso vamos escrever uma função para obter potências reais de uma matriz $\mathbf{A}$ simétrica e positiva definida:

$$\mathbf{A} = \mathbf{P}\mathbf{\Lambda}\mathbf{P}^\top,$$

sendo a potência de ordem $\alpha \in \mathbb{R}$ dada por

$$\mathbf{A}^\alpha = \mathbf{P}\mathbf{\Lambda}^\alpha\mathbf{P}^\top.$$

Deve receber $\mathbf{A}$ e retornar $\mathbf{A}^\alpha$. O `script` a seguir ilustra uma função para obtermos estas potências matriciais. Observe que não é uma potência elemento a elemento. Também devemos observar que não é importante que os autovalores estejam ordenados. Mas é óbvio que os autovetores associados a cada autovalor deve estar corretamente associado e preservado e isso é feito pela biblioteca `numpy` por meio da função `eig()`.

```
def mat_power(A, alpha = 0.5):
  e_val, e_vec = np.linalg.eig(A)
  if  any(e_val) < 0:
    print('Matriz não é positiva definida!')
    return
  Ap = e_vec.dot(np.diag(e_val**alpha)).dot(np.transpose(e_vec))
  return Ap
A = [[4,1],[1,2]]
print('A = ',A)
mat_power(A) # raiz quadrada
A3 = mat_power(A, 1/3) # raiz cúbica
A3
A3.dot(A3).dot(A3) # verificando
```

```
A =  [[4, 1], [1, 2]]
```

```
array([[1.97773553, 0.29759397],
       [0.29759397, 1.38254759]])
```

```
array([[1.57094963, 0.16768037],
       [0.16768037, 1.23558888]])
```

```
array([[4., 1.],
       [1., 2.]])
```

1.9 Estatística Computacional

Os métodos de computação intensiva têm desempenhado um papel cada vez mais importante para resolver problemas de diferentes áreas da ciência. Vamos apresentar algoritmos para gerar realizações de variáveis aleatórias de diversas distribuições de probabilidade, para realizar operações matriciais, para realizar inferências utilizando métodos de permutação e bootstrap, etc. Assim, buscamos realizar uma divisão deste material em uma seção básica e em outra aplicada. As técnicas computacionais são denominadas de estatística computacional se forem usadas para realizarmos inferências, para gerarmos realizações de variáveis aleatórias ou para compararmos métodos e técnicas estatísticas.

Vamos explorar métodos de geração de realizações de variáveis aleatórias de diversos modelos probabilísticos, para manipularmos matrizes, para obtermos quadraturas de funções de distribuição de diversos modelos probabilísticos e de funções especiais na estatística e finalmente vamos apresentar os métodos de computação intensiva para realizarmos inferências em diferentes situações reais. Temos a intenção de criar algoritmos em linguagem Python e posteriormente, quando existirem, apresentar os comandos para acessarmos os mesmos algoritmos já implementados.

Vamos apresentar os métodos de *bootstrap* e Monte Carlo, os testes de permutação e o procedimento *jackknife* para realizarmos inferências nas mais diferentes situações reais. Assim, este curso tem basicamente duas intenções: possibilitar ao aluno realizar suas próprias simulações e permitir que realizem suas inferências de interesse em situações em que seria altamente complexo o uso da inferência clássica.

Seja na inferência frequentista ou na inferência Bayesiana, os métodos de simulação de números aleatórios de diferentes modelos probabilísticos assumem grande importância. Para utilizarmos de uma forma mais eficiente a estatística computacional, um conhecimento mínimo de simulação de realizações de variáveis aleatórias é uma necessidade que não deve ser ignorada. Vamos dar grande ênfase a este assunto, sem descuidar dos demais. Apresentaremos neste material diversos algoritmos desenvolvidos e adaptados para a linguagem Python.

Simular é a arte de construir modelos segundo Naylor et al. [1971], com o objetivo de imitar o funcionamento de um sistema real, para averiguarmos o que aconteceria se fossem feitas alterações no seu funcionamento (Dachs [1988]). Este tipo de procedimento pode ter um custo baixo, evitar prejuízos por não utilizarmos procedimentos inadequados e otimizar a decisão e o funcionamento do sistema real.

Precauções contra erros devem ser tomadas quando realizamos algum tipo de simulação. Podemos enumerar:

1. escolha inadequada das distribuições;
2. simplificação inadequada da realidade; e
3. erros de implementação.

Devemos fazer o sistema simulado operar nas condições do sistema real e verificar por meio de alguns testes se os resultados estão de acordo com o que se observa no sistema real. A este processo denominamos de validação. A simulação é uma técnica que usamos para a solução de problemas. Se a solução alcançada for mais rápida, com eficiência igual ou superior, de menor custo e de fácil interpretação em relação a outro método qualquer, o uso de simulação é justificável.

1.10 Exercícios

1. Criar no Python os vetores $\mathbf{a}^\top = [4, 2, 1, 5]$ e $\mathbf{b}^\top = [6, 3, 8, 9]$ e concatená-los formando um único vetor. Obter o vetor $\mathbf{c} = 2\mathbf{a} - \mathbf{b}$ e o vetor $\mathbf{d} = \mathbf{b}^\top\mathbf{a}$. Criar uma sequência cujo valor inicial é igual a 2 e o valor final é 30 e cujo passo é igual a 2. Replicar cada valor da sequência 4 vezes de duas formas diferentes (valores replicados ficam agregados e a sequência toda se replica sem que os valores iguais fiquem agregados).

2. Selecionar o subvetor de $\mathbf{x}^\top = [4, 3, 5, 7, 9, 10]$ cujos elementos são menores ou iguais a 7.
3. Criar a matriz

$$\mathbf{A} = \begin{bmatrix} 10 & 1 \\ 1 & 2 \end{bmatrix}$$

   e determinar os autovalores e a decomposição espectral de $\mathbf{A}$.
4. Construir uma função para verificar quantos elementos de um vetor de dimensão n são menores ou iguais a uma constante k, real. Utilize as estruturas de repetições `for` e `while` para realizar tal tarefa (cada uma destas estruturas deverá ser implementada em uma diferente função). Existe algum procedimento mais eficiente para gerarmos tal função sem utilizar estruturas de repetições? Se sim, implementá-lo.
5. Implementar uma função R para realizar o teste t de *Student* para duas amostras independentes. Considerar os casos de variâncias heterogêneas e homogêneas. Utilizar uma estrutura condicional para aplicar o teste apropriado, caso as variâncias sejam heterogêneas ou homogêneas. A decisão deve ser baseada em um teste de homogeneidade de variâncias. Para realizar tal tarefa implementar uma função específica assumindo normalidade das amostras aleatórias.
6. Criar uma função para obter a inversa de Moore-Penrose de uma matriz qualquer $n \times m$, baseado na decomposição do valor singular, função `svd` do `np.linalg.svd`. Seja para isso uma matriz $\mathbf{A}$, cuja decomposição do valor singular é $\mathbf{A} = \mathbf{UDV}^\top$, em que $\mathbf{D}$ é a matriz diagonal dos valores singulares e $\mathbf{U}$ e $\mathbf{V}$ são os vetores singulares correspondentes. A inversa de Moore-Penrose de $\mathbf{A}$ é definida por $\mathbf{A}^+ = \mathbf{VD}^{-1}\mathbf{U}^\top$.

Chapter 2

Variáveis Aleatórias Uniformes

Neste capítulo vamos considerar a geração de números aleatórios para o modelo probabilístico uniforme. A partir do modelo uniforme podemos gerar realizações de variáveis aleatórias de qualquer outro modelo probabilístico. Para gerar realizações de uma distribuição uniforme, precisamos gerar números aleatórios. Isso não pode ser realizado por máquinas. Na verdade qualquer sequência produzida por uma máquina é na verdade uma sequência previsível. Dessa forma, a denominamos de sequência de números pseudo-aleatórios.

Uma sequência de números será considerada "aleatória" do ponto de vista computacional se o programa que a gerar for diferente e estatisticamente não correlacionado com o programa que a usará. Assim, dois geradores de números aleatórios deveriam produzir os mesmos resultados nas suas aplicações. Se isso não ocorrer, um deles não pode ser considerado um bom gerador de números aleatórios [Press et al., 1992].

Os conceitos de números uniformes e números aleatórios podem ser muitas vezes confundidos. Números uniformes são aqueles que variam aleatoriamente em uma faixa determinada de valores com probabilidade constante. No entanto, devemos diferenciar números aleatórios uniformes de outros tipos de números aleatórios, como por exemplo, números aleatórios normais ou gaussianos. Estes outros tipos são geralmente provenientes de transformações realizadas nos números aleatórios uniformes. Então, uma fonte confiável para gerar números aleatórios uniformes determina o sucesso de métodos estocásticos de inferência e de todo processo de simulação Monte Carlo.

2.1 Números Aleatórios Uniformes

Números uniformes aleatórios são aqueles que, a princípio, se situam dentro de um intervalo real, geralmente, entre 0 e 1, para os quais não podemos produzir uma sequência previsível de valores e cuja função densidade é constante. Em vários programas de computadores estes números são gerados utilizando o comando `random` ou comandos similares. Em Pascal, por exemplo, se este comando for utilizado com o argumento n, `random(n)`, números aleatórios inteiros U do intervalo $0 \leq U \leq n-1$ são gerados e se o argumento n não for usado, os números gerados são valores aleatórios reais do intervalo $[0, 1)$.

Em geral, os programas utilizam o método congruencial. Vamos considerar os números uniformes inteiros $U_1, U_2, U_3, \ldots$ entre 0 e $m-1$, em que m representa um grande número inteiro. Podemos gerar estes números utilizando o método congruencial por meio da relação recursiva:

$$U_{i+1} = (aU_i + c) \mod m \tag{2.1}$$

em que m é chamado de módulo, a e c são inteiros positivos denominados de multiplicador e incremento, respectivamente. O operador `mod` retorna o resto da divisão do argumento $(aU_i + c)$ por m. A sequência

recorrente (Equation 2.1) se repete em um período que não é maior que m, por razões óbvias. Se a, c e m são adequadamente escolhidos, a sequência tem tamanho máximo igual a m. A escolha do valor inicial U_0 é também muito importante. O valor do número uniforme correspondente no intervalo de 0 a 1 é dado por U_{i+1}/m, que é sempre menor que 1, mas podendo ser igual a zero.

Vamos apresentar um exemplo didático para ilustrar um gerador de números aleatórios. Sejam $U_0 = a = c = 7$ e $m = 10$, logo,

$$U_1 = (7 \times 7 + 7) \mod 10 = 56 \mod 10 = 6$$
$$U_2 = (7 \times 6 + 7) \mod 10 = 49 \mod 10 = 9$$

e assim sucessivamente. Obtemos a sequência de números aleatórios:

$$\{7, 6, 9, 0, 7, 6, 9, 0, 7, 6, 9, \cdots\}$$

e verificamos que o período é igual a 4, $\{7,6,9,0,\cdots\}$, que é menor do que $m = 10$.

Este método tem a desvantagem de ser correlacionado serialmente. Se m, a ou c não forem cuidadosamente escolhidos a correlação pode comprometer a sequência gerada. Por outro lado, o método tem a vantagem de ser muito rápido. Podemos perceber que a cada chamada do método, somente alguns poucos cálculos são executados. Escolhemos, em geral, o valor de m pelo maior inteiro que pode ser representado pela máquina de 32 bits, qual seja, 2^{32}. Um exemplo que foi utilizado por muitos anos nos computadores `IBM mainframe`, que representam uma péssima escolha é $a = 65.539$ e $m = 2^{31}$.

A correlação serial não é o único problema desse método. Os bits de maior ordem são mais aleatórios do que os bits de menor ordem (mais significantes). Devemos gerar inteiros entre 1 e 20 por $j = 1 + int(20 \times random(semente))$, ao invés de usar o método menos acurado $j = 1 + \mod (int(1000000 \times random(semente)),20)$, que usa bits de menor ordem. Existem fortes evidências, empíricas e teóricas, que o método congruencial

$$U_{i+1} = aU_i \mod m \tag{2.2}$$

é tão bom quanto o método congruencial com $c \neq 0$, se o módulo m e o multiplicador a forem escolhidos com cuidado [Press et al., 1992]. Park and Miller [1988] propuseram um gerador "padrão" mínimo baseado nas escolhas:

$$a = 7^5 = 16.807 \quad m = 2^{31} - 1 = 2.147.483.647 \tag{2.3}$$

Este gerador de números aleatórios não é perfeito, mas passou por todos os testes a qual foi submetido e tem sido usado como padrão para comparar e julgar outros geradores. Um problema que surge e que devemos contornar é que não é possível implementarmos diretamente em uma linguagem de alto-nível a equação (Equation 2.2) com as constantes de (Equation 2.3), pois o produto de a e U_i excede, em geral, o limite máximo de 32 bits para inteiros. Podemos usar um truque, devido a Schrage [1979], para multiplicar inteiros de 32 bits e aplicar o operador de módulo, garantindo portabilidade para implementação em praticamente todas as linguagens e todas as máquinas. O algoritmo de Schrage baseia-se na fatoração de m dada por:

$$m = aq + r; \qquad \text{i.e.,} \qquad q = \lfloor m/a \rfloor; \quad r = m \mod a$$

em que $\lfloor z \rfloor$ denota a parte inteira do número z utilizado como argumento. Para um número U_i entre 1 e $m - 1$ e para r pequeno, especificamente para $r < q$, Schrage [1979] mostrou que ambos $a(U_i \mod q)$ e $r\lfloor U_i/q \rfloor$ pertencem ao intervalo $0 \cdots m - 1$ e que

$$aU_i \mod m = \begin{cases} a(U_i \mod q) - r\lfloor U_i/q \rfloor & \text{se maior que 0} \\ a(U_i \mod q) - r\lfloor U_i/q \rfloor + m & \text{caso contrário.} \end{cases} \tag{2.4}$$

Computacionalmente observamos que a relação:

$$a(U_i \mod q) = a(U_i - q\lfloor U_i/q \rfloor)$$

se verifica. No Python pode-se optar por usar o operador `%` (`mod`), que retorna o resto da operação entre dois inteiros e o operador `//` (`div`), que retorna o resultado do dividendo para operações com inteiros. A quantidade $U_i \mod q = U_i - (U_i \text{ div } q) \times q$ pode ser obtida em Python simplesmente com $U_i \% q$. Atribuímos o resultado a uma variável qualquer definida como inteiro. Para aplicarmos o algoritmo de Schrage às constantes de (Equation 2.3) devemos usar os seguintes valores: $q = 127.773$ e $r = 2.836$.

A seguir apresentamos o algoritmo do gerador padrão mínimo de números aleatórios:

```
# gerador padrão mínimo de números aleatórios adaptado de Park and
# Miller. Retorna desvios aleatórios uniformes entre 0 e 1. Fazer
# "sem" igual a qualquer valor inteiro para iniciar a sequência;
# "sem" não pode ser alterado entre sucessivas chamadas da sequência
# se "sem" for zero ou negativo, um valor dependente do valor do relógio
# do sistema no momento da chamada é usado como semente. Constantes
# usadas a = 7^5 = 16.807; m = 2^31 - 1 = 2.147.483.647
# e c = 0

def  gnup(sem, q, a, r, m):
  k = sem // q # divisão por inteiros
  sem = a * (sem % q) - r * k
  if sem < 0:
    sem = sem + m
  u =  sem / m
  res = {'sem': sem, 'u': u}
  return res

from datetime import datetime # obter o data/horário do sistema

def gnap(n, sem = 0):
  a = 16807; m = 2147483647
  q = 127773; r = 2836
  mr = 1 / m
  if sem <= 0:
    t = datetime.now()
    sem = t.second+t.minute*60+t.hour*3600+t.day*86400
  u = []
  for i in range(n):
    x = gnup(sem, q, a, r, m)
    u.append(x['u'])
    sem = x['sem']
  return u
# Exemplos de uso
n = 5
x = gnap(n,0)
# Formatando a saída para 5 casas decimais
print(["%0.5f" % v for v in x])
#  especificando a semente
y = gnap(n, 1001)
# Formatando a saída para 5 casas decimais
print(["%0.5f" % v for v in y])
```

```
# tempo de execução para cada número aleatório
n = 100000
t1 = datetime.now()
x = gnap(n)
t2 = datetime.now()
t = t2-t1
print('tempo médio (micros): ',t.microseconds / n)
print('tempo total (micros): ',t.microseconds)
```

```
['0.59606', '0.02729', '0.61578', '0.35600', '0.35379']
['0.00783', '0.66933', '0.36093', '0.10878', '0.30000']
tempo médio (micros):  0.25909
tempo total (micros):  25909
```

Foi computado o tempo médio e o tempo total para rodar `100000` números aleatórios uniformes. Para capturar o tempo, foi usado o pacote datetime e também para obter a diferença de tempo entre o início e o fim do processamento de nossa função. O bloco para fazer isso deve ser executado de uma só vez, ou seja, marcando o bloco e teclando enter no `Positron`.

Algumas considerações a respeito desse algoritmo: a) a função gnap (gerador de números aleatórios mínima) retorna um número real entre 0 e 1. Este tipo de especificação determina que as variáveis possuam precisão dupla. A precisão dupla (`double-precision float`) possui números na faixa de $\pm 1{,}7{,}0 \times 10^{-308}$ $\cdot \pm 1{,}7 \times 10^{308}$, ocupa 24 bytes de memória e possui $15 - 16$ dígitos significantes; b) o valor da semente é definido pelo usuário e é passado como parâmetro para a função. Isso significa que a variável do programa que chama a função e que é passada como semente deve ser atualizada com o novo valor modificado em `gnup`. Se o seu valor inicial for zero ou negativo, a função atribui um inteiro dependente da hora do sistema no momento da chamada; c) a função tem dependência do pacote `datetime`, que foi usado para capturar a hora e dia do sistema.

A rotina é iniciada com os valores de n e da semente fornecidos pelo usuário. Se a semente for nula ou negativa, atribuímos um novo valor dependente do relógio do sistema no momento da chamada. A partir deste ponto o programa deve chamar reiteradas vezes a função `gnap`, que retorna o valor do número aleatório 0 e 1 utilizando o algoritmo descrito anteriormente, até que a sequência requerida pelo usuário seja completada. Nas sucessivas chamadas desta função, o valor da semente é sempre igual ao valor do último passo.

O período de `gnap` é da ordem de $2^{31} \approx 2{,}15 \times 10^9$, ou seja, a sequência completa é um pouco superior a 2 bilhões de números aleatórios. Assim, podemos utilizar `gnap` para alguns poucos propósitos práticos. Como já salientamos o gerador padrão mínimo de números aleatórios possui duas limitações básicas, quais sejam, sequência curta e correlação serial. Assim, como existem métodos para eliminar a correlação serial e que aumentam o período da sequência, recomendamos que sejam adotados. Claro que a função apresentada teve por objetivo ilustrar como podemos programar nossas próprias funções para gerarmos números aleatórios uniformes. O Python, no entanto, possui seu próprio gerador de números uniformes, que veremos na sequência. Um dos melhores e mais interessantes geradores de números aleatórios é o `Mersenne Twister` (MT). Mersenne Twister é um gerador de números pseudo-aleatórios desenvolvido por Makoto Matsumoto e Takuji Nishimura nos anos de 1996 e 1997 [Matsumoto and Nishimura, 1998]. O MT possui os seguintes méritos segundo seus desenvolvedores:

- foi desenvolvido para eliminar as falhas dos diferentes geradores existentes;
- possui a vantagem de apresentar o maior período e maior ordem de equidistribuição do que de qualquer outro método implementado. Ele fornece um período que é da ordem de $2^{19.937} - 1 \approx 4{,}3154 \times 10^{6001}$, e uma equidistribuição 623-dimensional;
- é um dos mais rápido geradores existentes, embora complexo;

- faz uso de forma muito eficiente da memória.

Existem muitas versões implementadas deste algoritmo, inclusive em Fortran e C e que estão disponíveis na internet. Felizmente, o Python já possui este algoritmo implementado. Por se tratar de um tópico mais avançado, que vai além do que pretendemos apresentar nestas notas de aulas, não descreveremos este tipo de procedimento para incorporações de funções escritas em outras linguagens.

2.2 Números Aleatórios Uniformes no Python

No Python podemos gerar números aleatórios uniformes contínuos utilizando uma função pré-programada. Os números aleatórios uniformes são gerados pelo comando `random.uniform(low=0.0, high=1.0, size=None)` da biblioteca `numpy`, em que `None` é para gerar apenas um valor, podendo ser substituído por n, entre `low` e `high` (excluído), que são argumentos que delimitam o valor mínimo e máximo da sequência a ser gerada. O controle da semente para se gerar uma sequência reproduzível de números uniformes é dada pelo comando `random.seed(seed=None)` do `numpy`, em que o argumento`seed` deve ser um número inteiro. O Python automaticamente determina a cada chamada uma nova semente. Conseguimos gerar diferentes sequências em cada chamada do comando gerador, sem nos preocuparmos com a semente aleatória. O gerador de números aleatórios uniformes usa o algoritmo Mersenne-Twister por padrão.

No programa apresentado a seguir ilustramos como podemos gerar n números aleatórios uniformes entre 0 e 1 de forma compacta, simples e eficiente:

```
import numpy as np
n = 5
x = np.random.uniform(low=0.0, high=1.0, size=n)
print(x)
# Fixando a semente
np.random.seed(seed=1000)
np.random.uniform(low=0.0, high=1.0, size=n)
np.random.seed(seed=1000)
np.random.uniform(low=0.0, high=1.0, size=n) # igual a anterior (mesma semente)
```

```
[0.25905951 0.43646754 0.35069046 0.51189972 0.2101137 ]
```

```
array([0.65358959, 0.11500694, 0.95028286, 0.4821914 , 0.87247454])
```

```
array([0.65358959, 0.11500694, 0.95028286, 0.4821914 , 0.87247454])
```

Fizemos um programa para comparar o tempo de execução das funções `gnap` e `random.uniform()` e retornamos o tempo médio para cada realização da variável aleatória. Desta forma verificamos que o algoritmo `random.uniform` é mais rápido de todos, conforme valores relativos apresentados a seguir. Obviamente temos que considerar que o algoritmo do `numpy` é uma função compilada. O algoritmo`gnap` por sua vez foram implementados em Python, que usa uma linguagem interpretada. As comparações de tempo podem ser vistas no programa a seguir. O programa utilizado foi:

```
# tempo de execução para cada número aleatório
# comparativo entre nosso gerador e o do np
n = 100000
t1 = datetime.now()
x = gnap(n)
t2 = datetime.now()
tgp = t2-t1
print('tempo médio gnap: ',tgp.microseconds / n)
t1 = datetime.now()
x = np.random.uniform(low=0.0, high=1.0, size=n)
```

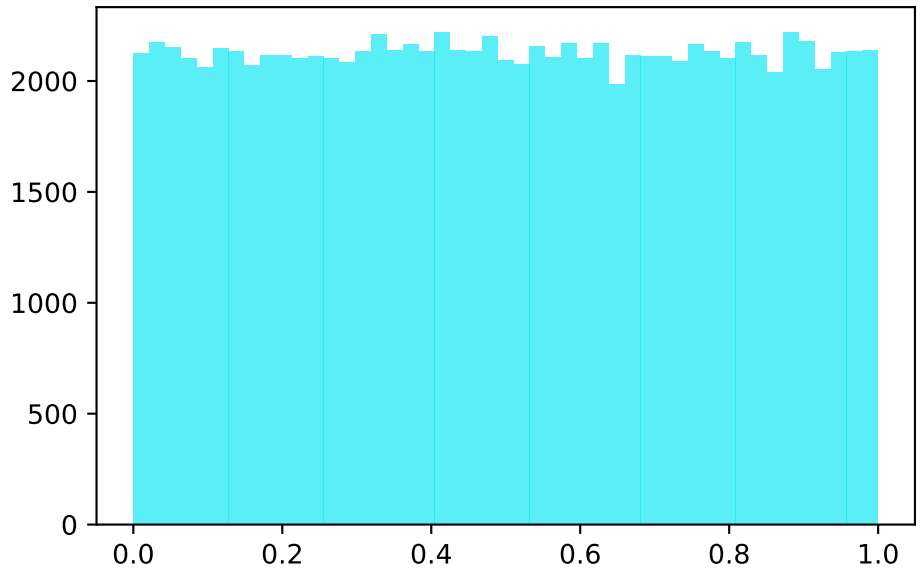
```
t2 = datetime.now()
tnp = t2-t1
print('tempo médio numpy: ',tnp.microseconds / n)
# tempo relativo
tgp / tnp
```

```
tempo médio gnap:  0.25557
tempo médio numpy:  0.00968
```

```
26.401859504132233
```

Podemos fazer um histograma usando o programa a seguir. Para isso usamos a biblioteca `matplotlib` e o resultado foi:

```
import matplotlib.pyplot  as plt
n = 100000
x = np.random.uniform(low=0.0, high=1.0, size=n) # gnap(n) troque para testar
graf = plt.hist(x, bins='auto', color='#14e8f3',rwidth=0.95,alpha=0.7)
```



2.3 Exercícios

1. Utilizar o gerador `gnap` para gerar n realizações de uma distribuição exponencial $f(x) = \lambda e^{-\lambda x}$. Sabemos do teorema da transformação de probabilidades, que se U tem distribuição uniforme, $X = F^{-1}(U)$ tem distribuição de probabilidade com densidade $f(x) = F'(x)$; em que $F(x) = \int_{-\infty}^{x} f(t)dt$ é a função de distribuição de X e $F^{-1}(y)$ é a sua função inversa para o valor y. Para a exponencial a função de distribuição de probabilidade é: $F(x) = \int_0^x \lambda e^{-\lambda t} dt = 1 - e^{-\lambda x}$. Para obtermos a função inversa temos que igualar u a $F(x)$ e resolver para x. Assim, $u = 1 - e^{-\lambda x}$ e resolvendo para x temos: $x = -\ln(1-u)/\lambda$. Devido à simetria da distribuição uniforme $1-u$ pode ser trocado por u. O resultado final é: $x = -\ln(u)/\lambda$. Para gerar números da exponencial basta gerar números uniformes e aplicar a relação $x = -\ln(u)/\lambda$. Fazer isso para construir uma função que gera n realizações exponenciais. Aplicar a função para obter amostras aleatórias da exponencial de tamanho $n = 100$ e obter o histograma da amostra simulada. Calcule a média e a variância e confronte com os valores teóricos da distribuição exponencial.

2. Para gerar números de uma distribuição normal, cuja densidade é dada por $f(x) = 1/(\sqrt{2\pi\sigma^2})\exp\{-(x-\mu)^2/(2\sigma^2)\}$, qual seria a dificuldade para podermos utilizar o teorema anunciado no exercício proposto anterior?

3. Como poderíamos adaptar o algoritmo apresentados nesse capítulo para gerar números aleatórios uniformes utilizando os valores propostos por Park e Miller, ou seja, $a = 48.271$ e $m = 2^{31} - 1$? Implementar o algoritmo, tomando cuidado em relação aos novos multiplicador q e resto r da fatoração de m?

4. Como você poderia propor um teste estatístico simples para avaliar a aleatoriedade da sequência de números uniformes gerados por esses algoritmos apresentados no capítulo? Implementar sua ideia.

Chapter 3

Variáveis Aleatórias Não-Uniformes

Neste capítulo vamos apresentar alguns métodos gerais para gerarmos realizações de variáveis aleatórias de outras distribuições de probabilidade, como, por exemplo, dos modelos exponencial, normal e binomial. Implementaremos algumas funções em Python e finalizaremos com a apresentação das rotinas otimizadas e já implementadas.

3.1 Introdução

Vamos estudar a partir deste instante um dos principais métodos, determinado pela lei fundamental de transformação de probabilidades, para gerarmos dados de distribuições de probabilidades contínuas ou discretas. Para alguns casos específicos, vamos ilustrar com procedimentos alternativos, que sejam eficientes e computacionalmente mais simples. Esta transformação tem como modelo fundamental a distribuição uniforme (0, 1). Por essa razão a geração de números uniformes contínuos é tão importante.

Veremos posteriormente nestas notas de aulas algoritmos para obtermos numericamente a função de distribuição $F(x)$ e a sua função inversa $x = F^{-1}(p)$, em que p pertence ao intervalo que vai de 0 a 1. Este conhecimento é fundamental para a utilização deste principal método.

Neste capítulo limitaremos a apresentar a teoria para alguns poucos modelos probabilísticos, para os quais podemos facilmente obter a função de distribuição de probabilidade e a sua inversa analiticamente. Para os modelos mais complexos, embora o método determinado pela lei fundamental de transformação de probabilidades seja adequado, apresentaremos apenas métodos alternativos, uma vez que, em geral, ele é pouco eficiente em relação ao tempo gasto para gerarmos cada realização da variável aleatória. Isso se deve ao fato de termos que obter a função inversa numericamente da função de distribuição de probabilidade dos modelos probabilísticos mais complexos.

3.2 Métodos Gerais para Gerar Realizações de Variáveis Aleatórias

Podemos obter realizações de variáveis aleatórias de qualquer distribuição de probabilidade a partir de números aleatórios uniformes. Para isso um importante teorema pode ser utilizado: o teorema fundamental da transformação de probabilidades.

Theorem 3.1 (Teorema fundamental da Transformação de Probabilidades)**.** *Sejam U uma variável uniforme $U(0,1)$ e X uma variável aleatória com densidade f e função de distribuição F contínua e invertível, então $X = F^{-1}(U)$ possui densidade f. Sendo F^{-1} a função inversa da função de distribuição F.*

Demonstração: Seja X uma variável aleatória com função de distribuição F e função densidade f. Se $u = F(x)$, então o jacobiano da transformação é $du/dx = F'(x) = f(x)$, em que U é uma variável aleatória uniforme $U(0,1)$, com função densidade $g(u) = 1$, para $0 < u < 1$ e $g(u) = 0$ para outros valores de u. Assim, a variável aleatória $X = F^{-1}(U)$ tem densidade f dada por:

$$f_X(x) = g(u)\left|\frac{du}{dx}\right| = g\left[F_X(x)\right] f(x) = f(x).$$

Em outras palavras a variável aleatória $X = F^{-1}(U)$ possui função densidade $f_X(x)$, estabelecendo o resultado almejado e assim, a prova fica completa. ■

Para variáveis aleatórias discretas, devemos modificar o teorema para podermos contemplar funções de distribuições F em escada, como são as funções de distribuição de probabilidades associadas a essas variáveis aleatórias.

Na Figura Figure 3.1 representamos como gerar uma realização de uma variável aleatória X com densidade f e função de distribuição F. Assim, basta gerarmos um número uniforme u_0 e invertermos a função de distribuição F neste ponto. Computacionalmente a dificuldade é obtermos analiticamente uma expressão para a função F^{-1} para muitos modelos probabilísticos. Em geral, essas expressões não existem e métodos numéricos são requeridos para inverter a função de distribuição. Neste capítulo vamos apresentar este método para a distribuição exponencial.

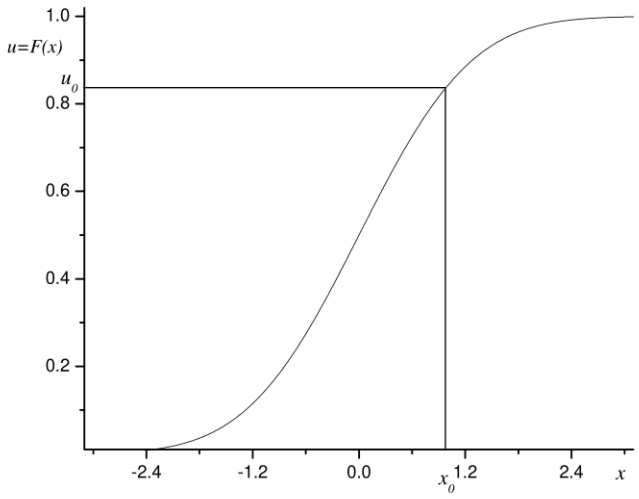


Figure 3.1: Ilustração do teorema fundamental da transformação de probabilidades para gerar uma variável aleatória X com densidade $f(x) = F'(x)$. A partir de um número aleatório uniforme u_0 a função de distribuição é invertida neste ponto para se obter x_0, com densidade $f(x)$.

Um outro método bastante geral que utilizaremos é denominado de método da amostragem por rejeição. Esse método tem um forte apelo geométrico. Procuraremos, a princípio, descrever esse método de uma forma bastante geral. Posteriormente, aplicaremos este método para gerarmos variáveis aleatórias de alguns modelos probabilístico. A grande vantagem deste método contempla o fato de não precisarmos obter a função de distribuição de probabilidade e nem a sua inversa. Estas estratégias só podem ser aplicadas em muitos dos modelos probabilísticos existentes, se utilizarmos métodos numéricos iterativos. Seja $f(x)$ a função densidade de probabilidade para a qual queremos gerar uma amostra aleatória. A área sob a curva para um intervalo qualquer de x corresponde à probabilidade de gerar um valor x nesse

intervalo. Se pudéssemos gerar um ponto em duas dimensões, digamos (X,Y), com distribuição uniforme sob a área, então a coordenada X teria a distribuição desejada.

Para realizarmos de uma forma eficiente a geração de variáveis aleatórias com densidade $f(x)$, evitando as complicações numéricas mencionadas anteriormente, poderíamos definir uma função qualquer $g(x)$. Essa função tem que ter algumas propriedades especiais para sua especificação. Deve possuir área finita e ter para todos os valores x densidade $g(x)$ superior a $f(x)$. Essa função é denominada de função de comparação. Outra característica importante que $g(x)$ deve ter é possuir função de distribuição $G(x)$ analiticamente computável e invertível, ou seja, $x = G^{-1}(u)$. Como a função $g(x)$ não é necessariamente uma densidade, vamos denominar a área sob essa curva no intervalo para x de interesse por $A = \int_{-\infty}^{\infty} g(x)dx$. Como G^{-1} é conhecida, podemos gerar pontos uniformes (x,y) que pertencem à área sob a curva $g(x)$ facilmente. Para isso basta gerarmos um valor de uma variável aleatória uniforme u_1 entre 0 e A e aplicarmos o teorema (Theorem 3.1). Assim, obtemos o primeiro valor do ponto (x_0,y_0) por $x_0 = G^{-1}(u_1)$. Para gerarmos a segunda coordenada do ponto não podemos gerar um valor de uma variável aleatória uniforme no intervalo de 0 a A, sob pena de gerarmos um ponto que não está sob a curva $g(x)$. Assim, calculamos o valor de g no ponto x_0 por $g(x_0)$. Geramos $y_0 = u_2$, sendo u_2 o valor de uma variável aleatória uniforme entre 0 e $g(x_0)$. Assim, obtemos um ponto (x_0,y_0) uniforme sob a curva $g(x)$. A dificuldade deste método é justamente estabelecer essa função $g(x)$ com as propriedades exigidas.

Vamos agora traçar as curvas correspondentes a $g(x)$ e $f(x)$ no mesmo gráfico. Se o ponto uniforme (x_0,y_0) está na área sob a curva $f(x)$, ou seja se $y_0 \leq f(x_0)$, então aceitamos x_0 como um valor válido de $f(x)$; se por outro lado o ponto estiver na região entre as densidades $f(x)$ e $g(x)$, ou seja se $f(x_0) < y_0 \leq g(x_0)$, então rejeitamos x_0. Uma forma alternativa de apresentarmos esse critério é tomarmos y_0 de uma distribuição $U(0,1)$ e aceitarmos ou rejeitarmos x_0 se $y_0 \leq f(x_0)/g(x_0)$ ou se $y_0 > f(x_0)/g(x_0)$, respectivamente. Ilustramos esse método na Figura Figure 3.2, sendo que a A representa a área total sob a curva $g(x)$.

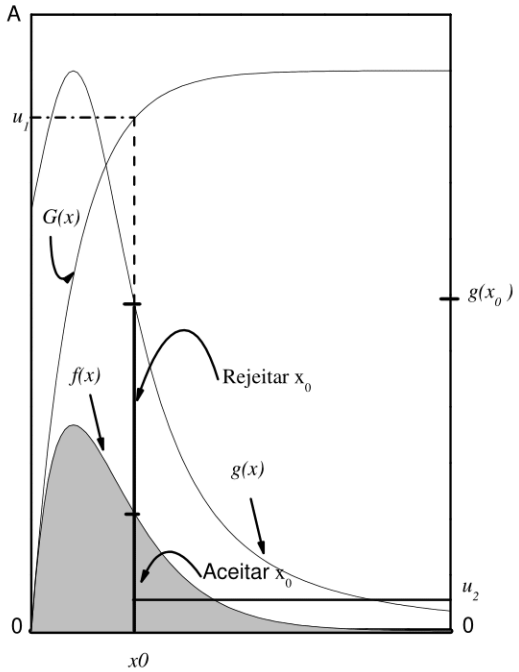


Figure 3.2: Método da rejeição para gerar um valor x_0 da variável aleatória X com função densidade $f(x)$ que é menor do que $g(x)$ para todo x. Nessa ilustração, x_0 deve ser aceito.

Vamos ilustrar o primeiro método e salientar que o segundo método é o que ocorre na maioria dos casos

de geração de variáveis aleatórias. A exponencial é uma distribuição de probabilidade em que facilmente podemos aplicar o teorema Theorem 3.1 para gerarmos amostras aleatórias. Assim, optamos por iniciar o processo de geração de números aleatórios nesta distribuição, uma vez que facilmente podemos obter a função de distribuição e a sua inversa. Seja X uma variável aleatória cuja densidade apresentamos por:

$$f(x) = \lambda e^{-\lambda x} \tag{3.1}$$

em que $\lambda > 0$ é o parâmetro da distribuição exponencial e $x > 0$.

A função de distribuição exponencial é dada por:

$$F(x) = \int_0^x \lambda e^{-\lambda t} dt$$

$$F(x) = 1 - e^{-\lambda x}. \tag{3.2}$$

A função de distribuição inversa $x = F^{-1}(u)$ é dada por:

$$x = F^{-1}(u) = \frac{-\ln(1-u)}{\lambda} \tag{3.3}$$

em que u é um número uniforme $(0, 1)$.

Devido à distribuição uniforme ser simétrica, podemos substituir $1 - u$ na equação (Equation 3.3) por u. Assim, para gerarmos uma realização de uma variável exponencial X, a partir de uma variável aleatória uniforme, utilizamos o teorema Theorem 3.1 por intermédio da equação:

$$x = \frac{-\ln(u)}{\lambda}. \tag{3.4}$$

O algoritmo Python para gerarmos realizações de variáveis aleatórias exponenciais é dado por:

```
# programa demonstrando a geração de n realizações de variáveis
# aleatórias exponenciais com parâmetro lamb, utilizamos
# a função np.random.uniform() para
# gerarmos números aleatórios uniformes
import numpy as np
def rexpon(n, lamb = 1.0):
  u = np.random.uniform(0.0, 1.0, n) # gera vetor u (U(0,1))
  x = -np.log(u) / lamb  # gera vetor x com distrib. exp.
  return x            # retorna o vetor x

# exemplo
rexpon(5, 0.1)
```

```
array([ 0.52764905,  1.77227152, 11.87125976,  1.44435612, 20.3013401 ])
```

Podemos utilizar a função pré-existente do Python (`np.random.exponential(scale=1.0, size=None)`) para realizarmos a mesma tarefa. Simplesmente digitamos `np.random.exponential(scale=10, size=5)` e teremos uma amostra aleatória de tamanho n de uma exponencial com parâmetro $\lambda = 1/\text{scale}$. O Python faz com que a estatística computacional seja ainda menos penosa e portanto acessível para a maioria dos pesquisadores.

3.3 Variáveis Aleatórias de Algumas Distribuições Importantes

Vamos descrever nesta seção alguns métodos específicos para gerarmos algumas realizações de variáveis aleatórias. Vamos enfatizar a distribuição normal. Apesar de o mesmo método apresentado na seção Section 3.2 poder ser usado para a distribuição normal, daremos ênfase a outros processos. Para utilizarmos o mesmo método anterior teríamos que implementar uma função parecida com `rexpon`, digamos `rnormal`. No lugar do comando `x = -log(u)/lamb` deveríamos escrever $x = \text{invnorm}(u) \times \sigma + \mu$, sendo que `invnorm(u)` é a função de distribuição normal padrão inversa. A distribuição normal padrão dentro da família normal, definida pelos seus parâmetros μ e σ^2, é aquela com média nula e variância unitária. Esta densidade será referenciada por $N(0,1)$. Essa deve ser uma função externa escrita pelo usuário. Os argumentos μ e σ da função são a média e o desvio padrão da distribuição normal que pretendemos gerar. A função densidade da normal é:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.5}$$

Nenhuma outra função utilizando o teorema Theorem 3.1 será novamente apresentada, uma vez que podemos facilmente adaptar as funções `rexpon` se tivermos um eficiente algoritmo de inversão da função de distribuição do modelo probabilístico alvo. A dificuldade deste método é a necessidade de uma enorme quantidade de cálculo para a maioria das densidades. Isso pode tornar ineficiente o algoritmo, pois o tempo de processamento é elevado.

Podemos ainda aproveitar a relação entre algumas funções de distribuições para gerarmos realizações de variáveis aleatórias de outras distribuições. Por exemplo, se X é normal com densidade (Equation 3.5), $N(\mu,\sigma^2)$, podemos gerar realizações de $Y = e^X$. Sabemos que fazendo tal transformação Y terá distribuição log-normal, cuja densidade com parâmetros de locação α (μ) e escala β (σ) é:

$$f(y) = \frac{1}{y\beta\sqrt{2\pi}} e^{-\frac{1}{2}\left[\frac{\ln(y)-\alpha}{\beta}\right]^2}, \qquad y > 0. \tag{3.6}$$

Um importante método usado para gerar dados da distribuição normal é o de Box-Müller, que é baseado na generalização do método da transformação de variáveis para mais de uma dimensão. Para apresentarmos esse método, vamos considerar p variáveis aleatórias X_1, $X_2, \ldots,$ X_p com função densidade conjunta $f(x_1,x_2,\ldots,x_p)$ e p variáveis Y_1, Y_2, ..., Y_p, funções de todos os X's, então a função densidade conjunta dos Y's é:

$$f(y_1,\ldots,y_p) = f(x_1,\ldots,x_p)\text{abs}\begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \cdots & \frac{\partial x_1}{\partial y_p} \\ \vdots & \ddots & \vdots \\ \frac{\partial x_p}{\partial y_1} & \cdots & \frac{\partial x_p}{\partial y_p} \end{vmatrix} \tag{3.7}$$

em que $J = |\partial()/\partial()|$ é o Jacobiano da transformação dos X's em relação aos Y's.

Vamos considerar a transformação (Equation 3.7) para gerar dados normais com densidade dadas por (Equation 3.5). Para aplicarmos a transformação de Box-Müller, vamos considerar duas variáveis aleatórias uniformes entre 0 e 1, representados por X_1 e X_2 e duas funções delas, representadas por Y_1 e Y_2 e dadas por:

$$\begin{cases} y_1 &= \sqrt{-2\ln x_1}\cos(2\pi x_2) \\ y_2 &= \sqrt{-2\ln x_1}\sin(2\pi x_2) \end{cases} \tag{3.8}$$

Ao explicitarmos X_1 e X_2 em (Equation 3.8) obtemos alternativamente:

$$\begin{cases} x_1 &= e^{-\frac{1}{2}\left(y_1^2+y_2^2\right)} \\ \\ x_2 &= \frac{1}{2\pi}\arctan\left(\frac{y_2}{y_1}\right) \end{cases} \tag{3.9}$$

Sabendo que a seguinte derivada $dk\arctan(g)/dx$ é dada por $k(dg/dx)/(1+g^2)$, em que g é uma função de X, então o Jacobiano da transformação é:

$$\begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = \begin{vmatrix} -y_1 e^{-0{,}5(y_1^2+y_2^2)} & -y_2 e^{-0{,}5(y_1^2+y_2^2)} \\ -\frac{y_2}{2\pi y_1^2\left(1+\frac{y_2^2}{y_1^2}\right)} & \frac{1}{2\pi y_1\left(1+\frac{y_2^2}{y_1^2}\right)} \end{vmatrix} = -\frac{1}{2\pi}e^{-0{,}5(y_1^2+y_2^2)} \tag{3.10}$$

Assim, a função densidade conjunta de Y_1 e Y_2 é dada por $f(x_1{,}x_2)|J|$, sendo

$$f(y_1{,}y_2) = \left[\frac{1}{\sqrt{2\pi}}e^{-\frac{y_1^2}{2}}\right]\left[\frac{1}{\sqrt{2\pi}}e^{-\frac{y_2^2}{2}}\right]. \tag{3.11}$$

Desde que a densidade conjunta de Y_1 e Y_2 é o produto de duas normais independentes, podemos afirmar que as duas variáveis geradas são normais padrão independentes, como pode ser visto em Johnson and Wichern [1998] e Ferreira [2018].

Assim, podemos usar esse resultado para gerarmos variáveis aleatórias normais. A dificuldade, no entanto, é apenas computacional. A utilização de funções trigonométricas como seno e cosseno pode limitar a performance do algoritmo gerado tornando-o lento. Um truque apresentado por Press et al. [1992] é bastante interessante para evitarmos diretamente o uso de funções trigonométricas. Esse truque representa uma melhoria do algoritmo de Box-Müller e é devido a Marsaglia and Bray [1964].

Ao invés de considerarmos os valores das variáveis aleatórias uniformes x_1 e x_2 de um quadrado de lado igual a 1 (quadrado unitário), tomarmos u_1 e u_2 como coordenadas de um ponto aleatório em um círculo unitário (de raio igual a 1). A soma de seus quadrados $R^2 = U_1^2 + U_2^2$ é uma variável aleatória uniforme que pode ser usada como X_1. Já o ângulo que o ponto (u_1, u_2) determina em relação ao eixo 1 pode ser usado como um ângulo aleatório dado por $\Theta = 2\pi X_2$. Podemos apontar que a vantagem da não utilização direta da expressão (Equation 3.8) refere-se ao fato do cosseno e do seno poderem ser obtidos alternativamente por: $\cos(2\pi x_2) = u_1/\sqrt{r^2}$ e $\sin(2\pi x_2) = u_2/\sqrt{r^2}$. Evitamos assim as chamadas de funções trigonométricas. Na Figura Figure 3.3} ilustramos os conceitos apresentados e denominamos o ângulo que o ponto $(u_1{,}u_2)$ determina em relação ao eixo u_1 por θ.

Agora podemos apresentar o função `boxmuller` para gerar dados de uma normal utilizando o algoritmo de Box-Müller. Essa função utiliza a função `polar` para gerar dois valores aleatórios, de variáveis aleatórias independentes normais padrão Y_1 e Y_2. A função `boxmuller` é:

```
# Função boxmuller retorna uma amostra de tamanho n de
# uma distribuição normal com média mu e variância sigma^2
# utilizando o método Polar Box-Müller

import matplotlib.pyplot as plt
def polar():
  r2 = -1
  while r2 <= 0 or r2 >= 1:
    u = np.random.uniform(-1.0,1.0,2)
    r2 = u[0]**2+u[1]**2
  y = (-2 * np.log(r2) / r2)**0.5 * u
  return y
```

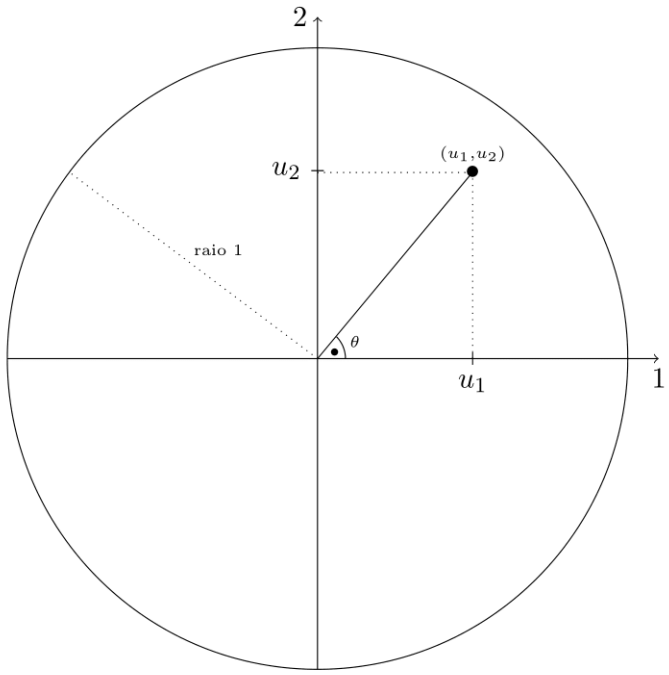
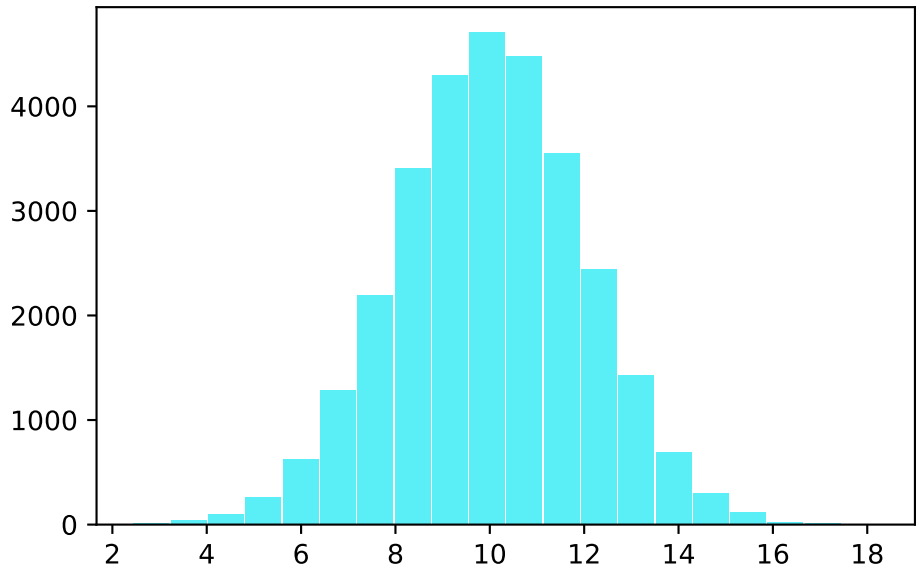


Figure 3.3: Círculo unitário mostrando um ponto aleatório (u_1, u_2) com $r^2 = u_1^2 + u_2^2$ representando x_1 e θ o ângulo que o ponto (u_1, u_2) determina em relação ao eixo 1. No exemplo, o ponto está situado no círculo unitário, conforme é exigido.

```
def boxmuller(n, mu = 0, sigma = 1):
  if n % 2 == 0: # n é par
    k = n // 2
    for i in range(k):
      if i == 0:
         x = polar()
      else:
        x = np.append(x,polar())
  else: # n é ímpar
    k = n // 2
    if k == 0:
      x = polar()[0]
    else:
      for i in range(k):
        if i == 0:
         x = polar()
        else:
         x = np.append(x,polar())
      x = np.append(x,polar()[0])
  x = x * sigma + mu
  return x

n = 30000
x = boxmuller(n, 10, 2)
```

```
graf = plt.hist(x, bins=20, color='#14e8f3',rwidth=0.95,alpha=0.7)
plt.show()
```



Algumas aproximações são apresentadas em Atkinson and Pearce [1976] e serão apenas descritas na sequência. Uma das aproximações faz uso da densidade Tukey-lambda e aproxima a normal igualando os quatro primeiros momentos. Esse algoritmo, além de ser uma aproximação, tem a desvantagem de utilizar a exponenciação que é uma operação lenta. Utilizando essa aproximação podemos obter uma variável normal X a partir de uma variável uniforme $U \sim U(0,1)$ por:

$$X = [U^{0,135} - (1-U)^{0,135}]/0{,}1975.$$

Outro método é baseado na soma de 12 ou mais variáveis uniformes ($U_i \sim U(0,1)$) independentes. Assim, a variável $X = \sum_i^{12} U_i - 6$ tem distribuição aproximadamente normal com média 0 e variância 1. Isso ocorre em decorrência do teorema do limite central e em razão de cada uma das 12 variáveis uniformes possuírem média 1/2 e variância 1/12.

Estas duas aproximações tem valor apenas didático e não devem ser recomendadas como uma forma de gerar variáveis normais. Muitas vezes esse fato é ignorado em problemas que requerem elevada precisão e confiabilidade dos resultados obtidos. Quando isso acontece conclusões incorretas ou no mínimo imprecisas podem ser obtidas.

De uma forma geral temos

$$X = \frac{\sum_{i=1}^{k} U_i - \frac{k}{2}}{\sqrt{\frac{k}{12}}} \sim N(0,1),$$

pois $E(U_i) = (b-a)/2$ e $V(U_i) = (b-a)^2/12$, em que k é o número de uniformes que devem ser somadas para cada realização da variável normal. A seguir, implementamos essas duas aproximações para fins de treinamento em criação de funções com o Python.

A primeira é a Tukey-Lambda:

```
# Aproximação baseada na Tukey-Lambda
def tukeylambda(n, mu = 0, sigma = 1):
  x = np.random.uniform(0.0,1.0,n)
  x = (x**0.135 -(1-x)**0.135) / 0.1975
  x = x * sigma  + mu
  return x

n = 1000000
x = tukeylambda(n, 10,10)
np.mean(x)
np.std(x)
```

```
np.float64(9.990788357702453)
```

```
np.float64(9.996598992292157)
```

A segunda é a aproximação da soma de `k` uniformes, usando o teorema do limite central:

```
# Aproximação baseada na soma de k
# uniformes
def soma_un(u, k):
  if k == 12:
    rn = np.sum(u) - 6
  else:
    rn  = (np.sum(u) - k / 2) / (k / 12)**0.5
  return rn

def norm_tlc(n, mu = 0, sigma = 1, k = 12):
  x = []
  for i in range(n):
    u = np.random.uniform(0.0,1.0,k)
    x. append(soma_un(u, k))
  x = np.asarray(x) * sigma  + mu
  return x

n = 100000
k = 12
x = norm_tlc(n, 10, 10, k)
np.mean(x)
np.std(x)
```

```
np.float64(10.012679858038275)
```

```
np.float64(9.988686420192224)
```

3.4 Distribuição Binomial

Vamos ilustrar a partir da binomial a geração de realizações de variáveis aleatórias discretas. A distribuição binomial é surpreendentemente importante nas aplicações da estatística. Esta distribuição aparece nas mais variadas situações reais e teóricas. Por exemplo, o teste não-paramétrico do sinal utiliza a distribuição binomial, a ocorrência de animais doentes em uma amostra de tamanho n pode ser muitas vezes modelada pela distribuição binomial. Inúmeros outros exemplos poderiam ser citados. A distribuição binomial é a primeira distribuição discreta de probabilidade, entre as distribuições já estudadas. A variável aleatória X com distribuição de probabilidade binomial tem a seguinte função de probabilidade:

$$P(X = x) = \binom{n}{x} p^x (1-p)^{n-x}, \qquad x = 0,1,\cdots,n \quad \text{e} \quad n \geq 1, \tag{3.12}$$

em que os parâmetros n e p referem-se, respectivamente, ao tamanho da amostra e a probabilidade de sucesso de se obter um evento favorável em uma amostragem de 1 único elemento ao acaso da população. O termo $\binom{n}{x}$ é o coeficiente binomial definido por:

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}.$$

A probabilidade de sucesso p, em amostras de tamanho n da população, ou seja, em n ensaios de Bernoulli, deve permanecer constante e os sucessivos ensaios devem ser independentes. O parâmetro n em geral é determinado pelo pesquisador e deve ser um inteiro maior ou igual a 1. Se $n = 1$, então a distribuição binomial se especializa na distribuição Bernoulli. Assim, a distribuição binomial é a repetição de n ensaios Bernoulli independentes e com probabilidade de sucesso constante. Os seguintes teoremas e lemas são importantes, para a definição de alguns métodos que apareceram na sequência. Estes lemas serão apresentados sem as provas.

Theorem 3.2 (Gênese)**.** *Seja X o número de sucessos em uma sequência de n ensaios Bernoulli com probabilidade de sucesso p, ou seja,*

$$X = \sum_{i=1}^{n} I(U_i \leq p)$$

em que U_1, U_2,$\cdots$, U_n são variáveis uniformes $(0,1)$ i.i.d. e $I(\bullet)$ é uma função indicadora. Então, X tem distribuição binomial (n,p).

Lemma 3.1 (Soma de binomiais)**.** *Se X_1, X_2, $\cdots$, X_k são variáveis aleatórias binomiais independentes com (n_1,p), $\cdots$, (n_k,p), então $\sum_{i=1}^{k} X_i$ tem distribuição binomial, com parâmetros $(\sum_{i=1}^{k} n_i, p)$.*

Lemma 3.2 (Tempo de Espera - Propriedade 1)**.** *Sejam G_1, G_2, $\cdots$ variáveis aleatórias geométricas independentes e, X, o menor inteiro tal que*

$$\sum_{i=1}^{X+1} G_i > n.$$

Assim, X tem distribuição binomial (n,p).

As variáveis geométricas citadas no lema Lemma 3.2 são definidas a seguir. Se G tem distribuição geométrica com probabilidade de sucesso constante $p \in (0,1)$, então, a função de probabilidade é:

$$P(G = g) = p(1-p)^{g-1}, \qquad g = 1,2\cdots \tag{3.13}$$

A geométrica é a distribuição do tempo de espera até a ocorrência do primeiro sucesso no g-ésimo evento, numa sequência de ensaios Bernoulli independentes, incluindo o primeiro sucesso. Assim, supõe-se que venham a ocorrer $g-1$ fracassos, cada um com probabilidade de ocorrência constante $1-p$, antes da ocorrência de um sucesso no g-ésimo ensaio, com probabilidade p. Finalmente, o segundo lema do tempo de espera pode ser anunciado por:

Lemma 3.3 (Tempo de Espera - Propriedade 2)**.** *Sejam E_1, E_2, $\cdots$ variáveis aleatórias exponenciais i.i.d., e X o menor inteiro tal que*

$$\sum_{i=1}^{X+1} \frac{E_i}{n-i+1} > -ln(1-p).$$

Logo, X tem distribuição binomial (n,p).

As propriedades especiais da distribuição binomial, descritas no teorema e nos lemas, formam a base para dois algoritmos binomiais. Estes dois algoritmos são baseados na propriedade de que uma variável aleatória binomial é á soma de n variáveis Bernoulli obtidas em ensaios independentes e com probabilidade de sucesso constante p. O algoritmo binomial mais básico baseia-se na geração de n variáveis independentes $U(0,1)$ e no cômputo do total das que são menores ou iguais a p. Este algoritmo denominado por Kachitvichyanukul and Schmeiser [1988] de BU é dado por:

1. Faça $x = 0$ e $k = 0$
2. Gere u de uma $U(0,1)$ e faça $k = k + 1$
3. Se $u \leq p$, então faça $x = x + 1$
4. Se $k < n$ vá para o passo 2
5. Retorne x de uma binomial (n, p)

O algoritmo BU tem velocidade proporcional a n e depende da velocidade do gerador de números aleatórios, mas possui a vantagem de não necessitar de variáveis de `setup`. O segundo algoritmo atribuído a Devroy 1980 [Devroy, 1980] é denominado de BG e é baseado no lema Lemma 3.2. O algoritmo BG pode ser descrito por:

1. Faça $y = 0$, $x = 0$ e $c = ln(1 - p)$
2. Se $c = 0$, vá para o passo 6
3. Gere u de uma $U(0,1)$
4. $y = y + \lfloor \ln(u)/c \rfloor + 1$, em que $\lfloor \bullet \rfloor$ denotam a parte inteira do argumento $\bullet$
5. Se $y \leq n$, faça $x = x + 1$ e vá para o passo 3
6. Retorne x de uma binomial (n, p)

O algoritmo utilizado no passo 4 do algoritmo BG é baseado em truncar uma variável exponencial para gerar uma variável geométrica, conforme descrição feita por Devroy [1986]. O tempo de execução desse algoritmo é proporcional a np, o que representa uma considerável melhoria da performance. Assim, $p > 0{,}5$, pode-se melhorar o tempo de execução de BG explorando a propriedade de que se X é binomial com parâmetro n e p, então $n - X$ é binomial com parâmetros n e $1 - p$. Especificamente o que devemos fazer é substituir p pelo $\min(p,1-p)$ e retornar x se $p \leq \frac{1}{2}$ ou retornar $n - x$, caso contrário. A velocidade, então, é proporcional a n vezes o valor $\min(p,1 - p)$. A desvantagem desse procedimento é que são necessárias várias chamadas do gerador de realizações de variáveis aleatórias uniformes, até que um sucesso seja obtido e o valor x seja retornado. Uma alternativa a esse problema pode ser conseguida se utilizarmos um gerador baseado na inversão da função de distribuição binomial.

Como já havíamos comentado em outras oportunidades o método da inversão é o método básico para convertermos uma variável uniforme U em uma variável aleatória X, invertendo a função de distribuição. Para uma variável aleatória contínua, temos o seguinte procedimento:

- Gerar um número uniforme u
- Retornar $x = F^{-1}(u)$

O procedimento análogo para o caso discreto requer a busca do valor x, tal que:

$$F(x-1) = \sum_{i<x} P(X = i) < u \leq \sum_{i \leq x} P(X = i) = F(x).$$

A maneira mais simples de obtermos uma solução no caso discreto é realizarmos uma busca sequencial a partir da origem. Para o caso binomial, este algoritmo da inversão, denominado de BINV, pode ser implementado se utilizarmos a fórmula recursiva:

$$\begin{cases} P(X = 0) & = (1 - p)^n \\ P(X = x) & = P(X = x - 1)\frac{n-x+1}{x}\frac{p}{1-p} \end{cases} \tag{3.14}$$

para $x = 1,2,\cdots,n$ da seguinte forma:

1. Faça $pp = \min(p,1-p)$, $qq = 1-pp$, $r = pp/qq$, $g = r(n+1)$ e $f = qq^n$
2. Gere u de uma $U(0,1)$ e faça $x = 0$
3. Se $u \leq f$, então vá para o passo 5
4. Faça $u = u - f$, $x = x + 1$, $f = f\left(\frac{g}{x} - r\right)$ e vá para o passo 3
5. Se $p \leq \frac{1}{2}$, então retorne x de uma binomial (n,p), senão retorne $n - x$ de uma binomial (n,p)

A velocidade deste algoritmo é proporcional a n vezes o valor $\min(p,1-p)$. A vantagem desse algoritmo é que apenas uma variável aleatória uniforme é gerada para cada variável binomial requerida. Um ponto importante é o tempo consumido para gerar qq^n é substancial e dois problemas potenciais podem ser destacados. O primeiro é a possibilidade de *underflow* no cálculo de $f = qq^n$, quando n é muito grande e, o segundo, é a possibilidade do cálculo recursivo de f ser uma fonte de erros de arredondamento, que se acumulam e que se tornam sérios na medida que n aumenta [Kachitvichyanukul and Schmeiser, 1988]. Devroy [1986] menciona que o algoritmo do tempo de espera BG baseado no lema Lemma 3.2 deve ser usado no lugar de BINV para evitarmos esses fatos. Por outro lado, Kachitvichyanukul and Schmeiser [1988] mencionam que basta implementar o algoritmo em precisão dupla que esses problemas são evitados.

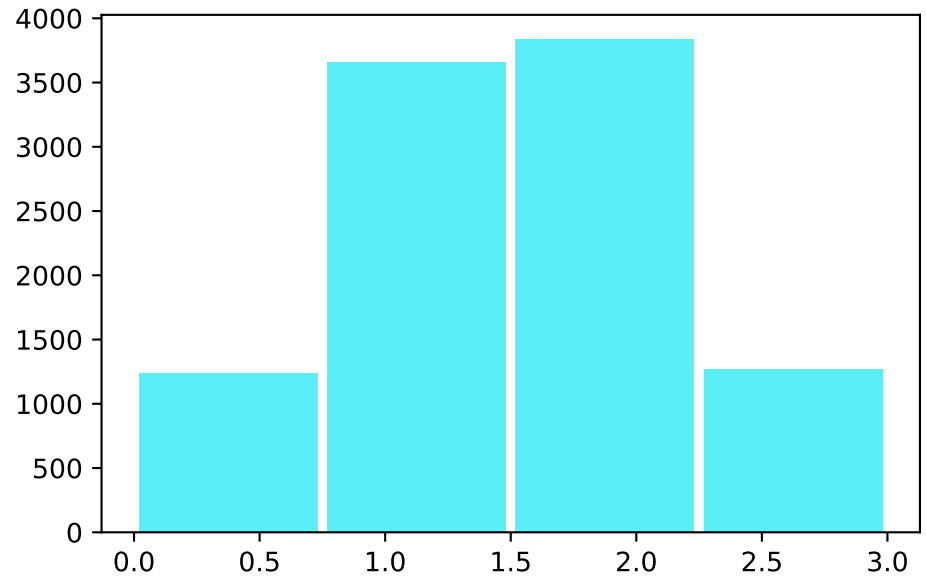
```
# Exemplificação de algoritmos para gerar realizações de
# variáveis aleatórias binomiais binom(n, p).
# Os algoritmos BU, BG e BINV foram implementados
# Exemplificação de algoritmos para gerar realizações de
# variáveis aleatórias binomiais binom(n, p).
# Os algoritmos BU, BG e BINV foram implementados
import numpy as np
def bu(n, p):
  x = 0
  k = 0
  while k < n:
    u = np.random.uniform(0.0,1.0,1)
    k = k + 1
    if u <= p:
      x += 1
  return(x)

def bg(n, p):
  if p > 0.5:
    pp = 1 - p
  else:
    pp = p
  y = 0
  x = 0
  c = np.log(1 - pp)
  if c < 0:
    while y <= n:
       u = np.random.uniform(0.0,1.0,1)
       y += np.trunc(np.log(u) /c) + 1
       if y <= n:
        x += 1
    if p > 0.5:
      x = n - x
  return x
```

```
def binv(n, p):
  if p > 0.5:
    pp = 1 - p
  else:
    pp = p
  q = 1 - pp
  f = q**n
  r = pp / q
  g = r * (n + 1)
  u = np.random.uniform(0.0,1.0,1)
  x = 0
  Fx = f
  while Fx < u:
    x += 1
    f *= (g / x - r)
    Fx += f
  if p > 0.5:
    x = n - x
  return x

# gerador de uma amostra binomial n, p
# recebe uma das três funções implementas
# como argumento, size e prob
# são os parâmetros da binomial e n é o
# tamanho da amostra ser gerada
# recebe bg por default
def rbinom(n, size, prob, func = bg):
  x = []
  for i in range(n):
    x.append(func(size, prob))
  return x

# Exemplo
prob = 0.5
size = 3
bu(size, prob)
bg(size, prob)
binv(size, prob)
n = 10000 # sample size
x = rbinom(n, size, prob, bg) # pode trocar bg por binv ou bu
graf = plt.hist(x,bins=size+1,color='#14e8f3',rwidth=0.95,alpha=0.7)
```

1

2

2



Procedimentos de geração de números aleatórios poderiam ser apresentados para muitas outras distribuições de probabilidades. Felizmente no python não temos este tipo de preocupação, pois estas rotinas já existem e estão implementadas em linguagem não interpretada. Inclusive para os modelos considerados temos rotinas prontas. Veremos uma boa parte delas na próxima seção.

3.5 Rotinas Python para Geração de Realizações de Variáveis Aleatórias

Nesta seção, veremos alguns dos principais comandos Python para acessarmos os processos de geração de realizações de variáveis aleatórias de diferentes modelos probabilísticos. Os modelos probabilísticos contemplados pelo R estão apresentados na Tabela Table 3.1.

Table 3.1: Distribuições de probabilidades, nome Python (`np.random.col2nome`) e parâmetros (argumentos da função) dos principais modelos probabilístico, sendo `size` o tamanho da amostra.

Distribuição	Nome `numpy`	Parâmetros
beta	beta(*a*, *b*, *size=None*)	a, b, size
binomial	binomial(*n*, *p*, *size=None*)	n, p,size
Cauchy Padrão	standard_cauchy(*size=None*)	size
qui-quadrado	chisquare(*df*, *size=None*)	df, size
exponencial	exponential(*scale=1.0*, *size=None*)	scale (1/lambda), size
F	f(*dfnum*, *dfden*, *size=None*)	dfnum, dfden, size
gama	gamma(*shape*, *scale=1.0*, *size=None*)	shape, scale (1/beta), size
geométrica	geometric(*p*, *size=None*)	p, size
hipergeométrica	hypergeometric(*ngood*, *nbad*, *nsample*, *size=None*)	ngood, nbad, nsample, size
log-normal	lognormal(*mean=0.0*, *sigma=1.0*, *size=None*)	mean, sigma, size

Distribuição	Nome `numpy`	Parâmetros
logística	logistic(*loc=0.0*, *scale=1.0*, *size=None*)	loc, scale, size
binomial negativa	negative_binomial(*n*, *p*, *size=None*)	n, p, size
normal	normal(*loc=0.0*, *scale=1.0*, *size=None*)	loc, scale, size
Poisson	poisson(*lam=1.0*, *size=None*)	lam, size
t de Student	standard_t(*df*, *size=None*)	df, size
uniform	uniform(*low=0.0*, *high=1.0*, *size=None*)	low, high, size
Weibull	weibull(*a*, *size=None*)	a, size

No Python, a biblioteca `scipy.stats` nos permite realizarmos os cálculos da função de probabilidade (caso discreto) com o comando, por exemplo, para a binomial, de `sp.stats.binom.pmf(x, n,p)`. Neste caso, o `scipy`foi importado com `sp` e `pmf` é acrônimo de probability mass function, do inglês. Para o caso contínuo, usamos `pdf`. Para a função de distribuição, usamos `cdf`, sejam as variáveis contínuas ou discretas. Para a inversa da função de distribuição, usamos `ppf`, percent point function. Neste caso o primeiro argumento é o valor da probabilidade acumulada. No caso particular deste capítulo temos interesse na geração de realizações de variáveis aleatórias. Existem ainda modelos probabilísticos não centrais, para os quais devemos utilizar o parâmetro de não-centralidade (ncp). Para mais detalhes, visite a página do `scipy.stats` clicando aqui.

O uso destas funções para gerarmos números aleatórios é bastante simples. Se, por exemplo, quisermos gerar dados de uma distribuição beta com parâmetros $\alpha = 1$ e $\beta = 2$, podemos utilizar o programa ilustrativo apresentado na sequência. Podemos utilizar funções semelhantes a função `beta`, de acordo com a descrição feita na Tabela Table 3.1, para gerarmos n dados de qualquer outra função densidade ou função de probabilidade. Este procedimento é mais eficiente do que utilizarmos nossas próprias funções, pois estas funções foram implementadas em geral em C. Se para algum modelo particular desejarmos utilizar nossas próprias funções e se iremos chamá-las milhares ou milhões de vezes é conveniente que implementemos em C e as associemos ao Python. A forma de associarmos as rotinas escritas em C ao Python foge do escopo deste material e por isso não explicaremos como fazê-lo.

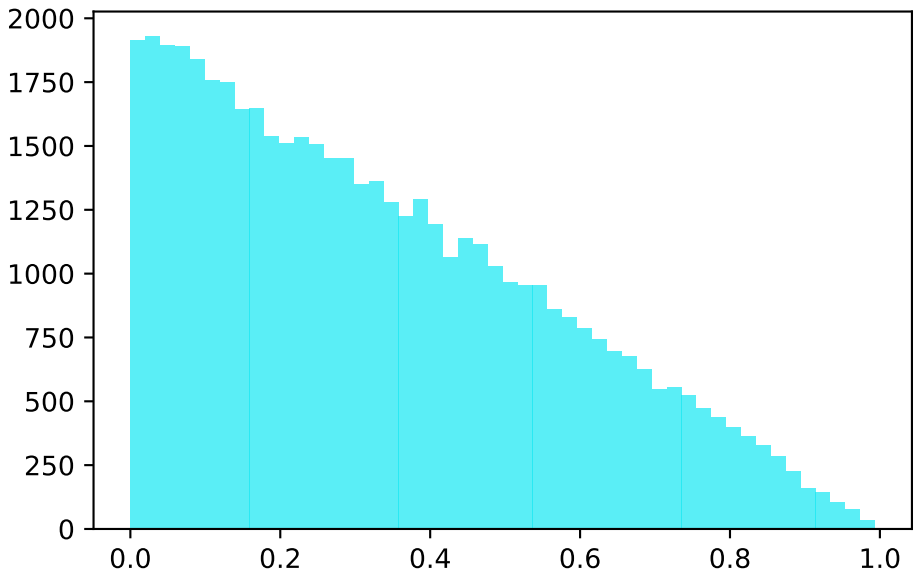
```
n = 50000
alpha = 1.0
beta = 2.0
x = np.random.beta(alpha, beta, n)
graf = plt.hist(x, bins='auto', color='#14e8f3',rwidth=0.95,alpha=0.7)
np.mean(x)
alpha/(alpha+beta) # verdadeira
np.var(x)
alpha*beta/((alpha+beta)**2*(alpha+beta+1)) # verdadeira
```

```
np.float64(0.33427782244226617)
```

```
0.3333333333333333
```

```
np.float64(0.05553407156478516)
```

```
0.05555555555555555
```



3.6 Exercícios

1. Seja $f(x) = 3x^2$ uma função densidade de uma variável aleatória contínua X com domínio definido no intervalo $[0; 1]$. Aplicar o método da inversão e descrever um algoritmo para gerar realizações de variáveis aleatórias dessa densidade. Implementar em R e gerar uma amostra de tamanho $n = 1.000$. Estimar os quantis 1%, 5%, 10%, 50%, 90%, 95% e 99%. Confrontar com os quantis teóricos.

2. Os dados a seguir referem-se ao tempo de vida, em dias, de $n = 40$ insetos. Considerando que a distribuição do tempo de vida é a exponencial e que o parâmetro λ pode ser estimado pelo estimador de máxima verossimilhança $\hat{\lambda} = 1/\bar{X}$, em que $\bar{X} = \sum_{i=1}^{n} X_i/n$, obter o intervalo de 95% de confiança utilizando o seguinte procedimento: i) gerar uma amostra da exponencial de tamanho $n = 40$, utilizando o algoritmo `rexpon`, considerando o parâmetro igual a estimativa obtida; ii) determinar a estimativa da média $\mu = 1/\lambda$ por $\bar{X}$ nesta amostra simulada de tamanho $n = 40$; iii) repetir 1.000 vezes os passos (i) e (ii) e armazenar os valores obtidos; iv) ordenar as estimativas e tomar os quantis 2,5% e 97,5%. Os valores obtidos são o intervalo de confiança almejado, considerando como verdadeira a densidade exponencial para modelar o tempo de vida dos insetos. Este procedimento é denominado de bootstrap paramétrico. Os dados em dias do tempo de vida dos insetos são:

8,521	4,187	2,516	1,913	8,780	5,912	0,761	12,037
2,604	1,689	5,626	6,361	5,068	3,031	1,128	1,385
12,578	2,029	0,595	0,445	3,601	7,829	1,383	1,934
0,864	8,514	4,977	0,576	1,503	0,475	1,041	0,301
1,781	2,564	5,359	2,307	1,530	8,105	3,151	8,628

   Repetir esse processo, gerando 100.000 amostras de tamanho $n = 40$. Compare os resultados e verifique se o custo adicional de ter aumentado o número de simulações compensou a possível maior precisão obtida.

3. Gerar uma amostra de $n = 5.000$ realizações de variáveis normais padrão utilizando as aproximações: $X = [U^{0,135} - (1 - U)^{0,135}]/\ 0{,}1975$ e da soma de 12 ou mais variáveis uniformes ($U_i \sim U(0{,}1)$) independentes, dada por $X = \sum_{i}^{12} U_i - 6$. Confrontar os quantis 1%, 5%, 10%, 50%, 90%, 95% e

99% esperados da distribuição normal com os estimados dessa distribuição. Gerar também uma amostra de mesmo tamanho utilizando o algoritmo Polar-Box-Müller. Estimar os mesmos quantis anteriores nesta amostra e comparar com os resultados anteriores.

4. Se os dados do exercício 2 pudessem ser atribuídos a uma amostra aleatória da distribuição log-normal, então estimar os parâmetros da log-normal e utilizar o mesmo procedimento descrito naquele exercício, substituindo apenas a distribuição exponencial pela log-normal para estimar por intervalo a média populacional. Para estimar os parâmetros da log-normal utilizar o seguinte procedimento: a) transformar os dados originais, utilizando $X_i^* = \ln(X_i)$; b) determinar a média e o desvio padrão amostral dos dados transformados - estas estimativas são as estimativas de μ e σ. Utilizar estas estimativas para gerar amostras log-lognormais. Realizar os mesmos procedimentos descritos para exponencial, confrontar os resultados e discutir a respeito da dificuldade de se tomar uma decisão da escolha da distribuição populacional no processo de inferência. Como em situações reais nunca se sabe de qual distribuição os dados são provenientes com precisão, então você teria alguma ideia de como fazer para determinar qual a distribuição que melhor modela os dados do tempo de vida dos insetos? Justificar sua resposta adequadamente com os procedimentos numéricos escolhidos.

5. Fazer reamostragens com reposição a partir da amostra do exercício 2 e estimar o intervalo de 95% para a média populacional, seguindo os passos descritos a seguir e utilizar um gerador de números uniformes para determinar quais elementos amostrais devem ser selecionados: i) reamostrar com reposição os $n = 40$ elementos da amostra original e compor uma nova amostra por: X_i^*; ii) calcular a média desta nova amostra por $\bar{X}^* = \sum_{i=1}^{n} X_i^*/n$; iii) armazenar este valor e repetir os passos (i) e (ii) $B - 1$ vezes; iv) agrupar os valores com o valor da amostra original; e v) ordenar os valores obtidos e determinar os quantis 2,5% e 97,5% desse conjunto de B valores. Escolher $B = 1.000$ e $B = 100.000$ e confrontar os resultados obtidos com os obtidos nos exercícios anteriores para a distribuição exponencial e log-normal. Os resultados que estiverem mais próximos deste resultado devem fornecer um indicativo da escolha da distribuição mais apropriada para modelar o tempo de vida de insetos. Este procedimento sugerido neste exercício é o bootstrap não-paramétrico. A sua grande vantagem é não precisar fazer suposição a respeito da distribuição dos dados amostrais.

6. Uma importante relação para obtermos intervalos de confiança para a média de uma distribuição exponencial, $f(x) = \lambda e^{-\lambda x}$, refere-se ao fato de que a soma de n variáveis exponenciais com parâmetro λ é igual a uma gama $f(y) = \frac{1}{\lambda\Gamma(\alpha)} (y/\beta)^{\alpha-1} e^{-\frac{y}{\beta}}$ com parâmetros $\alpha = n$ e $\beta = 1/\lambda$. Assim, assumir que os dados do exercício 2 têm distribuição exponencial com parâmetro λ estimado pelo recíproco da média amostral, ou seja, $\hat{\beta} = 1/\bar{X}$. Considerando que a variável X tem distribuição gama padrão com parâmetro $\alpha = n$, então obtenha $Y = \hat{\beta}X = X/\hat{\lambda}$. Neste caso $Y|\hat{\beta}$ tem distribuição da soma de n variáveis exponenciais, ou seja, distribuição gama com parâmetros $\alpha = n$ e $\beta = \hat{\beta}$. Como queremos a distribuição da média, devemos obter a transformação $\bar{Y} = Y/n$. Gerar amostras de tamanho $n = 1.000$ e $n = 100.000$ e estimar os quantis 2,5% e 97,5% da distribuição de $\bar{Y}$, em cada uma delas. Confrontar os intervalos de confiança, dessa forma obtidos, com os do exercício 2. Quais são as suas conclusões? Qual é a vantagem de utilizar a distribuição gama?

7. Duas amostras binomiais foram realizadas em duas (1 e 2) diferentes populações. Os resultados do número de sucesso foram $y_1 = 2$ e $y_2 = 3$ em amostras de tamanho $n_1 = 12$ e $n_2 = 14$, respectivamente, de ambas as populações. Estimar os parâmetros p_1 e p_2 das duas populações por: $\hat{p_1} = y_1/n_1$ e $\hat{p_2} = y_2/n_2$. Para testarmos a hipótese $H_0 : p_1 = p_2$ ou $H_0 : p_1 - p_2 = 0$, podemos utilizar o seguinte algoritmo *bootstrap* paramétrico: a) utilizar $\hat{p_1}$ e $\hat{p_2}$ para gerarmos amostras de tamanho n_1 e n_2 de ambas as populações; b) estimar $\hat{p_{1j}} = y_{1j}/n_1$ e $\hat{p_{2j}} = y_{2j}/n_2$ na j-ésima repetição desse processo; c) calcular $d_j = \hat{p_{1j}} - \hat{p_{2j}}$; d) repetir os passos de (a) a (c) $B - 1$ vezes; e) unir com o valor da amostra original; f) ordenar os valores e obter os quantis 2,5% e 97,5% da distribuição bootstrap de d_j; e g) se o valor hipotético 0 estiver contido nesse intervalo, não rejeitar H_0, caso contrário, rejeitar a hipótese de igualdade das proporções binomiais das duas populações.

Chapter 4

Geração de Amostras Aleatórias de Variáveis Multidimensionais

Os modelos multivariados ganharam grande aceitação no meio científico em função das facilidades computacionais e do desenvolvimento de programas especializados nesta área. Os fenômenos naturais são em geral multivariados. Um tratamento aplicado em um ser, a um solo ou a um sistema não afeta isoladamente apenas uma variável, e sim todas as variáveis. Ademais, as variáveis possuem relações entre si e qualquer mudança em uma ou algumas delas, afeta as outras. Assim, a geração de realizações de vetores ou matrizes aleatórias é um assunto que não pode ser ignorado. Vamos neste capítulo disponibilizar ao leitor mecanismos para gerar realizações de variáveis aleatórias multidimensionais.

4.1 Introdução

Os processos para gerarmos variáveis aleatórias multidimensionais são muitas vezes considerados difíceis pela maioria dos pesquisadores. Uma boa parte deles, no entanto, pode ser realizada no Python com apenas uma linha de comando. Embora tenhamos estas facilidades, nestas notas vamos apresentar detalhes de alguns processos para gerarmos dados dos principais modelos probabilísticos multivariados como, por exemplo, a normal multivariada, a Wishart e a Wishart invertida, a t de Student multivariada e algumas outras distribuições.

Uma das principais características das variáveis multidimensionais é a correlação entre seus componentes. A importância destes modelos é praticamente indescritível, mas podemos destacar a inferência paramétrica, a inferência bayesiana, a estimação de regiões de confiança, entre outras. Vamos abordar nas próximos seções formas de gerarmos realizações de variáveis aleatórias multidimensionais para determinados modelos utilizando o Python para implementarmos as rotinas ou para utilizarmos as rotinas pré-existentes. Nossa aparente perda de tempo, descrevendo funções menos eficientes do que as pré-existentes no Python, tem como razão fundamental permitir ao leitor ir além de simplesmente utilizar rotinas previamente programadas por terceiros. Se o leitor ganhar, ao final da leitura deste material, a capacidade de produzir suas próprias rotinas e entender como as rotinas pré-existentes funcionam, nosso objetivo terá sido alcançado.

4.2 Distribuição Normal Multivariada

A função densidade de probabilidade normal multivariada de um vetor aleatório $\mathbf{X}$ é dada por:

$$f_{\mathbf{X}}(\mathbf{x}) = (2\pi)^{-\frac{p}{2}} |\mathbf{\Sigma}|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\mu)^\top \mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)\right\} \tag{4.1}$$

em que μ e $\mathbf{\Sigma}$ são, respectivamente, o vetor de média e a matriz de covariâncias, simétrica e positiva definida, e p é a dimensão do vetor aleatório $\mathbf{X}$.

Um importante resultado diz respeito a combinações lineares de variáveis normais multivariadas e será apresentado no seguinte teorema.

Theorem 4.1 (Combinações Lineares)**.** *Considere o vetor aleatório normal multivariado* $\mathbf{X} = [X_1, X_2, \cdots, X_p]^\top$ *com média* μ *e covariância* $\mathbf{\Sigma}$ *e considere* $\mathbf{C}$ *uma matriz* $(p \times p)$ *de posto* p*, então a combinação linear* $\mathbf{Y} = \mathbf{C}\mathbf{X}$ $(p \times 1)$ *tem distribuição normal multivariada com média* $\mu_{\mathbf{Y}} = \mathbf{C}\mu$ *e covariância* $\mathbf{\Sigma}_{\mathbf{Y}} = \mathbf{C}\mathbf{\Sigma}\mathbf{C}^\top$*.*

Proof. Se $\mathbf{C}$ possui posto p, então existe $\mathbf{C}^{-1}$ e, portanto,

$$\mathbf{X} = \mathbf{C}^{-1}\mathbf{Y}.$$

O Jacobiano da transformação é $J = |\mathbf{C}|^{-1}$ e a distribuição de $\mathbf{Y}$ é dada por $f_{\mathbf{Y}}(\mathbf{y}) = f_{\mathbf{X}}(\mathbf{x})|J|$. Se $\mathbf{X}$ tem distribuição normal multivariada, então

$$\begin{aligned}
f_{\mathbf{Y}}(\mathbf{y}) =& (2\pi)^{-p/2} |\mathbf{\Sigma}|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\mathbf{C}^{-1}\mathbf{y}-\mu)^\top \mathbf{\Sigma}^{-1}(\mathbf{C}^{-1}\mathbf{y}-\mu)\right\} |\mathbf{C}|^{-1} \\
=& (2\pi)^{-p/2} |\mathbf{C}|^{-1/2} |\mathbf{\Sigma}|^{-\frac{1}{2}} |\mathbf{C}|^{-1/2} \\
& \times \exp\left\{-\frac{1}{2}(\mathbf{y}-\mathbf{C}\mu)^\top \left(\mathbf{C}^\top\right)^{-1} \mathbf{\Sigma}^{-1}\mathbf{C}^{-1}(\mathbf{y}-\mathbf{C}\mu)\right\} \\
=& (2\pi)^{-p/2} \left|\mathbf{C}\mathbf{\Sigma}\mathbf{C}^\top\right|^{-\frac{1}{2}} \exp\left\{-\frac{1}{2}(\mathbf{y}-\mathbf{C}\mu)^\top \left(\mathbf{C}\mathbf{\Sigma}\mathbf{C}^\top\right)^{-1} (\mathbf{y}-\mathbf{C}\mu)\right\}
\end{aligned}$$

que é a densidade normal multivariada do vetor aleatório $\mathbf{Y}$ com média $\mu_{\mathbf{Y}} = \mathbf{C}\mu$ e covariância $\mathbf{\Sigma}_{\mathbf{Y}} = \mathbf{C}\mathbf{\Sigma}\mathbf{C}^\top$. Portanto combinações lineares de variáveis normais multivariadas são normais multivariadas. □

O teorema Theorem 4.1 nos fornece o principal resultado para gerarmos dados de uma normal multivariada. Assim, como nosso objetivo é gerar dados de uma amostra normal multivariada com vetor de médias μ e matriz de covariâncias $\mathbf{\Sigma}$ pré-estabelecidos, devemos seguir os seguintes procedimentos. Inicialmente devemos obter a matriz raiz quadrada de $\mathbf{\Sigma}$, representada por $\mathbf{\Sigma}^{1/2}$. Para isso, vamos considerar a decomposição espectral da matriz de covariâncias dada por $\mathbf{\Sigma} = \mathbf{P}\mathbf{\Lambda}\mathbf{P}^\top$. Logo, podemos definir a matriz raiz quadrada de $\mathbf{\Sigma}$ por $\mathbf{\Sigma}^{1/2} = \mathbf{P}\mathbf{\Lambda}^{1/2}\mathbf{P}^\top$, em que $\mathbf{\Lambda}$ é a matriz diagonal dos autovalores, $\mathbf{\Lambda}^{1/2}$ é a matriz diagonal contendo a raiz quadrada destes elementos e $\mathbf{P}$ é a matriz de autovetores, cada um destes vetor disposto em uma de suas colunas. Desta decomposição facilmente podemos observar que $\mathbf{\Sigma} = \mathbf{\Sigma}^{1/2}\mathbf{\Sigma}^{1/2}$.

Assim, podemos utilizar o seguinte método para gerarmos uma realização p-variada de uma normal multivariada. Inicialmente devemos gerar um vetor aleatório $\mathbf{Z} = [Z_1, Z_2, \cdots, Z_p]^\top$ de p variáveis normais padrão independentes utilizando, por exemplo, o algoritmo de Box-Müller. Isto que dizer que $\mu_{\mathbf{Z}} = \mathbf{0}$ e que $\text{Cov}(\mathbf{Z}) = \mathbf{I}$. Este vetor deve sofrer a seguinte transformação linear:

$$\mathbf{Y} = \mathbf{\Sigma}^{1/2}\mathbf{Z} + \mu. \tag{4.2}$$

De acordo com o teorema Theorem 4.1, o vetor $\mathbf{Y}$ possui distribuição normal multivariada com média $\mu_{\mathbf{Y}} = \mathbf{\Sigma}^{1/2}\mu_{\mathbf{Z}}+\mu = \mu$ e matriz de covariâncias $\mathbf{\Sigma}^{1/2}\mathbf{I}\mathbf{\Sigma}^{1/2} = \mathbf{\Sigma}$. Este será o método que usaremos para obter a amostra p-dimensional de tamanho n de uma normal multivariada com média μ e covariância $\mathbf{\Sigma}$. Para obtermos a matriz raiz quadrada no Python, podemos utilizar o comando para a obtenção da decomposição espectral `np.linalg.svd()` ou alternativamente o comando `linalg.cholesky(a,/,*,upper=False)`, que retorna o fator de Cholesky de uma matriz positiva definida, que na verdade é um tipo de raiz quadrada. A decomposição do valor singular neste caso se especializa na decomposição espectral, pois a matriz $\mathbf{\Sigma}$ é simétrica. Geraremos um vetor de variáveis aleatórias normal padrão independentes $\mathbf{Z}$ utilizando o comando `np.random.normal()`. Em seguida a transformação Equation 4.2 é realizada.

Vamos ilustrar e apresentar o programa de geração de variáveis normais multivariada para um caso particular bivariado ($p = 2$) e com vetor de médias $\mu = [10, 50]^\top$ e matriz de covariâncias dada por:

$$\mathbf{\Sigma} = \begin{bmatrix} 4 & 1 \\ 1 & 1 \end{bmatrix}.$$

O programa resultante é dado por:

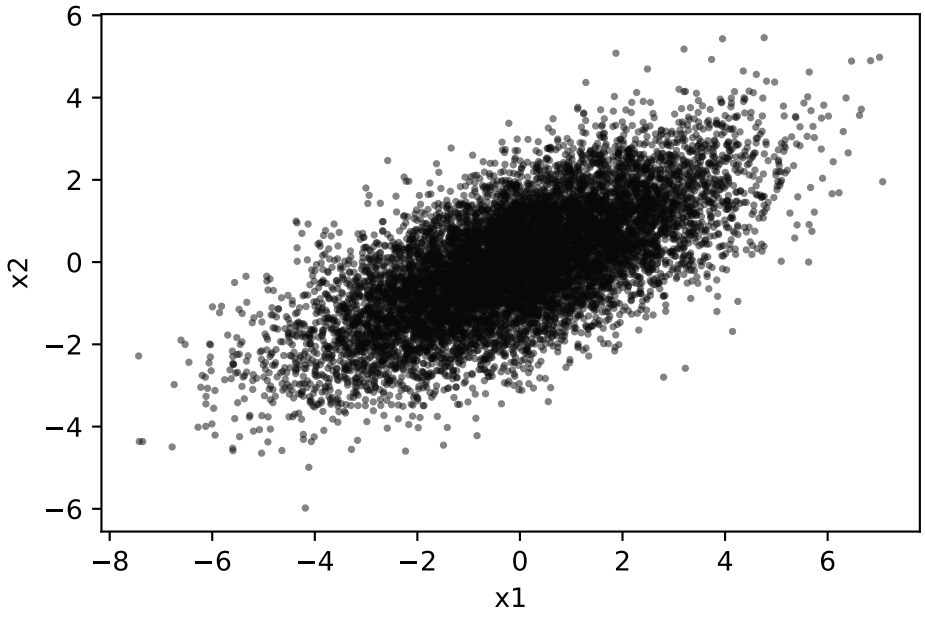
```
# Função Python para gerar n vetores aleatórios normais
# multivariados com vetor de médias mu e covariância sigma.
# o resultado é uma matriz n x p, sendo p o número de variáveis
import numpy as np
import matplotlib.pyplot as plt
def rnormmv(n, mu, sigma):
  p = sigma.shape[0]
  u, d, vt = np.linalg.svd(sigma)
  sigmaroot = u @ np.diag(d**0.5) @ vt
  for i in range(n):
    z = np.random.normal(0,1,p)
    if i == 0:
      x = sigmaroot @ z + mu
    else:
      li = sigmaroot @ z + mu
      x = np.vstack((x, li))
  return x

# Exemplo de uso
n = 10000
sigma = np.array([[4,1.9],[1.9,2]])
mu = np.full(2, 0) # cria o vetor com 2 valores 0
x = rnormmv(n, mu, sigma)
np.mean(x, axis = 0)
np.cov(x,rowvar = False)
a = x[:,0]
b = x[:,1]
plt.scatter(a, b, s = 3, c = '#070808', alpha = 0.5)
plt.xlabel('x1')
plt.ylabel('x2')
plt.show()
```

```
array([-0.0298272 , -0.01346456])
```

```
array([[4.06939457, 1.95852754],
       [1.95852754, 2.03020781]])
```

```
Text(0.5, 0, 'x1')
```

```
Text(0, 0.5, 'x2')
```



No python temos o gerador do `numpy`, `random.multivariate_normal(mean, cov, size=None, check_valid='warn', tol=1e-8)` para gerarmos dados da distribuição normal multivariada. No script a seguir aplicamos a nossa função e a do `numpy` para gerarmos um número grande de observações e comparamos o desempenho em termos de tempo de processamento. A função do `numpy` protege o processo, com, por exemplo verificando se a matriz de covariâncias é positiva definida. Em nossa função não utilizamos nenhuma proteção, embora seja possível fazer isso.

```
# Comparativo de desempenho dos dois geradores de
# normais multivariadas
from datetime import datetime
n = 60000
sigma = np.array([[4,1.9],[1.9,2]])
mu = np.full(2, 0) # cria o vetor com 2 valores 0
t1 = datetime.now()
x = rnormmv(n, mu, sigma)
t2 = datetime.now()
trnm = t2-t1
print('tempo médio da rnormmv: ',trnm.microseconds / n)
t1 = datetime.now()
x = np.random.multivariate_normal(mu, sigma, size=n)
t2 = datetime.now()
tnp = t2-t1
print('tempo médio da numpy: ',tnp.microseconds / n)
# tempo relativo
print('numpy é mais rápido ',trnm.microseconds / tnp.microseconds ,' vezes')
```

```
tempo médio da rnormmv:  5.65985
tempo médio da numpy:  0.03783333333333333
numpy é mais rápido  149.59955947136564  vezes
```

A rotina do `numpy` foi muitas vezes superior a nossa implementação. Isso em parte é devido ao fato de estar compilada e, talvez, também ao possível método utilizado para obter a matriz raiz quadrada. A troca da matriz raiz quadrada de `svd` para `Cholesky` ou pela decomposição espectral, poderia ser feita e o desempenho médio dos procedimentos avaliados, para definirmos a melhor estratégia de implementação. Fizemos uma alteração na nossa implementação. Colocamos um argumento com uma das três opções, para que o usuário escolha qual método utilizar. A obtenção da raiz quadrada foi implementada em uma função separada.

```
# Função Python para gerar n vetores aleatórios normais
# multivariados com vetor de médias mu e covariância sigma.
# o resultado é uma matriz n x p, sendo p o número de variáveis
def sigmaroot(sigma, metodo='chol'):
  if metodo == 'svd':
    u, d, vt = np.linalg.svd(sigma)
    root = u @ np.diag(d**0.5) @ vt
  elif metodo == 'eig':
    d, u = np.linalg.eig(sigma)
    root = u @ np.diag(d**0.5) @ np.transpose(u)
  else:
    root = np.linalg.cholesky(sigma)
  return  root

# São métodos válidos: 'chol', 'svd' e 'eig'
def rnormmv_2(n, mu, sigma, meth ='chol'):
  p = sigma.shape[0]
  sigmar = sigmaroot(sigma, meth)
  for i in range(n):
    z = np.random.normal(0,1,p)
    if i == 0:
      x = sigmar @ z + mu
    else:
      li = sigmar @ z + mu
      x = np.vstack((x, li))
  return x
```

O comparativo entre eles foi apresentado no seguinte `script`:

```
# Comparativo de desempenho dos dois geradores de
# normais multivariadas
n = 60000
sigma = np.array([[4,1.9],[1.9,2]])
mu = np.full(2, 0) # cria o vetor com 2 valores 0
t1 = datetime.now()
x = rnormmv_2(n, mu, sigma,'svd')
t2 = datetime.now()
tsvd = t2-t1
t1 = datetime.now()
x = rnormmv_2(n, mu, sigma,'eig')
t2 = datetime.now()
teig = t2-t1
t1 = datetime.now()
x = rnormmv_2(n, mu, sigma,'chol')
t2 = datetime.now()
tchol = t2-t1
```

```
t1 = datetime.now()
x = np.random.multivariate_normal(mu, sigma, size=n)
t2 = datetime.now()
tnp = t2-t1
print('tempo médio da svd: ',tsvd.microseconds / n)
print('tempo médio da eig: ',teig.microseconds / n)
print('tempo médio da chol: ',tchol.microseconds / n)
print('tempo médio da numpy: ',tnp.microseconds / n)
# tempo relativo
print('numpy é mais rápido ',tsvd/tnp ,' vezes que svd')
print('numpy é mais rápido ',teig/tnp ,' vezes que eig')
print('numpy é mais rápido ',tchol/tnp,' vezes que chol')
```

```
tempo médio da svd:  6.21645
tempo médio da eig:  6.019716666666667
tempo médio da chol:  9.997766666666667
tempo médio da numpy:  0.09428333333333333
numpy é mais rápido  242.70585115785752  vezes que svd
numpy é mais rápido  417.3913735195333  vezes que eig
numpy é mais rápido  813.1281598020151  vezes que chol
```

Embora nossa função seja relativamente rápida, levando entre 2 e 14 micro segundos para rodar cada observação bivariada neste caso, ela ainda foi bem menos eficiente que rotina `numpy`. A compilação é fundamental em relação à interpretação. É extremamente simples gerarmos dados de normais multivariadas utilizando a função `numpy`, o que nos desobriga de programar os nossos próprios geradores aleatórios. Reiteramos que fizemos isso, pois queremos que o nosso leitor e nosso estudante consigam desvendar o que está por trás de cada método deste e também que consiga desenvolver suas habilidades em programação, implementando rotinas sofisticadas como estas.

4.3 Distribuição Wishart e Wishart Invertida

As distribuições Wishart e Wishart invertida são relacionadas às distribuições de matrizes de somas de quadrados e produtos não-corrigidas $\mathbf{W}$ obtidas de amostras de tamanho $n-1$ da distribuição normal multivariada com média $\mathbf{0}$. Considere $\mathbf{X}_j = [X_1, X_2, \cdots, X_p]^\top$ o j-ésimo vetor ($j = 1, 2, \cdots, \nu$) de uma amostra aleatória de tamanho ν de uma normal com média $\mathbf{0}$ e covariância $\mathbf{\Sigma}$, então a matriz aleatória

$$\mathbf{W} = \sum_{j=1}^{n-1} \mathbf{X}_j \mathbf{X}_j^\top$$

possui distribuição Wishart com $n-1$ graus de liberdade e parâmetro $\mathbf{\Sigma}$ (matriz positiva definida).

Da mesma forma, se temos uma amostra aleatória de tamanho n de uma distribuição normal multivariada com média μ e covariância $\mathbf{\Sigma}$, a distribuição da matriz aleatória

$$\mathbf{W} = \sum_{j=1}^{n} (\mathbf{X}_j - \bar{\mathbf{X}})(\mathbf{X}_j - \bar{\mathbf{X}})^\top$$

é Wishart com $\nu = n-1$ graus de liberdade e parâmetro $\mathbf{\Sigma}$.

A função densidade Wishart de uma matriz aleatória $\mathbf{W}$ de somas de quadrados e produtos e representada por $W_p(\nu,\mathbf{\Sigma})$ é definida por:

$$f_{\mathbf{W}}(\mathbf{w}|\nu,\mathbf{\Sigma}) = \frac{|\mathbf{\Sigma}|^{-\nu/2}|\mathbf{w}|^{(\nu-p-1)/2}}{2^{\nu p/2}\pi^{p(p-1)/4}\prod_{i=1}^{p}\Gamma\left(\frac{\nu-i+1}{2}\right)}\exp\left\{-\frac{tr\left(\mathbf{\Sigma}^{-1}\mathbf{w}\right)}{2}\right\} \tag{4.3}$$

em que $\Gamma(x) = \int_0^\infty t^{x-1}e^{-x}dt$ é função gama.

Assim, para gerarmos variáveis Wishart com parâmetros $n-1$ graus liberdade inteiro e matriz $\mathbf{\Sigma}$ positiva definida, podemos utilizar um gerador de amostras aleatórias normais multivariadas e obter a matriz de somas de quadrados e produtos amostrais. Esta matriz será uma realização de uma variável aleatória Wishart, que é uma matriz de dimensão $p \times p$. A seguinte função pode ser utilizada para obtermos realizações aleatórias de uma Wishart:

```
# Exemplificação para gerarmos matrizes de somas de quadrados e produtos
# aleatórias W com distribuição Wishart(nu, Sigma), nu = n - 1
# utiliza o random.multivariate_normal para gerar normais multivariadas.
def rwishart(nu, sigma):
  p = sigma.shape[0]
  mu = np.full(p, 0)
  x = np.random.multivariate_normal(mu, sigma, size=nu + 1)
  w = nu * np.cov(x, rowvar=False)
  return w

# Exemplo de uso
sigma = np.array([[4, 1], [1, 1]])
nu    = 5
w     = rwishart(nu, sigma)
print(w)
```

```
[[10.87107909  3.19722672]
 [ 3.19722672  1.98382574]]
```

Outra distribuição relacionada que aparece frequentemente na inferência multivariada é a Wishart invertida. Considere $\mathbf{W}$ uma matriz aleatória $W_p(\nu,\mathbf{\Sigma})$, então a distribuição de $\mathbf{S} = \mathbf{W}^{-1}$, dada pela função densidade

$$f_{\mathbf{S}}(\mathbf{s}|\nu,\mathbf{\Sigma}) = \frac{|\mathbf{\Sigma}^{-1}|^{\nu/2}|\mathbf{s}|^{-(\nu+p+1)/2}}{2^{\nu p/2}\pi^{p(p-1)/4}\prod_{i=1}^{p}\Gamma\left(\frac{\nu-i+1}{2}\right)}\exp\left\{-\frac{tr\left(\mathbf{\Sigma}^{-1}\mathbf{s}^{-1}\right)}{2}\right\} \tag{4.4}$$

é a Wishart invertida que vamos representar por $W_p^{-1}(\nu,\mathbf{\Sigma})$.

Se quisermos gerar uma matriz aleatória de uma distribuição Wishart invertida em vez de uma Wishart precisamos simplesmente realizar a transformação $\mathbf{S} = \mathbf{W}^{-1}$. Assim, vamos alterar o programa anterior para gerar simultaneamente realizações aleatórias de distribuição Wishart e Wishart invertida. É importante salientar que em nossa notação os parâmetros da Wishart invertida são aqueles da Wishart. Isto é relevante, pois alguns autores apresentam os parâmetros da Wishart invertida e não da Wishart e devemos estar atentos para este fato, senão geraremos variáveis aleatórias com densidades diferentes.

```
# Exemplificação para gerarmos matrizes de somas de quadrados e produtos
# aleatórias W com distribuição Wishart(nu, Sigma), nu = n - 1
# utiliza o pacote mvtnorm para gerar amostras normais.
# Exemplificação para gerarmos matrizes de somas de quadrados e produtos
# aleatórias W com distribuição Wishart(nu, Sigma), nu = n - 1
# e Wishart invertida
```

```
def rw_wi(nu, sigma, wi = True):
  p = sigma.shape[0]
  mu = np.full(p, 0)
  x = np.random.multivariate_normal(mu, sigma, size=nu+1)
  w = nu * np.cov(x, rowvar=False)
  if wi == True:
    iw = np.linalg.pinv(w)
    res = {'W': w, 'WI': iw}
    return res
  else:
    return w

# Exemplo de uso
sigma = np.array([[4, 1], [1, 1]])
nu    = 5
wi = True
res     = rw_wi(nu, sigma, wi)
print('Ambas, W e WI', res)
wi = False
res     = rw_wi(nu, sigma, wi)
print('Só a Wishart: ', res)
```

```
Ambas, W e WI {'W': array([[7.86262938, 3.31061534],
       [3.31061534, 5.466944  ]]), 'WI': array([[ 0.17071194, -0.10337797],
       [-0.10337797,  0.24552011]])}
Só a Wishart:  [[26.57609811  2.93602364]
 [ 2.93602364  5.60585008]]
```

Um aspecto importante que precisamos mencionar é sobre a necessidade de gerarmos variáveis Wishart ou Wishart invertida com graus de liberdade reais. Para isso podemos utilizar o algoritmo descrito por Smith and Hocking [1972]. Considere a decomposição da matriz $\mathbf{\Sigma}$ dada por $\mathbf{\Sigma} = \mathbf{\Sigma}^{1/2}\mathbf{\Sigma}^{1/2}$, que utilizaremos para realizarmos uma transformação na matriz aleatória gerada, que de acordo com propriedades de matrizes Wishart, será Wishart.

Devemos inicialmente construir uma matriz triangular inferior $\mathbf{T} = [t_{ij}]$ $(i,j = 1, 2, \cdots, p)$, com $t_{ii} \sim \sqrt{\chi^2_{\nu+1-i}}$ e $t_{ij} \sim N(0,1)$ se $i > j$ e $t_{ij} = 0$ se $i < j$. Na sequência devemos obter a matriz $\mathbf{\Gamma} = \mathbf{T}\mathbf{T}^\top$, que possui distribuição $W_p(\nu, \mathbf{I})$. Desta forma obteremos $\mathbf{W} = \mathbf{\Sigma}^{1/2}\mathbf{\Gamma}\mathbf{\Sigma}^{1/2}$ com a distribuição desejada, ou seja, com distribuição $W_p(\nu,\mathbf{\Sigma})$. Fica claro que com esse algoritmo podemos gerar variáveis Wishart com graus de liberdade reais ou inteiros.

```
# Função mais eficiente para gerarmos matrizes de somas de quadrados
# e produtos aleatórias W com distribuição Wishart(nu, Sigma) e
# Wishart invertida WI(nu, Sigma)
def rwwi_sh(nu, sigma, wi = True):
  tip = type(sigma)
  if tip == int or tip == float:
    p = 1
  else:
    p = sigma.shape[0]
  df = np.flip(np.arange(nu - p + 1, nu + 1, 1))
  if p > 1:
    t = np.diag(np.random.chisquare(df, p)**0.5)
    t[np.tril_indices(t.shape[0], -1)]=np.random.normal(0,1,int(p*(p-1)/2))
```

```
    s = np.linalg.cholesky(sigma)
  else:
    t = np.random.chisquare(df)**0.5
    s = sigma**0.5
  if tip == int or tip == float:
    w = s**2 * t**2
  else:
    w = s @ t @ np.transpose(t) @ np.transpose(s)
  if wi == True:
    if tip == int or tip == float:
      iw = 1 / w
    else:
      iw = np.linalg.pinv(w)
    res = {'W': w, 'WI': iw}
    return res
  else:
    return w

# Exemplo de uso
sigma = np.array([[4, 1], [1, 1]])
nu    = 5
wi = True
res     = rwwi_sh(nu, sigma, wi)
print('Ambas, W e WI', res)
wi = False
res     = rwwi_sh(nu, sigma, wi)
print('Só a Wishart: ', res)
```

```
Ambas, W e WI {'W': array([[21.18876273,  9.0943717 ],
       [ 9.0943717 ,  5.9279009 ]]), 'WI': array([[ 0.13818824, -0.21200341],
       [-0.21200341,  0.49394177]])}
Só a Wishart:  [[46.9033493  17.71148872]
 [17.71148872  7.33890111]]
```

Vamos chamar a atenção para alguns fatos sobre esta função, pois utilizamos alguns comandos e recursos não mencionados até o presente momento. Inicialmente utilizamos o comando `np.diag(np.random.chisquare(df, p)**0.5)` para preencher a diagonal da matriz $\mathbf{T}$ com realizações de variáveis aleatórias qui-quadrado e em seguida do método diag para transformar o vetor em uma matriz diagonal.

Outro aspecto interessante que merece ser mencionado é o uso do fator de Cholesky no lugar de obter a matriz raiz quadrada de $\mathbf{\Sigma}$. O fator de Cholesky utiliza a decomposição $\mathbf{\Sigma} = \mathbf{S}\mathbf{S}^\top$, em que $\mathbf{S}$ é uma matriz triangular inferior. O Python, por meio da função `np.linalg.cholesky(sigma)`, retorna a matriz $\mathbf{S}$. Finalmente, se o número de variáveis é igual a 1, a distribuição Wishart se especializa na qui-quadrado e a Wishart invertida na qui-quadrado invertida. Assim, quando $p = 1$ o algoritmo retornará variáveis $\sigma^2 X$ e $1/(\sigma^2 X)$ com distribuições proporcionais a distribuição qui-quadrado e qui-quadrado invertida, respectivamente, sendo X uma variável qui-quadrado com ν graus de liberdade.

O processamento para o caso escalar foi controlado com o uso do `if`, pois operações matriciais não são aplicáveis se os argumentos forem escalares inteiros ou `float` (reais).

4.4 Distribuição t de Student Multivariada

A família de distribuições elípticas é muito importante na multivariada e nas aplicações Bayesianas. A distribuição t de Student multivariada é um caso particular desta família. Esta distribuição tem particular interesse nos procedimentos de comparação múltipla dos tratamentos com alguma testemunha avaliada no experimento. A função densidade do vetor aleatório $\mathbf{X} = [X_1, X_2, \cdots, X_p]^\top \in \mathbb{R}^p$ com parâmetros dados pelo vetor de médias $\mu = [\mu_1, \mu_2, \cdots, \mu_p]^\top \in \mathbb{R}^p$ e matriz simétrica e positiva definida $\mathbf{\Sigma}$ $(p \times p)$ é

$$\begin{aligned} f_{\mathbf{X}}(\mathbf{x}) =& \frac{g((\mathbf{x}-\mu)^\top \mathbf{\Sigma}^{-1}(\mathbf{x}-\mu))}{|\mathbf{\Sigma}|^{1/2}} \\ =& \frac{\Gamma\left(\frac{\nu+p}{2}\right)}{(\pi\nu)^{p/2}\Gamma(\nu/2)|\mathbf{\Sigma}|^{1/2}} \left[1 + \frac{1}{\nu}(\mathbf{x}-\mu)^\top \mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)\right]^{-\frac{\nu+p}{2}} \end{aligned}$$

em que a função g é dada por

$$g(z) = \frac{\Gamma\left(\frac{\nu+p}{2}\right)}{(\pi\nu)^{p/2}\Gamma(\nu/2)} \left(1 + \frac{z}{\nu}\right)^{-(\nu+p)/2}.$$

A variável aleatória $\mathbf{X}$ tem média μ e matriz de covariâncias $\nu\mathbf{\Sigma}/(\nu-2)$ no caso de $\nu > 2$.

Se efetuarmos a transformação $\mathbf{Y} = \mathbf{\Sigma}^{-1/2}(\mathbf{X} - \mu)$ obteremos a distribuição t multivariada esférica simétrica, cuja densidade é dada por:

$$f_{\mathbf{Y}}(\mathbf{y}) = \frac{\Gamma\left(\frac{\nu+p}{2}\right)}{(\pi\nu)^{p/2}\Gamma(\nu/2)} \left[1 + \frac{1}{\nu}\mathbf{y}^\top \mathbf{y}\right]^{-\frac{\nu+p}{2}}. \tag{4.5}$$

A variável aleatória $\mathbf{Y}$ terá vetor de médias nulo e covariâncias $\nu\mathbf{I}/(\nu-2)$ e a densidade terá contornos esféricos de mesma probabilidade.

Vamos apresentar a forma geral para gerarmos variáveis aleatórias p-dimensionais t multivariada com ν graus de liberdade e parâmetros μ e $\mathbf{\Sigma}$. Seja um vetor aleatório $\mathbf{Z}$ com distribuição $N_p(\mathbf{0}, \mathbf{I})$ e a variável aleatória U com distribuição qui-quadrado com ν graus de liberdade, então o vetor aleatório $\mathbf{Y}$, dado pela transformação

$$\mathbf{Y} = \sqrt{\nu}\frac{\mathbf{Z}}{\sqrt{U}}, \tag{4.6}$$

possui distribuição t multivariada esférica com ν graus de liberdade. O vetor $\mathbf{X}$ obtido pela transformação linear

$$\mathbf{X} = \mathbf{\Sigma}^{1/2}\mathbf{Y} + \mu, \tag{4.7}$$

possui distribuição t multivariada elíptica com ν graus de liberdade e parâmetros μ e $\mathbf{\Sigma}$.

Assim, devemos aplicar a transformação (Equation 4.7) n vezes a n diferentes vetores aleatórios $\mathbf{Y}$ e variáveis U. Ao final deste processo teremos uma amostra de tamanho n da distribuição t multivariada almejada com ν graus de liberdade. Assim, para gerarmos dados de uma t multivariada com dimensão p, graus de liberdade ν (não necessariamente inteiro), vetor de média μ qualquer e matriz positiva definida $\mathbf{\Sigma}$ podemos utilizar a seguinte função Python, substituindo na expressão (Equation 4.7) a matriz raiz quadrada pelo fator de Cholesky $\mathbf{F}$ de $\mathbf{\Sigma}$:

```
# Função para gerarmos variáveis aleatórias t
# multivariadas (n, mu, Sigma, nu).
def rtmult(n, mu=[0,0,0], sigma=np.identity(3), df = 1):
```

```
  f = np.linalg.cholesky(sigma)
  p = sigma.shape[0]
  x = np.random.multivariate_normal(np.zeros(p),np.identity(p),n)
  q = (np.random.chisquare(df,n) / df)**0.5
  x =  x / q[:, np.newaxis]  @ np.transpose(f)
  x += np.tile(mu, n).reshape(n,p)
  return x

# Exemplo de uso
n = 3000
nu = 3
mu = [0,0]
sigma = np.identity(2)
x = rtmult(n, mu, sigma, nu)
fig, (ax1, ax2) = plt.subplots(1, 2)
fig.suptitle('Distribuições t multivariadas')
ax1.scatter(x[:,0], x[:,1], s = 3, c = '#070808', alpha = 0.5)
ax1.set_xlabel('x1')
ax1.set_ylabel('x2')
mu = [10, 5]
sigma = np.array([[4, 1.9], [1.9,1]])
x = rtmult(n, mu, sigma, nu)
ax2.scatter(x[:,0], x[:,1], s = 3, c = '#070808', alpha = 0.5)
ax2.set_xlabel('x1')
ax2.set_ylabel('x2')
np.cov(x, rowvar = False)
nu * sigma
```

```
Text(0.5, 0.98, 'Distribuições t multivariadas')
```

```
Text(0.5, 0, 'x1')
```

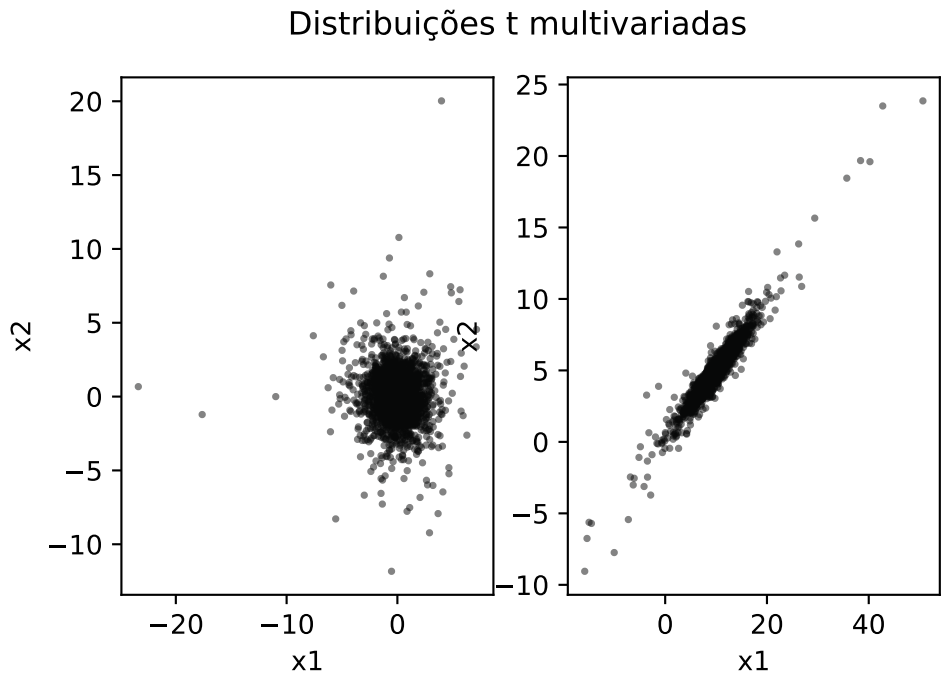
```
Text(0, 0.5, 'x2')
```

```
Text(0.5, 0, 'x1')
```

```
Text(0, 0.5, 'x2')
```

```
array([[11.74146176,  5.57705378],
       [ 5.57705378,  2.87989181]])
```

```
array([[12. ,  5.7],
       [ 5.7,  3. ]])
```



Graus de liberdade reais positivos podem ser utilizados como argumento da função criada. Foram geradas, no exemplo anterior, duas amostras aleatórias para ilustrar. Foram definidos para os parâmetros **mu**, $\mathbf{\Sigma}$ e ν, os valores $[0,, 0, 0]^\top$, $\mathbf{I}_3$ e 3, respectivamente, como `default`. Podemos também utilizar a implementação determinada pela função `multivariate_t.rvs(mu, sigma, df n)`, da biblioteca `scipy`, para gerarmos dados da distribuição t de Student multivariada.

```
from scipy.stats import multivariate_t
n = 5
X = multivariate_t.rvs([1.0, -0.5], [[2.1, 0.3], [0.3, 1.5]], df=3, size = n)
print(X)
```

```
[[ 0.11049954  2.85994019]
 [ 0.48393524 -2.15062667]
 [ 4.20035761  1.5564479 ]
 [ 1.87226554 -0.84375489]
 [ 0.95358703  1.80208699]]
```

4.5 Outras Distribuições Multivariadas

Existem muitas outras distribuições multivariadas. Vamos mencionar apenas mais duas delas: a log-normal e a normal contaminada multivariadas. A geração de um vetor aleatório de dimensão p com distribuição log-normal multivariada é feita tomando-se o seguinte vetor $\mathbf{Y} = [\exp(Z_1), \exp(Z_2), \cdots, \exp(Z_p)]^\top$, em que $\mathbf{Z} \sim N_p(\mu, \mathbf{\Sigma})$.

```
# gerador de realizações de variáveis log-normal
# multivariada com parâmetros mu e sigma
def rlgnormalmv(n, mu=np.zeros(3), sigma=np.identity(3)):
  p = sigma.shape[0]
  x = np.exp(np.random.multivariate_normal(np.zeros(p),np.identity(p),n))
  return x
```

```
# Exemplo de uso
n = 7
print(rlgnormalmv(n)) # lgnormalmv padrão
```

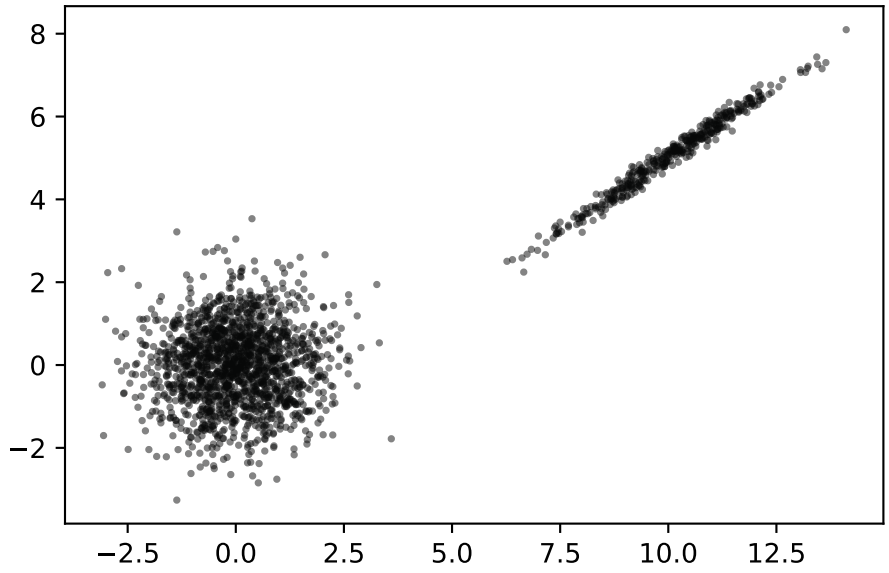
```
[[0.09433153 0.44188747 0.43161949]
 [0.18086973 1.39997425 2.39060125]
 [4.18450485 0.19009278 2.19513551]
 [2.43979725 4.16713122 0.79155077]
 [0.38956237 1.2708211  6.04592779]
 [1.49091478 0.366016   0.49925148]
 [0.17307373 0.29556996 4.5553044 ]]
```

Para gerarmos realizações da normal contaminada multivariada consideramos a simulação de um vetor aleatório $\mathbf{X}$ cuja densidade de probabilidade é dada por:

$$f_{\mathbf{X}}(\mathbf{x}) = \delta(2\pi)^{-p/2}|\mathbf{\Sigma}_1|^{-1/2}\exp\left\{-\frac{(\mathbf{x}-\mu_1)^\top\mathbf{\Sigma}_1^{-1}(\mathbf{x}-\mu_1)}{2}\right\} + (1-\delta)(2\pi)^{-p/2}|\mathbf{\Sigma}_2|^{-1/2}\exp\left\{-\frac{(\mathbf{x}-\mu_2)^\top\mathbf{\Sigma}_2^{-1}(\mathbf{x}-\mu_2)}{2}\right\}$$

em que $\mathbf{\Sigma}_i$ positiva definida, $i = 1, 2$ e $0 \leq \delta \leq 1$.

```
# gerador de variáveis normal contaminada multivariada
# com delta sendo a proporção de não-contaminantes: (0, 1)
def rncm(n, mu1, mu2,  sig1,  sig2, delta=0.8):
  u = np.random.uniform(size=n)
  p = sig1.shape[0]
  n1 = len(u[u <= delta])
  n2 = n - n1
  x  = np.zeros(n * p).reshape(n, p)
  if (n1 > 0):
    li = np.arange(n)[u <= delta]
    x[li,:] = np.random.multivariate_normal(mu1, sig1, n1)
  if (n2 > 0):
    li = np.arange(n)[u > delta]
    x[li,:] = np.random.multivariate_normal(mu2, sig2, n2)
  return x

# Exemplo
n = 2000
delta = 0.8
mu1 = np.zeros(2)
sig1 = np.identity(2)
mu2 = [10, 5]
sig2 = np.array([[2, 1.4], [1.4, 1]])
x = rncm(n, mu1, mu2, sig1, sig2, delta)
plt.scatter(x[:,0], x[:,1], s = 3, c = '#070808', alpha = 0.5)
plt.show()
```



4.6 Exercícios

1. Gerar uma amostra de tamanho ($n = 10$) uma distribuição normal trivariada com vetor de médias $\mu = [5{,}10, 15]^\top$ e matriz de covariâncias

$$\mathbf{\Sigma} = \begin{bmatrix} 5 & -1 & 2 \\ -1 & 3 & -2 \\ 2 & -2 & 7 \end{bmatrix}.$$

Estimar a média e a covariância amostral. Repetir este processo 1.000 vezes e estimar a média das médias amostrais e a média das matrizes de covariâncias amostrais. Estes valores correspondem exatamente aos respectivos valores esperados? Se não, apresentar a(s) principal(is) causa(s).

2. A partir da alteração da função `rnormmv` que realizamos, comparar o tempo de processamento médio quando for utilizado `svd`, `eig` ou `cholesky` que corresponde a um possível tipo de matriz raiz quadrada de $\mathbf{\Sigma}$. Verificar se houve melhoria no desempenho em relação ao tempo médio de processamento de cada variável aleatória gerada, vetor p-dimensional. Testar isso usando modificando n e p, pois só apresentamos o teste para um único valor de p e pequeno, $p = 2$.

3. Sabemos que variáveis Wishart possuem média $\nu\mathbf{\Sigma}$. Apresentar um programa para verificar se o valor esperado, ignorando o erro de Monte Carlo, é alcançado com o uso das funções apresentadas para gerar variáveis Wishart. Utilizar 10.000 repetições de Monte Carlo. Isso seria uma forma simples, embora não conclusiva, de checar se a função está realizando as simulações corretamente. Serve ao menos para indicar a presença de erro, mas não garante a assertividade.

4. Implementar funções Python para gerarmos variáveis aleatórias log-normal e normal-contaminada elípticas multivariadas.

Chapter 5

Algoritmos para Médias, Variâncias e Covariâncias

Muitos algoritmos para o cálculo de médias, variâncias e covariâncias são imprecisos, podendo gerar resultados finais contendo grandes erros. A necessidade de utilizarmos algoritmos eficientes para realizarmos estas tarefas simples são evidentes e serão descritos neste capítulo.

5.1 Introdução

Felizmente o Python utiliza algoritmos precisos para cálculo da média, da variância e de covariância. Vamos buscar esclarecer como a utilização de algoritmo ineficientes podem levar a resultados inconsistentes e imprecisos. Nosso objetivo neste capítulo é apresentar os algoritmos eficientes para estimarmos estes parâmetros quando as fórmulas convencionais podem falhar se utilizadas nos algoritmos diretamente.

Estes algoritmos eficientes são particularmente úteis quando os dados possuem grande magnitude ou estão muito próximos de zero. Neste caso particular, algumas planilhas eletrônicas, como o Excel nas suas versões mais antigas, podiam falhar [McCullough and Wilson, 1999]. O conhecimento de algoritmos que conduzirão a maiores precisões numéricas pode levar o pesquisador a não cometer os mesmos erros encontrados em alguns *softwares*.

5.2 Algoritmos Univariados

A ideia básica de utilizarmos algoritmos para média, para a soma de potências dos desvios em relação a média ou para soma de produtos de desvios é aplicarmos recursivamente as fórmulas existentes. Se desejamos, por exemplo, obter a média de n observações, podemos inicialmente calcular a média da primeira observação, que é a própria. Em um segundo estágio, propor uma expressão para atualizarmos a média da primeira observação, contemplando a segunda e assim sucessivamente. O mesmo procedimento de atualização é aplicado às variâncias ou às covariâncias, no caso de termos mais de uma variável.

Para uma amostra de tamanho n dada por X_1, X_2, $\cdots$, X_n, a média e a variância amostral convencionais são obtidas, respectivamente, por:

$$\bar{X}_{\cdot} = \frac{\sum_{i=1}^{n} X_i}{n}, \tag{5.1}$$

e

$$S^2 = \frac{1}{n-1}\left[\sum_{i=1}^{n} X_i^2 - \frac{\left(\sum_{i=1}^{n} X_i\right)^2}{n}\right]. \tag{5.2}$$

Alguns algoritmos existentes procuraram melhorar os algoritmos dos livros textos que são as fórmulas das equações (Equation 5.1 e Equation 5.2) procurando fazer adaptações, repassando a amostra duas vezes. Estes algoritmos podem ser eficientes do ponto de vista da precisão, mas não são rápidos, justamente por repassarem duas vezes os dados. West [1979] propôs utilizar um algoritmo que faz uma única passada, atualizando a média e a variância em cada nova observação. Podemos facilmente mostrar que para uma amostra de tamanho n, que a média é igual a X_1 se $n = 1$, $(X_1 + X_2)/2$ se $n = 2$ e assim por diante. No $(k-1)$-ésimo passo podemos especificar o estimador da média por:

$$\bar{X}_{k-1} = \frac{\sum_{i=1}^{k-1} X_i}{k-1}.$$

No k-ésimo passo teremos observado X_k e a média atualizada é:

$$\bar{X}_k = \frac{\sum_{i=1}^{k} X_i}{k}. \tag{5.3}$$

A pergunta que fazemos é "podemos expressar a média do k-ésimo passo em função da média do $(k-1)$-ésimo passo?" A resposta a esta pergunta é sim e o resultado nos fornece o algoritmo desejado. A partir da equação (Equation 5.3) obtemos:

$$\begin{aligned}
\bar{X}_k =& \frac{\sum_{i=1}^{k-1} X_i + X_k}{k} \\
=& \frac{(k-1)\sum_{i=1}^{k-1} X_i}{(k-1)k} + \frac{X_k}{k} \\
=& \frac{(k-1)\bar{X}_{k-1}}{k} + \frac{X_k}{k} \\
=& \bar{X}_{k-1} - \frac{\bar{X}_{k-1}}{k} + \frac{X_k}{k}
\end{aligned}$$

resultando na equação recursiva final

$$\bar{X}_k = \bar{X}_{k-1} + \frac{X_k - \bar{X}_{k-1}}{k}, \tag{5.4}$$

para $2 \leq k \leq n$, sendo que $\bar{X}_1 = X_1$.

Da mesma forma se definirmos a soma de quadrados corrigidas das k primeiras observações amostrais

$1 < k \leq n$ por:

$$W2_k = \sum_{i=1}^{k} X_i^2 - \frac{\left(\sum_{i=1}^{k} X_i\right)^2}{k}$$

veremos que a variância correspondente é dada por $S_k^2 = W2_k/(k-1)$. Se expandirmos esta expressão isolando o k-ésimo termo e simplificarmos a expressão resultante teremos:

$$\begin{aligned} W2_k &= \sum_{i=1}^{k-1} X_i^2 + X_k^2 - \frac{\left(\sum_{i=1}^{k-1} X_i + X_k\right)^2}{k} \\ &= W2_{k-1} + k\left(X_k - \bar{X}_k\right)^2/(k-1) \end{aligned} \tag{5.5}$$

A expressão que desenvolvemos (Equation 5.5) é equivalente a apresentada por West [1979]. Isso pode ser demonstrado facilmente se substituirmos $\bar{X}_k$ obtida na equação (Equation 5.4) na equação (Equation 5.5), de onde obtivemos:

$$W2_k = W2_{k-1} + (k-1)\left(X_k - \bar{X}_{k-1}\right)^2/k \tag{5.6}$$

para $2 \leq k \leq n$, sendo que $W2_1 = 0$. A variância é obtida por $S^2 = W2_n/(n-1)$.

Para demonstrarmos diretamente a expressão (Equation 5.6) temos:

$$\begin{aligned} W2_k =& \sum_{i=1}^{k-1} X_i^2 + X_k^2 - \frac{\left(\sum_{i=1}^{k-1} X_i + X_k\right)^2}{k} \\ =& \sum_{i=1}^{k-1} X_i^2 + X_k^2 - \frac{\left(\sum_{i=1}^{k-1} X_i\right)^2 + 2X_k \sum_{i=1}^{k-1} X_i + X_k^2}{k} \\ =& \sum_{i=1}^{k-1} X_i^2 + X_k^2 - \frac{(k-1)\left(\sum_{i=1}^{k-1} X_i\right)^2 + 2(k-1)X_k \sum_{i=1}^{k-1} X_i + (k-1)X_k^2}{k(k-1)} \\ =& \sum_{i=1}^{k-1} X_i^2 - \frac{\left(\sum_{i=1}^{k-1} X_i\right)^2}{k-1} + X_k^2 + \frac{\left(\sum_{i=1}^{k-1} X_i\right)^2}{k(k-1)} - \frac{2(k-1)X_k \sum_{i=1}^{k-1} X_i}{k(k-1)} - \frac{X_k^2}{k} \\ =& W2_{k-1} + \frac{\left(\sum_{i=1}^{k-1} X_i\right)^2}{k(k-1)} - \frac{2(k-1)X_k \sum_{i=1}^{k-1} X_i}{k(k-1)} + \frac{(k-1)X_k^2}{k} \\ =& W2_{k-1} + \frac{(k-1)\bar{X}_{k-1}^2}{k} - \frac{2(k-1)X_k\bar{X}_{k-1}}{k} + \frac{(k-1)X_k^2}{k}, \end{aligned}$$

resultando em

$$W2_k = W2_{k-1} + (k-1)\left(X_k - \bar{X}_{k-1}\right)^2/k.$$

Podemos generalizar essa expressão para computarmos a covariância $S_{(x,y)}$ entre uma variável X e outra Y. A expressão para a soma de produtos é dada por:

$$W2_{k,(x,y)} = W2_{k-1,(x,y)} + (k-1)\left(X_k - \bar{X}_{k-1}\right)\left(Y_k - \bar{Y}_{k-1}\right)/k \tag{5.7}$$

para $2 \leq k \leq n$, sendo $W2_{1,(x,y)} = 0$. O estimador da covariância é obtido por $S_{(x,y)} = W2_{n,(x,y)}/(n-1)$.

Podemos observar, analisando todas estas expressões, que para efetuarmos os cálculos da soma de quadrados e da soma de produtos corrigida necessitamos do cálculo das médias das variáveis no k-ésimo passo ou no passo anterior. A vantagem é que em uma única passagem pelos dados, obtemos todas as estimativas. Podemos ainda estender estes resultados para obtermos somas das terceira e quarta potências dos desvios em relação a média. As expressões que derivamos para isso são:

$$W3_k = W3_{k-1} + \frac{(k^2-3k+2)(X_k - \bar{X}_{k-1})^3}{k^2} - \frac{3(X_k - \bar{X}_{k-1})W2_{k-1}}{k} \tag{5.8}$$

e

$$\begin{aligned} W4_k =& W4_{k-1} + \frac{(k^3-4k^2+6k-3)(X_k - \bar{X}_{k-1})^4}{k^3} + \\ &+ \frac{6(X_k - \bar{X}_{k-1})^2 W2_{k-1}}{k^2} - \frac{4(X_k - \bar{X}_{k-1})W3_{k-1}}{k} \end{aligned} \tag{5.9}$$

para $2 \leq k \leq n$, sendo $W3_1 = 0$ e $W4_1 = 0$.

Desta forma podemos implementar a função `medsq()` que retorna a média, a soma de quadrado, cubo e quarta potência dos desvios em relação a média e variância a partir de um vetor de dados $\mathbf{X}$ de dimensão n.

```
# função para retornar a média, somas de desvios em relação
#  a média # ao quadrado, ao cubo e quarta potência e variância
def medsq(x):
  n = len(x)
  if (n <= 1):
    print('Vetor deve ter mais de 1 elemento!')
    return
  xb = x[0]
  w2 = 0
  w3 = 0
  w4 = 0
  for j in range(1, n):
    d = x[j] - xb
    i = j + 1.0
    w4  = w4 + (i**3 - 4 * i**2 + 6 * i - 3) * d**4 / i**3 + \
      6 * w2 * d**2 / i**2 - 4 * w3 * d / i
    w3 = w3 + (i**2 - 3 * i + 2) * d**3 / i**2 - 3 * w2 * d / i
    w2 = w2 + (i - 1) * d**2 / i
    xb = xb + d / i
  s2 = w2 / (n - 1)
  res = {'media': xb, 'variância': s2, 'SQ2': w2, 'W3': w3, 'W4': w4}
  return res
# Exemplo
x = [1, 2, 3, 4, 5, 7, 8]
res = medsq(x)
for key in res.keys():
    print(key, ":", res[key])
```

```
media : 4.285714285714286
variância : 6.571428571428572
SQ2 : 39.42857142857143
W3 : 22.04081632653064
W4 : 391.4518950437319
```

5.3 Algoritmos para Vetores Médias e Matrizes de Covariâncias

Vamos apresentar nesta seção a extensão multivariada para obtermos o vetor de médias e as matrizes de somas de quadrados e produtos e de covariâncias. Por essa razão não implementamos uma função específica para obtermos a covariância entre duas variáveis X e Y. Seja uma amostra aleatória no espaço $\mathbb{R}^p$ dada por $\mathbf{X}_1$, $\mathbf{X}_2$, $\cdots$, $\mathbf{X}_j$, $\cdots$, $\mathbf{X}_n$, sendo que estes vetores serão dispostos em uma matriz $\mathbf{X}$ de dimensões $(n \times p)$. Para estendermos os resultados da seção anterior, utilizaremos as mesmas expressões, tomando o devido cuidado para adaptá-las para lidar com operações matriciais e vetoriais. Assim, para estimarmos o vetor de médias populacionais, devemos em vez de utilizar o estimador clássico dos livros textos dado por

$$\bar{\mathbf{X}} = \frac{\sum_{i=1}^{n} \mathbf{X}_i}{n},$$

utilizaremos a expressão recursiva dada por

$$\bar{\mathbf{X}}_k = \bar{\mathbf{X}}_{k-1} + \frac{\mathbf{X}_k - \bar{\mathbf{X}}_{k-1}}{k}, \tag{5.10}$$

para $2 \leq k \leq n$, sendo que $\bar{\mathbf{X}}_1 = \mathbf{X}_1$.

Da mesma forma, a adaptação para p dimensões da expressão (Equation 5.6) é direta e o resultado obtido é:

$$\mathbf{W}_k = \mathbf{W}_{k-1} + (k-1)\left(\mathbf{X}_k - \bar{\mathbf{X}}_{k-1}\right)\left(\mathbf{X}_k - \bar{\mathbf{X}}_{k-1}\right)^\top / k \tag{5.11}$$

para $2 \leq k \leq n$, sendo $\mathbf{W}_1 = \mathbf{0}$, uma matriz de zeros de dimensões $(p \times p)$. O estimador da matriz de covariâncias é obtido por $\mathbf{S} = \mathbf{W}_n/(n-1)$.

Implementamos a função `medcov()` apresentada a seguir para obtermos o vetor de médias, a matriz de somas de quadrados e produtos e a matriz de covariâncias. O argumento desta função deve ser uma matriz de dados multivariados com n linhas (observações) e p colunas (variáveis). O programa resultante e um exemplo são apresentados na sequência. Escolhemos um exemplo onde geramos uma matriz de dados de uma normal multivariada.

```
# função para retornar o vetor de médias, a matriz de
# somas de # quadrados e produtos e a matriz de covariâncias
import numpy as np
def medcov(x):
  n = x.shape[0]
  p = x.shape[1]
  if (n <= 1):
    print('Matriz deve ter mais de 1 linha!')
    return
  xb = x[0,:]
  w  = np.zeros((p, p))
  for j in range(1, n):
```

```
    d = x[j,:] - xb
    i = j + 1.0
    w = w + (i - 1) * np.outer(d, d)  / i
    xb = xb + d / i
  s = w / (n - 1)
  res = {'media': xb, 'covariância': s, 'SQP': w}
  return res
# Exemplo
n = 1000
p = 5
x = np.random.multivariate_normal(np.zeros(p),np.identity(p),n)
medcov(x)
# comparar com resultado da nossa função
np.cov(x, rowvar = False)
np.mean(x, axis = 0)
```

```
{'media': array([-0.00227144, -0.01316051,  0.00693671,  0.0375574 ,  0.01663061]),
 'covariância': array([[ 1.01410199, -0.07485936, -0.05071402, -0.04940439,  0.00772572],
        [-0.07485936,  1.0405782 ,  0.04005046, -0.0493902 , -0.00492861],
        [-0.05071402,  0.04005046,  0.97944331, -0.07421177, -0.0313028 ],
        [-0.04940439, -0.0493902 , -0.07421177,  1.0137693 , -0.00514097],
        [ 0.00772572, -0.00492861, -0.0313028 , -0.00514097,  0.98960797]]),
 'SQP': array([[1013.08789019,  -74.78449591,  -50.66330947,  -49.35498766,
           7.71799519],
        [ -74.78449591, 1039.53762103,   40.01041354,  -49.34080555,
          -4.92368055],
        [ -50.66330947,   40.01041354,  978.46386529,  -74.13756065,
          -31.27149488],
        [ -49.35498766,  -49.34080555,  -74.13756065, 1012.75553022,
          -5.13583104],
        [   7.71799519,   -4.92368055,  -31.27149488,   -5.13583104,
         988.61836613]])}
```

```
array([[ 1.01410199, -0.07485936, -0.05071402, -0.04940439,  0.00772572],
       [-0.07485936,  1.0405782 ,  0.04005046, -0.0493902 , -0.00492861],
       [-0.05071402,  0.04005046,  0.97944331, -0.07421177, -0.0313028 ],
       [-0.04940439, -0.0493902 , -0.07421177,  1.0137693 , -0.00514097],
       [ 0.00772572, -0.00492861, -0.0313028 , -0.00514097,  0.98960797]])
```

```
array([-0.00227144, -0.01316051,  0.00693671,  0.0375574 ,  0.01663061])
```

Felizmente, devido ao Python (`numpy`) usar precisão dupla e utilizar algoritmos de ótima qualidade para estas tarefas, não precisaremos nos preocupar com a implementação de funções como as apresentadas neste capítulo. Mas se formos utilizar um compilador da linguagem Pascal, Fortran ou C e C^{++}, deveremos fazer uso destes algoritmos, pois somente assim alcançaremos elevada precisão, principalmente se tivermos lidando com dados de grande magnitude ou muito próximos de zero.

5.4 Exercícios

1. Mostrar a equivalência existente entre as expressões (Equation 5.5) e (Equation 5.6).
2. Implementar em Python uma função para obtermos a soma de produtos, equação (Equation 5.7), e covariância entre as n observações de duas variáveis e cujos valores estão dispostos em dois vetores $\mathbf{X}$

e $\mathbf{Y}$. Criar uma matriz com n linhas e $p = 2$ colunas de algum exemplo e utilizar a função `medcov()` para comparar os resultados obtidos.

3. Os coeficientes de assimetria e curtose amostrais são funções das somas de potências dos desvios em relação a média. O coeficiente de assimetria é dado por $\sqrt{b_1} = (W3_n/n)/(W2_n/n)^{3/2}$ e o coeficiente de curtose é $b_2 = (W4_n/n)/(W2_n/n)^2$. Implementar uma função Python, utilizando a função `medsq()` para estimar o coeficiente de assimetria e curtose univariados.

4. Utilizar uma amostra em uma área de sua especialidade e determinar a média, variância, soma de quadrados, soma de desvios ao cubo e na quarta potência. Determinar também os coeficientes de assimetria e curtose.

Chapter 6

Aproximação de Distribuições

Algoritmos para a obtenção de probabilidades foram alvos de pesquisas de muitas décadas e ainda continuam sendo. A importância de se ter algoritmos que sejam rápidos e precisos é indescritível. A maior dificuldade é que a maioria dos modelos probabilísticos não possui função de distribuição explicitamente conhecida. Assim, métodos numéricos sofisticados são exigidos para calcularmos probabilidades. Um outro aspecto é a necessidade de invertermos as funções de distribuição para obter quantis da variável aleatória para uma probabilidade acumulada conhecida. Nos testes de hipóteses e nos processos de estimação por intervalo e por região quase sempre utilizamos estes algoritmos indiretamente sem nos darmos conta disso.

Neste capítulo vamos introduzir estes conceitos e apresentar algumas ideias básicas de métodos gerais para realizar as quadraturas necessárias. Métodos numéricos particulares serão abordados para alguns poucos modelos. Também abordaremos separadamente os casos discreto e contínuo. Para finalizarmos apresentaremos as principais funções pré-existentes do R para uma série de modelos probabilísticos.

6.1 Introdução

Não temos muitas preocupações com a obtenção de probabilidades e quantis, quando utilizamos o R, pois a maioria dos modelos probabilísticos e das funções especiais já está implementada. Nosso objetivo está além deste fato, pois nossa intenção é buscar os conhecimentos necessários para ir adiante e para entendermos como determinada função opera e quais são suas potencialidades e limitações.

Vamos iniciar nossa discussão com a distribuição exponencial para chamarmos a atenção para a principal dificuldade existente neste processo. Assim, escolhemos este modelo justamente por ele não apresentar tais dificuldades. Considerando uma variável aleatória X com distribuição exponencial com parâmetro λ, temos a função densidade

$$f(x) = \lambda e^{-\lambda x}, \quad x > 0 \tag{6.1}$$

e a função de distribuição

$$F(x) = 1 - e^{-\lambda x}. \tag{6.2}$$

Para este modelo probabilístico podemos calcular probabilidades utilizando a função de distribuição (Equation 6.2) e obter quantis com a função de distribuição inversa que é dada por:

$$q = F^{-1}(p) = \frac{-\ln(1-p)}{\lambda}. \tag{6.3}$$

Implementamos, em Python, as funções densidade, de distribuição e inversa da distribuição e as denominamos `dexp`, `pexp` e `qexp`, respectivamente. Estas funções são:

```
# função densidade exponencial
# f(x) = lambda * exp(-lamb * x)
def dexp(x, lamb = 1):
  if any(x < 0):
    print('Elementos de x devem ser todos não-negativos!')
    return
  else:
    fx = lamb * np.exp(-lamb * x)
  return fx


# função de distribuição exponencial
# F(x) = 1 - exp(-lamb * x)
def pexp(x, lamb = 1):
  if any(x < 0):
    print('Elementos de x devem ser todos não-negativos!')
    return
  else:
    Fx = 1.0 - np.exp(-lamb * x)
  return Fx

# função inversa da distribuição exponencial
# q = -log(1 - p) / lamb
def qexp(x, lamb = 1):
  if any(p >= 1) or any(p < 0):
    print('Elementos de p devem estar [0, 1)!')
    return
  else:
    q = np.log(1 - p) / lamb
  return q

# Exemplo
import numpy as np
lamb = 0.1
x = np.array([29.95732])
p  = np.array([0.95])
dexp(x, lamb)
pexp(x, lamb)
qexp(p, lamb)
```

```
array([0.005])
```

```
array([0.94999999])
```

```
array([-29.95732274])
```

Estas três funções foram facilmente implementadas, pois conseguimos explicitamente obter a função de distribuição e sua inversa. Se por outro lado tivéssemos o modelo normal

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\} \tag{6.4}$$

não poderíamos obter explicitamente a função de distribuição e muito menos a função inversa da função de distribuição de probabilidade. Como para a grande maioria dos modelos probabilísticos encontramos os

mesmos problemas, necessitamos de métodos numéricos para obter estas funções. Por essa razão iremos apresentar alguns detalhes neste capítulo sobre alguns métodos gerais e específicos de alguns modelos. No caso discreto podemos utilizar algoritmos sequenciais, porém como boa parte dos modelos probabilísticos possuem relação exata com modelos contínuos, encontramos os mesmos problemas.

6.2 Modelos Probabilísticos Discretos

Vamos apresentar algoritmos para obtenção da função de probabilidade, função de distribuição de probabilidade e sua inversa para os modelos discretos que julgamos mais importantes, os modelos binomial e Poisson. Vamos iniciar nossa discussão pelo modelo binomial. Considerando uma variável aleatória X com distribuição binomial, então a sua função de probabilidade é dada por:

$$P(X = x) = \binom{n}{x} p^x (1-p)^{(n-x)} \tag{6.5}$$

em que n é o tamanho da amostra ou números de ensaios de Bernoulli independentes com probabilidade de sucesso p e x é o número de sucessos, que pertence ao conjunto finito $0, 1, \cdots, n$. A função de distribuição de probabilidade é dada por:

$$F(x) = \sum_{t=0}^{x} \binom{n}{t} p^t (1-p)^{n-t}. \tag{6.6}$$

Vimos no capítulo 3, equação (Equation 3.14) que podemos obter as probabilidades acumuladas de forma recursiva. Sendo $P(X = 0) = (1-p)^n$, obtemos as probabilidades para os demais valores de X, ou seja, para $x = 1, \cdots, n$ de forma recursiva utilizando a relação $P(X = x) = P(X = x-1)[(n-x+1)/x][p/(1-p)]$. Este mesmo algoritmo apropriadamente modificado é utilizado para obtermos as probabilidades acumuladas e a inversa da função de distribuição. Se os valores de n e de p forem grandes, este algoritmo pode ser ineficiente. Para o caso do parâmetro p ser grande (próximo de um) podemos utilizar a propriedade da binomial dada por: se $X \sim Bin(n, p)$, então $Y = n - X \sim Bin(n, 1-p)$. Assim, podemos, por exemplo, obter $P(X = x)$ de forma equivalente por $P(Y = n - x)$ e $P(X \leq x) = P(Y \geq n - x)$, que pode ser reescrito por $F_X(x) = 1 - F_Y(n - x - 1)$, exceto para $x = n$, em que $F_X(x) = 1$. Desta forma trocamos de variável para realizar o cálculo e retornamos o valor correspondente a do evento original.

```
# função de probabilidade e distribuição
# da binomial(n, p)
def dpbinom(x, n, p = 0.5):
  if p > 0.5:
    pp = 1 - p
    x  = n - x
  else:
    pp = p
  qq = 1 - pp
  if (x < 0):
    f = 0
  else:
    f  = qq**n
  r = pp / qq
  g = r * (n + 1)
  cdf = f
  if x > 0:
    u = 0
```

```
    while(u < x):
      u += 1
      f *= (g / u - r)
      cdf += f
  if p > 0.5:
    cdf = 1 - cdf + f
  return {'pdf': f,'cdf': cdf}

# inversa da função distribuição
# binomial(n, p)
def qbinom(prob, n, p):
  q = 1 - p
  f = q**n
  r = p / q
  g = r * (n + 1)
  x = 0
  cdf = f
  while ((prob - cdf) >= 1.e-11):
    x += 1
    f *= (g / x - r)
    cdf += f
  return x

# Exemplo de uso
p = 0.5713131
x = 5 # retire a tolerância, qbinom falha
n = 20
res = dpbinom(x, n, p)
print(res)
prob = res['cdf']
qbinom(prob, n, p)
```

```
{'pdf': 0.0028636362867051975, 'cdf': 0.0036697111708624904}
```

```
5
```

Se usarmos a propriedade de que se X é binomial com parâmetros n e p, então $n - X$ também é binomial com parâmetros n e $1 - p$. Como o algoritmo é mais eficiente quando $p < 0{,}5$, pois $q = 1 - p$ está mais próximo de 1 e a velocidade de execução do algoritmo é proporcional a np. Assim, se ganha em tempo de processamento e precisão, se $p > 0{,}5$ e nós realizarmos a troca para $n - X$. O `script` a seguir apresenta esta alteração na função `qbinom`:

```
# inversa da função distribuição
# binomial(n, p), com troca se p > 0.5
def qqbinom(prob, n, p):
  if p > 0.5:
    pp = 1 - p
    probq  = 1 - prob
  else:
    pp = p
    probq = prob
  qq = 1 - pp
  f = qq**n
```

```
  r = pp / qq
  g = r * (n + 1)
  x = 0
  cdf = f
  while ((probq - cdf) >= 1.e-11):
    x += 1
    f *= (g / x - r)
    cdf += f
  if p > 0.5:
    if (probq - cdf) <= 1.e-11:
      x = n - x - 1
    else:
      x = n - x
  return x

# Exemplo de uso
p = 0.5713131
x = 5 # retire a tolerância, qbinom falha
n = 20
res = dpbinom(x, n, p)
print(res)
prob = res['cdf']
qqbinom(prob, n, p)
```

```
{'pdf': 0.0028636362867051975, 'cdf': 0.0036697111708624904}
```

```
5
```

A função de distribuição binomial possui uma relação exata com a função distribuição beta. Se tivermos algoritmos para obter a beta incompleta podemos utilizar esta relação para obter probabilidades da distribuição binomial. A função de distribuição binomial, $F(x)$, se relaciona com a função beta incompleta, $Ip(\alpha,\beta)$, da seguinte forma:

$$F(x;n,p) = \begin{cases} 1 - I_p(x+1,n-x), & \text{se } 0 \leq x < n \\ 1, & \text{se } x = n. \end{cases} \tag{6.7}$$

Desta forma podemos antever a importância de conhecermos algoritmos de integração numérica das distribuições contínuas. Isso fica mais evidente quando percebemos que para grandes valores de n os algoritmos recursivos são ineficientes e podem se tornar imprecisos. Na seção Section 6.3 apresentaremos alguns métodos gerais de integração para funções contínuas. No `script` seguinte utilizamos a relação (Equation 6.7}) para a obtenção da função de distribuição da binomial.

```
# função de probabilidade e distribuição da
# binomial(n, p) a partir da relação com a
# função beta incompleta - cdf da beta
from scipy.stats import beta
def pbinombeta(x, n, p = 0.5):
  if p <= 0 or p >= 1:
    print('p deve estar no intervalo (0, 1)')
    return
  if (x < n):
    a =  x + 1
    b  = n - x
```

```
    cdf = 1 - beta.cdf(p, a, b)
  else:
    cdf = 1
  return cdf

# Exemplo
p = 0.5713131
n = 20
x = 3
pbinombeta(x, n, p)
import scipy.stats as sps
sps.binom.cdf(x, n, p)
```

```
np.float64(0.00013459360572398715)
```

```
np.float64(0.00013459360572400553)
```

A distribuição Poisson é a segunda que consideraremos. Existe relação da função de distribuição da Poisson com a gama incompleta. Se uma variável aleatória discreta X com valores $x = 0,\ 1,\ 2,\ 3,\ \cdots$ possui distribuição Poisson com parâmetro λ, então podemos definir a função de probabilidade por:

$$P(X = x) = \frac{\lambda^x e^{-\lambda}}{x!}. \tag{6.8}$$

Podemos obter probabilidades desta distribuição utilizando a relação da gama incompleta com a função de distribuição Poisson

$$F(x|\lambda) = 1 - I_\lambda(\alpha = x + 1), \qquad x = 0, 1, 2, \cdots . \tag{6.9}$$

sendo

$$I_x(\alpha) = \frac{1}{\Gamma(\alpha)} \int_0^x e^{-t} t^{\alpha-1} dt,$$

a função gama incompleta, para valores de $x \geq 0$. O `script` para isso é

```
# função de probabilidade e distribuição da
# Poisson(lamb) usando a relação com a função
# gama incompleta - cdf da gama
from scipy.stats import gamma
def pgama(x, lamb = 1.0):
  if lamb <= 0:
    print('lambda deve ser maior que 0')
    return
  if (x >= 0):
    a =  x + 1.0
    cdf = 1 - gamma.cdf(lamb, a)
  else:
    cdf = 0
  return cdf

# Exemplo
lamb = 0.85
x = 3.0
pgama(x, lamb)
```

```
import scipy.stats as sps
sps.poisson.cdf(x, lamb)
```

```
np.float64(0.9888689674521238)
```

```
np.float64(0.9888689674521238)
```

6.3 Modelos Probabilísticos Contínuos

Para obtermos a função de distribuição ou a inversa da função de distribuição de modelos probabilísticos contínuos via de regra devemos utilizar métodos numéricos. Existem exceções como, por exemplo, o modelo exponencial descrito no início deste capítulo. A grande maioria dos modelos probabilísticos utilizados atualmente faz uso de algoritmos especialmente desenvolvidos para realizar quadraturas em cada caso particular. Estes algoritmos são, em geral, mais precisos do que os métodos de quadraturas, como são chamados os processos de integração numérica. Existem vários métodos de quadraturas numéricas como o método de Simpson, as quadraturas gaussianas e os métodos de Monte Carlo. Vamos apresentar apenas os métodos de Simpson e de Monte Carlo, que são mais simples e são menos precisos.

Vamos utilizar o modelo normal para exemplificar o uso desses métodos gerais de integração nas funções contínuas de probabilidades. Uma variável aleatória X com distribuição normal com média μ e variância σ^2 possui função densidade dada por:

$$f(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(x-\mu)^2}{2\sigma^2}\right\}. \tag{6.10}$$

A função de distribuição de probabilidade não pode ser obtida explicitamente e é definida por:

$$F(x) = \int_{-\infty}^{x} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left\{-\frac{(t-\mu)^2}{2\sigma^2}\right\} dt. \tag{6.11}$$

Em geral utilizamos a normal padrão para aplicarmos as quadraturas. Neste caso a média é $\mu = 0$ e a variância é $\sigma^2 = 1$. Assim, representamos frequentemente a densidade por $\phi(z)$, a função de distribuição por $\Phi(z)$ e a sua inversa por $\Phi^{-1}(p)$, em que $0 < p < 1$. A variável Z é obtida de uma transformação linear de uma variável X normal por $Z = (X - \mu)/\sigma$.

Vamos apresentar de forma bastante resumida a regra trapezoidal estendida para realizarmos quadraturas de funções. Seja f_i o valor da função alvo no ponto x_i, ou seja, $f_i = f(x_i)$, então a regra trapezoidal é dada por:

$$\int_{x_1}^{x_n} f(x)dx = \frac{h}{2}(f_1 + f_n) + O(h^3 f''), \tag{6.12}$$

em que $h = x_n - x_1$ e o termo de erro $O(h^3 f'')$ significa que a verdadeira resposta difere da estimada por uma quantidade que é o produto de h^3 pelo valor da segunda derivada da função avaliada em algum ponto do intervalo de integração determinado por x_1 e x_n, sendo $x_1 < x_n$.

Esta equação retorna valores muito pobres para as quadraturas da maioria das funções de interesse na estatística. Mas se utilizarmos esta função $n - 1$ vezes para fazer a integração nos intervalos (x_1, x_2), (x_2, x_3), $\cdots$, (x_{n-1}, x_n) e somarmos os resultados, obteremos a fórmula composta ou a regra trapezoidal estendida por:

$$\int_{x_1}^{x_n} f(x)dx = h\left(\frac{f_1}{2} + f_2 + f_3 + \cdots + f_{n-1} + \frac{f_n}{2}\right) + O\left(\frac{(x_n - x_1)^3 f''}{n^2}\right). \tag{6.13}$$

em que $h = (x_n - x_1)/(n-1)$.

Uma das melhores forma de implementar a função trapezoidal é discutida e apresentada por Press et al. [1992]. Nesta implementação inicialmente é tomada a média da função nos seus pontos finais de integração x_1 e x_n. São realizados refinamentos sucessivos. No primeiro estágio devemos acrescentar o valor da função avaliada no ponto médio dos limites de integração e no segundo estágio, os pontos intermediários 1/4 e 3/4 são inseridos e assim sucessivamente. Sejam `func()` a função de interesse, a e b os limites de integração e n o número de intervalos de integração previamente definido, então podemos obter a função `trapzd()` adaptando a mesma função implementada em Fortran por Press et al. [1992] da seguinte forma:

```
# Esta rotina calcula o n-ésimo estágio de refinamento da
# integração trapezoidal estendida, em que, func é uma
# função externa de interesse que deve ser chamada para
# n=1, 2, etc. e o valor de s deve ser retornado a função
# em cada nova chamada.
import math
def trapzd(func, a, b, s, n):
  if n == 1:
    s = 0.5 * (b - a) * (func(a) + func(b))
  else:
    it  = 2**(n - 2)
    dl = (b - a) / it   # espaço entre pontos
    x   = a + 0.5 * dl
    soma = 0.0
    for j in range(it):
      soma += func(x)
      x += dl
    # refina o valor de s
    s = 0.5 * (s + (b - a) * soma / it)
  return s
# função para executar quadraturas de
# funções definidas em func()
# até que uma determinada
# precisão tenha sido alcançada
def qtrap(func, a, b):
  eps  = 1.0e-11
  nmax = 25
  olds = -1.e30 # impossível valor para quadratura inicial
  n    = 1
  fim  = 0
  s = -1.0e15  # valor arbitrário para iniciar o loop
  while abs(s-olds) >= eps * abs(olds) and fim == 0:
    olds = s
    s = trapzd(func, a, b, s, n)
    # evitar convergência prematura espúria
    if n > 5:
      if s < 0.0 and math.isclose(olds, 0.0):
        fim = 1
    n = n  + 1
    if n > nmax:
      fim  = 1
  if n > nmax:
    print('Limite de passos ultrapassado!')
```

```
    return
  return s
```

Devemos chamar a função `qtrap()` especificando a função de interesse `func()` e os limites de integração. Assim, para a normal padrão devemos utilizar as seguintes funções:

```
# fdp da normal padrão
def dnorm(x):
  fx = (1.0 / (2.0 * math.pi)**0.5) * math.exp(-x**2 / 2)
  return fx

def pnorm(x):
  p = qtrap(dnorm, 0.0, abs(x))
  if x  > 0:
    p += 0.5
  else:
    p = 0.5 - p
  return p

# exemplo
z = 1.96
print(pnorm(z))
print(sps.norm.cdf(z))
```

```
0.9750021048512256
0.9750021048517795
```

É evidente que temos métodos numéricos gerais mais eficientes do que o apresentado. As quadraturas gaussianas representam uma classe de métodos que normalmente são mais eficientes que este apresentado. Não vamos nos atentar para estes métodos gerais, por duas razões básicas. A primeira é que existem métodos específicos para obtermos as quadraturas dos principais modelos probabilísticos que são mais eficientes. A maior eficiência destes métodos específicos se dá por dois aspectos: velocidade de processamento e precisão. A segunda razão refere-se ao fato de que no Python estas rotinas específicas já estão implementadas. Como ilustração, podemos substituir a função que implementamos `pnorm()` pela pré-existente no Python `normal.cdf()` do `scipy.stats`. Vamos ilustrar um destes algoritmos especializados para obtermos a função de distribuição da normal padrão. O algoritmo de Hasting possui erro máximo de 1×10^{-6} e é dado por:

$$\Phi(x) = \begin{cases} G & \text{se } x \leq 0 \\ 1 - G & \text{se } x > 0 \end{cases} \tag{6.14}$$

sendo G dado por

$$G = (a_1\eta + a_2\eta^2 + a_3\eta^3 + a_4\eta^4 + a_5\eta^5)\phi(x)$$

em que

$$\eta = \frac{1}{1 + 0{,}2316418|x|}$$

e $a_1 = 0{,}319381530$, $a_2 = -0{,}356563782$, $a_3 = 1{,}781477937$, $a_4 = -1{,}821255978$ e $a_5 = 1{,}330274429$.

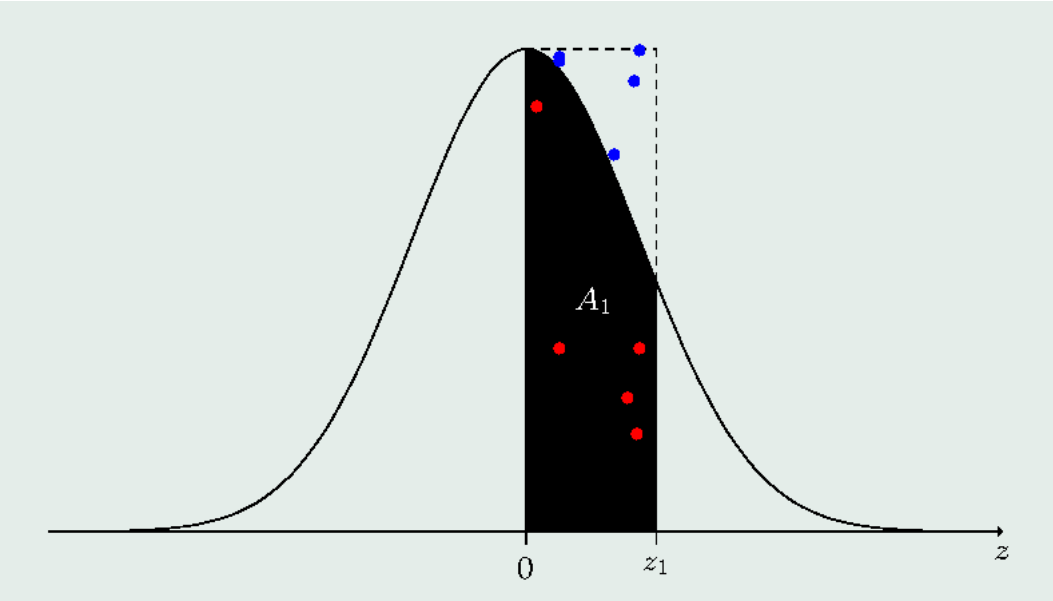
O resultado da implementação deste procedimento é:

```
# CDF da normal padrão:
# aproximação de Hasting
def phnorm(x):
  eta = 1 / (1 + abs(x) * 0.2316418)
  a1  = 0.319381530; a2 = -0.356563782
  a3 = 1.781477937; a4 = -1.821255978
  a5  = 1.330274429
  phi = 1 / (2 * math.pi)**0.5 * math.exp(-x * x / 2)
  g = (a1*eta+a2*eta**2+a3*eta**3+a4*eta**4+a5*eta**5)*phi
  if (x <= 0):
    cdf = g
  else:
    cdf = 1 - g
  return cdf

# exemplo
z = 1.96
p = phnorm(z)
p
```

```
0.9750021668514388
```

Com o uso desta função ganhamos em precisão, principalmente para grandes valores em módulo do limite de integração superior e principalmente ganhamos em tempo de processamento. Podemos ainda abordar os métodos de Monte Carlo, que são especialmente úteis para integrarmos funções complexas e multidimensionais. Vamos apresentar apenas uma versão bastante rudimentar deste método. A ideia é determinar um retângulo que engloba a função que desejamos integrar e bombardearmos a região com pontos aleatórios (u_1, u_2) provenientes da distribuição uniforme.

Contamos o número de pontos sob a função e determinamos a area correspondente, a partir da proporcionalidade entre este número de pontos e o total de pontos simulados em relação a área sob a função na região de interesse em relação à área total do retângulo. Se conhecemos o máximo da função f_{max}, podemos determinar este retângulo completamente. Assim, o retângulo de interesse fica definido pela base (valor entre 0 e z_1 em módulo, sendo z_1 fornecido pelo usuário) e pela altura (valor da densidade no ponto de máximo). Assim, a área deste retângulo é $A = |z_1| f_{max}$. No caso da normal padrão, o máximo obtido para $z = 0$ é $f_{max} = 1/\sqrt{2\pi}$ e a área do retângulo $A = |z_1|/\sqrt{2\pi}$. Se a área sob a curva, que desejamos estimar, for definida por A_1, podemos gerar números uniformes u_1 entre 0 e $|z_1|$ e números uniformes u_2 entre 0 e f_{max}. Para cada valor u_1 gerado calculamos a densidade $f_1 = f(u_1)$. Assim, a razão entre as áreas A_1/A é proporcional a razão n/N, em que n representa o número de pontos (u_1, u_2) para os quais $u_2 \leq f_1$ e N o número total de pontos gerados. Logo, a integral é obtida por $A_1 = |z_1| \times f_{max} \times n/N$, em que z_1 é o valor da normal padrão para o qual desejamos calcular a área que está compreendida entre 0 e $|z_1|$, para assim obtermos a função de distribuição no ponto z_1, ou seja, para obtermos $\Phi(z_1)$. Assim, se $z_1 \leq 0$, então $\Phi(z_1) = 0{,}5 - A_1$ e se $z_1 > 0$, então $\Phi(z_1) = 0{,}5 + A_1$. Veja a seguinte figura ilustrativa:



Para o caso particular da normal padrão, implementamos a seguinte função:

```
# Quadratura da normal padrão
# via simulação Monte Carlo
import numpy as np
def mcpnorm(x, N = 2000):
  max = 1 / (2 * math.pi)**0.5
  z  = abs(x)
  u1  = np.random.uniform(0, z, N)   # 0 < u1 < z
  u2  = np.random.uniform(0, max, N) # 0 < u2 < max
  f1  = (1/(2*math.pi)**0.5)*np.exp(-u1**2/2) # f1(u1): N(0,1)
  n   = len(f1[u2 <= f1])
  g   = n / N * max * z
  if (x < 0):
    cdf = 0.5 - g
  else:
    cdf = 0.5 + g
  return(cdf)

# exemplo
z = 1.96
N = 1500000
mcpnorm(z, N)
print('Erro de MC: ',1 / N**0.5)
```

```
0.974727090055601
```

```
Erro de MC:  0.0008164965809277261
```

Outra forma de obtermos uma aproximação da integral

$$\int_0^{abs(z)} \frac{1}{\sqrt{2\pi}} e^{-t^2/2} dt = \int_0^{abs(z)} \phi(t) dt$$

por Monte Carlo é gerarmos m números uniformes entre 0 e $abs(z)$, digamos z_1, z_2, ..., z_m e obter

$$\int_0^{abs(z)} \phi(t)dt \approx \Delta z \left[\frac{1}{m} \sum_{i=1}^{m} \phi(z_i) \right],$$

em que $\phi(t) = 1/\sqrt{2\pi} \times \exp\{-t^2/2\}$ é a função densidade normal padrão avaliada no ponto t e $\Delta z = abs(z) - 0$. A ordem de erro desse processo é dada por $O(m^{-1/2})$. O programa Python para obter o valor da função de distribuição normal padrão $\Phi(z)$ utilizando essas ideias é apresentado a seguir. Por meio de uma comparação dessa alternativa Monte Carlo com a primeira podemos verificar que houve uma grande diminuição do erro de Monte Carlo no cálculo dessa integral, nessa nova abordagem. Muitas variantes e melhorias nesse processo podem ser implementadas, mas nós não iremos discuti-las aqui.

```
# Quadratura da normal padrão via simulação
# Monte Carlo. Segunda forma de obter a
# integral: forma clássica
import numpy as np
import math
import scipy.stats as sps
import matplotlib
import matplotlib.pyplot as plt
def pmcnorm(z, m = 2000):
  x = np.random.uniform(0, abs(z), m)
  p = (1/(2 * math.pi)**0.5)*np.exp(-(x**2)/2)
  p = abs(z) * np.mean(p)
  if z < 0:
    p = 0.5 - p
  else:
    p += 0.5
  return p

# Exemplo
m = 15000   # número de pontos muito inferior ao caso anterior
z = 1.96
pmcnorm(z, m)  # Estimativa de Monte Carlo
p = sps.norm.cdf(z)       # valor real
p
# Ordem de erro: O(m^(-1/2))
1/m**0.5
# verificação da convergência
np.random.seed(1000)
x = np.random.uniform(0, abs(z), m)
pdf = (1/(2*math.pi)**0.5) * np.exp(-(x**2)/2)
if (z < 0):
   cdf = 0.5 - abs(z) * np.cumsum(pdf) / np.arange(1,m+1)
else:
  cdf = 0.5 + abs(z) * np.cumsum(pdf) / np.arange(1,m+1)
plt.plot(np.arange(1,m+1), cdf)
plt.plot(np.arange(1,m+1), [p]*m)
plt.xlabel('Iterações', fontsize=15)
plt.ylabel('P(Z<=z)', fontsize=15)
```

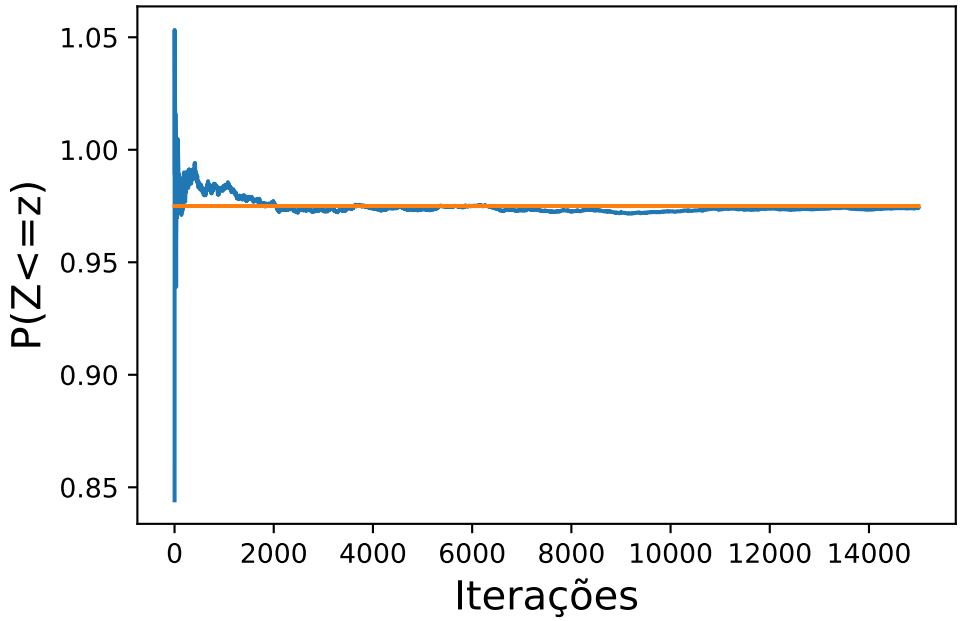
```
np.float64(0.9716597131275779)
```

```
np.float64(0.9750021048517795)
```

```
0.00816496580927726
```

```
Text(0.5, 0, 'Iterações')
```

```
Text(0, 0.5, 'P(Z<=z)')
```



6.4 Quadraturas Gaussianas

As quadraturas gaussianas desempenham um papel preponderante na estatística, pois os principais algoritmos de obtenção de probabilidades e quantis utilizam tais métodos implicitamente. Nesta seção, apresentaremos, de forma bastante simplificada e sem aprofundar nos aspectos matemáticos mais técnicos, o principal método de quadratura gaussiana. Os interessados em uma maior pormenorização desses aspectos podem consultar inúmeros livros específicos. Recomendamos por exemplo a leitura de Quarteroni et al. [2000].

A base dessas quadraturas são as interpolações baseadas em polinômios ortogonais. Seu uso extrapola o tema das quadraturas numéricas, podendo ser usados para aproximar soluções de quadrados mínimos e diferenciação numérica. Uma quadratura gaussiana de n pontos é aquela que fornece resultados exatos para a integração de um polinômio de grau igual ou inferior a $2n-1$. O domínio das quadraturas é, sem perda de generalidade e por convenção, assumido como sendo $[-1,1]$. A regra geral para a quadratura gaussiana de n pontos para uma função $f(x)$, no domínio $[-1,1]$, é dada por

$$\int_{-1}^{1} f(x)dx \approx \sum_{i=1}^{n} w_i f(x_i), \tag{6.15}$$

em que x_i e w_i são denominados de nós e pesos, respectivamente, da quadratura.

Os nós são valores do intervalo $[-1,1]$ e os pesos são positivos. Devemos escolhê-los de forma apropriada para que o resultado da integral seja aproximado de forma acurada pela soma da direita. Muitas vezes,

usamos uma função peso $w(x_i)$, tal que a integral (Equation 6.15) possa ser representada por

$$\int_{-1}^{1} f(x)dx = \int_{-1}^{1} w(x)g(x)dx \approx \sum_{i=1}^{n} w_i g(x_i), \tag{6.16}$$

em que $g(x)$ é tal que $f(x) = w(x)g(x)$ e w_i nesse caso são pesos alternativos.

Por exemplo, para o caso específico da função $f(x) = e^{2x}$, usando pesos $w_1 = w_2 = 1$ e nós $x_1 = \sqrt{3}/3$ e $x_2 = -\sqrt{3}/3$, no domínio de -1 a 1 resulta em

$$\int_{-1}^{1} e^{2x}dx \approx f(\sqrt{3}/3) + f(-\sqrt{3}/3) = 3{,}4882.$$

O valor exato da integral é

$$\int_{-1}^{1} e^{2x}dx = \left[\frac{e^{2x}}{2}\right]_{-1}^{1} = 3{,}6269.$$

É surpreendente que a soma de $f(x_1) + f(x_2)$ resulte em valores exatos para polinômios de grau até 3, $2n-1$, pois $n = 2$. O que temos que fazer é encontrar mecanismos para obter os pesos e os nós de integração. Para um intervalo de integração diferente de $[-1, 1]$, ou seja, $[a, b]$, $b > a$, temos que a seguinte transformação não altera a precisão da integração

$$\begin{aligned} \textstyle\int_a^b f(x)dx = & \quad \frac{b-a}{2} \textstyle\int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right) dx \\ \approx & \quad \frac{b-a}{2} \textstyle\sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}x_i + \frac{a+b}{2}\right). \end{aligned}$$

Vale a pena ressaltar que os nós utilizados nas quadraturas são as raízes dos polinômios ortogonais de alguma das famílias gaussianas, normalmente empregadas. Entre elas, podemos citar os polinômios de Legendre, Laguerre, Hermite, Chebyshev e Jacobi. Descreveremos apenas a quadratura Gauss-Legendre na sequência.

A quadratura denominada Gauss-Legendre é utilizada para intervalos de integração definidos por $[-1, 1]$, podendo ser extrapolado para intervalos mais gerais, se utilizarmos a expressão (Equation 6.17). Assim, para o domínio $[-1, 1]$, a quadratura Gauss-Legendre pode ser aplicada por

$$\int_a^b f(x)dx \approx \begin{cases} \displaystyle\sum_{i=1}^{n} w_i f(x_i) & \text{para } a = -1 \text{ e } b = 1 \\ \displaystyle\sum_{i=1}^{n} w_i g(y_i) & \text{para } a \text{ e } b \text{ reais quaisquer,} \end{cases} \tag{6.17}$$

em que $g(y)$ é dada por

$$g(y) = \frac{b-a}{2} f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right), \tag{6.18}$$

em que $y = (b-a)x/2 + (b+a)/2$.

O próximo passo, na aplicação das quadraturas gaussianas, é calcular os nós e pesos para cada uma delas. Apresentaremos isso, na forma de uma sequência de passos. Utilizaremos as seguintes quantidades para todos os métodos. O escalar μ_0 que é definido por

$$\mu_0 = \int_a^b w(x)dx$$

e os vetores de coeficientes $\mathbf{a}$ $(n \times 1)$ e $\mathbf{b}$ $(n-1 \times 1)$, sendo n o número de pontos da quadratura a ser realizada.

Posteriormente, ainda utilizaremos uma matriz denotada por $\mathbf{J}$, $(n \times n)$, para a qual obteremos os autovalores e autovetores. Como utilizaremos um problema de álgebra, solucionado pela obtenção de autovalores e autovetores, esse método é conhecido como algoritmo de Golub–Welsch. Descreveremos, inicialmente, a obtenção dos vetores $\mathbf{a}$ e $\mathbf{b}$ e da constante μ_0 para o método de quadratura Gauss-Legendre. Detalhes técnicos de como isso pode ser realizado e as bases teóricas podem ser encontradas em Quarteroni et al. [2000].

Para a quadratura Gauss-Legendre temos:

1. faça $\mu_0 = 2$;
2. $\mathbf{a} = \mathbf{0}$ $(n \times 1)$;
3. $\mathbf{b} = [b_j]$ $(n-1 \times 1)$, tal que

$$b_j = \frac{j}{\sqrt{4j^2 - 1}}, \qquad j = 1, 2, \ldots, n-1.$$

Definidas as quantidades necessárias $\mathbf{a}$, $\mathbf{b}$ e μ_0, devemos aplicar a segunda parte do algoritmo para obter os nós x_i's e os pesos w_i's, $i = 1, 2, \ldots, n$. Utilizamos, para isso, o algoritmo de Golub–Welsch, pelo qual determinamos a matriz $\mathbf{J}$ a partir dessas quantidades e determinamos seus autovalores e autovetores. A matriz $\mathbf{J}$ é tridiagonal simétrica definida por

$$\mathbf{J} = \begin{bmatrix} a_1 & b_1 & 0 & \ldots & \ldots & \ldots \\ b_1 & a_2 & b_2 & 0 & \ldots & \ldots \\ 0 & b_2 & a_3 & b_3 & 0 & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & 0 \\ \ldots & \ldots & 0 & b_{n-2} & a_{n-1} & b_{n-1} \\ \ldots & \ldots & \ldots & 0 & b_{n-1} & a_n \end{bmatrix}.$$

Determinamos os autovalores e autovetores de $\mathbf{J}$ e os denotamos por

$$\mathbf{x}^* = \begin{bmatrix} x_1^* \\ x_2^* \\ \vdots \\ x_n^* \end{bmatrix} \qquad \text{e} \qquad \mathbf{P} = \begin{bmatrix} p_{11} & p_{21} & \cdots & p_{n1} \\ p_{12} & p_{22} & \cdots & p_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ p_{1n} & p_{2n} & \cdots & p_{nn} \end{bmatrix},$$

respectivamente.

Modificamos a notação usual para representar os elementos de uma matriz. Assim, verificamos que na matriz $\mathbf{P}$, o primeiro índice do elemento p_{ij} refere-se ao índice do i-ésimo autovetor e o segundo índice ao j-ésimo elemento desse autovetor. Por isso, a notação não usual dos índices nessa matriz. Portanto, cada coluna de $\mathbf{P}$, corresponde a um dos n autovetores associados a cada um dos autovalores x_i^* ordenados em ordem decrescente.

Em seguida, podemos obter os nós (pontos), tomando simplesmente os autovalores em ordem crescente, ou seja, o vetor dos nós $\mathbf{x}$, com componentes x_i, é definido por

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_n^* \\ x_{n-1}^* \\ \vdots \\ x_1^* \end{bmatrix}, \tag{6.19}$$

em que x_i^* é o i-ésimo autovalor da matriz tridiagonal $\mathbf{J}$.

Podemos constatar que os nós são os autovalores dessa matriz apresentados em ordem reversa, ou seja, em ordem crescente ao invés da habitual forma de apresentá-los, que é a ordem decrescente. Finalmente, podemos obter os pesos como sendo uma função de μ_0 e dos primeiros elementos de cada autovetor de $\mathbf{J}$, também considerado em ordem reversa, considerando as ordenações clássicas do maior para o menor autovalor. Essa função busca realizar uma normalização adequada para garantir a validade da quadratura.

Assim, temos que os pesos das quadraturas são obtidos por

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_{n-1} \\ w_n \end{bmatrix} = \mu_0 \begin{bmatrix} p_{n,1}^2 \\ p_{n-1,1}^2 \\ p_{n-2,1}^2 \\ \vdots \\ p_{2,1}^2 \\ p_{1,1}^2 \end{bmatrix}, \tag{6.20}$$

sendo p_{ij} o j-ésimo elemento do i-autovetor de $\mathbf{J}$ correspondente ao i-autovalor x_i^*.

Podemos observar que os valores w_i's dependem de p_{ij} em ordem reversa de importância, sendo essa ordem determinada pelos autovalores de $\mathbf{J}$.

Com os pesos, nós e funções pesos, quando ela não for a unidade, podemos aplicar os métodos de quadraturas apresentados anteriormente. Nosso próximo passo é implementar funções para obter os pesos e os nós e exemplificarmos.

```
# Função para obter nós e pesos
# da quadratura Gauss-Legendre
import numpy as np
import scipy as sp
def gausslegendquad(n):
  n = int(n)
  if n < 0:
    print('É necessário um número não negativo de nós!')
    return
  if n == 0:
    return {'xi':[], 'wi': []}
  i = np.arange(1, n + 1)
  i1 = np.delete(i,n - 1)
  mu0 = 2
  b = i1 / np.sqrt(4 * i1**2 - 1)
  J = np.zeros((n*n))
  J[(n + 1) * (i1-1) + 1] = b
  J[(n + 1) * i1 - 1] = b
  J = J.reshape(n, n)
  va, ve = sp.linalg.eigh(J)
  w = ve[0,:]
  w = mu0 * w**2
  return {'xi': va, 'wi': w}

# função genérica para obter a
# quadratura no [-1,1]
def quadgl(func, n = 8):
  gl = gausslegendquad(n)
  res = np.sum(gl['wi'] * func(gl['xi']))
```

```
  return res

# exemplo: f(x) = exp(2x)
# int_{-1}^{1} f(x) dx - vetorizada
def e2x(x):
  return np.exp(2*x)

# função polinomial
def p3(x):
  return x**3-0.4*x**2+2.3*x+5
# Exemplos
n = 4
gausslegendquad(n)
n = 8
quadgl(e2x, n)
print('Valor exato: ',(e2x(1)-e2x(-1))/2)
quadgl(p3, 2)
print('Valor exato wolfram: ',9.73333)
```

```
{'xi': array([-0.86113631, -0.33998104,  0.33998104,  0.86113631]),
 'wi': array([0.34785485, 0.65214515, 0.65214515, 0.34785485])}
```

```
np.float64(3.626860407846865)
```

```
Valor exato:  3.626860407847019
```

```
np.float64(9.73333333333333)
```

```
Valor exato wolfram:  9.73333
```

No exemplo anterior, usamos a função `sp.linalg.eigh()` do `scipy.linalg`, pois a função `np.linalg.eig()` não retorna os autovalores necessariamente em ordem. A função `eigh`, por sua vez, retorna os autovalores ordenados, mas em ordem crescente de valores e não na ordem decrescente, como é feito em muitos outros programas. Podemos aplicar na normal as quadraturas, para ilustrarmos os casos em que os limites não são -1 e 1, incluindo os exemplos anteriores.

```
# função genérica para trocar os
# limites de integração de -1 a 1
# para a e b (finitos)
def hx(func, x, a = -1, b = 1):
  aux = (b - a) / 2
  h = aux * func(aux * x + (a + b) / 2)
  return h

# modificação da quadGL para contemplar
# outros limites que não sejam o -1 e 1
def quadglab(func, n = 8, a = -1, b = 1):
  gl = gausslegendquad(n)
  h = hx(func, gl['xi'], a, b)
  res = np.sum(gl['wi'] * h)
  return res

# X ~ N(mu, sig)
import math
import scipy as sp
```

```
def pnormal(x, mu = 0, sigma = 1, n = 16):
  if math.isclose(x, mu):
    return 0.5
  else:
    if math.isclose(mu, 0) and math.isclose(sigma, 1):
      z = x
    else:
      z = (x - mu)   / sigma
  res = quadglab(sp.stats.norm.pdf, n, 0, abs(z))
  if (x < mu):
    res = 0.5 - res
  else:
    res += 0.5
  return res

# integrar e2x de 1 até 3
a = 1
b = 3
quadglab(e2x, 8, a, b)
print('Valor exato: ',(e2x(b)-e2x(a))/2)
quadglab(p3, 2, a, b) # valor exato 35,73333
x = 1.96
pnormal(x)
print('Valor exato: ',sp.stats.norm.cdf(x))
```

```
np.float64(198.01986869689384)
```

```
Valor exato:  198.01986869690222
```

```
np.float64(35.73333333333333)
```

```
np.float64(0.9750021048517794)
```

```
Valor exato:  0.9750021048517795
```

Nas quadraturas anteriores, podemos substituir nossa função que calcula os nós e os pesos, por outra, que utiliza um método de obtenção de autovalores e autovetores mais eficiente. Esse método é aplicável a matrizes simétricas tridiagonais. O `script` a seguir apresenta essa implementação. Precisamos apenas fornecer um vetor com a diagonal da matriz e outro com os elementos das duas diagonais secundárias, que são idênticos. A função utilizada foi a `sp.linalg.eigh_tridiagonal()` da biblioteca `scipy`.

```
# Função para obter nós e pesos
# da quadratura Gauss-Legendre
# Essa versão dispensa a criação
# da matriz  J, por aplicar um
# processo de obtenção de
# autovalores e autovetores em
# matrizes tridiagonais - + rápido
import numpy as np
import scipy as sp
def gausslegendquad2(n):
  n = int(n)
  if n < 0:
    print('É necessário um número não negativo de nós!')
    return
```

```
  if n == 0:
    return {'xi':[], 'wi': []}
  i = np.arange(1, n + 1)
  i1 = np.delete(i,n - 1)
  mu0 = 2
  b = i1 / np.sqrt(4 * i1**2 - 1)
  a = np.zeros(n)
  va, ve = sp.linalg.eigh_tridiagonal(a, b)
  w = ve[0,:]
  w = mu0 * w**2
  return {'xi': va, 'wi': w}
n = 4
gausslegendquad2(n)
```

```
{'xi': array([-0.86113631, -0.33998104,  0.33998104,  0.86113631]),
 'wi': array([0.34785485, 0.65214515, 0.65214515, 0.34785485])}
```

6.5 Newton-Raphson

Vamos apresentar o método numérico de Newton-Raphson para obtermos a solução da equação

$$z = \Phi^{-1}(p),$$

em que $0 < p < 1$ é o valor da função de distribuição da normal padrão, $\Phi^{-1}(p)$ é a função inversa da função de distribuição normal padrão no ponto p e z o quantil correspondente, que queremos encontrar dado um valor de p. Podemos apresentar esse problema por meio da seguinte equação

$$\Phi(z) - p = 0,$$

em que $\Phi(z)$ é a função de distribuição normal padrão avaliada em z. Assim, nosso objetivo é encontrar os zeros da função $f(z) = \Phi(z) - p$. Em geral, podemos resolver essa equação numericamente utilizando o método de Newton-Raphson, que é um processo iterativo. Assim, devemos ter um valor inicial para o quantil para iniciarmos o processo e no $(n+1)$-ésimo passo do processo iterativo podemos atualizar o valor do quantil por

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)}, \tag{6.21}$$

em que $f'(z_n)$ é derivada de primeira ordem da função para a qual queremos obter as raízes avaliada no ponto z_n. Para esse caso particular temos que

$$z_{n+1} = z_n - \frac{[\Phi(z_n) - p]}{\phi(z_n)}, \tag{6.22}$$

sendo $\phi(z_n)$ a função densidade normal padrão. Como valores iniciais usaremos uma aproximação grosseira, pois nosso objetivo é somente demonstrar o método. Assim, se p for inferior a 0,5 utilizaremos $z_0 = -0{,}1$, se $p > 0{,}5$, utilizaremos $z_0 = 0{,}1$. Obviamente se $p = 0$, a função deve retornar $z = 0$. A função *qPhi* a seguir retorna os quantis da normal, dados os valores de p entre 0 e 1, da média real μ e da variância real positiva σ^2, utilizando para isso o método de Newton-Raphson e a função `phnorm` de Hasting para obtermos o valor da função de distribuição normal padrão.

```
# função auxiliar para retornar o valor da
# função densidade normal padrão
def phi(z):
  return (1 / (2 * math.pi)**0.5 * np.exp(-z * z / 2))

# Método de Newton-Raphson para obtermos quantis
# da distribuição normal com média real mu e desvio
# padrão real positivo sig, dado 0 < p < 1
# utiliza as funções phi(z) e  phnorm(z),
# apresentadas anteriormente
def qphi(p, mu = 0, sig = 1, func = phnorm):
  eps = 1e-11
  if p <= 0 or p >= 1:
    print('Valor de p deve estar entre 0 e 1!')
    return
  if sig  <= 0:
    print('Desvio padrão deve ser maior que 0!')
    return
  if abs(p - 0.5)  <= eps:
    z1 = 0
  else:
    if p < 0.5:
       z0 = -0.1
    else:
      z0 = 0.1
    it = 1
    itmax = 2000
    convergiu = False
    while convergiu == False:
      z1 = z0 - (func(z0) - p) / phi(z0)
      if abs(z0 - z1) <= eps * abs(z0) or it > itmax:
        convergiu = True
      it = it + 1
      z0 = z1
  return {'x': z1 * sig + mu, 'iter': it - 1}

# Exemplo
p   = 0.975
mu  = 0
sig = 1
qphi(p, mu, sig, phnorm)
sps.norm.ppf(p, mu, sig) # pra fins de comparação
```

```
{'x': np.float64(1.9599629237446643), 'iter': 8}
```

```
np.float64(1.959963984540054)
```

Poderíamos, ainda, ter utilizado o método da secante, uma vez que ele não necessita da derivada de primeira ordem, mas precisa de dois valores iniciais para iniciar o processo e tem convergência mais lenta. O leitor é incentivado a consultar Press et al. [1992] para obter mais detalhes. Também poderia ter sido usada a função `pnorm`, que utiliza o método trapezoidal. Nesse caso a precisão seria maior, mas o tempo de processamento também é maior. Para isso bastaria usar na chamada como valor do argumento `func` a função `pnorm`. Também seria possível chamar `mcpnorm` ou `pmcnorm` ou qualquer outra implementação

eficiente que tivermos para a quadratura.

Felizmente o Python também possui rotinas pré-programadas para este e para muitos outros modelos probabilísticos, que nos alivia da necessidade de programar rotinas para obtenção das funções de distribuições e inversas das funções de distribuições dos mais variados modelos probabilísticos existentes.

6.6 Funções Pré-Existentes no Python

Na Tabela Table 3.1 apresentamos uma boa parte das rotinas para gerarmos dados dos mais variados modelos probabilísticos contemplados pelo Python. Logo após a tabela mencionada, apresentamos os procedimentos Python da biblioteca `scipy.stats` para obtermos a função de probabilidade ou densidade, a função de distribuição e a função quantil. Devemos consultar os recursos desta biblioteca para conhecermos os mais diferentes métodos associados a cada um dos seus objetos.

6.7 Exercícios

1. Comparar a precisão dos três algoritmos de obtenção da função de distribuição normal apresentados neste capítulo. Utilizar a função `normal.cdf()` como referência.
2. Utilizar diferentes números de simulações Monte Carlo para integrar a função de distribuição normal e estimar o erro de Monte Carlo relativo e absoluto máximo cometidos em 30 repetições para cada tamanho. Utilize os quantis 1,00, 1,645 e 1,96 e a função `normal.cdf()` como referência.
3. A distribuição Cauchy é um caso particular da t de Student com $\nu = 1$ grau de liberdade. A densidade Cauchy é dada por:

$$f(x) = \frac{1}{\pi(1 + x^2)}$$

   Utilizar o método trapezoidal estendido para implementar a obtenção dos valores da distribuição Cauchy. Podemos obter analiticamente também a função de distribuição e sua inversa. Obter tais funções e implementá-las no Python. Utilize as funções `scipy.stats` pré-existentes para checar os resultados obtidos.
4. Utilizar o método Monte Carlo descrito nesse capítulo, para obter valores da função de distribuição Cauchy, apresentada no exercício proposto 3. Utilizar alguns valores numéricos para ilustrar e comparar com funções implementadas em Python para esse caso. Qual deve ser o número mínimo de simulações Monte Carlo requeridas para se ter uma precisão razoável.

Chapter 7

Conjuntos e Elementos de Análise Combinatória em Python

Neste capítulo, pretendemos apresentar os conceitos básicos de análise combinatória e de conjuntos, de uma forma bastante simples. Vamos fazer algumas funções simples para cálculo de probabilidades computacionalmente em alguns casos particulares. Vimos no capítulo 1 os objetos `conjuntos` (`set`) do Python. O operador `set()` é usado para criar um conjunto. O argumento da função `set()` é uma lista ou uma tupla. São imutáveis, não ordenados e não possuem elemento duplicados.

Várias funções podem ser usadas para este tipo de objeto em Python, como pertencimento (`in`), união (`union` ou `|`), interseção (`intersection` ou `&`) e diferença simétrica (`symmetric_difference` ou `^`), como foi ilustrado no capítulo 1. Também é possível obter diferenças, tipo `A-B` entre dois conjuntos `A` e `B`.

A análise combinatória nos permite resolver inúmeros problemas de probabilidade. Problemas que ocorrem normalmente são listados a seguir:

a. selecionar entre n elementos x deles, $0 \leq x \leq n$, sem repetir nenhum elemento (amostragem sem reposição) onde a ordem não importa: combinação;
b. selecionar entre n elementos x deles, $0 \leq x \leq n$, podendo repetir os elementos selecionados (amostragem sem reposição) onde a ordem importa: arranjos;
c. distribuir n elementos em x, $x = n$, posições de formas diferentes: permutações;
d. caminhos diferentes possíveis de se percorrer com n_1 possibilidades na primeira etapa, n_2 possibilidades na segunda etapa e assim sucessivamente até n_k possibilidades na k-ésima etapa: contagem.

Assim, queremos obter todas as possíveis combinações, permutações, arranjos ou caminhos (contagens) possíveis na resolução de algum problema de probabilidade ou de outra situação em geral.

7.1 Introdução a Análise Combinatória no Python

A análise combinatória e os métodos de contagem são essenciais para se entender probabilidade. No Python podemos computar o número de combinações dado por

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}, \text{ para } 0 \leq x \leq n,$$

por `math.comb(n, x)`. Já á listagem das combinações de n tomados x a x é obtida pelo comando `combinations(range(1,n+1), x)` do pacote `itertools` que deve ser importado. O primeiro argumento da função `combinations` é uma lista, que no caso foi de uma lista indo de 1 a n, em que usamos a função

`range` para obter essa lista. Podemos, em vez disso, usar qualquer outra lista. Podemos usar ainda a função `comb` do `scipy.special` para obtermos o número de combinações de forma alternativa.

O comando `combinations(n, x)` do pacote `itertools` retorna um objeto que deve ser convertido para uma lista com $\binom{n}{x}$ elementos, em que cada elemento da lista corresponde a uma tupla de x elementos (a combinação).

O `script` a seguir ilustra um caso particular destes comandos:

```
from itertools import combinations
from scipy import special as sps
import math
n = 4
x = 2
math.comb(n, x)
sps.comb(n, x) # resultado é um float
comb = list(combinations(range(1,n+1), x))
print(comb)
# listando a combinação 0 da lista
print(comb[0])
len(comb) # número de combinações alternativo
# retorna uma combinação de letras
list(combinations(['a','b','c'], 2))
```

```
6
```

```
np.float64(6.0)
```

```
[(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
(1, 2)
```

```
6
```

```
[('a', 'b'), ('a', 'c'), ('b', 'c')]
```

7.2 Permutações e Arranjos

O número total de permutações de n objetos é dado por $n!$ e com o uso do pacote `itertools`, podemos listá-las em uma lista de tamanho $n!$ com cada elemento sendo uma tupla de tamanho n, com a função `permutations()`;

Também podemos tomar os arranjos de n tomados x a x, com o mesmo comando, cujo número de arranjos é dado por

$$A_{n,x} = \frac{n!}{(n-x)!},$$

sendo que no Python esse valor é obtido por meio da função `math.perm(n, x)`.

O `script` a seguir ilustra estes comandos:

```
# Obter todas as permutações de n
# em n posições - permutação
# ou em x posições x<n, arranjo
from itertools import permutations
import math
n = 3
# permutações
```

```
perm = list(permutations(range(1,n+1)))
print(perm)
# arranjos
x = 2 # x < n
math.perm(n, x)
arran = list(permutations(range(1,n+1), x))
print(arran)
# Imprime as permutações
for i in list(perm):
  print(i)
type(perm[0])
```

```
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
```

```
6
```

```
[(1, 2), (1, 3), (2, 1), (2, 3), (3, 1), (3, 2)]
(1, 2, 3)
(1, 3, 2)
(2, 1, 3)
(2, 3, 1)
(3, 1, 2)
(3, 2, 1)
```

```
tuple
```

Para sortearmos permutações ou arranjos de forma aleatória, o comando `random.sample(range(1,n+1),n)` retorna uma permutação de n tomados n a n, ou seja, é uma amostra aleatória sem reposição. Se quisermos um sorteio de n tomados x a x `random.sample(range(1,n+1),x)`, ou seja, um arranjo de um resultado com x elementos sem repetição (sem reposição). Não é referente ao caso em questão (permutações e arranjos), mas se desejarmos amostragens com reposição, como requerido nos métodos *bootstrap*, usamos a função `random.choices(range(1,n+1),x)`, em que x é um valor entre `1` e `n`. Ambos os comando requerem que importemos a biblioteca `random` do Python. O `script` a seguir ilustra estes comandos:

```
# Obter amostras aleatórias de
# permutações ou arranjos
# (sem reposição), ou amostras com
# reposição
import random
n = 5
# amostras de permutações
amostra = random.sample(range(1,n+1), n)
print(amostra)
x = 3 # x < n
# amostras de arranjos
amostra = random.sample(range(1,n+1), x)
print(amostra)
# amostras com reposição
x =  3
amos = random.choices(range(1,n+1), k=x)
print(amos)
```

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

```
[1, 4, 3]
```

7.3 Contagem

O princípio fundamental da contagem, conhecido por princípio multiplicativo, é utilizado para encontrar o número de possibilidades para um evento constituído de k etapas. As etapas devem ser sucessivas e independentes. Se um evento tem duas ($k = 2$) etapas, sendo que a primeira possui n_1 possibilidades e a segunda, n_2 possibilidades, então existem $n_1 \times n_2$ possibilidades.

Assim, o princípio fundamental da contagem é a multiplicação das opções de cada etapa para determinar o total de possibilidades. Esse conceito é importante para a análise combinatória, área da matemática que reúne os métodos para resolução de problemas incluindo a contagem entre eles e, por isso, é muito útil na investigação de possibilidades para determinar a probabilidade de fenômenos naturais.

Para obtermos todas as possibilidades em k etapas, cada uma com a mesma quantidade n de possibilidades criamos uma classe Python com algumas funções. A função `contag(n, k)` retorna todas as contagens a partir de n possibilidades em k etapas, como, por exemplo, com $n = 2$ sexos de animais em $k = 3$ nascimentos, ou $n = 2$ portas de um prédio em $k = 2$ possibilidades, correspondentes às entradas e às saídas do prédio. Também fizemos o mesmo quando temos uma lista (iterador qualquer) de tamanho k correspondendo ás k etapas em que cada elemento refere-se ao número de possibilidades daquela etapa.

A pergunta, do segundo exemplo, de quais maneiras diferentes uma pessoa pode entrar e sair do edifício com 2 entradas e duas saídas é o que pretendemos listar (enumerar). A função proposta usa um processo recursivo, em que em cada chamada é atualizada a contagem em particular. A recursividade ocorre na função `permuta` ou `permutav`, para o caso de uma lista de etapas.

```
# Classe para obter as contagens de x
# possibilidades em cada k etapas: x, k vezes
# ou x = [n1,n2,...,nk](lista com k nós).
# Usa overload e ilustra a criação
# de uma classe Python
from multipledispatch import dispatch
import numpy as np
class Cont:
  def permuta(self,z, pos, permn, n, k, prim):
    if prim == True:
      z = np.full(k + 1, 1)
      prim = False
      permun = np.append(permn,[z[1:(k+1)]],axis=0)
    else:
      permun = permn
      if z[pos] < n:
        z[pos] = z[pos] + 1
        permun = np.append(permun,[z[1:(k+1)]],axis=0)
      else:
        nachei = True
        if pos == k:
          i = pos - 1
          paratras = True
        else:
          i = pos + 1
          paratras = False
        while nachei == True and i >= 1 and i <= k:
          if z[i] < n:
```

```
            z[i] = z[i] + 1
            if paratras == True:
              z[(i+1):(k+1)] = 1
            permun = np.append(permun,[z[1:(k+1)]],axis=0)
            pos  = k
            nachei = False
          else:
            if (pos == k):
              i = i - 1
            else:
              i = i + 1
    if np.sum(z[1:(k+1)]) == n * k:
      return permun
    else:
      return self.permuta(z, pos, permun, n, k, prim)
  def permutav(self,z, pos, permn, x, k, prim):
    if prim == True:
      k = len(x)
      pos  = k
      z = np.full(k + 1, 1)
      prim = False
      permun = np.append(permn,[z[1:(k+1)]],axis=0)
    else:
      permun = permn
      if z[pos] < x[pos-1]:
        z[pos] = z[pos] + 1
        permun = np.append(permun,[z[1:(k+1)]],axis=0)
      else:
        nachei = True
        if pos == k:
          i = pos - 1
          paratras = True
        else:
          i = pos + 1
          paratras = False
        while nachei == True and i >= 1 and i <= k:
          if z[i] < x[i-1]:
            z[i] = z[i] + 1
            if paratras == True:
              z[(i+1):(k+1)] = 1
              permun = np.append(permun,[z[1:(k+1)]],axis=0)
              pos  = k
              nachei = False
          else:
            if pos == k:
              i = i - 1
            else:
              i = i + 1
    if np.sum(z[1:(k+1)]) == sum(x):
      return permun
    else:
```

```
      return self.permutav(z, pos, permun, x, k, prim)
  @dispatch(int, int)
  def contag(self, x, k):
    y = np.full(k, x)
    prim = True
    permn = np.empty((0, k), int)
    z = np.array([])
    pos = k
    n = x
    res = self.permuta(z, pos, permn, n, k, prim)
    return res
  @dispatch(list)
  def contag(self, x):
    pos = k = len(x)
    prim = True
    permn = np.empty((0, k), int)
    z = np.array([])
    res = self.permutav(z, pos, permn, x, k, prim)
    return res
# Exemplo de uso
res  = Cont()
k = 3
x = 3
val = res.contag(x, k)
print(val)
x = [3, 4, 2]
res.contag(x)
```

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

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

array([[1, 1, 1],
       [1, 1, 2],
       [1, 2, 1],
       [1, 2, 2],
       [1, 3, 1],
       [1, 3, 2],
       [1, 4, 1],
       [1, 4, 2],
       [2, 1, 1],
       [2, 1, 2],
       [2, 2, 1],
       [2, 2, 2],
       [2, 3, 1],
       [2, 3, 2],
       [2, 4, 1],
       [2, 4, 2],
       [3, 1, 1],
       [3, 1, 2],
       [3, 2, 1],
       [3, 2, 2],
       [3, 3, 1],
       [3, 3, 2],
       [3, 4, 1],
       [3, 4, 2]])
```

No exemplo anterior tivemos várias novidades. Criamos uma classe pela primeira vez. Nesta classe implementamos quatro funções. Entre elas, as duas com a função de realizar as listagens de todas as contagens foram implementadas com chamadas recursivas. As outras duas, foram duas versões de uma mesma função com diferentes argumentos, a função `contag`. Para isso usamos a técnica de overloading. Assim, usamos o pacote `multipledispatch` de onde importamos `from multipledispatch import dispatch`. O comando `@dispatch(int, int)` ou `@dispatch(list)` antes da definição da função `contag` indica que ela tem diferentes argumentos. A primeira definição possui dois argumentos inteiros e a segunda, um argumento, correspondente a uma lista. Para exemplificar, criamos um objeto da classe `Cont`, objeto `res`, por meio do qual chamamos o método `contag` duas vezes com argumentos diferentes, o que faz com que ao ser executado, o interpretador escolha a opção apropriada.

Se, por exemplo, quiséssemos obter todas as possibilidades de nascimento quanto ao sexo de famílias de até 4 filhos, teríamos o seguinte espaço amostral de nosso experimento com $2^4 = 16$ elementos:

```
# Espaço amostral dos filhos
# quanto ao sexo
# com n = 4 e 0 <= x <= n
filhos = Cont()
n = 4
res = filhos.contag(2, n) -  1
sexo = ['F','M']
type(res[0])
print(np.array(sexo)[res])
```

```
numpy.ndarray
```

```
[['F' 'F' 'F' 'F']
 ['F' 'F' 'F' 'M']
 ['F' 'F' 'M' 'F']
 ['F' 'F' 'M' 'M']
 ['F' 'M' 'F' 'F']
 ['F' 'M' 'F' 'M']
 ['F' 'M' 'M' 'F']
 ['F' 'M' 'M' 'M']
 ['M' 'F' 'F' 'F']
 ['M' 'F' 'F' 'M']
 ['M' 'F' 'M' 'F']
 ['M' 'F' 'M' 'M']
 ['M' 'M' 'F' 'F']
 ['M' 'M' 'F' 'M']
 ['M' 'M' 'M' 'F']
 ['M' 'M' 'M' 'M']]
```

O Python já tem implementado esses recursos para realizar contagens. A biblioteca `itertools` possui a função `product` que faz o mesmo papel de nossas funções implementadas na classe `cont`.

```
import itertools
l1 = ['F','M']
list(itertools.product(l1, repeat=4))
n = 3
l2 = list(range(1, n+1))
list(itertools.product(l2, repeat=3))
x = [3, 4, 2]
d3 = {}
for i in range(len(x)):
  d3[i]= list(range(1, x[i]+1))
l3 = list(d3.values())
result = itertools.product(*l3)
print(list(result))
```

```
[('F', 'F', 'F', 'F'),
 ('F', 'F', 'F', 'M'),
 ('F', 'F', 'M', 'F'),
 ('F', 'F', 'M', 'M'),
 ('F', 'M', 'F', 'F'),
 ('F', 'M', 'F', 'M'),
 ('F', 'M', 'M', 'F'),
 ('F', 'M', 'M', 'M'),
 ('M', 'F', 'F', 'F'),
 ('M', 'F', 'F', 'M'),
 ('M', 'F', 'M', 'F'),
 ('M', 'F', 'M', 'M'),
 ('M', 'M', 'F', 'F'),
 ('M', 'M', 'F', 'M'),
 ('M', 'M', 'M', 'F'),
 ('M', 'M', 'M', 'M')]
```

```
[(1, 1, 1),
 (1, 1, 2),
 (1, 1, 3),
```

```
 (1, 2, 1),
 (1, 2, 2),
 (1, 2, 3),
 (1, 3, 1),
 (1, 3, 2),
 (1, 3, 3),
 (2, 1, 1),
 (2, 1, 2),
 (2, 1, 3),
 (2, 2, 1),
 (2, 2, 2),
 (2, 2, 3),
 (2, 3, 1),
 (2, 3, 2),
 (2, 3, 3),
 (3, 1, 1),
 (3, 1, 2),
 (3, 1, 3),
 (3, 2, 1),
 (3, 2, 2),
 (3, 2, 3),
 (3, 3, 1),
 (3, 3, 2),
 (3, 3, 3)]
```

```
[(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2), (1, 3, 1), (1, 3, 2), (1, 4, 1), (1, 4, 2), (2, 1, 1), (
```

7.4 Conjuntos em Python

Vamos relembrar as principais funções dos objetos `set`, conjuntos em Python, para podermos realizar algumas operações simples. Os conjuntos no Python é uma coletânea de elementos sem a ocorrência de valores com multiplicidades. Os conjuntos são criados pelo método `set` ou pelo uso de chaves. Portanto, `x = set([1,2,3])` ou `x={1,2,3}` é um conjunto, mas `y= [1,2,3,3]` não é, por ter elementos repetidos. Se considerassemos o comando `y= set([1,2,3,3])`, este objeto seria um conjunto, pois o Python elimina automaticamente os elementos repetidos na criação do conjunto.

Os argumentos da função `set` são as listas ou as tuplas. Podemos de forma eficiente usar os conjuntos para remover elementos duplicados de uma lista ou tupla. Primeiro, aplicamos o operador`set()` com o argumento sendo uma lista ou uma tupla e aplicamos o argumento `list()` ou `tupla()` no resultado. Outra vantagem dos conjuntos é que eles podem ter elementos de diferentes tipos. Os conjuntos são imutáveis, pois seus elementos não podem ser trocados, mas podemos adicionar e remover elementos de um conjunto em Python. Além do mais, os conjuntos são não ordenados. Vejamos um exemplo de como criar um conjunto.

```
# Criar conjuntos em Python
A = {'Laranja', 'Maçã', 'Pera'}
B = set(('Laranja'))
C = set(('Laranja',))
D = set((1,2,3,'Frutas'))
A
B
C
D
```

```
{'Laranja', 'Maçã', 'Pera'}
```

```
{'L', 'a', 'j', 'n', 'r'}
```

```
{'Laranja'}
```

```
{1, 2, 3, 'Frutas'}
```

Podemos adicionar ou remover elementos em um conjunto. Os elementos que são argumentos das funções `set.add()` ou `set.update()` devem ser imutáveis como as tuplas ou as strings para o primeiro método e qualquer objeto iterador como as listas, para o segundo. Se o argumento for uma lista, ocorrerá um erro no método `add`. Para remover um elemento de um conjunto, podemos usar métodos como o `set.remove()` ou `set.discard()` ou `set.pop()`. O primeiro comando remove um elemento específico do conjunto e retorna um erro, se o elemento não existir. O segundo remove o elemento, mas não acusa erro se o elemento não existir, deixando o conjunto como estava anteriormente a sua aplicação. O terceiro remove um elemento de forma aleatória do conjunto e não tem argumento. O método `pop` retorna o elemento removido e atualiza (modifica) o conjunto no qual foi aplicado. Os métodos `add` e `update` diferem no sentido do primeiro não suportar acrescentar uma lista ao conjunto. O método `set.clear()` remove todos os elementos do conjunto. Veja o `script` com alguns exemplos:

```
# adição e remoção de elementos
# de um conjunto
x = {1,2,3,4}
y = [4,5,6,7]
x.update(y)
x
x.add(8)
x
x.remove(8)
x
x.discard(9) # não causa erro por não existir em x
x
x.pop()
x
```

```
{1, 2, 3, 4, 5, 6, 7}
```

```
{1, 2, 3, 4, 5, 6, 7, 8}
```

```
{1, 2, 3, 4, 5, 6, 7}
```

```
{1, 2, 3, 4, 5, 6, 7}
```

```
1
```

```
{2, 3, 4, 5, 6, 7}
```

O Python inclui algumas operações com conjuntos, sendo algumas delas:

- `set.union(x, y)`: união dos conjuntos x e y, que corresponde ao conjunto de todos os elementos presentes em x e em y, cosniderando apenas uma vez aqueles elementos com multiplicidade maior que 1;
- `set.intersect(x,y)`: interseção dos conjuntos x e y, que corresponde ao conjunto dos elementos presentes simultaneamente em x e y;
- `set.difference(x,y)`: conjunto da diferença entre os conjuntos x e y, consistindo no conjunto de todos os elementos de x que não estão em y;
- `x == y` ou `x != y`: testa se dois conjuntos x e y são iguais ou se são diferentes, respectivamente;

- `c in y`: pertencimento, ou seja, testa se c é um elemento do conjunto y;
- `set.symmetric_difference(x, y)`: é o conjunto dos elementos que estão ou em x ou em y, mas não está em ambos simultaneamente.

```
# operações com conjuntos
# ilustrativas
y = set([1,2,3,3])
x = {3,4,5}
set.union(x,y)
x.union(y)
x # não modifica x
set.intersection(x,y)
set.difference(x,y)
x == y
x != y
t = tuple(set((1,2,3,3))) # removendo duplicados de uma tupla
t
type(t)
y = set([1,2,3])
x = {3,4,5}
set.symmetric_difference(y,x)
```

```
{1, 2, 3, 4, 5}
```

```
{1, 2, 3, 4, 5}
```

```
{3, 4, 5}
```

```
{3}
```

```
{4, 5}
```

```
False
```

```
True
```

```
(1, 2, 3)
```

```
tuple
```

```
{1, 2, 4, 5}
```

Outras opções de operações básicas com conjuntos são as seguintes:

- `y.issubset(x)`: retorna `True` se y está contido ou é igual ao conjunto x;
- `x.issuperset(y)`: retorna `True` se x contém ou é igual ao conjunto y;
- `x.isdisjoint(y)`: retorna `True` se x e y não tiverem elementos comuns, ou seja, se a interseção for um conjunto vazio.

```
# outras operações com conjuntos
x = {1,2,3,4,5}
y = {1, 3, 5}
# y está contido em x
y.issubset(x)
# x contém y
x.issuperset(y)
# se x e y são disjuntos
x.isdisjoint(y)
```

```
True

True

False
```

7.5 Alguns Problemas de Probabilidade

Vamos apresentar alguns casos particulares como o caso das coincidências de aniversários. Para a coincidência de aniversários, vamos considerar a coincidência de nascimento de um grupo de n pessoas na mesma data e que o ano tem 365 dias. Consideramos também que a distribuição dos nascimentos ao longo dos anos dos aniversários é aleatória uniforme, o que é uma forte suposição. A probabilidade de que nenhuma pessoa tenha a mesma data de nascimento (evento A) em um grupo de n pessoas é:

$$P(A) = \frac{A_{365,n}}{365^n} = \frac{365!}{(365-n)!365^n}.$$

Desejamos a probabilidade do evento complementar, que é dada por $P(A^c) = 1 - P(A)$. Logo,

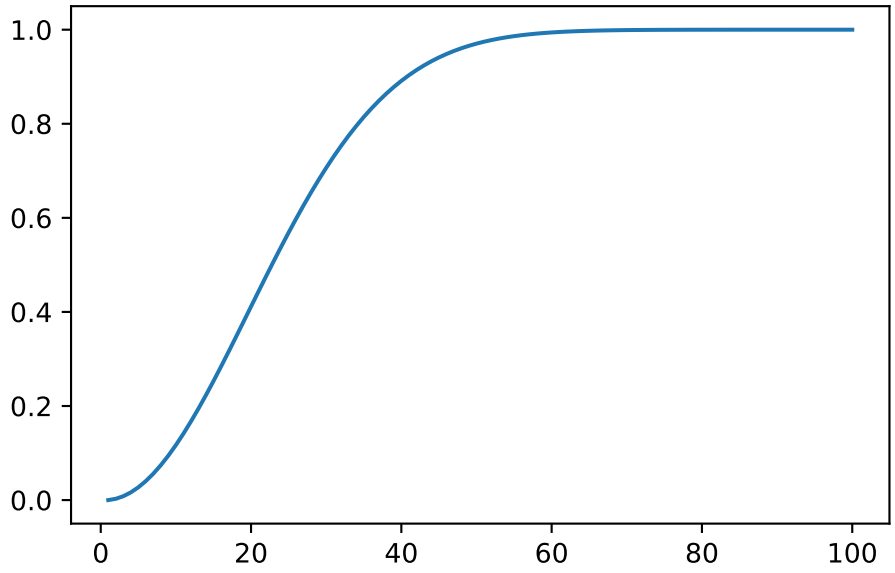
$$P(A^c) = 1 - \frac{A_{365,n}}{365^n} = 1 - \frac{365!}{(365-n)!365^n},$$

que é a probabilidade de haver pelo menos uma coincidência de aniversários na mesma data.

Implementamos duas formas. Na primeira usamos a fórmula completa. Tomamos o logaritmo (usamos a função `gammaln` da `scipy.special`) e ao final recuperamos tomando o `exp` do resultado. Na segunda alternativa usamos a biblioteca `math`, em que o total de arranjos $A_{365,n}$ é obtido pelo comando `math.perm(365, n)`.

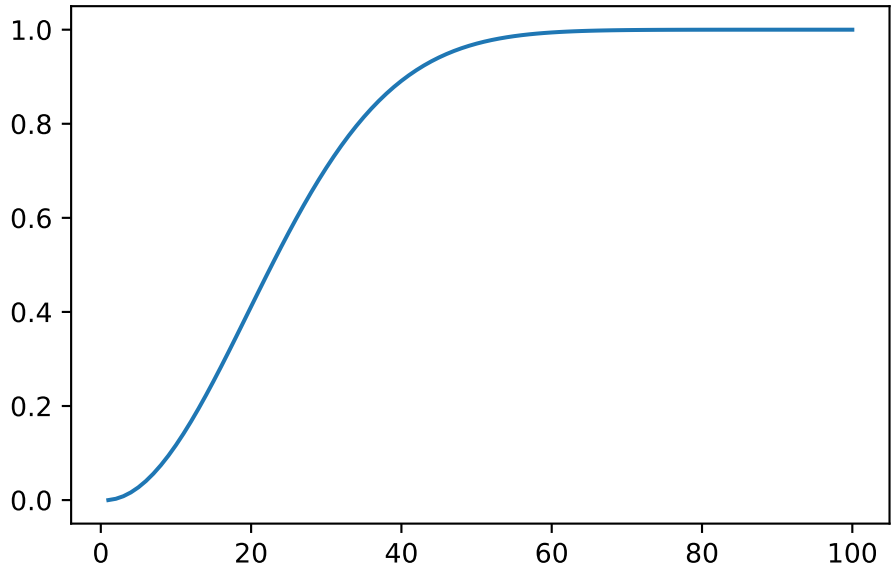
O programa da primeira alternativa em Python é:

```
import numpy as np
import matplotlib.pyplot as plt
from numpy.lib.scimath import log
from numpy import exp
from scipy.special import gammaln
# Uso do log of gama para obter o log do fatorial
def ca(n):
  # invertendo o log (i.e. exp)
  return 1 - exp(gammaln(365+1) - gammaln(365-n+1) -n*log(365) )

n = np.arange(1, 100+1)
# Gráfico
plt.plot(n, ca(n))
plt.show()
```



Nossa segunda implementação é:

```
# coincidência de aniversários
# probabilidades
import math
def pca(n):
  if type(n) == int:
    n = [n]
  log365 = math.log(365)
  pc  = np.empty(0)
  for k in range(len(n)):
    pc=np.append(pc,1-exp(math.log(math.perm(365, n[k]))-n[k]*log365))
  return pc
n = np.arange(1, 100+1)
# Gráfico
plt.plot(n, pca(n))
plt.show()
```



No segundo exemplo, vamos considerar a probabilidade de que uma comissão de tamanho m ao ser retirada de um grupo de tamanho n aleatoriamente não contenha representantes de um grupo existente específico de tamanho k entre os n elementos do grupo todo. Essa probabilidade é:

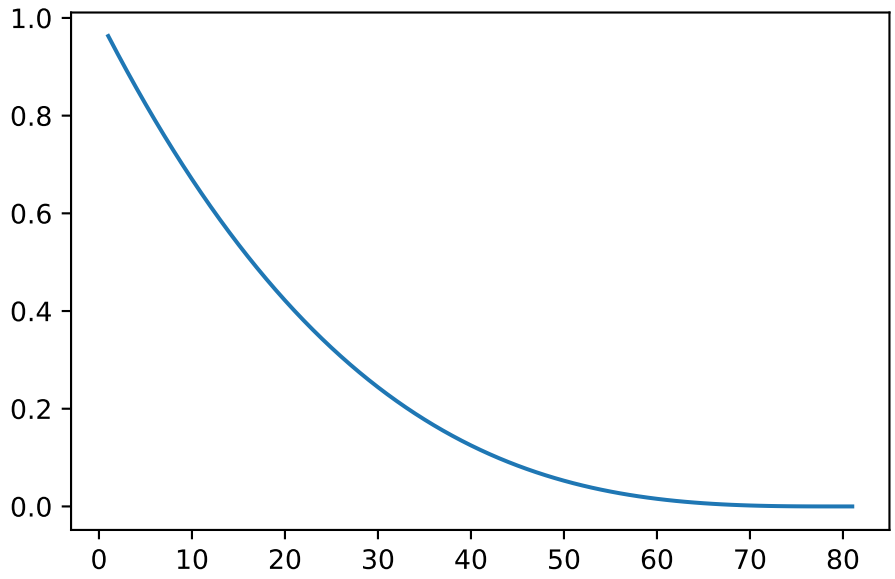
$$P(\text{Não ser representado}) = \frac{\binom{n-k}{m}}{\binom{n}{m}}.$$

O programa Python foi implementado e ilustramos com o exemplo dos senadores do Brasil, que possui representantes das diferentes unidades federativas brasileiras. No Brasil temos $n = 27 \times 3$ senadores, ou seja, 81 senadores no total, sendo 3 de cada estado ou do Distrito Federal. Se vamos formar uma comissão aleatória de $m = 10$ senadores, qual é a probabilidade de que uma unidade federativa qualquer contendo $k = 3$ senadores não seja representada. Vimos por meio da análise da fórmula anterior que temos $\binom{n}{m}$ maneiras diferentes de sortear comissões de tamanho m em n elementos do grupo todo. Se eliminarmos do total n, os k que serão excluídos da composição da comissão, sobram $n - k$ que podem ser amostrados m a m. Assim, o número de comissões sem os k do grupo considerado é $\binom{n-k}{m}$. A razão entre estas duas quantidades nos fornece a probabilidade desejada. O `script` com o exemplo dos senadores está apresentado a seguir.

```
# Probabilidade de uma comissão de m pessoas
# não ter representante de um subgrupo de tamanho
# k quando retirada uma comissão de tamanho m de
# um grupo contendo n elementos, k < n e m <= n-k
from scipy import special as sps
def pnr(n , m, k):
  pr = sps.comb(n-k, m) / sps.comb(n, m)
  return pr
# Exemplo
n = 27*3 # número de senadores no Brasil, 81 (3 em cada estado ou DF)
m = 10 # comissão de m deles
k = 3 # um estado qualquer fixado não ser representado
pnr(n,m,k)
m = range(1, 81+1)
plt.plot(m, pnr(n,m,k))
```

```
plt.show()
# probabilidade de MG e DF
# não serem representados na comissão
k = 6
m = 10
pnr(n,m,k)
```

```
np.float64(0.6698898265353962)
```




```
np.float64(0.44129815640475195)
```

Nosso próximo exemplo refere-se às probabilidades em uma mão de pôquer, construindo as possibilidades e probabilidades. Neste jogo podemos ter uma mão sem nada (sem pares, sem dois pares, sem trincas, etc.), uma com um par, uma com dois pares, uma com uma trinca, uma com uma sequência (qualquer de naipe), uma com *flush* (diferentes valores que não estão em sequência do mesmo naipe), uma com um *full house* (trinca e par), uma com o *four* (quadra), uma com uma sequência do mesmo naipe e uma sequência real do mesmo naipe (*royal flush* - do 10 ao Ás).

O número de possibilidades totais de distribuir 52 em um sorteio de 5 cartas (uma mão) é:

$$\binom{52}{5} = \frac{52!}{5!(52-5)!} = 2.598.960.$$

- Para obtermos uma mão sem valor, temos que entender que o baralho é constituído de 52 cartas, sendo 13 valores das cartas (Ás, 1, 2, $\cdots$, 10, Valete, Dama e Rei) e 4 naipes para cada valor (espadas, paus, ouros e copas). O número de possibilidades de uma mão de cinco cartas (sorteio de 5 cartas sem reposição) conter uma mão sem valor, ouseja, sem pares, sem trincas, sem quadras, sem sequências de naipes diferentes ou sem sequências do mesmo naipe é:

$$\binom{13}{5}\left(\binom{4}{1}\right)^5 - 10\left(\binom{4}{1}\right)^5 - 4\binom{13}{5} + 4 \times 10 = = 1.302.540,$$

ou seja, é o número de cartas de diferentes valores com qualquer um dos 4 naipes $\binom{13}{5}4^5$, subtraído de todas as possibilidades das sequências de naipes diferentes 10×4^5 e das possibilidades de ter cartas de

diferentes valores com o mesmo naipe $4\binom{13}{5}$. Este valor deve ser adicionado das dez possíveis sequências para cada um dos naipes 4×10. Esta adição se dá em razão de termos retirado as possibilidades de de ter cartas de diferentes valores com o mesmo naipe, que incluí as 10 sequências do mesmo naipe possíveis e novamente foi retirada quando eliminamos as possibilidades de termos 5 cartas de valores diferentes com o mesmo naipe, que incluí as 10 sequências do mesmo naipe possíveis. Daí precisamos adiciona-las, uma vez que elas foram retiradas duas vezes.

- Para a mão de um par simples temos:

$$\binom{13}{1}\binom{4}{2}\binom{12}{3}4^3,$$

em que $\binom{13}{1}$ escolhe o valor do par, $\binom{4}{2}$ escolhe os naipes para o par, $\binom{12}{3}$ escolhe entre os 12 valores remanescentes (não escolhido para o par) e 4^3, escolhe os naipes dos 3 valores diferentes do par.

- Para os dois pares temos:

$$\binom{13}{1}\binom{4}{2}\binom{12}{1}\binom{4}{2}\binom{11}{1}4,$$

em que $\binom{13}{1}$ escolhe o valor do primeiro par, $\binom{4}{2}$ escolhe os naipes para o primeiro par, $\binom{12}{1}$ escolhe o segundo par entre os 12 valores remanescentes, $\binom{4}{2}$ escolhe os naipes para o segundo par e $\binom{11}{1}$ escolhe os valores entre os 11 valores remanescentes (não escolhido para os 2 pares) e 4, escolhe os naipes do valor diferente dos dois pares.

- Para uma tripla temos:

$$\binom{13}{1}\binom{4}{3}\binom{12}{2}4^2,$$

em que $\binom{13}{1}$ escolhe o valor da tripla, $\binom{4}{3}$ escolhe os naipes para a tripla, $\binom{12}{2}$ escolhe os 2 valores entre os 12 valores remanescentes, 4^2 escolhe os naipes para estes valores.

- Para a sequência:

$$10 \times 4^5 - 10 \times 4,$$

em que 10 escolhe entre as 10 possíveis sequências (do Ás ao 5, do 2 ao 6, até do 10 ao Ás) e o 4^5 escolhe os naipes de cada valor. O resultado incluí as sequências do mesmo naipe, que devem ser retiradas, que correspondem à 10×4 (sequências de mesmo naipe incluindo as reais).

- Para o *flush* (cartas do mesmo naipe):

$$\binom{13}{5} \times 4 - 10 \times 4,$$

em que $\binom{13}{5}$ escolhe os 5 diferentes valores entre os 13 e 4 escolhe um dos naipes para os 5 valores. O resultado incluí as sequências do mesmo naipe, que devem ser retiradas, que correspondem à 10×4 (sequências de mesmo naipe incluindo as reais).

- Para o *full house* (trinca e par):

$$\binom{13}{1}\binom{4}{3}\binom{12}{1}\binom{4}{2}$$

em que $\binom{13}{1}$ escolhe o valor para a trinca, $\binom{4}{3}$ escolhe os naipes da trinca, $\binom{12}{1}$ escolhe o valor do para entre os 12 valores remanescentes e $\binom{4}{2}$ escolhe os naipes da dupla.

- Para a quadra (*four*):

$$\binom{13}{1}\binom{12}{1}4$$

em que $\binom{13}{1}$ escolhe o valor para a quadra que necessariamente terá uma de cada naipe, $\binom{12}{1}$ escolhe um valor para a carta remanescente dos 12 valores restantes e 4 escolhe um dos naipes da carta remanescente, que não é o da quadra.

- Para as sequências do mesmo naipe (*straight flush*):

$$10 \times 4 - 4,$$

pois são 10 sequências para um dos 4 naipes, subtraída das 4 sequências reais entre elas, do 10 ao Ás de cada um naipe entre os quatro naipes possíveis.

- A sequência real (*royal straight flush*): tem uma única sequência possível do 10 ao Ás para cada um dos quatro naipes, totalizando 4 possíveis sequências reais.

O programa a seguir cria um `dataframe` com todas essas contagens e calcula as probabilidades dividindo-as pelo número total de combinações:

```
#Mão de pôquer
# probabilidades
import pandas as pd
import numpy as np
from scipy import special as sps
poker = {
  'none': sps.comb(13,5)*4**5 - 10*4**5 - 4*sps.comb(13,5) + 10*4,
  'pair': 13*sps.comb(4,2)*sps.comb(12,3)*4**3,
  'two.pairs': sps.comb(13,2)*sps.comb(4,2)**2*11*4,
  'triple': 13*sps.comb(4,3)*sps.comb(12,2)*4**2,
  'straight': 10*4**5 - 10*4,
  'flush': 4*sps.comb(13,5) - 10*4,
  'full.house': 13*sps.comb(4,3)*12*sps.comb(4,2),
  'four': 13*sps.comb(4,4)*12*4,
  'straight.flush': 10*4 - 4,
  'royal.flush': 4 }
sum(poker.values()) - sps.comb(52,5) # conferir
poker
mão = list(poker.keys())
possibilidades = list(poker.values())
```

```
datapoker = pd.DataFrame({'mão': mão, 'possibilidades': possibilidades})
datapoker['probabilidades'] = list(poker.values()) / sps.comb(52,5)
datapoker
sum(datapoker['probabilidades'])
```

```
np.float64(0.0)
```

```
{'none': np.float64(1302540.0),
 'pair': np.float64(1098240.0),
 'two.pairs': np.float64(123552.0),
 'triple': np.float64(54912.0),
 'straight': 10200,
 'flush': np.float64(5108.0),
 'full.house': np.float64(3744.0),
 'four': np.float64(624.0),
 'straight.flush': 36,
 'royal.flush': 4}
```

	mão	possibilidades	probabilidades
0	none	1302540.0	0.501177
1	pair	1098240.0	0.422569
2	two.pairs	123552.0	0.047539
3	triple	54912.0	0.021128
4	straight	10200.0	0.003925
5	flush	5108.0	0.001965
6	full.house	3744.0	0.001441
7	four	624.0	0.000240
8	straight.flush	36.0	0.000014
9	royal.flush	4.0	0.000002

```
1.0
```

Nosso próximo exemplo refere-se à probabilidade da máxima diferença de uma lista enumerada ordenada. Obter o maior *gap* (salto) (diferença máxima) entre dois números consecutivos no sorteio sem reposição de m números entre os n primeiros inteiros de 1 a n, para $m < n$. No Python, o comando `np.diff(x)`, retorna as diferenças entre números consecutivos da lista `x`.

O programa a seguir, usa a função `checkgap` para retornar a máxima diferença de um interador qualquer. A função `maxgap`, recebe n, m e k e calcula as probabilidades de cada caso. Para isso ela percorre todas as combinações geradas por `combinations` retornando o máximo de cada combinação obtida em `y` e com o comando `mean(y == k)` é obtida a probabilidade exata de $P(X = k)$, sendo X a variável que representa a maior diferença entre números inteiros consecutivos de uma amostra sem reposição de tamanho m obtida entre n números inteiros, para m de 1 a n e k o valor da máxima diferença (valor da variável aleatória) para o qual queremos calcular a probabilidade de ocorrência. Se $m = 2$, então a máxima diferença será considerada a diferença consecutiva, pois a diferença em cada resultado do espaço amostral é única, em razão de termos amostras de tamanho 2.

```
# Max gap probabilidades
from itertools import combinations
import pandas as pd
def checkgap(cmb):
  x = np.diff(cmb)
  return max(x)
```

```
def maxgap(n, m, k):
  if m <= 1 or m > n:
    print('m deve estar entre 1 e n!')
    return
  comb = list(combinations(range(1,n+1), m))
  y = np.empty(0)
  for cmb in comb:
    y = np.append(y,checkgap(cmb))
  return np.mean(y == k)

# função para todo o suporte
def maxgapsx(n, m):
  if m <= 1 or m > n:
    print('m deve estar entre 1 e n!')
    return
  comb = list(combinations(range(1,n+1), m))
  y = np.empty(0)
  for cmb in comb:
    y = np.append(y,checkgap(cmb))
  res = {'x':[],'P(X=x)':[]}
  for k in range(1,n-m+2):
    res['P(X=x)'].append(np.mean(y == k))
    res['x'].append(k)
  return pd.DataFrame(res)

# exemplo de uso
n = 6
m = 3
k = 2
maxgap(n,m,k)
maxgapsx(n,m)
```

```
np.float64(0.4)
```

	x	P(X=x)
0	1	0.2
1	2	0.4
2	3	0.3
3	4	0.1

Nosso último exemplo é para a probabilidade das somas das faces no lançamento de n dados e da diferença em valor absoluto das faces no lançamento de 2 dados. Para este caso, podemos usar uma versão similar, ou aplicando a função `contag` anteriormente apresentada ou aplicando a função `product` da biblioteca `intertools`. Optamos por usar a função `contag` da classe `cont`.

Com uso de alguns pequenos detalhes adicionais, obtivemos os resultados para os dois casos. O segundo caso, da diferença, por razões óbvias, são resultantes do lançamento de apenas 2 dados. Nos dois casos apresentamos também os resultados das probabilidades obtendo todas as possibilidades que corresponde ao espaço amostral do experimento aleatório. Temos uma função para obter uma probabilidade para um valor específico da variável aleatória e outra para todos os valores de probabilidade relativos ao suporte da

variável aleatória, nos dois casos, o da soma de n dados e o da diferença de dois dados. Os resultados estão apresentados no `script` a seguir:

```
# Probabilidades para o lançamento de
# n dados e obtenção da soma e também
# para a diferença absoluta de 2 dados
# somafaces: retorna P(X=x)
# depende da class cont
def somafaces(n, x):
  if x<n or x>6*n:
    print('x deve estar entre n e 6n!')
    return
  res  = Cont()
  perm = res.contag(6, n)
  y = np.empty(0)
  for pmb in perm:
    y = np.append(y,np.sum(pmb))
  return np.mean(y == x)
def somafacessx(n):
  if x<n or x>6*n:
    print('x deve estar entre n e 6n!')
    return
  res  = Cont()
  perm = res.contag(6, n)
  y = np.empty(0)
  for pmb in perm:
    y = np.append(y,np.sum(pmb))
  res = {'x':[],'P(X=x)':[]}
  for k in range(n,6*n+1):
    res['P(X=x)'].append(np.mean(y == k))
    res['x'].append(k)
  return pd.DataFrame(res)
# Probabilidades da diferença entre
# duas faces de dois dados equilibrados
# em módulo - retorna P(X=x), X=|i-j|
def difabs2faces(x):
  if x<0 or x>5:
    print('x deve estar entre 0 e 5!')
    return
  res  = Cont()
  perm = res.contag(6, 2)
  y = np.empty(0)
  for pmb in perm:
    y = np.append(y,abs(np.diff(pmb)))
  return np.mean(y == x)
# Probabilidades da diferença entre
# duas faces de dois dados equilibrados
# em módulo - retorna P(X=x), para todo
# o suporte de x
def difabs2facessx():
  if x<0 or x>5:
    print('x deve estar entre 0 e 5!')
    return
```

```
  res  = Cont()
  perm = res.contag(6, 2)
  y = np.empty(0)
  for pmb in perm:
    y = np.append(y,abs(np.diff(pmb)))
  res = {'x':[],'P(X=x)':[]}
  for k in range(0,5+1):
    res['P(X=x)'].append(np.mean(y == k))
    res['x'].append(k)
  return pd.DataFrame(res)
# Exemplo
n = 4
x = 4
print('P(X =',x,') = ',somafaces(n,x))
dist = somafacessx(n)
dist
sum(dist['P(X=x)']) # checando
print('P(X =',x,') = ',difabs2faces(x))
difabs2facessx()
```

```
P(X = 4 ) =  0.0007716049382716049
```

	x	P(X=x)
0	4	0.000772
1	5	0.003086
2	6	0.007716
3	7	0.015432
4	8	0.027006
5	9	0.043210
6	10	0.061728
7	11	0.080247
8	12	0.096451
9	13	0.108025
10	14	0.112654
11	15	0.108025
12	16	0.096451
13	17	0.080247
14	18	0.061728
15	19	0.043210
16	20	0.027006
17	21	0.015432
18	22	0.007716
19	23	0.003086
20	24	0.000772

```
1.0
P(X = 4 ) =  0.1111111111111111
```

	x	P(X=x)
0	0	0.166667

	x	P(X=x)
1	1	0.277778
2	2	0.222222
3	3	0.166667
4	4	0.111111
5	5	0.055556

7.6 Exercícios

1. Obter uma função para obter por meio de simulação Monte Carlo as probabilidades dos lançamentos de n dados e obtenção da soma e também para as diferenças em módulo entre as faces nos lançamentos de dois dados. Para fazer isso escolha um grande número de repetições e em cada uma delas simule os resultados dos lançamentos dos dados. Obtenha os resultados das variáveis aleatórias e ao final calcule as proporções de cada ocorrência, conforme fizemos de forma exata anteriormente. Estas proporções são estimativas empíricas das probabilidades almejadas.

2. Obtenha também por Monte Carlo as probabilidades empíricas de uma mão de pôquer, simulando um grande número de mãos. Para isso é preciso criar um baralho, que pode ser feito com um dicionário e depois amostrar as mãos de cinco cartas e calcular as probabilidades empíricas de cada possível resultado. Confrontar com as probabilidades exatas obtidas neste capítulo. O que você espera que ocorra com essa comparação quando aumenta-se o número de simulações?

References

A. C. Atkinson and M. C. Pearce. the computer generation of beta, gamma and normal random variable. *journal of the royal statistical society, series a*, 139(4):431–461, 1976.

J. N. W. Dachs. *Estatìstica computacional: uma introdução em turbo Pascal*. LTC, Rio de Janeiro, 1988.

L. Devroy. generating the maximum of independent identically distributed random variables. *Computers and mathematics with applications*, 6:305–315, 1980.

L. Devroy. *Non-uniform random variate generation*. springer-verlag, new york, 1986.

D. F. Ferreira. *Estatística multivariada*. Editora UFLA, Lavras, 3 edition, 2018.

R. A. Johnson and D. W. Wichern. *Applied multivariate statistical analysis*. Prentice Hall, New Jersey, 4 edition, 1998.

V. Kachitvichyanukul and B. W. Schmeiser. binomial random variate generation. *Communications of the ACM*, 31(2):216–222, 1988.

Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97—111, May 1984.

G. Marsaglia and T. A. Bray. A convenient method for generating normal variables. *Siam Review*, 6(3): 260–264, 1964.

M. Matsumoto and T. Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions On Modeling and Computer Simulation*, 8(1): 3–30, 1998.

B. D. McCullough and B. Wilson. on the accuracy of statistical procedures in Microsoft Excel 97. *Computational Statistics and Data Analysis*, 31:27–37, 1999.

T. H. J. Naylor, J. L. Balintfy, D. S. Burdick, and K. Chu. *Técnicas de simulação em computadores*. Vozes, Petrópolis, 1971.

S. K. Park and K. W. Miller. Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10):1192–1201, 1988.

W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical recipes in Fortran: the art of scientific computing*. Cambridge University Press, Cambridge, 1992.

A. Quarteroni, R. Sacco, and F. Saleri. *Numerical mathematics*. springer, Berlin, 2000.

L. Schrage. A more portable FORTRAN random number generator. *ACM transactions on mathematical software*, 5(2):132–138, 1979.

W. B. Smith and R. R. Hocking. Algorithm AS 53: Wishart variate generator. *Applied Statistics - Journal of the Royal Statistical Society - Series C*, 21(3):341–345, 1972.

D. H. D. West. Updating mean and variance estimates: an improved method. *ACM transactions on mathematical software*, 22(9):532–535, 1979.

Index