Computational Statistics in Python

Daniel Furtado Ferreira

Date: 11/11/2026

Table of contents

Preface **1**

1 Introduction to Python **3**
1.1 Introduction to Python Statements and Objects . . . 4
1.1.1 Basic Arithmetic Operations . . . 5
1.2 Boolean Variables . . . 7
1.3 Strings . . . 7
1.4 Lists, Tuples, Sets and Dictionaries . . . 9
1.4.1 Lists . . . 9
1.4.2 Tuples . . . 15
1.4.3 Sets . . . 16
1.4.4 Dictionaries . . . 18
1.5 Matrices and Arrays . . . 19
1.6 Data Files . . . 23
1.7 Programming Control Structures . . . 28
1.8 Functions . . . 31
1.9 Computational Statistics . . . 34
1.10 Exercises . . . 35

2 Uniform Random Variables **37**
2.1 Uniform Random Numbers . . . 37
2.2 Uniform Random Numbers in Python . . . 42
2.3 Genuinely Random Numbers . . . 44
2.4 Comparative Performance Analysis . . . 45
2.5 Exercises . . . 46

3 Non-Uniform Random Variables **49**
3.1 Introduction . . . 49
3.2 General Methods for Generating Realizations of Random Variables . . . 49
3.3 Random Variables from Important Distributions . . . 52
3.4 Binomial Distribution . . . 59
3.5 Python Routines for Generating Random Variables . . . 67
3.6 Exercises . . . 69

4 Generating Random Samples of Multidimensional Variables **71**
4.1 Introduction . . . 71
4.2 Multivariate Normal Distribution . . . 71
4.3 Wishart and Inverse Wishart Distribution . . . 76
4.4 Multivariate Student's t Distribution . . . 80
4.5 Other Multivariate Distributions . . . 83

4.6 Exercises . . . 85

5 Algorithms for Means, Variances, and Covariances **87**
5.1 Introduction . . . 87
5.2 Univariate Algorithms . . . 87
5.3 Algorithms for Mean Vectors and Covariance Matrices . . . 91
5.4 Case Studies . . . 92
5.4.1 Log-Sum-Exp (LSE) . . . 93
5.4.2 Kahan Summation (Compensated Summation) . . . 93
5.4.3 Practical Demonstration in Python . . . 94
5.5 Exercises . . . 95

6 Approximation of Distributions **97**
6.1 Introduction . . . 97
6.2 Discrete Probabilistic Models . . . 102
6.3 Continuous Probabilistic Models . . . 107
6.4 Gaussian Quadratures . . . 114
6.5 Newton-Raphson . . . 120
6.6 Pre-Existing Functions in Python . . . 123
6.7 Exercises . . . 123

7 Sets and Elements of Combinatorial Analysis in Python **125**
7.1 Introduction to Combinatorial Analysis in Python . . . 125
7.2 Permutations and Arrangements . . . 126
7.3 Counting . . . 128
7.4 Sets in Python . . . 134
7.5 Some Probability Problems . . . 137
7.6 Exercises . . . 147

8 Bootstrap Methods in Python **149**
8.1 Introduction . . . 149
8.2 Non-Parametric Bootstrap . . . 149
8.3 Estimation . . . 150
8.3.1 Standard Error . . . 150
8.3.2 Bias Correction . . . 153
8.3.3 Standard Bootstrap Confidence Interval . . . 155
8.3.4 Confidence Interval Based on Bootstrap Percentiles . . . 157
8.3.5 Basic Bootstrap Confidence Interval . . . 159
8.3.6 Bias-Corrected and Accelerated Bootstrap Confidence Interval . . . 160
8.3.7 Bootstrap Confidence Interval with Bias Correction . . . 162
8.3.8 Bootstrap in Python . . . 163

References **165**

Preface

The book *Computational Statistics in Python* is a summary and adaptation of the book *Computational Statistics in Java*, published by Editora UFLA. The objective of this book is, first, to improve the author's own knowledge of Quarto and, second, to consolidate and advance the learning of Python.

We will initially adapt the course notes *Computational Statistics in R* to accommodate the Python code. Subsequently, we will expand the content of this first edition of the book.

To learn more about Quarto Books, visit https://quarto.org/docs/books.

In these lecture notes, our intention was to address the topic of computational statistics, which is so crucial to the scientific community and especially to graduate students in statistics. We can state without hesitation that computational statistics has become, and is today, one of the main subfields of statistics. Furthermore, knowledge from this area can be—and frequently is—applied to other fields of statistics, engineering, and physics. Bayesian inference is a prime example where a computational approach is typically required. When planning these lecture notes, we had many doubts about what topics to cover and how to approach each selection. Ultimately, we chose to write something that enables the reader to go beyond a simple recipe book while avoiding getting bogged down in a tangle of proofs. On the other hand, we sought to present models and methods in a broad, non-restrictive manner.

Another motivation that guided and encouraged us to develop this project was our personal research experience in computational statistics. We also cannot deny that we undertook this project for personal benefit, coming into direct contact with the vast body of literature published in this branch of statistics. However, we do not intend to cover every subject, nor do we aim to exhaust all possibilities for any given topic. On the contrary, we hope these notes serve as an introduction to computational statistics and inspire undergraduate statistics students to delve even deeper into this field.

These notes are based on a book we wrote on computational statistics using Java. Adapting some of the routines implemented in that book to Python was a very enjoyable and revealing task. We took this opportunity to uncover some of the many secrets of this powerful tool and discover its vast potential—as well as its limitations. In this first version, we do not expect perfection; we are fully aware that flaws may exist and rely on reader feedback to help correct them. We are launching this initial, unrevised, and unexpanded version using Quarto and transitioning from R to Python. These notes will be updated continuously online. We hope this manuscript contributes to the professional growth of students in our graduate program in Statistics and Agricultural Experimentation, as well as students from other programs at UFLA or other institutions. If so, our goal will have been achieved.

Chapter 1

Introduction to Python

`Python` was chosen to teach this course due to a number of advantages. In addition to being free, in the sense of open distribution and open-source code, it can be used on both `Windows` and `Linux` platforms. Furthermore, `Python` offers great versatility by providing numerous ready-to-use libraries while also allowing us to build new methods and functions. `PyPI` is the official `Python` repository where all libraries are stored. We can think of it as a `GitHub` for `Python` libraries. `Python` was created by Dutch programmer Guido van Rossum to be a simple, readable, and highly productive programming language. `Python` evolved and became a highly attractive language and a primary choice for web development, data analysis, and artificial intelligence applications, among others. Being genuinely an object-oriented language, it allows us to program with great efficiency and versatility, although it presents some differences in its implementation compared to other object-oriented languages. Another aspect that is very attractive in `Python` refers to the fact that it receives contributions from researchers all over the world in the form of libraries. This is a characteristic that drives a great development of the language in relatively short periods of time and enables us to find solutions for almost all problems we encounter in real situations. For the problems for which we cannot find solutions, the `Python` programming environment allows us to create our own solutions.

In these lecture notes, we intend to present the basic concepts of computational statistics in a very simple way. Initially, we will obtain our own solutions for a given method or technique, and in a second moment, we will show that we can have the same ready-made solution from `Python` when it is available. Particularly in this chapter, we will present some characteristics of the environment and the language to implement our solutions. Our course does not intend to provide advanced and maximally efficient solutions for the problems we will address, but rather to provide students with a first contact with the `Python` language and with the basic problems of computational statistics.

Some initial efforts are necessary before we can obtain any benefit. We do not intend to present in this course the `Python` resources for analyzing full or non-full rank linear models, non-linear models, generalized linear models, or graphs. Eventually, we may use some of these functions as an intermediate step to solve the problem for which we are presenting solutions. This material will be built with a brief and simplified theoretical approach to the topic and will associate practical examples of `Python` programming resources to solve a formulated problem, in particular cases of the studied theory.

This material is only a first version that will likely have many flaws. Thus, the reader who finds them or has a better solution to the problem can contribute by sending an email to danielff@ufla.br.

Visit my homepage https://des.ufla.br/~danielff/.

1.1 Introduction to Python Statements and Objects

In `Python`, objects can be of different types or structures, such as numbers (`int`, `float`, and complex), `boolean`, `string`, `list`, `tuple`, `set`, `dictionary`, `functions` (objects that encapsulate code), `dataframes`, and many others. We will succinctly and gradually describe some of these basic objects and statements.

`Python` statements can be written in a text editor and typed into the program's terminal, but it is better to use appropriate IDEs. To use `Python`, we initially install a distribution of the Program. We recommend the current version which can be downloaded from the website https://www.python.org/ using the link https://www.python.org/ftp/python/3.15.1/python-3.15.1-amd64.exe. To type the codes, we recommend downloading the free program **Positron** from the website https://posit.co/ using the link https://cdn.posit.co/positron/releases/win/x86_64/Positron-2026.10.1-2-UserSetup-x64.exe.

See a screenshot of Positron:

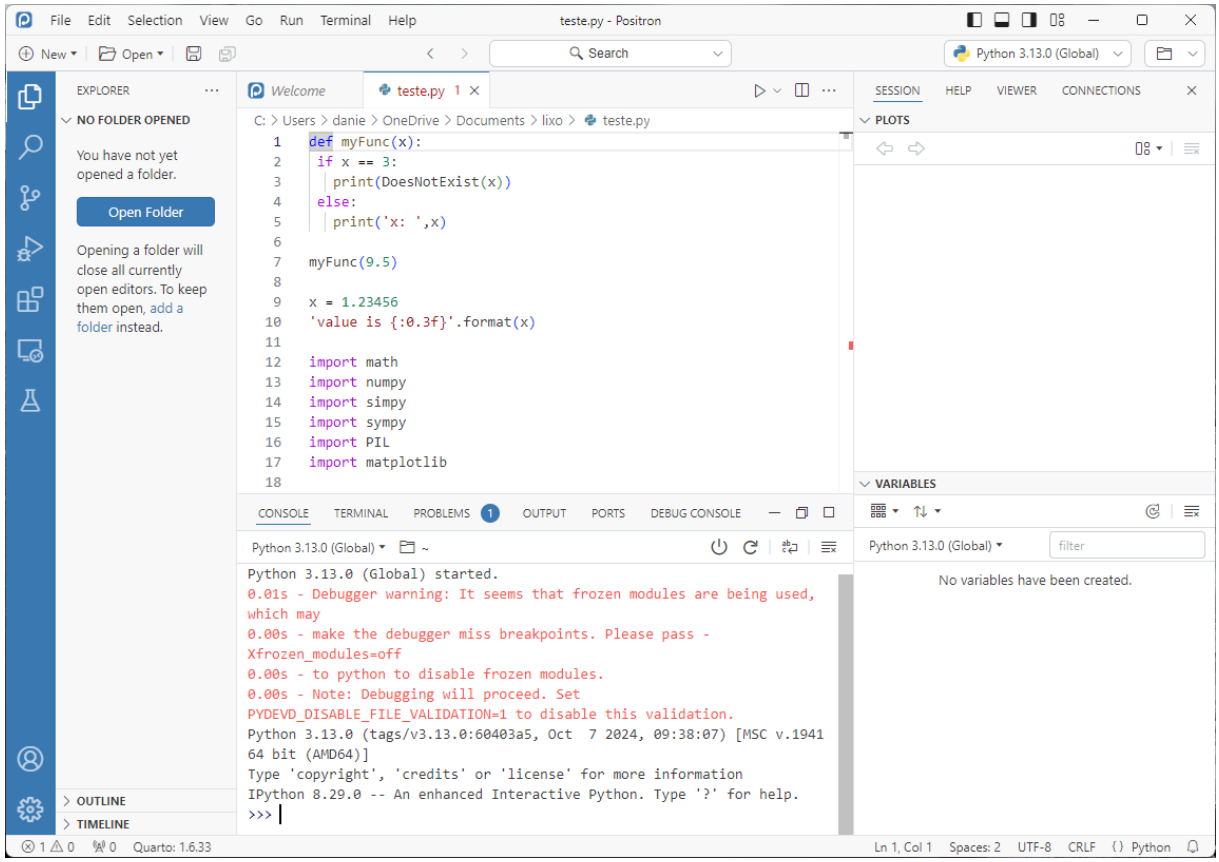


Figure 1.1: Positron: for code in `Python` or R

Once installed, we can type the codes and with **Ctrl Enter** we execute the codes line by line or a block of highlighted lines. It is important to install some basic libraries if they are not already installed. Next, we have a code for this purpose.

```
pip install numpy
pip install sympy
pip install PIL
pip install jupyter
pip install matplotlib
```

Next, we must import the libraries we need. In the following `script`, we consider the importation of all previously installed libraries. The `math` library does not need to be installed, as it already comes with the `Python` distributions.

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

⚠ Note on the functionality of scripts in real-time

Important: The `scripts` of this new version will run in real-time in the html version. For some functionalities, some changes were necessary in the generated `Python` codes for html compatibility. Furthermore, some `scripts` require previous ones to be run in order to load specific objects or libraries so that they are functional and do not present error messages.

1.1.1 Basic Arithmetic Operations

We can use `Python` as a calculator. In the following `script`, each physical line of the editor has a specific statement, illustrating this usage mode. The statements presented are of the type: expression statements. We can separate several expression statements on the same line by separating them with a semicolon.

```
# Illustrative program of elementary operations
# in Python
print(1 + 2 + 3)     # sum of the first 3 integers
print(3**10 - 1 + 8)
print(6 / 5 + 0.5**4)
```

```
6
59056
1.2625
```

We can observe that the `#` symbol is used to insert comments in the code and the `/` operator performs division using real operators. For integer division, we can use `//`, so `6 // 5` returns 1 and `6 / 5` returns `1.2`. The remainder of the integer division is obtained by the `%` operator. Thus, `6 % 5` returns `1`. We also have that the `**` operator is the power function, that is, for example, 3^{10} is `3**10` in `Python`.

`Python` can also perform operations with complex numbers, which in this case, are represented by $a + bj$, where a is the real part of the number and bj is the imaginary part, with $j = \sqrt{-1}$. The j is represented by i in mathematics and other field books. The following program illustrates an operation with complex numbers given by $(6 - 4i)^2$. Thus, we have:

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

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

Below we present a `script` that makes use of the `math` library, whose first statement was to import it, the penultimate to obtain the value of π, and the last one calculated $\sqrt{2}$. We also performed another operation with complex numbers.

```
import math

print(2 + 4 + 5.6)
```

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

```
11.6
-3.3333333333333335
(25+0j)
3.141592653589793

1.4142135623730951
```

The `NumPy` and `SymPy` libraries are for various mathematical calculations, the latter performing symbolic calculations.

```
import sympy
import numpy

numpy.set_printoptions(legacy='1.25')
print(sympy.sin(sympy.pi/5))
print(numpy.sin(numpy.pi/5))
# Note: In interactive mode (HTML/Pyodide), the output goes
# straight to innerHTML. To avoid '<' and '>'
# disappearing as invalid tags, we swap them for '[' and ']'.
# In static mode (PDF/LaTeX), just the pure command is enough:
# print(type(1.5 + 2.1j))
tipo = str(type(1.5 + 2.1j)).replace('<', '[').replace('>', ']')
print(tipo)
```

```
sqrt(5/8 - sqrt(5)/8)
0.5877852522924731
[class 'complex']
```

We can perform a basic mathematical operation like some of those presented above or even a more complex one and store the value in a variable, say `x`. These are assignment statements. The variable `x` can be used for other mathematical and even symbolic operations, if we use `SymPy`. See Knuth [1984] for a discussion on symbolic programming. The following `script` illustrates some cases of this procedure.

```
import sympy

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, 'and', 'z =', z)
```

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

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

We assign data to `Python` variables. `Python` is case-sensitive, so names like `X` and `x` are different. In `Python`, variables are pointers. Assignments like $x = 2$ create an object x and store the value 2 in it in other languages, but in `Python`, there is an integer object 2 and the variable x is a pointer pointing to it. See the consequences of this below, noting that `\n` performs a line break. There are no problems with scalar objects like these when we assign the same object to all of them and then change the assignment of

one of them to another object (value). However, when it comes to, for example, lists, our next object to be studied, the issue is quite different.

```
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
```

To deal with complex number functions, the `cmath` library must be imported, and trigonometric functions of complex numbers can be used based on this library rather than `math`, which is designated for real numbers (`float`). See the illustrative `script` below.

```
import cmath

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

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

1.2 Boolean Variables

Boolean variables point to `True` and `False` values only. Various comparison operations like `==`, `>=`, `<=`, `&` (`and`), `|` (`or`), and `!` (negation - `not`) can be used for this type of variable, the *booleans*. See a simple example of this. Later, we will return to talk about these *boolean* variable operators.

```
x = True
print('x = ',x)
y = False
print('y = ',y)
print(x != y) # or
z =  not(y)
print(x == z)
```

```
x =  True
y =  False
True
True
```

1.3 Strings

Strings (text variables) are an important type in `Python`. Once we have an object defined, the methods and functions are available to be used. Strings are denoted by `str`. See some examples, where strings were assigned or not to variables.

```
print('This is a string')
mensagem = 'Federal University '
# in normal mode use: print(type(mensagem))
print(str(type(mensagem)).replace('<', '[').replace('>', ']'))
print(mensagem)
UFLA = mensagem + 'of Lavras, MG.'
print(UFLA)
```

```
This is a string
[class 'str']
Federal University
Federal University of Lavras, MG.
```

Some methods that we can use with *strings* are illustrated below, among many others, showing the power of the object-oriented language.

```
mensagem = 'Federal University '
UFLA = mensagem + 'of Lavras, MG.' # new assignment to work independently
print(UFLA.capitalize())           # from the previous script
print(UFLA.lower())
print(UFLA.upper())
print(UFLA)
```

```
Federal university of lavras, mg.
federal university of lavras, mg.
FEDERAL UNIVERSITY OF LAVRAS, MG.
Federal University of Lavras, MG.
```

These methods act on the object, but as it was clear in the previous example, they do not change the content of the object. We can perform operations with strings, as illustrated below. The `str.format` method makes it possible to format `strings`, such as including `substrings` in the fields marked with `{}`.

```
mensagem = 'Federal University '
UFLA = mensagem + 'of Lavras, MG.'
print(UFLA * 2)
nome = 'First name: {}, Last name: {}'
nome.format('Daniel', 'Furtado Ferreira')
```

```
Federal University of Lavras, MG.Federal University of Lavras, MG.
```

```
'First name: Daniel, Last name: Furtado Ferreira'
```

If the user enters `Daniel`, the result will be `Daniel was approved!`. This version of `Markdown` does not yet support interactivity with the user. Therefore, the `input` was not evaluated in the output of this `script`. In the second statement, if the user enters a non-integer number, there will be an error message. There are options to deal with errors of this type and other causes as well.

To interact with the user, we use the `input()` function. It pauses the execution of the program and waits for a string to be typed on the keyboard.

See the standard code that would run in your local terminal (VS Code or Jupyter):

```
nome = input('enter your first name: ')
print(nome + ' was approved!')
x = int(input('enter an integer value: '))
  # transforms str into integer: int
x # if the input number is not int, it results in an error
```

> ⚠ Note on `input()`
>
> **Online environment:** Since our book runs directly in your internet browser, the native `input()` function cannot open a keyboard dialog box synchronously.

In the interactive console below, we simulate the behavior of `input()` by defining the values directly in the variables `input_nome` and `input_x`. Change the texts in quotes to test the code behavior and the

type conversion error:

```
# The student must change the string value directly in the variable!
nome = 'Daniel'  # Change your first name here

print(nome + ' was approved!')

# Simulation of integer input
entrada_usuario = '10' # Change the string value to test

x = int(entrada_usuario)
  # transforms str into integer: int
print(x) # if the input number is not int, it results in an error
```

```
Daniel was approved!
10
```

1.4 Lists, Tuples, Sets and Dictionaries

We will cover each of these objects separately. Let's start with lists.

1.4.1 Lists

Lists (`lists`) are the first building blocks for dealing with data manipulation. Lists are vectors whose first element starts at `0`, but whose elements in each position can be of different types, from integers, *booleans*, reals, complex, strings, characters, sets, tuples, and other lists. Therefore, the list can be mixed, that is, contain different types of objects throughout its positions. Lists are part of the quartet `list`, `tuple`, `set`, and `dictionary`. The `NumPy` library provides additional tools for dealing with large collections of data.

```
x = [1, 2, 3, 4]
print(x)
y = [1, 'Stat', 3.5, 4+5j]
print(y)
print(y[3])
print(str(type(x)).replace('<', '[').replace('>', ']'))
print(str(type(y)).replace('<', '[').replace('>', ']'))
```

```
[1, 2, 3, 4]
[1, 'Stat', 3.5, (4+5j)]
(4+5j)
[class 'list']
[class 'list']
```

The variable `x` is a list of integers with 4 elements, which are indexed by 0, 1, 2, 3. Thus, $x[1]$ points to the value 2 and $x[0]$ to the value 1. The variable `y` is also a list with 4 elements of different types, with $y[0]$ being an integer, $y[1]$ a `string`, $y[2]$ a `float`, and $y[3]$ a complex number. To create the list, we simply use the brackets `[]`, with each element of the list separated by a comma. This statement is the list class constructor. It is possible to create a list with elements having repeated values, and they will be identified as being different, because the list respects and preserves the input order of the values. We can check if an element belongs to the list with `in`, as the following `script` shows, among other examples.

```
print([2, 7] == [7, 2])
print([5, 7] == [5, 7, 7])
x = [1, 2, 3, 4]
```

```
y = [1, 'Stat', 3.5, 4+5j]
print(x)
print(2 in x)
print(7 in x)
print(y)
print(4+5j in y)
print('Daniel' in y)
```

```
False
False
[1, 2, 3, 4]
True
False
[1, 'Stat', 3.5, (4+5j)]
True
False
```

We can, in the same way as was done with `strings`, perform some arithmetic operations with lists, as shown in the example of the following `script`. These arithmetic operations do not act as mathematical arithmetic operations on `Python` lists. For example, `y*2` duplicates the list `y`, creating a new list with the duplicated values. The operation `x+y` concatenates the two lists `x` and `y`, forming a new list.

```
x = [1, 2, 3, 4]
y = [1, 'Stat', 3.5, 4+5j]
print(x+y)
print(x+[[0,1],'test',[1,0]])
print(y*2)
print('first element of the list',y[0]) # first element of the list
print('last element of the list',y[-1]) # last element of the list
# numbering -1,-2,-3,-4 for the index
# accesses the positions 3,2,1,0,
# respectively, of the list y
```

```
[1, 2, 3, 4, 1, 'Stat', 3.5, (4+5j)]
[1, 2, 3, 4, [0, 1], 'test', [1, 0]]
[1, 'Stat', 3.5, (4+5j), 1, 'Stat', 3.5, (4+5j)]
first element of the list 1
last element of the list (4+5j)
```

Let's now look at the problem of pointers, through the example of the `script` presented next.

```
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)
# all objects were changed and not just b
# because they point to the same list [1,2,3]
```

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

We observe that if `x`, `z`, and `b` point to the same object, then if we change the value `b[1]` from `2` to `7`, all three objects will be changed at position `1`, which corresponds to the second value of the list, since it starts at position `0`. However, if instead of `b[1] = 7` we had used the assignment `b = [7,9]`, then the

vectors `x` and `z` would not be changed with the new assignment of vector `b`. In the previous examples, we also saw that the elements of a list are accessed by their index, which varies from `0` to `n-1`, where `n` is its length. Thus, the list `x=[1,2,3,4]` has its elements `x[0]` equal to `1`, `x[1]` equal to `2`, `x[2]` equal to `3`, and `x[3]` equal to `4`. We can also vary the index from `-1` to `-n`, with `-1` meaning the last position of the list, that is, position `n-1`, `-2` corresponding to position `n-2`, and so on, until `-n`, which corresponds to position `0` of the list.

Lists are objects, and as such, we can use some methods associated with them. Lists are mutable and dynamic (we can change their elements), they are ordered (each element of the list has a defined order upon its creation), and they allow repeated elements. To use a method or a function, we must consider the difference between them. Although all methods are functions in `Python`, not every function is a method. Functions receive objects as inputs and do not modify them, while methods act on the objects. Below is a list of some methods or functions associated with lists:

- `sort()`: sorts the list in ascending order.
- `append()`: adds an element to the end of the list.
- `extend()`: adds multiple elements to the list.
- `index()`: used to find the index of an element in the list.
- `max(list)`: returns the maximum value of a list.
- `min(list)`: returns the minimum value of a list.
- `list(tuple)`: transforms a `tuple` into a list.
- `len(list)`: returns the size of the list (number of elements).
- `filter(fun,list)`: filters a list using a function `fun`.

Let's illustrate some of these methods with particular examples. We will consider a list and apply the `sort()` method to sort its values. In this example below, we will see the difference between a method and a function, observing how the method modifies the object that called it. In this case, a method call is given by the name of the list (object) followed by a dot and the name of the method: `lista.met()`.

```
x = [7.4, 5.8, 9.3, 3.2]
print('original x: ',x) # original object x
x.sort()
print('modified x: ',x) # object x modified by the sort() method, ordered
```

```
original x:  [7.4, 5.8, 9.3, 3.2]
modified x:  [3.2, 5.8, 7.4, 9.3]
```

For the `append` method, we have the following `script`, which added the month of April to the end of a list with the first three months of the year.

```
mes = ['January', 'February', 'March']
mes.append('April')
print(mes)
```

```
['January', 'February', 'March', 'April']
```

The `extend` method is applied to the previous list `x` and adds two more elements to its end.

```
x = [7.4, 5.8, 9.3, 3.2]
print(x)
x.extend([1.6, 11.6])
print(x)
```

```
x.sort() # sorts x, because it was no longer in order
print(x)
```

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

`index()` is a method to find the index of an element in the list. If the element sought is not in the list, `Python` will show an error message.

```
i = x.index(7.4) # depends on x from the previous script - execute it
print(i) # remember that the list starts at 0 and not at 1
```

```
3
```

Functions `len`, `max()`, and `min()`, on the other hand, act on the argument of the function, but do not modify it. See the effect of these functions on the previously presented list `x` in the example.

```
# execute the two previous scripts
print('Size of x: ',len(x))   # size of list x
print('Size of mes: ',len(mes)) # size of list mes
b = max(x)
a = min(x)
print('a=min(x) = ',a)
print('b=max(x) = ',b)
print('mes: ',mes)
print('x: ',x) # x and mes not modified
```

```
Size of x:  6
Size of mes:  4
a=min(x) =  1.6
b=max(x) =  11.6
mes:  ['January', 'February', 'March', 'April']
x:  [1.6, 3.2, 5.8, 7.4, 9.3, 11.6]
```

To illustrate the use of `filter()`, let's consider a function that returns `True` or `False` for a certain condition of interest. For example, if we wanted to know which numbers of the six elements of list `x` have a remainder of division by `2` less than `1.5`. This result is obtained with the comparison `a % 2 <= 1.5`, which will return true or false for the number represented by `a`. But we must do this for all elements of the list `x` or any other list. We must create a function to receive each element of the list and check the condition, returning `True` or `False`. This is done by passing the list `x` or any other list through the `filter()` function to perform the intended iteration and comparison on its elements. Let's create a first function and then apply the `filter()` function. The type of object returned from this function is `filter`, so the final result must be transformed into a list before printing.

```
# execute previous scripts to load x
def resto(a):
    if ((a % 2) <= 1.5):
        return True
    else:
        return False
# apply the filter function
x_filtrado = filter(resto, x)
print('filtered x: ',list(x_filtrado))
print('original x: ',x)
```

```
filtered x:  [3.2, 7.4, 9.3]
original x:  [1.6, 3.2, 5.8, 7.4, 9.3, 11.6]
```

We can use the constructor `list(object)` to construct a list from this object, as occurred with the object `x_filtrado` from the previous `script`. So `list()` is a list constructor. Note that the `%` operator returns the remainder of the integer division, which in this case, was by `2`. The function `resto` returns true or false according to the condition of the remainder of the division of `a` by `2`. To define a function we necessarily use `def` followed by the function name (`resto`) with the argument (`a`). Then comes the body of the function, which must have mandatory indentation. There is no separation with `{}` or `[]` or other characters to separate the body of the function. This separation is done solely using the appropriate indentations. We will talk more about functions in detail later.

We can take advantage of some ready-made list functions to calculate some quantities of interest, such as, for example, the sum of their elements. For this, we could use a loop structure, like the `for` loop that we will see later, and as the *loop* progresses, we update the sum. However, we can use the `sum()` function. A `for` *loop* is executed as interpreted ***bytecode***, while the `sum()` function is written purely in the C language, therefore, much faster and more efficient. See the code below to illustrate the sum of all values of the vector `x`. We will talk about conditional and repetition structures later.

```
# execute the two previous scripts
soma = sum(x)
print('Sum: ',soma)
# mean
print('Mean: ',soma / len(x))
```

```
Sum:  38.9
Mean:  6.483333333333333
```

Other methods like `count()` (counts the number of occurrences of a given value), `reverse()` (sorts the list in reverse order to the original order), `clear()` (clears all data from the list), `copy()` (copies all data from the list), `insert()` (inserts an element at a specific position in the list), and `pop()` (removes an element at a specific position) are presented in the `script` below.

```
L = [1, 2, 3, 5, 7, 2, 4.5]
print('How many 2s: ',L.count(2))
print('How many 2.1s: ',L.count(2.1))
L1 = L.copy()
L.reverse()
print('Reversed L: ',L)
L.clear()
print('Empty L: ',L)
print('L1: ',L1)
L1.insert(1, 3) # inserts the value 3 at position 1
print('L1 with + 3 at pos. 1: ',L1)
L1.pop(1) # removes the element 3 from position 1
print('L1 without the element from pos. 1 (3): ',L1)
```

```
How many 2s:  2
How many 2.1s:  0
Reversed L:  [4.5, 2, 7, 5, 3, 2, 1]
Empty L:  []
L1:  [1, 2, 3, 5, 7, 2, 4.5]
L1 with + 3 at pos. 1:  [1, 3, 2, 3, 5, 7, 2, 4.5]

3
```

```
L1 without the element from pos. 1 (3):  [1, 2, 3, 5, 7, 2, 4.5]
```

We can build a matrix using a list, since `Python` does not have a native matrix object. If we create a list of `n` components, with each of the components having `m` components, we will have an $n \times m$ matrix. Let's see in the `script` below the construction of the following matrix:

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

The `script` corresponding to the matrix defined in (1.1) is:

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

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

We will also present some extra details to access the values of a list. We can access the value of a list `x` by indicating the position of the element we want to access as follows: `x[i]`, where `i` represents an integer value between `0` and `len(x)-1`, say `n-1`, for lists of size `n`. To access a subset of a list `L` of size `n`, we can use the following statement `L[m:s]` which will access the elements from position `m` to position `s-1` and not `s`. If the indices are negative, then the list will be accessed from back to front, with `-1` corresponding to position `n-1`, `-2` to position `n-2`, and so on, until `-n`, which corresponds to position `0`. See some examples in the following `script`.

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

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

Accessing particular positions, for printing or for assignment:

```
print(L[0])
L[1] = 8
print(L)
```

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

Accessing positions with negative indices.

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

```
7
1
```

For blocks of elements we have that the statement `L[m:s:r]` accesses the elements at positions `m`, `m+r`, `m+2r`, ... until the `(s-1)`-th position or until the closest position to `s-1` possible. Great care must be taken, because the upper limit does not indicate where the subset ends among the valid indices of a list, but rather, that it ends at the highlighted position subtracted by `1`.

```
print(L[1:4])
print(L) # passes the limit len(L)
print(L[0:5:2])
print(L[-1:-8:-1])
print(L[4:])
print(L[:6])
```

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

If we omit the lower or upper limits of the sequence, then the selected list will start at index `0` (initial value) or end at the last index (final value of the list), as in the last two examples presented.

1.4.2 Tuples

Tuples are objects very similar to lists. Several methods and functions that apply to lists also apply to tuples. Unlike lists, tuples are immutable objects, meaning that once created, they cannot be modified. If we create a tuple by `t = (1,2,3)` we **cannot** assign a value, for example, in this way: `t[1] = 9`. They can contain more than one identical value and are ordered, like lists. The way to create a tuple compared to a list is the use of parentheses instead of square brackets.

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

```
True
DFF
1
(1, 2, 'DFF')
4
```

The `count()` and `index` methods can be used on tuples, as illustrated below. The `index()` method has the following syntax, with the last two arguments being optional: `tuple.index(element, start, end)`.

```
print(t.count(1)) # number of occurrences of 1
print(t.index('DFF')) # index of the position of 'DFF'
```

```
1
2
```

The tuple can have any type as its elements, including a tuple or a list.

```
t1 = ((1,2),'r', [2,3,4])
print(t1)
print('List component of the tuple ',t1[2])
print('Element 0 of the list component of the tuple ',t1[2][0])
t1[2].append(5)
print('Modifying the list component of the tuple ',t1)
t2 = (1,2,3)
print('Sum of t2: ',sum(t2))
```

```
((1, 2), 'r', [2, 3, 4])
List component of the tuple  [2, 3, 4]
Element 0 of the list component of the tuple  2
Modifying the list component of the tuple  ((1, 2), 'r', [2, 3, 4, 5])
Sum of t2:  6
```

The question that arises is why should we use tuples if they cannot be modified? The answer to this comes from the fact that data manipulation via tuples, which are immutable, is much faster than with lists. Even though the tuple points to the same memory ID, we were able to modify one of its elements, which was the list in its second position. This did not change the memory ID that tuple `t1` pointed to.

There are many methods or functions that work with tuples such as `len(t)` and `count()` as illustrated below.

```
t = (1,1,2,3,3,3,4,5)
print('Number of 3s: ',t.count(3))
print('Size of t: ',len(t))
```

```
Number of 3s:  3
Size of t:  8
```

The `any(t)` function returns `True` if there is any `True` item in the tuple and returns `False` otherwise. In this case, the tuple or any other appropriate type could have elements `0` and `1` or *booleans*. This function can be applied to lists, sets, and dictionaries.

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

```
True
```

We can also use the functions `min()`, `max()`, `sum()`, and `sorted()`, as illustrated below. The `sorted()` function sorts the tuple and returns a sorted **list** as a result. Note that the `lista.sort()` method alters the `list` object and cannot be applied to the tuple, because the tuple is immutable.

```
t = (2.3,4.5,1.2,1.1,9.7,5.3)
print('Min.: ',min(t))
print('Max.: ',max(t))
print('Sum: ',sum(t))
print('Sorted t: ',sorted(t))
print('Original t: ',t)
```

```
Min.:  1.1
Max.:  9.7
Sum:  24.099999999999998
Sorted t:  [1.1, 1.2, 2.3, 4.5, 5.3, 9.7]
Original t:  (2.3, 4.5, 1.2, 1.1, 9.7, 5.3)
```

1.4.3 Sets

`Set` objects have a connotation very close to the definition in mathematics. These are sets in which the order or duplication of their elements do not change the set. They are immutable, unordered, and do not have identical elements in their occurrences. We can use the constructor `set()` to create a set or use curly braces `{}` to type its elements separated by a comma.

```
phi = set()
print('Empty set: ', phi)
# it needs to be a list or tuple as argument
A = set(['A','D','B','C','E'])
print('A: ',A)
B = {1,2,3.4,5,6,6}
print('Element "A" belongs to A: ','A' in A)
print('B: ',B) # note that the repeated
```

```
  # element 6 appears only 1 time
print(B[0]) # this will result in an error
```

```
Empty set:  set()
A:  {'E', 'B', 'A', 'D', 'C'}
Element "A" belongs to A:  True
B:  {1, 2, 3.4, 5, 6}
```

```
TypeError: 'set' object is not subscriptable
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[36], line 10
      6 B = {1,2,3.4,5,6,6}
      7 print('Element "A" belongs to A: ','A' in A)
      8 print('B: ',B) # note that the repeated
      9   # element 6 appears only 1 time
---> 10 print(B[0]) # this will result in an error
TypeError: 'set' object is not subscriptable
```

We cannot access an element of a set by `B[0]`, for example, which causes an error, as can be seen in the result of the previous `script`. The order is not important, as can be verified by comparing the previous set `A` with the new set `C` created below.

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

```
C:  {'E', 'B', 'A', 'D', 'C'}
A:  {'E', 'B', 'A', 'D', 'C'}
A=C:  True
```

Some mathematical operations with sets are available in `Python`, such as union, intersection, difference ($A^c \cap B$), and symmetric difference ($(A^c \cap B) \cup (A \cap B^c)$), as illustrated in the example below.

```
A = {1,2,3,4,5,6}
B = {4,5,7,8,9,10}
print('A: ',A)
print('B: ',B)
print('A U B: ',A.union(B))
print('A intersect B: ',A.intersection(B))
print('A-B: ',A.difference(B)) # is in A, but not in B
print('B-A: ',B.difference(A)) # is in B, but not in A
print('Symmetric diff.: ',A.symmetric_difference(B)) # is only in A or only in B
print('A intersect B: ',A & B) # intersection
print('A U B: ',A | B) # union
print('A-B: ',A - B) # difference
print('B-A: ',B - A) # difference
print('Sym. diff.: ',A^B)   # symmetric difference
```

```
A:  {1, 2, 3, 4, 5, 6}
B:  {4, 5, 7, 8, 9, 10}
A U B:  {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
A intersect B:  {4, 5}
A-B:  {1, 2, 3, 6}
```

```
B-A:  {8, 9, 10, 7}
Symmetric diff.:  {1, 2, 3, 6, 7, 8, 9, 10}
A intersect B:  {4, 5}
A U B:  {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
A-B:  {1, 2, 3, 6}
B-A:  {8, 9, 10, 7}
Sym. diff.:  {1, 2, 3, 6, 7, 8, 9, 10}
```

1.4.4 Dictionaries

Dictionaries (`dictionary()`) are mutable and iterable objects. Their elements always come in pairs, the first value being a key and the second element, the key's value. Both the key and its value are `Python` objects. The key is immutable, but its associated values are mutable or immutable objects.

```
D = {1: [1,2,3.4,5], 2: 3.7, 3: {1,2,3}}
print(D[1])
print(D[2])
print(str(type(D[1])).replace('<', '[').replace('>', ']'))
print(str(type(D[3])).replace('<', '[').replace('>', ']'))
```

```
[1, 2, 3.4, 5]
3.7
[class 'list']
[class 'set']
```

We have a dictionary, with keys `1`, `2`, and `3`. For key `1` we have a list as its value, for key `2`, we have a `float` value, and for key `3`, we created a set object. Next, we add a key, named `4` with an associated *boolean* value. The `keys()` method retrieves the keys of the dictionary object `D`, transforms them into a list `C`, and prints the resulting list and its first element `C[0]`.

```
# execute the previous script
D[4] = True
print(D)
C = list(D.keys())
print(C)
print(C[0])
```

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

Similarly, we can easily get the values of the keys, as illustrated in the `script` below. Later, we will see how we can use a `for` loop structure to iterate through the elements of the dictionary and store the values in a list or process them one by one.

```
# execute the previous scripts
print('D: ',D)
print('Values of D: ',D.values())
print('List of values of D: ',list(D.values()))
print('D[C[0]]: ',D[C[0]]) # accessing the value with key 1, C[0]
print('C[0] belongs to D: ',C[0] in D) # checking if key 1 belongs to D
print('5 belongs to D:',5 in D)    # check if key 5 belongs to D
```

```
D:  {1: [1, 2, 3.4, 5], 2: 3.7, 3: {1, 2, 3}, 4: True}
Values of D:  dict_values([[1, 2, 3.4, 5], 3.7, {1, 2, 3}, True])
List of values of D:  [[1, 2, 3.4, 5], 3.7, {1, 2, 3}, True]
```

```
D[C[0]]:  [1, 2, 3.4, 5]
C[0] belongs to D:  True
5 belongs to D: False
```

We can delete the content of a key using the `del` function or using the `pop()` method. We can update the dictionary, creating new keys and values, with the `update` method.

```
# execute the previous scripts
del D[3] # deletes key 3
print(D)
print(D.pop(4)) # deletes key 4
print(D)
D.update({5: 'I am new', 6:'me too'})
print(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: 'I am new', 6: 'me too'}
```

We can create a dictionary object by creating two tuples, say `c` and `v`, with the keys and the key values (of the same size). Then we pair the elements with the `zip(c, v)` statement. Finally, we use the `dict` constructor to create the dictionary of paired values.

```
c = (1,2,3,4) # keys
v = ([1,2],True,4.5,('r','s'))
y = dict(zip(c,v))
print(y)
print(y.get(5,'Key does not exist'))
# get() does not generate an error for an inexistent key
# but, y[5] would generate an error
print(y.get(1))
print(y[1]) # as 1 exists, it is equivalent to get()
```

```
{1: [1, 2], 2: True, 3: 4.5, 4: ('r', 's')}
Key does not exist
[1, 2]
[1, 2]
```

Thus, having access to a value from the list, we can use the appropriate methods and functions to deal with them. More details of these objects will appear in due course as we advance into the next chapters with more features of `Python` programming.

1.5 Matrices and Arrays

Matrices in `Python`, as we said and showed earlier, can be created by lists. Next, we will create a 2×2 matrix using the `list` constructor to illustrate the creation procedure. If the dimension is just one, the lists are one-dimensional arrays, known as vectors.

```
A = [[4,1],[1,1]] # 2 x 2 matrix
print('A = ',A)
print(A[1][1])   # returns the value A[2,2]
A[1][1] = 2       # changes its value
print(A)
```

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

To deal with vector (`arrays`) or matrix functions, we can, among other possibilities, use the `NumPy` library. Our first step is to import the `NumPy` library with the alias `np`, which is conventionally used to facilitate calling its methods and functions. This can only be done if we already have the `NumPy` library installed.

```
import numpy as np
```

Then, we create a matrix using the `array()` and `asarray()` constructors. We will use our previous list `A` to do this to illustrate. It is important to note that lists accept data of various types, while `NumPy` lists or arrays are homogeneous structures more appropriate for mathematical and scientific calculations. Furthermore, calculations on `NumPy` arrays are vectorized, meaning we can perform element-by-element (vectorized) operations automatically and extremely fast. `NumPy` arrays are more efficient in memory usage because they store data in blocks and not as pointers where list data is scattered throughout memory.

```
# execute the previous script
import numpy as np

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

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

We can also create the `NumPy` matrix from tuples, instead of lists. In addition, there are functions specific to the library to create matrices, such as, for example, the 2×4 zero matrix below. There are also functions to create one-dimensional `arrays` (vectors), such as the `arange(n, m)` and `linspace(n, m, s)` methods. The first creates a vector going from `n` to the maximum `m` (integers) without including it by steps of 1, or from the minimum `n` to the maximum `m` (excluding the maximum) by steps of r: `arange(n,m)` or `arange(n,m,r)`. On the other hand, `linspace(n,m,s)` starts at `n`, ends at `m`, but with a step equal to `(m - n) / (s - 1)`. Without specifying the `s` argument, as in this command `linspace(n,m)`, `NumPy` uses the default value of `50` for it.

```
import numpy as np

C = np.zeros((2,4))
print(C)
print(np.arange(2,7)) # vector with elements 2,3,..,6
print(np.arange(2,7,0.5)) # from 2 to 7, in steps of 0.5 (except 7)
print(np.linspace(2,6,6))
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]]
[2 3 4 5 6]
[2.  2.5 3.  3.5 4.  4.5 5.  5.5 6.  6.5]
[2.  2.8 3.6 4.4 5.2 6. ]
```

Let's illustrate some simple calculations with vectors.

```
import numpy as np

x = np.arange(2,3,0.2)
y = np.arange(3,6,0.7)
print('', x,'\n', y)
```

```
print(x + y) # addition of vectors
print(x * y) # elementwise product
print(y ** x) # elementwise exponentiation
```

```
 [2.  2.2 2.4 2.6 2.8]
 [3.  3.7 4.4 5.1 5.8]
[5.  5.9 6.8 7.7 8.6]
[ 6.    8.14 10.56 13.26 16.24]
[  9.          17.78458735  35.01760371  69.13253113 137.27719037]
```

Matrix multiplication in turn can be done with the `np.dot` function, which has a method version as well, not modifying the object that triggered it.

```
import numpy as np

C = np.array([[2,1],[1,2]])
print('B: ',B)
print('C: ',C)
print('BC: ',np.dot(B,C))
print('BC: ',B.dot(C))
print('Original B: ',B)
```

```
B:  [[4 1]
 [1 2]]
C:  [[2 1]
 [1 2]]
BC:  [[9 6]
 [4 5]]
BC:  [[9 6]
 [4 5]]
Original B:  [[4 1]
 [1 2]]
```

Inverses can be obtained with the `np.linalg.inv()` function and the generalized Moore-Penrose inverses by the `numpy.linalg.pinv()` method. The determinant can be obtained by `np.linalg.det()` and the eigenvalues and eigenvectors (spectral decomposition) by `np.linalg.eig()`. It is important to note that the eigenvalues of `np.linalg.eig()` will not necessarily be sorted from largest to smallest, as is conventionally adopted in other programs. This method is very general and can be used on square real or complex matrices. Alternatively, for symmetric matrices `NumPy` has the `np.linalg.eigh()` method, which returns the eigenvalues in ascending order. See the example below.

```
# execute the previous scripts
import numpy as np

print(np.linalg.inv(B))
print(np.linalg.pinv(B)) # equals inverse (full rank)
print(np.linalg.det(B))
L, P = np.linalg.eig(B)
print('Eigenvalues: ',L,'\nEigenvectors: ',P)
L1, P1 = np.linalg.eigh(B)
print('Eigenvalues: ',L1,'\nEigenvectors: ',P1)
```

```
[[ 0.28571429 -0.14285714]
 [-0.14285714  0.57142857]]
```

```
[[ 0.28571429 -0.14285714]
 [-0.14285714  0.57142857]]
6.999999999999999
Eigenvalues:  [4.41421356+0.j 1.58578644+0.j]
Eigenvectors:  [[ 0.92387953+0.j -0.38268343+0.j]
 [ 0.38268343+0.j  0.92387953+0.j]]
Eigenvalues:  [1.58578644 4.41421356]
Eigenvectors:  [[ 0.38268343 -0.92387953]
 [-0.92387953 -0.38268343]]
```

We can also use the `scipy` library with the `sp.linalg.eig()` and `sp.linalg.eigh()` methods, which do exactly as their peers in the `NumPy` library. For this, we must import, after installing, the library using `import scipy as sp`.

For `n x m` matrices, we can use a matrix decomposition very useful for statistics, in methods such as principal components, AMMI, biplot, among others. This method is called singular value decomposition. In it we obtain the decomposition of a matrix $\mathbf{A}$ `(m x n)` as follows: $\mathbf{A} = \mathbf{U}\mathbf{\Lambda}\mathbf{V}^\top$, where the `np.linalg.svd()` method returns $\mathbf{U}$ an $m \times k$ orthonormal matrix by columns, $\mathbf{V}$ an $n \times k$ orthonormal matrix by columns and the vector corresponding to the diagonal of $\mathbf{\Lambda}$, which is a $k \times k$ diagonal matrix of real positive elements, if we use the `full_matrices=False` option. The matrices $\mathbf{U}$ and $\mathbf{V}$ are the singular vector matrices and the matrix $\mathbf{\Lambda}$ is the singular value matrix, where k is the rank of $\mathbf{A}$ which is $k \leq \min(n,m)$. The following `script` illustrates obtaining the singular decomposition of a 3×2 matrix.

```
import numpy as np

A = np.array([[1,4],[2,7],[9, 3]])
U,L,Vt = np.linalg.svd(A, full_matrices=False)
print('Left singular vectors: ',U,
'\nVector of singular values: ',L,
'\nRight singular vectors (transposed): ',Vt)
print(len(L)) # rank of A
print(np.diag(L)) # builds the Lambda matrix
```

```
Left singular vectors:  [[-0.30248631 -0.39963983]
 [-0.54614812 -0.67137377]
 [-0.78116852  0.62413562]]
Vector of singular values:  [11.19813546  5.88232625]
Right singular vectors (transposed):  [[-0.75238412 -0.65872463]
 [ 0.65872463 -0.75238412]]
2
[[11.19813546  0.        ]
 [ 0.          5.88232625]]
```

To verify that the decomposition is really adequate, we have the following `script`:

```
# execute the previous script
print('ULVt: ',np.dot(np.dot(U,np.diag(L)),Vt))
print('ULVt: ',U.dot(np.diag(L)).dot(Vt)) # alternative
print('ULVt: ',U @ np.diag(L) @ Vt)       # alternative
# Optimized vector-matrix multiplication followed by matrix multiplication
ULVt = U @ (L[:, None] * Vt)
print(ULVt)
```

```
ULVt:  [[1. 4.]
 [2. 7.]
```

```
 [9. 3.]]
ULVt:  [[1. 4.]
 [2. 7.]
 [9. 3.]]
ULVt:  [[1. 4.]
 [2. 7.]
 [9. 3.]]
[[1. 4.]
 [2. 7.]
 [9. 3.]]
```

Many other functions exist in the `NumPy` library to deal with or operate on matrices. If we need some other function for some matrix operation, we will present it on those occasions.

1.6 Data Files

We will use this data structure little in this material, but we will make a brief introduction and explanation about the `pandas` library to deal with `DataFrames`. We will present the reading of a particular file with two variables `X1` and `X2`, recorded as a text file separated by space between columns (variables). The path where this file is located on my computer is: `g:/Meu Drive/daniel/Cursos/Estatistica computacional/Apostila/` and its name is `dados.txt`. We must initially install the `pandas` library with the statement: `pip install pandas`. Subsequently, we load the `pandas` library, as presented in the `script` below. To change the directory, we need to import the `os` library and use `os.getcwd()` to get the current working directory and to change it `os.chdir('[path]')`.

```
import pandas as pd
import os

apath = os.getcwd() # project path
os.chdir('g:/Meu Drive/daniel/Cursos/Estatistica computacional/Apostila/')
print(os.getcwd())
```

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

To read a file from this directory into a `DataFrame` object, we can use the `pd.read_csv('dados.txt',r'\s+')` function, whose symbol `r'\s+'` means that the file is separated by spaces, and the number of spaces between columns from record (row) to record may vary, that is, if the number of spaces is not standardized between columns for the different lines of the data file.

```
# execute the previous script
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

	X1	X2
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

`DataFrames` are tabular data structures, where each column consists of a sequence of values of the same type (*boolean*, float, strings, etc.), where the different columns can be and potentially are of different types. `DataFrames` have an additional column called `index` which in the previous example, of the `dados` object, varied from `0` to `19`, because our `DataFrame` has `20` lines and two variables `X1` and `X2`, which in the `dados.txt` file were identified in the first physical line of the file that was read by the `read_csv()` method. The `index` maps the lines of the `DataFrame` with the mentioned `labels`.

We will show how to build a `DataFrame` directly from a `dict` object for the `pandas DataFrame()` constructor. We will create a `DataFrame` of a completely randomized design, with `2` treatments and `3` replicates of each, with the respective yields evaluated in the `6` experimental plots.

```
import pandas as pd

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
2	3	1	5.6
3	1	2	5.7
4	2	2	6.3
5	3	2	7.1

We can add a new column to our `DataFrame`, or eliminate an existing one, creating a new `DataFrame` to receive the result, as shown below, with the `columns` option, for any order of the keys (column names).

```
# execute the previous script

arqd
arqd1 = pd.DataFrame(arqd,columns=['trat','prod'])
arqd1 # selecting 2 variables in the DataFrame
arqd['alt'] = [1.5,1.3,1.4,1.2,1.1,1.6] # creating height variable
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

To access the keys (columns), indices, and values of the `DataFrame`, we can use the following illustrative codes. We observe that the statement `arqd.prod` to access the `prod` column results in an error, because `prod` is a reserved word (product). We must use the previous alternative for this key.

```
# execute the previous scripts

print(arqd.columns)
print(arqd.index)
print(arqd.values)
print(arqd['prod'][0]) # production of index 0
print(arqd['prod']) # entire production column
print(arqd.prod) # careful, because prod is a reserved word: error
print(arqd.rep)
```

```
Index(['rep', 'trat', 'prod', 'alt'], dtype='str')
RangeIndex(start=0, stop=6, step=1)
[[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
```

To extract an entire row we use the `loc` method of the `DataFrame` associated with the row index (record), as illustrated in the following `script`.

```
# execute the previous scripts
L  = arqd.loc[1] # second row of the DataFrame
print(L)
L2 = arqd.loc[[0,5]] # rows 1 and 6 of arqd
print(L2)            # with indices 0 and 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
```

To select a block of records (rows) going from `inicio` to `fim` (excluding the final limit) we use `arqd[inicio:fim]`. As illustrated below, where we extracted from index `0` to index `3`, that is, the first three rows with indices `0`, `1`, and `2` of `arqd`.

```
# execute the previous scripts
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

Missing values can be part of the `DataFrame`, and in this case, they assume the value `NaN`, which stands for `not a number`. We can delete a column with the `del` statement, as follows.

```
# execute the previous scripts
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

We can filter by imposing specific conditions on the `DataFrame`. For example, if we were interested in the resulting `DataFrame` of elements where production is greater than or equal to `5`, we would have the following result. The filtered result shows the records in the same original positions, and its original `DataFrame` `arqd` remains unchanged.

```
# execute the previous scripts
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
2	3	1	5.6
3	1	2	5.7
4	2	2	6.3
5	3	2	7.1

To save a `DataFrame` we can choose the directory (folder) and the file name and save it to a `csv` file (comma-separated file) with the statement `arqd.to_csv('nome.csv,index=False, header=True)` so as not to save the index and to save the header. We will recover in our code the original *path* with the statement `os.chdir(apath)`, where `apath` was obtained when we started the topic about `DataFrame` and refers to the directory of this project. We chose the name `dic.csv` for the file. We can read the file again, as we showed in the first steps of the `DataFrames` approach. Once saved, we repeat its reading and place it in the `dic` object, where we must pay attention to the column separator, which in this case is the comma.

```
# execute the previous scripts
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

In future editions, we will show more details of `DataFrames`. In later chapters, if we come to need a `DataFrame` and some of its properties, then we will add the necessary contents at that time. We will now talk about conditional structures and repetition structures.

1.7 Programming Control Structures

`Python` is a programming environment in which programs contain modules, modules contain methods and functions, methods contain instructions, which create and process objects. These assignment statements are of the type `a = b`. We can call methods, functions, and use statements such as `print(a)`, `if/elif/else` for action selection, `for/else` to perform iterations, `while/else` for `loops` in general, `break` and `continue` for `loop` control, `def` for defining functions and methods, `yield` for generator functions, among others. The statements are organized into expressions in groups of instructions, which unlike other languages are organized by indentation (paragraph indent). The body or group of instructions will be gathered together if they are indented in relation to the main instruction and with the same level of indentation. In other languages, instruction groups are gathered by curly braces: `{groups of instructions}`. The end of a line ends the statement, which can also be finalized by a semicolon. A statement can continue on a new line if the line ends with a `\` or with the use of parentheses.

Variable names in `Python` must start with an `underscore` or letter, followed by numbers or letters or an `underscore`. `Python` differentiates uppercase from lowercase, so `Y` is different from `y`, for example. Names should avoid `Python` reserved words, such as False, None, True, class, and, if, elif, yield, while, break, global, nor, try, return, break, in, etc. By convention, classes in `Python` start with uppercase, and modules and variable names with lowercase.

Conditional structures, `if/elif/else`, are structures for selecting actions. This instruction can contain other instructions of the same type or different instructions in its groups. The general format is given by

```
if condition1:
  instructions1
elif condition2: # optional elif
  instructions2
else:            # optional else
    instructions3
```

The `elif` statement means `else if` and is optional. If `condition1` is verified, the block named `instructions1` (one or several lines of instructions) is executed, which are indented in relation to `if`. If the condition is false, `condition2` is tested, and if it is true, the instructions denoted by `instructions2` are executed, which can be a simple instruction or several instructions, indented in relation to `elif`. Finally, if `condition2` is false, `instructions3` are executed. Each line of the `if`, `elif`, or `else` statements is followed by a colon. See the simple example below.

```
x = 5
if x > 6:
  print('Earns more than 6 minimum wages.')
  print('You are among the richest 20%!')
```

```
else:
  print('You earn 6 minimum wages or less.')
  print('You represent 80% of the population!')
```

```
You earn 6 minimum wages or less.
You represent 80% of the population!
```

For a probability model we have the following distribution function:

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

For this model, we have the following `script`, in which we decide which part of the program to run, according to the values assigned to `x`.

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

```
0.6400000000000001
```

Repetition structures are `for` and `while/else`. There is also a third type of procedure in `Python` for performing iterations. The general structure of a code for the `for` loop is presented in the `script` below.

```
x = [1,2,3,4,5,6]
for i in x:
  instructions1
else:            # optional else instruction
  instructions2
```

The objects (`x`) of the `for` statement are the iterators or iterable objects, which are those that contain a countable number of values. Lists, tuples, dictionaries, and sets are all iterable objects. We will illustrate with a simple example below, to calculate the sum, product, and mean of a list of values.

```
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('The sum is: ',soma)
print('The product is: ',prod)
print('The mean is: ',media)
```

```
The sum is:  14.899999999999999
The product is:  152.90744999999995
The mean is:  2.9799999999999995
```

To perform iterations on a dictionary, we have the following example:

```
D = {1: 1.3, 2: 3.1, 3: 1.7}
for i in D: # iterate over keys
  print(i, ' and ', D[i])
print('Now iterating over keys and values:')
for (i, valor) in D.items():
  print(i, ' and ', valor)  # iterate over key and value
```

```
1  and  1.3
2  and  3.1
3  and  1.7
Now iterating over keys and values:
1  and  1.3
2  and  3.1
3  and  1.7
```

Finally, an example on a set:

```
A = {1.1,2.2,3.4,4.7,5.3}
for i in A:
  print('element: ',i)
```

```
element:  1.1
element:  2.2
element:  3.4
element:  4.7
element:  5.3
```

We can create a sequence of values with `range(n)` that goes from `0` to `n-1`. We can also use `for` on this sequence, as illustrated in the example below.

```
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('The sum is: ',soma)
print('The mean is: ',media)
```

```
The sum is:  14.899999999999999
The mean is:  2.9799999999999995
```

The `while` is another repetition structure, in which the block of indented instructions will be executed until a condition is met. The general structure is given below.

```
while condition:
  instructions1
else:            # optional else instruction
  instructions2
```

The example below illustrates calculating the total and average of a list.

```
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('The sum is: ',soma)
print('The mean is: ',media)
```

```
The sum is:  14.899999999999999
The mean is:  2.9799999999999995
```

We can use `break` and `continue` within the `while` (or the `for`). `break` is used after a second condition is met in the block of instructions inside the repetition structure and jumps the program's execution to the first line of instruction after the `loop` block, that is, it ends the `loop`. `continue` executes the first line testing the primary condition of the `while` or taking the next iterator value in the `for`, that is, it goes to the beginning of the `loop`. See the example below.

```
y = 35 # try another integer > 1
x = y // 2
while x > 1:
  if y % x == 0:
    print(y, 'has factor ', x)
    break
  x = x - 1
else:
  print(y,' is prime')
```

```
35 has factor  7
```

We can use the `filter` function, as we have already illustrated earlier in this material, to perform iterations in our code. We will not give further details of this for now.

1.8 Functions

Functions in all languages are a powerful programming tool, which allows us to break down a large problem into small tasks (the functions), thus facilitating the resolution of the problem as a whole. We call this the divide-and-conquer strategy. Functions generally receive a variable (argument) and process it according to the rules defined in their instruction block. In this way, the language gains great power, convenience, and elegance. Learning to write useful functions is one of the many ways to make using `Python` comfortable and productive. The general syntax for a function is given by:

```
def name(arg1, arg2,...,argn):
  instructions
```

The *instructions* in the previous `script` mean a block of instructions (indented) and may or may not have the `return object` statement, which can occur in any part of the function or method's instruction block. The function may not have a `return`, if it only modifies a file by saving a new result or if it prints a message when called. Arguments or parameters are passed to the function, and calling it must strictly follow the order in which they appear, unless the call is key-based, that is, of the type `arg1 = 2.3`, for example. In this case, the arguments can be placed in any order when calling the function. The arguments of a function can contain `default` values, that is, in the declaration of the function's name we can have something like: `def name(x, theta = 0.5)`. The argument `theta=0.5` can be omitted when calling the function, and its default value of `0.5` will be assigned. By default, arguments with a `default` must obligatorily come last in the sequence of arguments.

We will present a simple function to test the hypothesis $H_0 : \mu = \mu_0$ from a simple sample of a normal distribution. Two arguments will be used: the data vector (list) x of size n and the hypothetical real value μ_0. The function will calculate the test statistic value t_c using:

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

The resulting function in `Python` is presented below. In this example, a sample of size $n = 8$ was used to obtain the statistic's value to test the hypothesis $H_0 : \mu = 3.0$ (if the sample came from the Na'vi people). We can observe that the final result of the function is the statistic values and p-value, returned through a dictionary-type variable. This function uses in its scope three `Python` functions (basic `NumPy` functions) not yet presented. The first two, `var()` and `mean()`, return the variance and mean of the vector used as an argument, respectively, and the third, `pt()`, returns the cumulative probability of the Student's t distribution for the first argument of the function with ν degrees of freedom, which is its second argument.

```
import scipy as sp # to calculate t probability

def t_test(x, mu0):
  n = len(x)
  s2 = np.var(x,ddof=1) # ddof=1, divisor n-1 for 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) # height of avatars
print('tc = ',t['tc'])
print('p.val = ',t['p.val'])
```

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

We can rewrite this function to include a `default` value for the `mu0` argument. If we chose the value `0` and in some function call this argument was omitted, the hypothesis test $H_0 : \mu = 0$ would be performed by default.

```
import scipy as sp

def t_test(x, mu0 = 0):
  n = len(x)
  s2 = np.var(x,ddof=1) # ddof=1, divisor n-1 for 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) # test 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)]
```

Let's perform a test for correlation in bivariate normal populations. Given a pair of vector variables `X` and `Y` taken on `n` individuals, we have the null hypothesis

$$H_0 : \rho = 0$$

and the test statistic under H_0 given by

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

where r is the sample correlation coefficient between X and Y, and n is the sample size. Under H_0, this statistic follows a Student's t distribution with $\nu = n - 2$ degrees of freedom.

The function should receive the vectors $\mathbf{x}$ and $\mathbf{y}$ and return the result of the test: statistic and p-value. One can use the `cor` function from `SciPy` to obtain the correlation between $\mathbf{x}$ and $\mathbf{y}$.

```
import scipy as sp

def cor_test(x, y):
  n = len(x)
  if n != len(y):
    print('Lists must have the same size!')
    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 =  0.0002923787083539686
r =  0.9997076212916461
```

Finally, let's get a function to obtain real powers of positive definite symmetric square matrices. Consider a symmetric and positive definite matrix $\mathbf{A}$:

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

then the power of order $\alpha \in \mathbb{R}$ is given by

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

It should receive $\mathbf{A}$ and return $\mathbf{A}^\alpha$. The following `script` illustrates a function to obtain these matrix powers. Note that it is not an element-by-element power. We must also note that it is not important that the eigenvalues are ordered. But it is obvious that the eigenvectors associated with each eigenvalue must be properly associated and preserved, and this is done by the `NumPy` library through the `eig()` function.

```
import numpy as np

def mat_power(A, alpha = 0.5):
  e_val, e_vec = np.linalg.eig(A)
  if  any(e_val) < 0:
    print('Matrix is not positive definite!')
    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)
print('A^(1/2): ',mat_power(A)) # square root
A3 = mat_power(A, 1/3) # cubic root
print('A^(1/3): ',A3)
print('A^(1/3)A^(1/3)A^(1/3): ',A3.dot(A3).dot(A3)) # checking
```

```
A =  [[4, 1], [1, 2]]
A^(1/2):  [[1.97773553+0.j 0.29759397+0.j]
 [0.29759397+0.j 1.38254759+0.j]]
A^(1/3):  [[1.57094963+0.j 0.16768037+0.j]
 [0.16768037+0.j 1.23558888+0.j]]
A^(1/3)A^(1/3)A^(1/3):  [[4.+0.j 1.+0.j]
 [1.+0.j 2.+0.j]]
```

1.9 Computational Statistics

Computationally intensive methods have played an increasingly important role in solving problems in different areas of science. We will present algorithms to generate realizations of random variables from various univariate and multivariate probability distributions, to perform matrix operations, to perform inferences using permutation and *bootstrap* methods, etc. Thus, we seek to carry out a division of this material into a basic section and an applied section. Computational techniques are called computational statistics if they are used to perform inferences, to generate realizations of random variables, or to compare statistical methods and techniques.

We will explore methods of generating realizations of random variables of various probability models, to manipulate matrices, to obtain quadratures of distribution functions of various probability models and special functions in statistics, and finally, we will present computationally intensive methods to perform inferences in different real situations. We intend to create algorithms in the `Python` language and later, when they exist, present the instructions for accessing the same algorithms already implemented.

We will present the *bootstrap* and Monte Carlo methods, permutation tests, and the *jackknife* procedure to perform inferences in the most varied real situations. This course has basically two primary intentions. The first is to enable students to perform their own simulations and, the second, to allow them to perform their inferences of interest in situations where using classical inference would be highly complex.

Whether in frequentist inference or Bayesian inference, simulation methods for random numbers of different probability models assume great importance. To use computational statistics more efficiently, a minimum knowledge of simulating random variable realizations is a necessity that should not be ignored. We will place a great emphasis on this subject without neglecting the others. We will present in this material several algorithms developed and adapted for the `Python` language.

Simulating is the art of building models according to Naylor et al. [1971], with the objective of imitating the functioning of a real system, to verify what would happen if alterations were made to its operation

(Dachs [1988]). This type of procedure can have a low cost, avoid losses by not using inadequate procedures, and optimize the decision and functioning of the real system.

Precautions against errors must be taken when we carry out some type of simulation. We can enumerate:

1. inadequate choice of distributions;
2. inadequate simplification of reality; and
3. implementation errors.

We must make the simulated system operate under the conditions of the real system and verify through some tests if the results are in agreement with what is observed in the real system. We call this process validation. Simulation is a technique we use to solve problems. If the achieved solution is faster, with equal or superior efficiency, of lower cost, and easy to interpret compared to any other method, the use of simulation is justifiable.

1.10 Exercises

1. Create in `Python` the vectors $\mathbf{a}^\top = [4, 2, 1, 5]$ and $\mathbf{b}^\top = [6, 3, 8, 9]$ and concatenate them forming a single vector. Obtain the vector $\mathbf{c} = 2\mathbf{a} - \mathbf{b}$ and the scalar $d = \mathbf{b}^\top\mathbf{a}$. Create a sequence whose initial value is equal to 2 and the final value is 30 and whose step is equal to 2. Replicate each value of the sequence 4 times in two different ways (replicated values are aggregated, and the whole sequence replicates without equal values being aggregated).
2. Select the subvector of $\mathbf{x}^\top = [4, 3, 5, 7, 9, 10]$ whose elements are less than or equal to 7.
3. Create the matrix

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

   and determine the eigenvalues and the spectral decomposition of $\mathbf{A}$.
4. Construct a function to verify how many elements of an n-dimensional vector are less than or equal to a real constant k. Use the repetition structures `for` and `while` to perform such a task (each of these structures must be implemented in a different function). Is there a more efficient procedure to generate such a function without using repetition structures? If yes, implement it.
5. Implement a function to perform Student's t-test for two independent samples. Consider the cases of heterogeneous and homogeneous variances. Use a conditional structure to apply the appropriate test, whether the variances are heterogeneous or homogeneous. The decision must be based on a test of homogeneity of variances. To perform this task, implement a specific function assuming normality of random samples.
6. Create a function to obtain the Moore-Penrose inverse of any $n \times m$ matrix, based on the singular value decomposition, function `svd` of `np.linalg.svd`. For this, let there be a matrix $\mathbf{A}$, whose singular value decomposition is $\mathbf{A} = \mathbf{UDV}^\top$, in which $\mathbf{D}$ is the diagonal matrix of the singular values and $\mathbf{U}$ and $\mathbf{V}$ are the corresponding singular vectors. The Moore-Penrose inverse of $\mathbf{A}$ is defined by $\mathbf{A}^+ = \mathbf{VD}^{-1}\mathbf{U}^\top$.

Chapter 2

Uniform Random Variables

In this chapter, we will consider the generation of random numbers for the uniform probability model. From the uniform distribution, we can generate realizations of random variables of any other probability model. The generation of realizations of random variables from the uniform distribution, in general, cannot be performed by machines in a purely algorithmic way, since any sequence produced by traditional *software* is a predictable sequence of pseudo-random numbers. Recently, however, it has been possible to capture thermal noise from processors to generate genuinely random numbers. In `Python`, this is possible through the `os` module, using the `os.urandom()` method.

Two random number generators must produce the same results in their applications. If this does not occur, one of them cannot be considered a good random number generator [Press et al., 1992].

The concepts of uniform numbers and random numbers are often confused. Uniform numbers are those that vary randomly in a real interval of values with constant probability. However, we must differentiate uniform random numbers from other types of random numbers, such as, for example, normal or Gaussian random numbers. These other types generally come from transformations performed on uniform random numbers. Therefore, a reliable source to generate uniform random numbers determines the success of stochastic inference methods and Monte Carlo simulation processes.

2.1 Uniform Random Numbers

Uniform random numbers are those that, in principle, lie within a real interval, generally between 0 and 1, for which we cannot produce a predictable sequence of values and whose density function is constant. In various computer programs, these numbers are generated using the `random` command or similar commands. In Pascal, for example, if this command is used with the argument n, `random(n)`, integer random numbers U in the interval $0 \leq U \leq n-1$ are generated, and if the argument n is not used, the generated numbers are real random values in the interval $[0, 1)$.

In general, programs use the congruential method. Let us consider integer uniform numbers $U_1, U_2, U_3, \ldots$ between 0 and $m-1$, where m represents a large integer. We can generate these numbers using the congruential method through the recursive relation:

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

where m is called the modulus, and a and c are positive integers called the multiplier and increment, respectively. The `mod` operator returns the remainder of the division of the argument $(aU_i + c)$ by m. The recurrent sequence (2.1) repeats in a period that is not greater than m, for obvious reasons. If a, c, and

m are appropriately chosen, the sequence has a maximum length equal to m. The choice of the initial value U_0 determines the sequence. The value of the corresponding uniform number in the interval from 0 to 1 is given by U_{i+1}/m, which is always less than 1, but can be equal to zero.

Let us present a didactic example to illustrate a random number generator. Let $U_0 = a = c = 7$ and $m = 10$, therefore,

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

and so on. We obtain the sequence of random numbers:

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

and we verify that the period is equal to 4, $\{7, 6, 9, 0, \cdots\}$, which is less than $m = 10$.

This method has the disadvantage of being serially correlated. If m, a, or c are not carefully chosen, the correlation can compromise the generated sequence. On the other hand, the method has the advantage of being very fast. We can notice that with each call of the method, only a few calculations are executed. We generally choose the value of m to be the largest integer that can be represented by a 32-bit machine, which is 2^{32}. An example that was used for many years on `IBM mainframe` computers, which represents a terrible choice, is $a = 65{,}539$ and $m = 2^{31}$.

Serial correlation is not the only problem with this method. The higher-order bits are more random than the lower-order (less significant) bits. We should generate integers between 1 and 20 by $j = 1 + \text{int}(20 \times \text{random(seed)})$, instead of using the less accurate method $j = 1 + \mod(\text{int}(1000000 \times \text{random(seed)}),20)$, which uses lower-order bits. There is strong evidence, empirical and theoretical, that the congruential method

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

is as good as the congruential method with $c \neq 0$, if the modulus m and the multiplier a are chosen with care [Press et al., 1992]. Park and Miller [1988] proposed a minimal "standard" generator based on the choices:

$$a = 7^5 = 16{,}807 \quad m = 2^{31} - 1 = 2{,}147{,}483{,}647 \tag{2.3}$$

This random number generator is not perfect, but it passed all the tests to which it was subjected and has been used as a standard to compare and judge other generators. A problem that arises and that we must overcome is that it is not possible to directly implement in a high-level language equation (2.2) with the constants from (2.3), because the product of a and U_i generally exceeds the maximum limit of 32 bits for integers. We can use a trick, due to Schrage [1979], to multiply 32-bit integers and apply the modulo operator, ensuring portability for implementation in virtually all languages and all machines. Schrage's algorithm is based on the factorization of m given by:

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

where $\lfloor z \rfloor$ denotes the integer part of the number z used as an argument. For a number U_i between 1 and $m - 1$ and for a small r, specifically for $r < q$, Schrage [1979] showed that both $a(U_i \mod q)$ and $r\lfloor U_i/q \rfloor$ belong to the interval $0, \cdots, m - 1$ and that

$$aU_i \mod m = \begin{cases} a(U_i \mod q) - r\lfloor U_i/q \rfloor, & \text{if greater than 0} \\ a(U_i \mod q) - r\lfloor U_i/q \rfloor + m, & \text{otherwise.} \end{cases} \tag{2.4}$$

Computationally we observe that the relation:

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

is verified. In Python, one can choose to use the `%` (`mod`) operator, which returns the remainder of the operation between two integers, and the `//` (`div`) operator, which returns the quotient result for integer operations. The quantity $U_i \mod q = U_i - (U_i \text{ div } q) \times q$ can be obtained in Python simply by $U_i \% q$. We assign the result to any variable defined as an integer. To apply Schrage's algorithm to the constants of (2.3) we must use the following values: $q = 127{,}773$ and $r = 2{,}836$.

Next, we present the minimal standard random number generator algorithm:

```
# minimal standard random number generator adapted from Park and
# Miller. Returns uniform random deviates between 0 and 1. Set
# "sem" equal to any integer value to start the sequence;
# "sem" cannot be changed between successive calls of the sequence
# if "sem" is zero or negative, a value dependent on the system clock
# at the time of the call is used as a seed. Constants
# used a = 7^5 = 16.807; m = 2^31 - 1 = 2.147.483.647
# and c = 0

def  gnup(sem, q, a, r, m):
  k = sem // q # integer division
  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 # get system date/time

def gnap(n, sem = 0):
  a = 16807; m = 2147483647
  q = 127773; r = 2836
  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
# Usage examples
n = 5
x = gnap(n,0)
# Formatting the output to 5 decimal places
print(["%0.5f" % v for v in x])
# specifying the seed
y = gnap(n, 1001)
# Formatting the output to 5 decimal places
print(["%0.5f" % v for v in y])
# execution time for each random number
```

```
n = 100000
t1 = datetime.now()
x = gnap(n)
t2 = datetime.now()
t = t2-t1
print('average time (micros): ',t.microseconds / n)
print('total time (micros): ',t.microseconds)
```

```
['0.85358', '0.14503', '0.56190', '0.85843', '0.66910']
['0.00783', '0.66933', '0.36093', '0.10878', '0.30000']
average time (micros):  0.20513
total time (micros):  20513
```

The average time and total time were computed to run `100000` uniform random numbers. To capture the time, the `datetime` package was used, as well as to obtain the time difference between the start and the end of the processing of our function. The block to do this must be executed all at once, that is, by marking the block and pressing enter in `Positron`.

Some considerations regarding this algorithm: a) the `gnap` function (minimal standard random number generator) returns a real number between 0 and 1. This type of specification determines that the variables have double precision. Double precision (`double-precision float`) has numbers in the range from $\pm 2.225 \times 10^{-308}$ to $\pm 1.798 \times 10^{308}$, occupies 24 bytes of memory, and has $15 - 17$ significant digits; b) the seed value is defined by the user and is passed as a parameter to the function. This means that the program variable that calls the function and is passed as a seed must be updated with the new modified value in `gnup`. If its initial value is zero or negative, the function assigns an integer dependent on the system time at the time of the call; c) the function has a dependency on the `datetime` package, which was used to capture the system time and day.

The routine is initialized with the values of n and the seed provided by the user. If the seed is null or negative, we assign a new value dependent on the system clock at the time of the call. From this point on, the program must repeatedly call the `gnap` function, which returns the value of the random number between 0 and 1 using the algorithm described above, until the sequence requested by the user is completed. In successive calls of this function, the seed value is always equal to the value of the last step.

The period of `gnap` is on the order of $2^{31} \approx 2.15 \times 10^9$, that is, the complete sequence is slightly over 2 billion random numbers. Thus, we can use `gnap` for a few practical purposes. As we have already pointed out, the minimal standard random number generator has two basic limitations, namely, short sequence and serial correlation.

Alternatively, since Python is capable of dealing with integers of arbitrary size without the limitations of 32 or 64 bits and without overflow, we can implement the standard random number generator without Schrage [1979]'s trick. In R, it is also possible to do something similar using coercion from double to integers. We should be concerned about using Schrage [1979]'s trick in languages where integers are 32 bits or to ensure code portability to any operating system or programming language. The simplified code is presented below, whose function was named `gna_ws` and whose arguments are the necessary constants and whose default values are the same as those previously used. The dependency on the `datetime` package remains in this version.

```
def gna_ws(n, sem = 0, m=2**31 - 1, a=7**5, c=0,  unit=True):
    """
    Linear congruential random number generator

    Parameters:
    n: number of values to be generated
    sem: initial seed
```

```
    m: modulus
    a: multiplier
    c: increment
    unit: if True, returns values between 0 and 1

    Returns:
    List with the generated values
    """
    x = [0] * (n + 1)
    if sem <= 0:
       t = datetime.now()
       sem = t.second+t.minute*60+t.hour*3600+t.day*86400
    x[0] = sem

    for i in range(1, n + 1):
        x[i] = (a * x[i-1] + c) % m

    if unit:
        return [xi / m for xi in x[1:]]
    return x[1:]
# Using the default values (unit=True)
resultado = gna_ws(n=4, sem=2**31)
print(resultado)
# Or explicitly with unit=False
resultado = gna_ws(n=4, sem=2**31, unit=False)
print(resultado)
```

```
[7.826369259425611e-06, 0.13153778814316625, 0.7556053221950332, 0.4586501319234493]
[16807, 282475249, 1622650073, 984943658]
```

As there are methods to eliminate serial correlation and that increase the period of the sequence, we recommend that they be adopted. Of course, the function presented was intended to illustrate how we can program our own functions to generate uniform random numbers. Python, however, has its own uniform number generator, which we will see next. One of the best and most interesting random number generators is the `Mersenne Twister` (MT). Mersenne Twister is a pseudo-random number generator developed by Makoto Matsumoto and Takuji Nishimura in the years 1996 and 1997 [Matsumoto and Nishimura, 1998]. The MT has the following merits according to its developers:

- it was developed to eliminate the flaws of different existing generators;
- it has the advantage of presenting the longest period and highest order of equidistribution of any other implemented method. It provides a period that is on the order of $2^{19.937} - 1 \approx 4.3154 \times 10^{6001}$, and a 623-dimensional equidistribution;
- it is one of the fastest existing generators, although complex;
- it makes very efficient use of memory.

There are many implemented versions of this algorithm, including in Fortran and C, which are available on the internet. Fortunately, Python already has this algorithm implemented. As it is a more advanced topic, which goes beyond what we intend to present in these lecture notes, we will not describe this type of procedure for incorporating functions written in other languages.

2.2 Uniform Random Numbers in Python

In Python, we can generate continuous uniform random numbers using a pre-programmed function. Uniform random numbers are generated by the command `random.uniform(low=0.0, high=1.0, size=None)` from the `numpy` library, where `None` is to generate just one value, and can be replaced by n, between `low` and `high` (excluded), which are arguments that delimit the minimum and maximum value of the sequence to be generated. Seed control to generate a reproducible sequence of uniform numbers is given by the command `random.seed(seed=None)` from `numpy`, where the `seed` argument must be an integer. Python automatically determines a new seed with each call. We can generate different sequences in each call of the generator command, without worrying about the random seed. The uniform random number generator uses the Mersenne-Twister algorithm by default.

In the program presented below, we illustrate how we can generate n uniform random numbers between 0 and 1 in a compact, simple, and efficient way:

```
import numpy as np
n = 5
x = np.random.uniform(low=0.0, high=1.0, size=n)
print(x)
# Fixing the seed
np.random.seed(seed=1000)
print('Sample: \n', np.random.uniform(low=0.0, high=1.0, size=n))
np.random.seed(seed=1000)
# (same seed)
print('Same as previous: \n', np.random.uniform(low=0.0, high=1.0, size=n))
```

```
[0.01963132 0.29288156 0.00278116 0.48763263 0.51003809]
Sample:
 [0.65358959 0.11500694 0.95028286 0.4821914  0.87247454]
Same as previous:
 [0.65358959 0.11500694 0.95028286 0.4821914  0.87247454]
```

We made a program to compare the execution time of the `gnap` and `random.uniform()` functions and we return the average time for each realization of the random variable. This way, we verify that the `random.uniform` algorithm is the fastest of all, according to the relative values presented below. Obviously, we have to consider that the `numpy` algorithm is a compiled function. The `gnap` algorithm, in turn, was implemented in Python, which uses an interpreted language. The time comparisons can be seen in the following program. The program used was:

```
# execution time for each random number
# comparison between our generator and the np one
import numpy as np

n = 1000000
t1 = datetime.now()
x = gnap(n)
t2 = datetime.now()
tgp = t2-t1
t1 = datetime.now()
x = gna_ws(n, sem=27589)
t2 = datetime.now()
tgna_ws = t2- t1
t1 = datetime.now()
x = np.random.uniform(low=0.0, high=1.0, size=n)
t2 = datetime.now()
```

```
tnp = t2-t1
print('average time gnap: ',tgp.microseconds / n)
print('average time gna_ws: ',tgna_ws.microseconds / n)
print('average time numpy: ',tnp.microseconds / n)
# relative time
print('Time ratio - gnap / ws:', tgp / tgna_ws)
print('Time ratio - gnap / np:', tgp / tnp)
print('Time ratio - ws / np ', tgna_ws / tnp)
```

```
average time gnap:  0.216253
average time gna_ws:  0.166641
average time numpy:  0.010782
Time ratio - gnap / ws: 1.2977178485486764
Time ratio - gnap / np: 20.056854015952513
Time ratio - ws / np  15.455481357818586
```

We can make a histogram using the program below. For this, we use the `matplotlib` library.

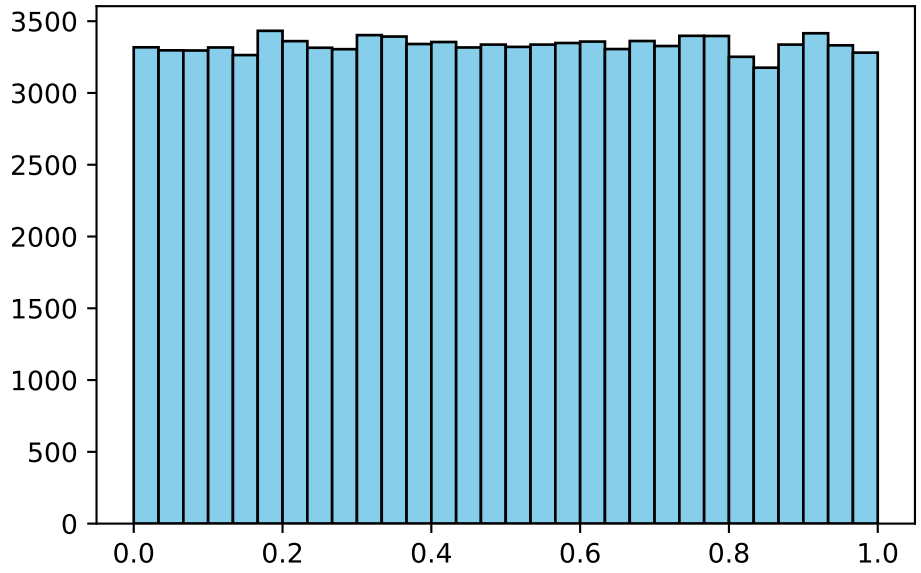
> ⚠ Note about graphs in the browser
>
> In some browsers (especially **Firefox**), displaying graphs with `plt.show()` in the interactive environment of this page may fail. If the graph does not appear, open the material in **Google Chrome**, **Microsoft Edge**, or **Safari**.

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

n = 100000
x = np.random.uniform(low=0.0, high=1.0, size=n) # gnap(n) change to test
# Creates an isolated space for the figure
fig, ax = plt.subplots()

# Draws the histogram
_ = ax.hist(x, bins=30, edgecolor='black', color='skyblue')

# Displays the graph natively
plt.show()
```



2.3 Genuinely Random Numbers

Using `Python`, we just need to use the `os` module. The `os.urandom(k)` function makes a system call (like `getrandom` in Linux or `BCryptGenRandom` in Windows) that returns true and uniform random bytes coming from the OS hardware ecosystem. The returned result contains `k` random bytes of uniform entropy from the processor/operating system. The `int.from_bytes()` method receives this generated value as an argument and transforms it into an $8k$-bit integer. If we divide this by the total number of combinations of the $8k$-bit representation, which is 2^{8k}, we will have a uniform random number in the interval $[0, 1)$.

Since each bit generated by the hardware has exactly a 50% chance of being 0 or 1 (independently of the previous bits), all bit combinations from 0 to $2^{8k} - 1$ are equally likely. By dividing by the maximum value, we map this discrete uniform distribution to the continuous interval $[0, 1)$ without distorting the distribution's probability. The option `byteorder='big'` in `int.from_bytes(dados_fisicos, byteorder='big')` makes `Python` read the most significant bits first, that is, from left to right in the value of the `dados_fisicos` variable that was generated. This option value in the context of generating continuous uniform numbers does not affect randomness, but was used because it is the natural logic of building integers in bits.

The `script` below illustrates the generation of a uniform random sample of size $n = 5$.

```
import os

def gerar_uniforme_verdadeiro(n_amostras=1, bytes_por_numero=8):
    # We use 8 bytes (64 bits) per number by default, which
    # already guarantees extremely high decimal precision
    amostra = []
    max_valor = 2 ** (bytes_por_numero * 8)

    for _ in range(n_amostras):
        # Pulls pure entropy from the hardware/OS
        dados_fisicos = os.urandom(bytes_por_numero)
        # Transforms into integer
        inteiro = int.from_bytes(dados_fisicos, byteorder='big')
```

```
        # Normalizes to the interval [0, 1)
        amostra.append(inteiro / max_valor)

    return amostra if n_amostras > 1 else amostra[0]

# Usage example: generating a sample of size 4 independent from U(0,1)
amostra = gerar_uniforme_verdadeiro(4)
print(amostra)
```

```
[0.12927058119103252, 0.3266227585332542, 0.6796226419774016, 0.7107338597017021]
```

Speed is the main point where the hardware generator and the `NumPy` generator drastically diverge. The hardware procedure (`os.urandom`) is much slower than `np.random.uniform`. We performed a comparative test identical to the previous one, in which we observed that `NumPy` can be from 10 to more than 100 times faster, depending on the sample size.

```
from datetime import datetime
import numpy as np

n_teste = 500000

# 1. Measuring the time of the True Hardware Generator
t1 = datetime.now()
x_hw = gerar_uniforme_verdadeiro(n_amostras=n_teste, bytes_por_numero=8)
t2 = datetime.now()
t_hardware = t2 - t1

# 2. Measuring the time of NumPy (Pseudo-random)
t1 = datetime.now()
x_np = np.random.uniform(low=0.0, high=1.0, size=n_teste)
t2 = datetime.now()
t_numpy = t2 - t1

print(f"Total time Hardware (50k samples): \n"
      f"{t_hardware.total_seconds():.5f} seconds")

print(f"Total time NumPy    (50k samples): "
      f"{t_numpy.total_seconds():.5f} seconds")

print(f"NumPy was about {t_hardware.total_seconds() / \
      t_numpy.total_seconds():.1f} times faster!")
```

```
Total time Hardware (50k samples): 0.13055 seconds
Total time NumPy    (50k samples): 0.00272 seconds
NumPy was about 48.0 times faster!
```

2.4 Comparative Performance Analysis

With the execution results of the different approaches presented in this chapter, we can build a computational overview of the efficiency of each method. The following table consolidates the estimated average time for generating each uniform random number in `Python`:

Generation Method	Generator Type	Average Time per Number	Relative Speed (vs. NumPy)
`np.random.uniform`	Pseudo-random (Vectorized in C)	~0.008 µs	**1.0x** (Speed Reference)
`gna_ws`	Pseudo-random (Congruential without Schrage)	~0.171 µs	~22 times slower
`gnap`	Pseudo-random (Minimal standard with Schrage)	~0.232 µs	~30 times slower
`os.urandom`	Genuinely Random (Via Hardware)	~0.304 µs	~39 times slower

This empirical comparison brings two fundamental lessons for computational statistics:

1. **The Cost of the Operating System Boundary:** The hardware procedure (`os.urandom`) proved to be the slowest of all. This occurs because `Python` needs to interrupt its flow and make a system call (`syscall`) so that the operating system kernel consults the hardware ecosystem. This process, added to the lack of vectorization (use of the `for` loop), creates a computational bottleneck.
2. **The Efficiency of NumPy:** The `numpy` library is extremely fast not only because it uses deterministic mathematical formulas (pseudo-random algorithm), but also because it delegates the creation of the numbers to optimized memory blocks compiled in the `C` language, operating on the nanosecond scale.

In short, the hardware generator is indispensable for information security, cryptography, and digital key generation purposes, scenarios in which unpredictability must be absolute. However, for heavy stochastic simulations, such as Monte Carlo methods that demand millions or billions of realizations quickly, the `np.random.uniform` command remains the correct choice due to its speed and efficiency.

2.5 Exercises

1. Use the `gnap` generator to generate n realizations of an exponential distribution $f(x) = \lambda e^{-\lambda x}$. We know from the probability integral transform theorem that if U has a uniform distribution, $X = F^{-1}(U)$ has a probability distribution with density $f(x) = F'(x)$; where $F(x) = \int_{-\infty}^{x} f(t)dt$ is the distribution function of X and $F^{-1}(y)$ is its inverse function for the value y. For the exponential, the probability distribution function is: $F(x) = \int_0^x \lambda e^{-\lambda t}dt = 1 - e^{-\lambda x}$. To obtain the inverse function we have to equate u to $F(x)$ and solve for x. Thus, $u = 1 - e^{-\lambda x}$ and solving for x we have: $x = -\ln(1-u)/\lambda$. Due to the symmetry of the uniform distribution, $1 - u$ can be replaced by u. The final result is: $x = -\ln(u)/\lambda$. To generate exponential numbers, simply generate uniform numbers and apply the relation $x = -\ln(u)/\lambda$. Do this to build a function that generates n exponential realizations. Apply the function to obtain random samples from the exponential of size $n = 100$ and obtain the histogram of the simulated sample. Calculate the mean and the variance and compare them with the theoretical values of the exponential distribution.

2. To generate numbers from a normal distribution, whose density is given by $f(x) = 1/(\sqrt{2\pi\sigma^2})\exp\{-(x-\mu)^2/(2\sigma^2)\}$, what would be the difficulty in being able to use the theorem announced in the previous proposed exercise?

3. How could we adapt the algorithm presented in this chapter to generate uniform random numbers using the values proposed by Park and Miller, that is, $a = 48.271$ and $m = 2^{31} - 1$? Implement the algorithm, taking care regarding the new multiplier q and remainder r of the factorization of m?

4. How could you propose a simple statistical test to evaluate the randomness of the sequence of uniform numbers generated by these algorithms presented in the chapter? Implement your idea.

Chapter 3

Non-Uniform Random Variables

In this chapter, we present general methods for generating realizations of random variables from other probability distributions, such as exponential, normal, and binomial models. We will implement several functions in Python and conclude by presenting optimized, ready-to-use routines available in standard libraries.

3.1 Introduction

From this point forward, we will study one of the primary methods—governed by the fundamental law of probability transformation—for generating data from continuous or discrete probability distributions. For specific cases, we will illustrate alternative procedures that are efficient and computationally simpler. This transformation relies on the continuous uniform distribution $(0, 1)$ as its foundational model. For this reason, generating continuous uniform numbers is of paramount importance.

Later in these lecture notes, we will examine algorithms to numerically obtain the cumulative distribution function $F(x)$ and its inverse function $x = F^{-1}(p)$, where p belongs to the interval from 0 to 1. This knowledge is essential for applying this primary method.

In this chapter, we limit ourselves to presenting the theory for a few probability models for which we can easily obtain the cumulative distribution function and its inverse analytically. For more complex models, although the method based on the fundamental law of probability transformation is conceptually appropriate, we present only alternative methods, as it is generally inefficient in terms of the computation time required to generate each realization of the random variable. This inefficiency stems from having to compute the inverse cumulative distribution function numerically for complex probability models.

3.2 General Methods for Generating Realizations of Random Variables

We can obtain realizations of random variables from any probability distribution starting from uniform random numbers. To this end, an important theorem can be utilized: the fundamental probability integral transform theorem.

Theorem 3.1 (Fundamental Probability Integral Transform Theorem)**.** *Let U be a uniform random variable $U(0, 1)$ and X be a random variable with density f and a continuous, invertible cumulative distribution function F. Then $X = F^{-1}(U)$ has density f, where F^{-1} is the inverse function of the distribution function F.*

Proof: Let X be a random variable with distribution function F and density function f. If $u = F(x)$, then the Jacobian of the transformation is $du/dx = F'(x) = f(x)$, where U is a uniform random variable $U(0,1)$ with density function $g(u) = 1$ for $0 < u < 1$ and $g(u) = 0$ elsewhere. Thus, the random variable $X = F^{-1}(U)$ has density f given by:

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

In other words, the random variable $X = F^{-1}(U)$ has density function $f_X(x)$, establishing the desired result and completing the proof. ■

For discrete random variables, the theorem must be modified to accommodate step-function cumulative distribution functions F, as is characteristic of discrete probability distributions.

In Figure 3.1, we illustrate how to generate a realization of a random variable X with density f and cumulative distribution function F. Thus, it suffices to generate a uniform random number u_0 and invert the distribution function F at that point. Computationally, the difficulty lies in analytically deriving an expression for F^{-1} for many probability models. In general, closed-form expressions do not exist, and numerical methods are required to invert the distribution function. In this chapter, we present this method for the exponential distribution.

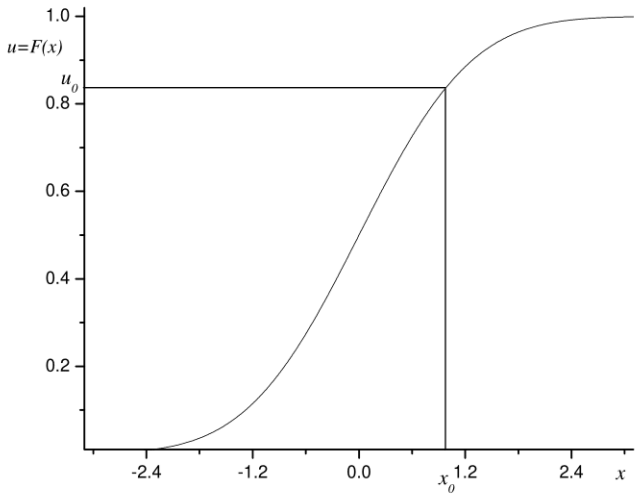


Figure 3.1: Illustration of the fundamental probability integral transform theorem for generating a random variable X with density $f(x) = F'(x)$. Starting from a uniform random number u_0, the distribution function is inverted at that point to obtain x_0, with density $f(x)$.

Another very general method that we will use is the rejection sampling method. This method has strong geometric appeal. We will first describe it in a broad, general manner. Later, we will apply it to generate random variables from specific probability models. The main advantage of this method is that it does not require computing the cumulative distribution function nor its inverse. These strategies can only be applied to many existing probability models if iterative numerical methods are used. Let $f(x)$ be the probability density function from which we want to generate a random sample. The area under the curve over any interval of x corresponds to the probability of generating a value x in that interval. If we could generate a two-dimensional point, say (X,Y), uniformly distributed over the area under the curve, then the X-coordinate would follow the target distribution.

To efficiently generate realizations of random variables with density $f(x)$ while avoiding the aforementioned numerical complications, we can define an arbitrary function $g(x)$. This function must satisfy specific properties: it must have finite area and, for all x, its values $g(x)$ must be greater than or equal to $f(x)$. This function is called a majorizing function (or comparison function). Another important property of $g(x)$ is that its distribution function $G(x)$ must be analytically computable and invertible, i.e., $x = G^{-1}(u)$. Since $g(x)$ is not necessarily a valid density function, let us denote the total area under this curve over the domain of interest as $A = \int_{-\infty}^{\infty} g(x)dx$. Since G^{-1} is known, we can easily generate uniform points (x,y) lying under the curve $g(x)$. To do so, we generate a uniform random variable u_1 between 0 and A and apply Theorem 3.1. This yields the first coordinate $x_0 = G^{-1}(u_1)$. To generate the second coordinate, we cannot simply draw a uniform variable on $[0, A]$, as that might produce a point outside the curve $g(x)$. Instead, we evaluate g at x_0, obtaining $g(x_0)$, and draw $y_0 = u_2$, where u_2 is a uniform random variable on $[0, g(x_0)]$. This yields a point (x_0,y_0) uniformly distributed under the curve $g(x)$. The challenge of this method lies precisely in defining a majorizing function $g(x)$ that satisfies these required properties.

Now, let us plot the curves corresponding to $g(x)$ and $f(x)$ on the same graph. If the uniform point (x_0,y_0) falls within the area under $f(x)$—that is, if $y_0 \leq f(x_0)$—we accept x_0 as a valid realization from $f(x)$. If, on the other hand, the point lies in the region between $f(x)$ and $g(x)$—that is, if $f(x_0) < y_0 \leq g(x_0)$—we reject x_0. An alternative way to express this criterion is to draw y_0 from a $U(0, 1)$ distribution and accept or reject x_0 depending on whether $y_0 \leq f(x_0)/g(x_0)$ or $y_0 > f(x_0)/g(x_0)$, respectively. We illustrate this method in Figure 3.2, where A represents the total area under $g(x)$.

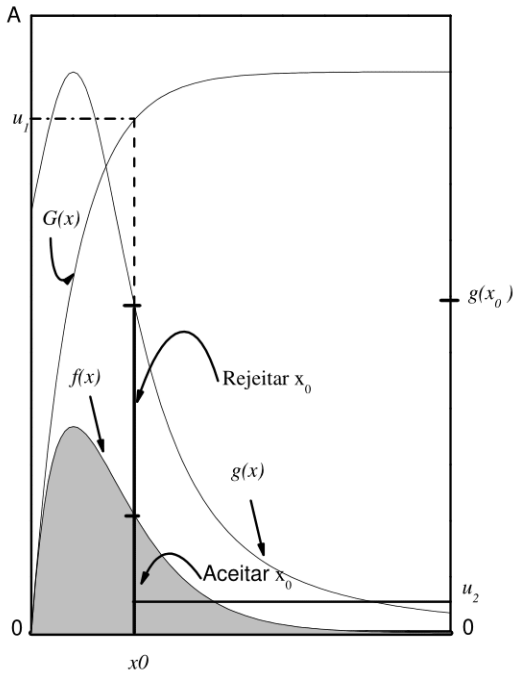


Figure 3.2: Rejection method for generating a value x_0 of a random variable X with density function $f(x)$ that is majorized by $g(x)$ for all x. In this illustration, x_0 should be accepted.

Let us illustrate the first method and note that the second method is the one most commonly encountered in random variable generation. The exponential distribution is a probability distribution to which we can easily apply Theorem 3.1 to generate random samples. We choose to start the random variable generation process with this distribution because its distribution function and inverse are straightforward to obtain analytically. Let X be a random variable whose exponential density is given by:

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

where $\lambda > 0$ is the rate parameter of the exponential distribution and $x > 0$.

The exponential cumulative distribution function is given by:

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

The inverse cumulative distribution function $x = F^{-1}(u)$ is given by:

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

where u is a uniform random number in $(0, 1)$.

Since the uniform distribution is symmetric on (0,1), we can replace $1 - u$ in equation (3.3) with u. Thus, to generate a realization of an exponential random variable X from a uniform random variable, we apply Theorem 3.1 via the expression:

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

The Python algorithm to generate realizations of exponential random variables is:

```
# Program demonstrating the generation of n exponential random
# variables with parameter lamb, using np.random.uniform()
# to generate uniform random numbers
import numpy as np

def rexpon(n, lamb = 1.0):
    u = np.random.uniform(0.0, 1.0, n) # generate vector u ~ U(0, 1)
    x = -np.log(u) / lamb  # generate vector x with exponential dist.
    return x               # return vector x

# Example
rexpon(5, 0.1)
```

```
array([2.08436254, 9.41345445, 6.33657011, 4.32181471, 9.91258879])
```

We can use Python's built-in function (`np.random.exponential(scale=1.0, size=None)`) to accomplish the same task. Simply typing `np.random.exponential(scale=10, size=5)` produces a random sample of size n from an exponential distribution with parameter $\lambda = 1/\text{scale}$. Python makes computational statistics straightforward and accessible to researchers.

3.3 Random Variables from Important Distributions

In this section, we describe specific methods for generating realizations of several key random variables, with emphasis on the normal distribution. Although the general method presented in Section 3.2 can also be applied to the normal distribution, we focus on alternative procedures. To use the inversion method, we would need a function similar to `rexpon`, say `rnormal`. In place of `x = -log(u)/lamb`, we would write $x = \text{invnorm}(u) \times \sigma + \mu$, where `invnorm(u)` is the inverse cumulative distribution function of the standard normal distribution. The standard normal distribution within the normal family—defined by parameters

μ and σ^2—is the one with zero mean and unit variance, denoted by $N(0,1)$. This inverse function would need to be written externally by the user. The parameters μ and σ passed to the function are the mean and standard deviation of the target normal distribution. The normal probability density function is:

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

We will not present additional functions relying directly on Theorem 3.1, as one can easily adapt functions like `rexpon` given an efficient algorithm for inverting the cumulative distribution function of the target model. The drawback of this approach is the substantial computational effort required for most densities, which can lead to high execution times and algorithmic inefficiency.

Furthermore, we can leverage relationships between probability distributions to generate realizations from other distributions. For example, if X is normal with density (3.5), $N(\mu,\sigma^2)$, we can generate realizations of $Y = e^X$. Under this transformation, Y follows a log-normal distribution with location parameter α (μ) and scale parameter β (σ), whose density function is:

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

An important algorithm used to generate normal variates is the Box-Muller method, which is based on generalizing the variable transformation technique to higher dimensions. To present this method, consider p random variables $X_1, X_2, \ldots, X_p$ with joint density function $f(x_1, x_2, \ldots, x_p)$ and p variables $Y_1, Y_2, \ldots, Y_p$ that are functions of all X's. The joint probability density function of the Y's is:

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

where $J = |\partial()/\partial()|$ is the Jacobian of the transformation from the X's to the Y's.

Let us apply transformation (3.7) to generate normal data with density given by (3.5). For the Box-Muller transformation, consider two uniform random variables on $[0,1]$, denoted X_1 and X_2, and two functions Y_1 and Y_2 defined by:

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

Solving for X_1 and X_2 in (3.8), we obtain:

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

Using the derivative rule $\frac{d}{dx}[k\arctan(g)] = \frac{kg'}{1+g^2}$, where g is a function of x, the Jacobian matrix of the transformation is:

$$\begin{vmatrix} \frac{\partial x_1}{\partial y_1} & \frac{\partial x_1}{\partial y_2} \\ \frac{\partial x_2}{\partial y_1} & \frac{\partial x_2}{\partial y_2} \end{vmatrix} = \begin{vmatrix} -y_1 e^{-0.5(y_1^2+y_2^2)} & -y_2 e^{-0.5(y_1^2+y_2^2)} \\ -\frac{y_2}{2\pi y_1^2\left(1+\frac{y_2^2}{y_1^2}\right)} & \frac{1}{2\pi y_1\left(1+\frac{y_2^2}{y_1^2}\right)} \end{vmatrix} = -\frac{1}{2\pi} e^{-0.5(y_1^2+y_2^2)} \tag{3.10}$$

Thus, the joint density function of Y_1 and Y_2 is given by $f(x_1,x_2)|J|$, which evaluates to:

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

Since the joint density of Y_1 and Y_2 is the product of two independent normal densities, Y_1 and Y_2 are independent standard normal random variables, as shown in Johnson and Wichern [1998] and Ferreira [2018].

Thus, we can use this result to generate normal random variables. The main drawback is computational: evaluating trigonometric functions such as sine and cosine can slow down the algorithm. A clever modification proposed by Marsaglia and Bray [1964] (and detailed in Press et al. [1992]) avoids direct evaluation of trigonometric functions. This variant is known as the Polar Box-Muller method.

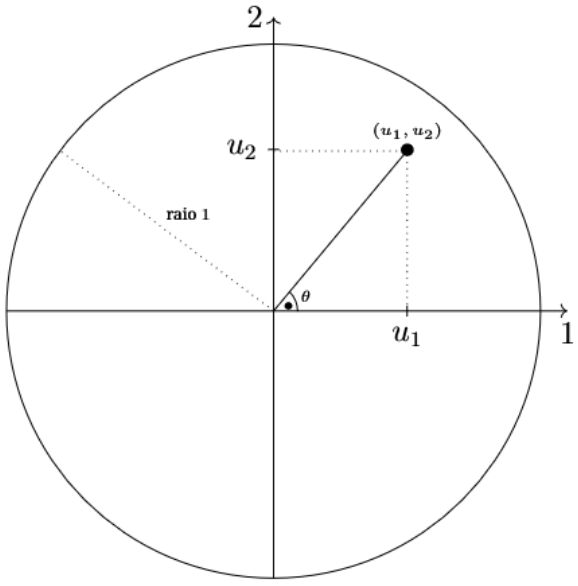
Instead of selecting uniform random variables x_1 and x_2 from a unit square, we sample points (u_1, u_2) uniformly distributed inside a unit circle (radius 1). The sum of their squares, $R^2 = U_1^2 + U_2^2$, is a uniform random variable that can serve as X_1, while the angle Θ formed by (u_1, u_2) relative to the horizontal axis serves as the random angle $\Theta = 2\pi X_2$. The key advantage of avoiding expression (3.8) directly is that cosine and sine can be obtained algebraically: $\cos(2\pi x_2) = u_1/\sqrt{r^2}$ and $\sin(2\pi x_2) = u_2/\sqrt{r^2}$. This eliminates trigonometric function calls. In Figure 3.3, we illustrate these concepts, denoting the angle between (u_1,u_2) and the u_1-axis as θ.

Now we present the `boxmuller` function to generate normal random variables using the Polar Box-Muller algorithm. This function relies on an inner helper `polar` to generate two independent standard normal variates Y_1 and Y_2:

```
# Function boxmuller returns a sample of size n from a normal
# distribution with mean mu and variance sigma^2 using the
# Polar Box-Muller method
import numpy as np
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

def boxmuller(n, mu = 0, sigma = 1):
    if n % 2 == 0: # n is even
        k = n // 2
        for i in range(k):
            if i == 0:
                x = polar()
            else:
                x = np.append(x, polar())
    else: # n is odd
```



$$R^2 = U_1^2 + U_2^2 = X_1$$

$$\Theta = 2\pi X_2$$

$$\sin(\Theta) = \sin(2\pi X_2) = \frac{U_2}{\sqrt{R^2}}$$

$$\cos(\Theta) = \cos(2\pi X_2) = \frac{U_1}{\sqrt{R^2}}$$

$$y_1 = \sqrt{-2\ln(x_1)}\cos(2\pi x_2)$$

$$y_2 = \sqrt{-2\ln(x_1)}\sin(2\pi x_2)$$

$$y_1 = \sqrt{\frac{-2\ln(r^2)}{r^2}}u_1$$

$$y_2 = \sqrt{\frac{-2\ln(r^2)}{r^2}}u_2$$

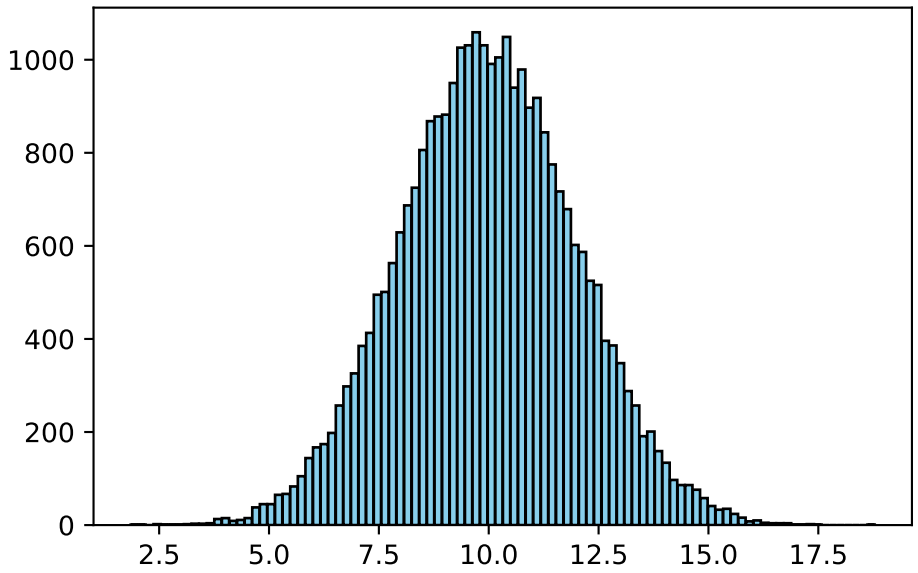
Figure 3.3: Unit circle showing a random point (u_1, u_2) with $r^2 = u_1^2 + u_2^2$ representing x_1 and θ representing the angle formed by (u_1, u_2) relative to the horizontal axis. In this illustration, the point lies inside the unit circle as required.

```
        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)
# Create an isolated space for the figure
fig, ax = plt.subplots()

# Draw the histogram
_ = ax.hist(x, bins='auto', rwidth=0.95, edgecolor='black', color='skyblue')

# Display the plot natively
plt.show()
```



Several approximations are outlined in Atkinson and Pearce [1976] and summarized below. One approximation relies on the Tukey-lambda distribution, fitting a normal curve by matching the first four moments. In addition to being an approximation, this method suffers from the computational expense of exponentiation. Using this approximation, a normal variable X can be generated from a uniform variable $U \sim U(0,1)$ via:

$$X = [U^{0.135} - (1-U)^{0.135}]/0.1975.$$

Another approximation is based on summing 12 or more independent uniform variables ($U_i \sim U(0,1)$). By the Central Limit Theorem, the variable $X = \sum_{i=1}^{12} U_i - 6$ is approximately standard normal $N(0,1)$, because each of the 12 uniform variables has mean $1/2$ and variance $1/12$.

These two approximations have primarily pedagogical value and are not recommended for production applications requiring high precision. In practice, relying on coarse approximations can yield inaccurate or misleading conclusions.

In general, summing k uniform variables yields:

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

since $E(U_i) = (b-a)/2$ and $V(U_i) = (b-a)^2/12$, where k is the number of uniform variables summed per normal realization. Below, we implement both approximations in Python as instructional exercises.

The first is the Tukey-Lambda approximation:

```
import numpy as np

# Approximation based on 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)
print('Mean: ', np.mean(x))
print('Standard deviation: ', np.std(x))
```

```
Mean:  9.991585920078752
Standard deviation:  10.001517489394036
```

The second is the sum of k uniform variables via the Central Limit Theorem:

```
import numpy as np

# Approximation based on the sum of k uniform variables
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)
print('Mean: ', np.mean(x))
print('Standard deviation: ', np.std(x))
```

```
Mean:  9.996139133239083
Standard deviation:  9.977244035992804
```

We can also implement a vectorized version of this CLT approximation (analogous to using `apply` in `R`), as shown in the script below:

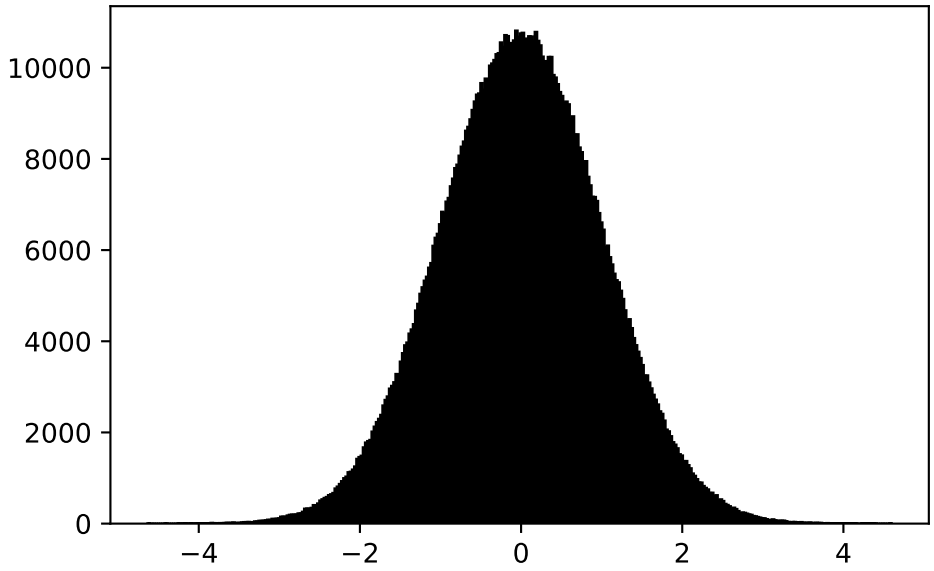
```
import numpy as np
import matplotlib.pyplot as plt

def rnorm_unif(n, k=12):
    u = np.random.uniform(size=(n, k))
    if k == 12:
        x = u.sum(axis=1) - 6
    else:
        x = (u.sum(axis=1) - k / 2) / np.sqrt(k / 12)
    return x

n = 1000000
k = 36
x = rnorm_unif(n, k)
# Create an isolated space for the figure
fig, ax = plt.subplots()

# Draw the histogram
_ = ax.hist(x, bins='auto', rwidth=0.95, edgecolor='black', color='skyblue')

# Display the plot natively
plt.show()
```



3.4 Binomial Distribution

We now move to discrete distributions, illustrating random variable generation with the binomial model. The binomial distribution plays a central role in applied statistics across diverse empirical contexts. For instance, the non-parametric sign test relies on the binomial distribution, and the count of diseased animals in a random sample of size n is frequently modeled binomially. A random variable X following a binomial distribution has probability mass function given by:

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

where the parameters n and p denote the number of trials and the success probability per trial, respectively. The term $\binom{n}{x}$ is the binomial coefficient:

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

The probability of success p must remain constant across the n independent Bernoulli trials. The parameter n is specified by the researcher as an integer $n \geq 1$. When $n = 1$, the binomial distribution reduces to a Bernoulli distribution. Thus, a binomial variable represents the total number of successes in n independent, identically distributed Bernoulli trials. The following theorems and lemmas underpin the generation algorithms presented below.

Theorem 3.2 (Genesis)**.** *Let X be the number of successes in a sequence of n Bernoulli trials with success probability p, i.e.,*

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

where $U_1, U_2, \ldots, U_n$ are i.i.d. uniform random variables $U(0,1)$ and $I(\bullet)$ is an indicator function. Then X follows a binomial distribution (n, p).

Lemma 3.1 (Sum of Binomials)**.** *If $X_1, X_2, \ldots, X_k$ are independent binomial random variables with parameters $(n_1, p), \ldots, (n_k, p)$, then $\sum_{i=1}^{k} X_i$ follows a binomial distribution with parameters $(\sum_{i=1}^{k} n_i, p)$.*

Lemma 3.2 (Waiting Time - Property 1)**.** *Let $G_1, G_2, \ldots$ be independent geometric random variables, and let X be the smallest integer such that*

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

Then X follows a binomial distribution (n, p).

The geometric variables in Lemma 3.2 are defined as follows: if G follows a geometric distribution with constant success probability $p \in (0, 1)$, its probability mass function is:

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

The geometric distribution models the waiting time (trial number) until the first success in a sequence of independent Bernoulli trials. It assumes $g - 1$ failures (each with probability $1 - p$) preceding a success on the g-th trial (with probability p). The second waiting time lemma is stated below:

Lemma 3.3 (Waiting Time - Property 2)**.** *Let $E_1, E_2, \ldots$ be i.i.d. exponential random variables, and let X be the smallest integer such that*

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

Then X follows a binomial distribution (n, p).

These theoretical properties form the foundation for several binomial sampling algorithms. The most basic binomial algorithm generates n independent $U(0, 1)$ variables and counts how many are $\leq p$. This algorithm, designated **BU** by Kachitvichyanukul and Schmeiser [1988], proceeds as follows:

1. Set $x = 0$ and $k = 0$;
2. Generate $u \sim U(0, 1)$ and set $k = k + 1$;
3. If $u \leq p$, set $x = x + 1$;
4. If $k < n$, return to step 2;
5. Return x as a draw from a binomial (n, p).

The BU algorithm has execution speed proportional to n and depends on the speed of the underlying uniform generator, but it requires no setup variables. Its Python implementation is:

```
import numpy as np

# Demonstration of algorithms for generating realizations of
# binomial random variables binom(size, p).
# BU Algorithm
def bu(size, p):
    x = 0
    k = 0
    while k < size:
        u = np.random.uniform(0.0, 1.0, 1)
        k = k + 1
        if u <= p:
            x += 1
    return x
```

```
# Example
prob = 0.5
size = 3
print('bu: ', bu(size, prob))
```

```
bu:  2
```

A second algorithm, developed by Devroye (1980) [Devroye, 1980] and named **BG**, is based on Lemma 3.2. The BG algorithm works as follows:

1. Set $y = 0$, $x = 0$, and $c = \ln(1 - p)$;
2. If $c = 0$, skip to step 6;
3. Generate $u \sim U(0, 1)$;
4. Set $y = y + \lfloor \ln(u)/c \rfloor + 1$, where $\lfloor \bullet \rfloor$ denotes the floor function;
5. If $y \leq n$, set $x = x + 1$ and return to step 3;
6. Return x as a draw from a binomial (n, p).

In step 4 of algorithm BG, a geometric random variable is generated by truncating an exponential variable, $G = \lfloor \ln(U)/c \rfloor + 1 \sim G(p)$, as described by Devroye [1986]. The execution time of this algorithm is proportional to $n \cdot p$, representing a substantial performance improvement. When $p > 0.5$, we can optimize BG further by exploiting the symmetry property: if $X \sim \text{Binomial}(n, p)$, then $n - X \sim \text{Binomial}(n, 1 - p)$. We replace p with $\min(p, 1 - p)$ and return x if $p \leq 0.5$, or $n - x$ otherwise. The overall speed is thus proportional to $n \cdot \min(p, 1 - p)$.

```
import numpy as np

# Demonstration of algorithms for generating realizations of
# binomial random variables binom(size, p).
# BG Algorithm
def bg(size, 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 <= size:
            u = np.random.uniform(0.0, 1.0, 1)
            y += np.trunc(np.log(u) / c) + 1
            if y <= size:
                x += 1
        if p > 0.5:
            x = size - x
    return x

# Example
prob = 0.5
size = 3
print('bg: ', bg(size, prob))
```

```
bg:  1
```

A disadvantage of algorithm `BG` is that multiple uniform random numbers may be drawn before returning a single binomial value x. An alternative approach is to use an inversion algorithm based on the discrete distribution function.

As noted previously, the inversion method is the standard approach for converting a uniform variable U into a random variable X by inverting F. For continuous variables, the steps are:

- Generate uniform $u \sim U(0,1)$
- Return $x = F^{-1}(u)$

For discrete variables, inversion requires searching for the integer value x such that:

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

The simplest search strategy is a sequential search starting at $x = 0$. For the binomial distribution, this inversion algorithm—designated **BINV**—uses the recursive recurrence relation:

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

for $x = 1, 2, \ldots, n$ through the following steps:

1. Set $pp = \min(p, 1-p)$, $qq = 1 - pp$, $r = pp/qq$, $g = r(n+1)$, and $f = qq^n$;
2. Generate $u \sim U(0,1)$, set $x = 0$, and set $F = f$;
3. If $F \geq u$, skip to step 5;
4. Set $x = x + 1$, $f = f\left(\frac{g}{x} - r\right)$, $F = F + f$, and return to step 3;
5. If $p \leq 0.5$, return x; otherwise return $n - x$.

The speed of this algorithm is proportional to $n \cdot \min(p, 1-p)$. Its advantage is that only a single uniform variable is drawn per binomial realization. However, computing qq^n can be computationally expensive, and two potential numerical issues may arise: (i) numerical underflow when computing $f = qq^n$ for large n, and (ii) accumulation of round-off errors during recursive updates of f as n grows [Kachitvichyanukul and Schmeiser, 1988]. Devroye [1986] suggests using the waiting-time algorithm BG based on Lemma 3.2 instead of BINV to avoid these issues. On the other hand, Kachitvichyanukul and Schmeiser [1988] note that double-precision arithmetic effectively mitigates these precision concerns.

```
# Requires loading the previous scripts defining bu or bg
import numpy as np

# Demonstration of algorithms for generating realizations of
# binomial random variables binom(size, p).
# Below is the BINV algorithm. Additionally, the rbinom
# function accepts BU, BG, or BINV routines.

def binv(size, p):
    if p > 0.5:
        pp = 1 - p
    else:
        pp = p
    q = 1 - pp
    f = q**size
```

```
    r = pp / q
    g = r * (size + 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 = size - x
    return x

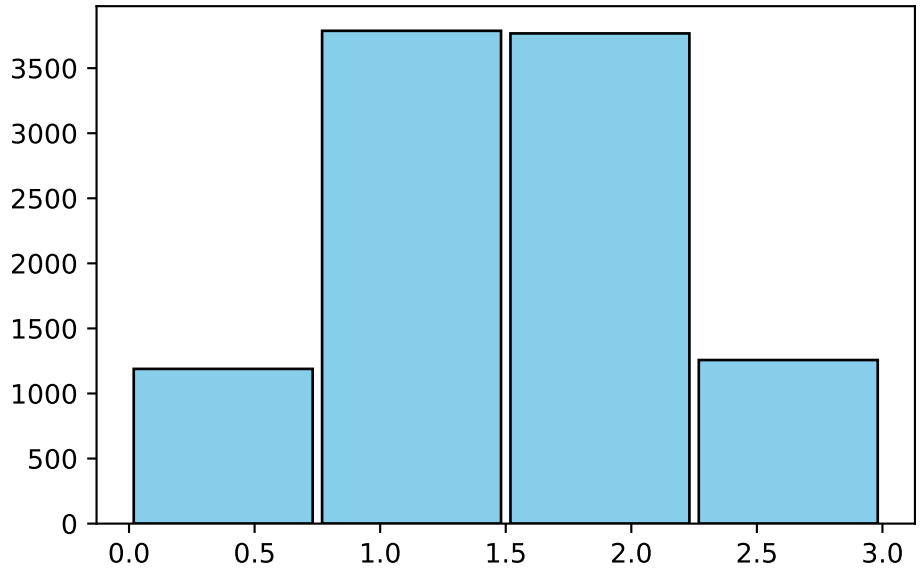
# Generator for a sample of binomial(size, prob) variates
# accepts one of the three implemented functions as an argument;
# size and prob are the binomial parameters and n is the sample size
# defaults to bg
def rbinom(n, size, prob, func = bg):
    x = []
    for i in range(n):
        x.append(func(size, prob))
    return x

# Example
prob = 0.5
size = 3
print('bu: ', bu(size, prob))
print('bg: ', bg(size, prob))
print('binv: ', binv(size, prob))
n = 10000 # sample size
x = rbinom(n, size, prob, bg) # can swap bg for binv or bu
# Create an isolated space for the figure
fig, ax = plt.subplots()

# Draw the histogram
_ = ax.hist(x, bins=size + 1, rwidth=0.95, edgecolor='black', color='skyblue')

# Display the plot natively
plt.show()
```

```
bu:  1
bg:  1
binv:  2
```



We can implement a parallelized version of `BG` using `Numba` (`pip install numba`). While vectorizing `BG` directly is difficult due to the random number of `while` loop iterations, Numba provides a straightforward alternative. Numba compiles pure Python loop constructs into machine code, enabling native parallelization across CPU cores via `prange` (*parallel range*). The `prange` construct informs Numba's compiler that iterations of the loop are mutually independent.

Combining `prange` with `parallel=True` allows Numba to automatically distribute n simulations across available CPU threads. The `@njit` decorator (short for no-Python mode JIT) compiles Python code into native machine code assembly, bypassing the Python interpreter entirely. If desired, the thread count can be fixed explicitly via `numba.set_num_threads(k)`. In the script below, this call is commented out, allowing Numba to utilize all available CPU cores.

The following code illustrates this implementation:

```
import numpy as np
import matplotlib.pyplot as plt
from numba import njit, prange

# Explicitly set the number of CPU threads used by Numba
# numba.set_num_threads(4)

@njit # Compiles the function for C-like execution speed
def bg_single(size, p):
    pp = 1 - p if p > 0.5 else p
    y, x = 0, 0
    c = np.log(1 - pp)
    if c < 0:
        while y <= size:
            u = np.random.uniform(0.0, 1.0)
            y += int(np.trunc(np.log(u) / c) + 1)
            if y <= size:
                x += 1
        if p > 0.5:
            x = size - x
```

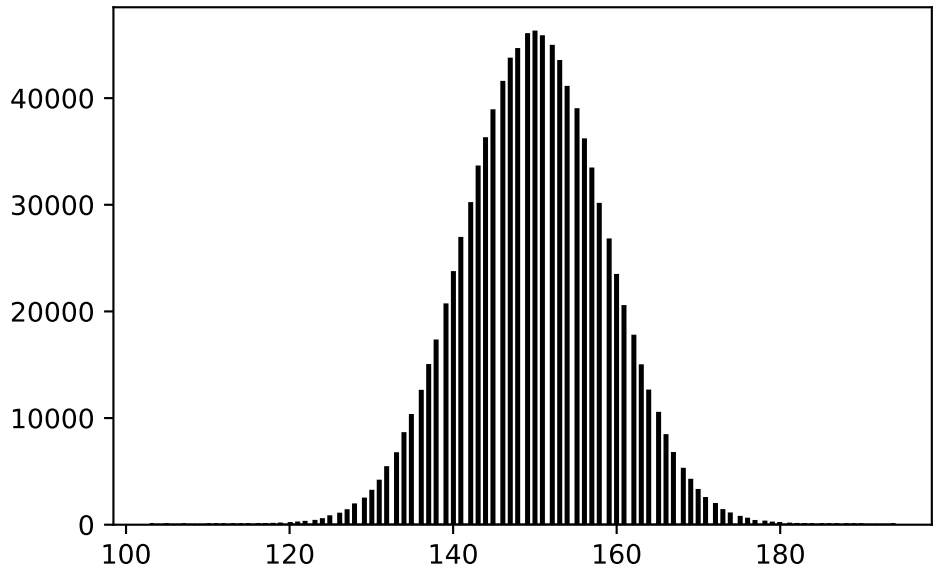
```
    return x

@njit(parallel=True) # Runs simulations in parallel across CPU cores
def rbinom_bg_numba(n, size, p):
    out = np.empty(n, dtype=np.int64)
    for i in prange(n):
        out[i] = bg_single(size, p)
    return out

# Usage example:
n = 1000000
size = 300
p = 0.5
x = rbinom_bg_numba(n, size, p)
# Create an isolated space for the figure
fig, ax = plt.subplots()

# Draw the histogram
_ = ax.hist(x, bins=size + 1, rwidth=0.95, edgecolor='black', color='skyblue')

# Display the plot natively
plt.show()
```



⚠ Pyodide Limitations (Numba and Memory Constraints)

Pyodide executes Python inside **WebAssembly (WASM)** within the browser environment, imposing strict operational limits:

- **Incompatibility with Numba:** Numba requires LLVM to generate native machine code dynamically (JIT). Because Pyodide operates within a browser sandbox, dynamic JIT compilation is unsupported. Remove or comment out decorators like `@jit` or `@njit`.
- **32-bit Memory Model:** WASM uses 32-bit memory addressing by default, constraining

available heap space.
- **2 GB to 4 GB Maximum Memory Limit:** Even if your physical system has 16 GB or 32 GB of RAM, Pyodide allocates a maximum of 2 GB to 4 GB for the entire Python session. Operations requesting large contiguous NumPy arrays can easily exceed this limit.

Conversely, for the `BU` method, a vectorized array implementation can be constructed as follows to generate samples of size n:

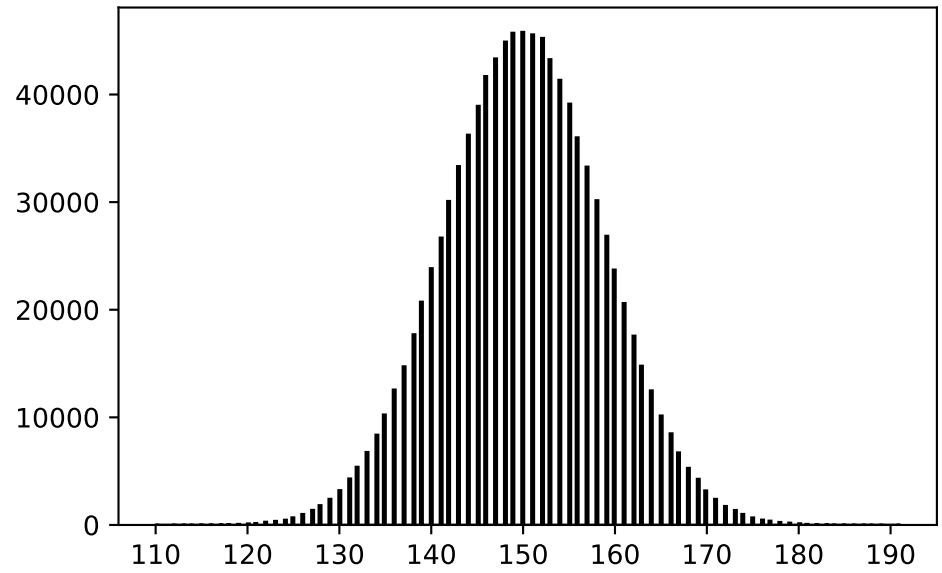
```
import numpy as np
import matplotlib.pyplot as plt

# Vectorized BU implementation combined with rbinom
def bu_vet(n, size, prob):
    u = np.random.uniform(size=(n, size))
    x = (u <= prob).sum(axis=1)
    return x

# Example
n = 100000
size = 300
p = 0.5
x = bu_vet(n, size, p)
# Create an isolated space for the figure
fig, ax = plt.subplots()

# Draw the histogram
_ = ax.hist(x, bins=size + 1, rwidth=0.95, edgecolor='black', color='skyblue')

# Display the plot natively
plt.show()
```



The command `np.random.uniform(size=(n, size))` creates an $n \times$ size matrix of uniform random numbers. The expression `(u <= prob).sum(axis=1)` evaluates to a boolean matrix of `True`/`False` values

and sums `True` (`=1`) entries across columns for each of the n rows. These two vectorized adaptations significantly boost computational efficiency.

While random number generators can be developed manually for many other distributions, Python provides optimized, pre-compiled routines for standard probability models in lower-level compiled code. We cover these built-in routines in the next section.

3.5 Python Routines for Generating Random Variables

In this section, we review the primary Python functions used to generate random variables from common probability distributions. The probability models supported by NumPy are summarized in Table 3.1.

Table 3.1: Probability distributions, NumPy function names (`np.random.col2name`), and arguments, where `size` specifies the output sample size.

Distribution	NumPy Name	Parameters
Beta	beta(*a*, *b*, *size=None*)	a, b, size
Binomial	binomial(*n*, *p*, *size=None*)	n, p, size
Standard Cauchy	standard_cauchy(*size=None*)	size
Chi-Square	chisquare(*df*, *size=None*)	df, size
Exponential	exponential(*scale=1.0*, *size=None*)	scale (1/lambda), size
F	f(*dfnum*, *dfden*, *size=None*)	dfnum, dfden, size
Gamma	gamma(*shape*, *scale=1.0*, *size=None*)	shape, scale (1/beta), size
Geometric	geometric(*p*, *size=None*)	p, size
Hypergeometric	hypergeometric(*ngood*, *nbad*, *nsample*, *size=None*)	ngood, nbad, nsample, size
Log-Normal	lognormal(*mean=0.0*, *sigma=1.0*, *size=None*)	mean, sigma, size
Logistic	logistic(*loc=0.0*, *scale=1.0*, *size=None*)	loc, scale, size
Negative Binomial	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
Student's t	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

In Python, the `scipy.stats` module enables evaluation of probability mass functions (discrete distributions) via functions like `sp.stats.binom.pmf(x, n, p)`, where `sp` represents `scipy` and `pmf` stands for Probability Mass Function. For continuous distributions, the corresponding method is `pdf` (Probability Density Function). Cumulative distribution functions are evaluated via `cdf` for both continuous and discrete models, while inverse cumulative distribution functions use `ppf` (Percent Point Function), which takes accumulated probability as its primary argument. Non-central distribution variants are also supported via non-centrality parameters (`ncp`). Additional information is available in the official `scipy.stats` documentation.

Using these routines to sample random variables is straightforward. For instance, to sample from a Beta distribution with parameters $\alpha = 1$ and $\beta = 2$, we can execute the following code script. Similar NumPy routines listed in Table 3.1 can generate samples of size n for other probability density or mass functions. Built-in functions are significantly faster than pure Python implementations because they execute optimized C code. If custom distributions must be sampled millions of times, compiling C routines and binding them to Python offers optimal performance.

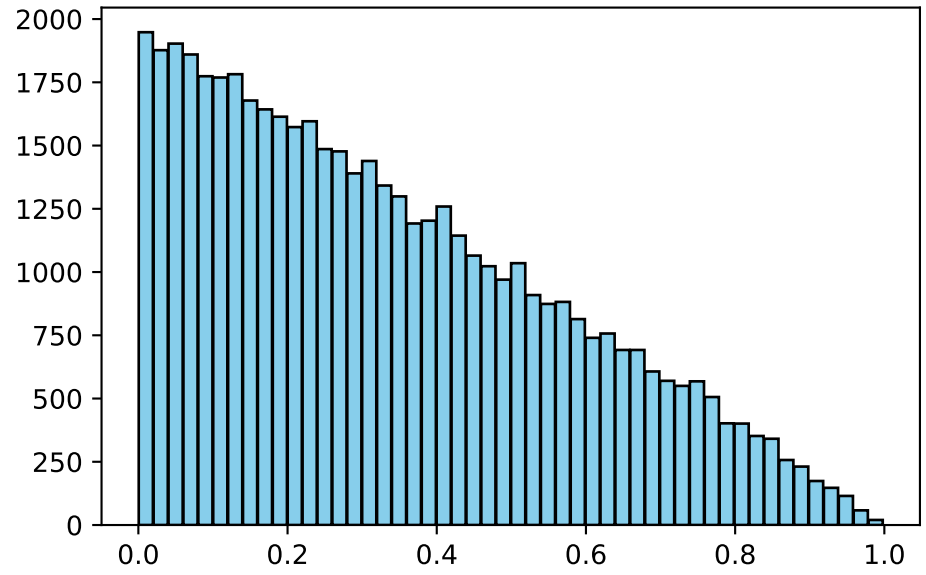
```
import numpy as np
import matplotlib.pyplot as plt

n = 50000
alpha = 1.0
beta = 2.0
x = np.random.beta(alpha, beta, n)
print('Mean: ', np.mean(x))
print('True mean: ', alpha / (alpha + beta)) # true mean
print('Variance: ', np.var(x))
print('True variance: ', alpha * beta / ((alpha + beta)**2 * (alpha + beta + 1)))
# Create an isolated space for the figure
fig, ax = plt.subplots()

# Draw the histogram
_ = ax.hist(x, bins='auto', rwidth=0.95, edgecolor='black', color='skyblue')

# Display the plot natively
plt.show()
```

```
Mean:  0.3350442104923811
True mean:  0.3333333333333333
Variance:  0.05597133886143762
True variance:  0.05555555555555555
```



3.6 Exercises

1. Let $f(x) = 3x^2$ be the probability density function of a continuous random variable X defined on $[0, 1]$. Apply the inversion method to derive an algorithm for generating random variables from this density. Implement the algorithm in Python and generate a sample of size $n = 1{,}000$. Estimate the 1%, 5%, 10%, 50%, 90%, 95%, and 99% quantiles and compare them against theoretical quantiles.

2. Use `Numba` to parallelize the Polar Box-Muller method and improve its efficiency. Compare execution speed against the original pure Python implementation and `NumPy`.

3. The dataset below contains the lifespan (in days) of $n = 40$ insects. Assuming lifespan follows an exponential distribution where the rate parameter λ is estimated by the maximum likelihood estimator $\hat{\lambda} = 1/\bar{X}$ ($\bar{X} = \sum_{i=1}^{n} X_i/n$), obtain a 95% confidence interval using the following parametric bootstrap procedure: (i) generate a sample of size $n = 40$ from an exponential distribution using `np.random.exponential` with parameter equal to the estimated rate; (ii) calculate the sample mean $\bar{X}$ to estimate $\mu = 1/\lambda$; (iii) repeat steps (i) and (ii) 1,000 times and store the resulting estimates; (iv) sort the estimates and extract the 2.5% and 97.5% quantiles. The recorded lifespans (in days) are:

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

Repeat this procedure using 100,000 bootstrap replicates of size $n = 40$. Compare results and evaluate whether the additional computational cost of increasing bootstrap replicates produced a meaningful gain in precision.

4. Generate a sample of $n = 5{,}000$ standard normal realizations using two approximations: (a) $X = [U^{0.135} - (1-U)^{0.135}]/0.1975$ and (b) the sum of 12 independent uniform variables $X = \sum_{i=1}^{12} U_i - 6$. Compare the empirical 1%, 5%, 10%, 50%, 90%, 95%, and 99% quantiles against theoretical standard normal quantiles. Generate an additional sample of size $n = 5{,}000$ using the Polar Box-Muller algorithm, calculate the same sample quantiles, and compare across all methods.

5. Assuming the insect lifespan dataset from Exercise 3 can be modeled by a log-normal distribution, estimate the log-normal parameters and apply the parametric bootstrap approach to construct a confidence interval for the population mean. To estimate log-normal parameters: (a) log-transform the original data, $X_i^* = \ln(X_i)$; (b) calculate the sample mean and standard deviation of X_i^*, using these as estimates for μ and σ. Use these parameters to generate log-normal samples by drawing standard normal variates $Z \sim N(0, 1)$, transforming to $Y = SZ + \bar{X}$, and setting $X = \exp(Y)$. Perform the bootstrap procedure, compare results against the exponential model, and discuss the challenges of selecting an underlying probability model for statistical inference. How might one empirically determine which probability distribution best fits the insect lifespan data? Justify your answer using appropriate statistical and numerical criteria.

6. Perform non-parametric bootstrap resampling with replacement from the sample in Exercise 3 to estimate a 95% confidence interval for the population mean, using a uniform random generator to select bootstrap sample indices: (i) resample $n = 40$ observations with replacement from the original dataset to form X_i^*; (ii) compute the bootstrap sample mean $\bar{X}^* = \sum_{i=1}^{n} X_i^*/n$; (iii) record this mean and repeat steps (i) and (ii) $B - 1$ times; (iv) combine the bootstrap estimates with the original sample mean; and (v) sort the values and extract the 2.5% and 97.5% quantiles across all B values. Perform the procedure for $B = 1{,}000$ and $B = 100{,}000$, comparing the resulting confidence bounds with those obtained for the exponential and log-normal models in previous exercises. Models

yielding confidence bounds closest to the non-parametric bootstrap provide empirical evidence regarding distribution fit. Discuss the primary advantage of non-parametric bootstrapping in avoiding distribution assumptions.

7. A fundamental property of the exponential distribution, $f(x) = \lambda e^{-\lambda x}$, is that the sum of n independent and identically distributed (i.i.d.) random variables $X_i \sim \text{Exp}(\lambda)$ follows a Gamma distribution with shape parameter $\alpha = n$ and scale parameter $\beta = 1/\lambda$:

$$Y = \sum_{i=1}^{n} X_i \sim \text{Gamma}\left(n, \frac{1}{\lambda}\right)$$

where the Gamma probability density function is:

$$f(y) = \frac{1}{\beta^{\alpha}\Gamma(\alpha)} y^{\alpha-1} e^{-\frac{y}{\beta}}$$

with $\alpha = n$. This property allows us to avoid drawing n individual random variates across each of the B parametric bootstrap replicates. Assuming the data in Exercise 3 follow an exponential distribution, we can estimate the population scale parameter β using the sample mean, $\hat{\beta} = \bar{X}$. Using this estimator, implement an optimized parametric bootstrap that directly simulates the sample mean distribution ($\bar{Y}$) via the following steps:

```
(i) Draw a vector of size $B$ directly from a **Standard Gamma** distribution ($\alpha = n$, scale $\be
(ii) Scale the sums by multiplying by $\hat{\beta}$ (i.e., $Y = \hat{\beta} \cdot \text{Standard Gamma}
(iii) Divide by sample size $n$ to obtain simulated sample means: $\bar{Y} = Y / n$;
(iv) Sort the resulting values and extract the $2.5\%$ and $97.5\%$ quantiles to construct a $95\%$ con

- **(a)** What conclusions can be drawn regarding the confidence interval bounds produced by both strat
- **(b)** What are the key computational advantages (in terms of memory usage and processing time) of s
```

8. Two binomial samples were collected from two distinct populations (1 and 2). The observed success counts were $y_1 = 2$ and $y_2 = 3$ in samples of size $n_1 = 12$ and $n_2 = 14$, respectively. Estimate population parameters p_1 and p_2 using sample proportions $\hat{p}_1 = y_1/n_1$ and $\hat{p}_2 = y_2/n_2$. To test the hypothesis $H_0 : p_1 = p_2$ (or $H_0 : p_1 - p_2 = 0$), implement the following parametric bootstrap algorithm: (a) use $\hat{p}_1$ and $\hat{p}_2$ to generate binomial samples of size n_1 and n_2 from both populations; (b) compute bootstrap estimates $\hat{p}_{1j} = y_{1j}/n_1$ and $\hat{p}_{2j} = y_{2j}/n_2$ on iteration j; (c) calculate $d_j = \hat{p}_{1j} - \hat{p}_{2j}$; (d) repeat steps (a) through (c) $B - 1$ times; (e) combine bootstrap difference estimates with the original sample difference; (f) sort the values and extract the 2.5% and 97.5% quantiles of the bootstrap distribution of d_j; and (g) if 0 lies within this confidence interval, do not reject H_0; otherwise, reject H_0 in favor of unequal population proportions.

Chapter 4

Generating Random Samples of Multidimensional Variables

Multivariate models have gained wide acceptance in the scientific community due to computational advances and the development of specialized software in this area. Natural phenomena are generally multivariate. A treatment applied to a living being, to a soil, or to a system does not affect only a single variable in isolation, but rather all variables. Moreover, the variables are related to one another, and any change in one or some of them affects the others. Thus, the generation of realizations of random vectors or matrices is a subject that cannot be ignored. In this chapter, we will provide the reader with mechanisms to generate realizations of multidimensional random variables.

4.1 Introduction

The procedures for generating multidimensional random variables are often considered difficult by most researchers. A good portion of them, however, can be carried out in Python with just a single line of code. Despite these conveniences, in these notes we will present details of some procedures for generating data from the main multivariate probabilistic models, such as the multivariate normal, the Wishart and inverse Wishart, the multivariate Student's t, and some other distributions.

One of the main characteristics of multidimensional variables is the correlation among their components. The importance of these models is practically indescribable, but we can highlight parametric inference, Bayesian inference, estimation of confidence regions, among others. In the following sections, we will address ways to generate realizations of multidimensional random variables for specific models using Python either to implement the routines or to use pre-existing routines. Our apparent waste of time, describing functions that are less efficient than those already available in Python, has as its fundamental reason allowing the reader to go beyond simply using routines previously programmed by third parties. If the reader gains, by the end of reading this material, the ability to produce their own routines and to understand how pre-existing routines work, our objective will have been achieved.

4.2 Multivariate Normal Distribution

The multivariate normal probability density function of a random vector $\mathbf{X}$ is given by:

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

where μ and $\mathbf{\Sigma}$ are, respectively, the mean vector and the covariance matrix, symmetric and positive definite, and p is the dimension of the random vector $\mathbf{X}$.

An important result concerns linear combinations of multivariate normal variables and will be presented in the following theorem.

Theorem 4.1 (Linear Combinations)**.** *Consider the multivariate normal random vector* $\mathbf{X} = [X_1, X_2, \cdots, X_p]^\top$ *with mean* μ *and covariance* $\mathbf{\Sigma}$*, and consider* $\mathbf{C}$ *a* $(p \times p)$ *matrix of rank* p*. Then the linear combination* $\mathbf{Y} = \mathbf{CX}$ $(p \times 1)$ *has a multivariate normal distribution with mean* $\mu_\mathbf{Y} = \mathbf{C}\mu$ *and covariance* $\mathbf{\Sigma_Y} = \mathbf{C\Sigma C}^\top$*.*

Proof. If $\mathbf{C}$ has rank p, then $\mathbf{C}^{-1}$ exists and, therefore,

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

The Jacobian of the transformation is $J = |\mathbf{C}|^{-1}$, and the distribution of $\mathbf{Y}$ is given by $f_\mathbf{Y}(\mathbf{y}) = f_\mathbf{X}(\mathbf{x})|J|$. If $\mathbf{X}$ has a multivariate normal distribution, then

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

which is the multivariate normal density of the random vector $\mathbf{Y}$ with mean $\mu_\mathbf{Y} = \mathbf{C}\mu$ and covariance $\mathbf{\Sigma_Y} = \mathbf{C\Sigma C}^\top$. Therefore, linear combinations of multivariate normal variables are multivariate normal. □

Theorem 4.1 provides us with the main result for generating data from a multivariate normal. Thus, since our objective is to generate data from a multivariate normal sample with a pre-established mean vector μ and covariance matrix $\mathbf{\Sigma}$, we should follow the following procedures. First, we must obtain the square root matrix of $\mathbf{\Sigma}$, denoted by $\mathbf{\Sigma}^{1/2}$. For this, we consider the spectral decomposition of the covariance matrix given by $\mathbf{\Sigma} = \mathbf{P\Lambda P}^\top$. Hence, we can define the square root matrix of $\mathbf{\Sigma}$ as $\mathbf{\Sigma}^{1/2} = \mathbf{P\Lambda}^{1/2}\mathbf{P}^\top$, where $\mathbf{\Lambda}$ is the diagonal matrix of eigenvalues, $\mathbf{\Lambda}^{1/2}$ is the diagonal matrix containing the square root of these elements, and $\mathbf{P}$ is the matrix of eigenvectors, each of these vectors arranged in one of its columns. From this decomposition, we can easily observe that $\mathbf{\Sigma} = \mathbf{\Sigma}^{1/2}\mathbf{\Sigma}^{1/2}$.

Thus, we can use the following method to generate a p-variate realization from a multivariate normal. First, we must generate a random vector $\mathbf{Z} = [Z_1, Z_2, \cdots, Z_p]^\top$ of p independent standard normal variables using, for example, the Box–Müller algorithm. This means that $\mu_\mathbf{Z} = \mathbf{0}$ and that $\text{Cov}(\mathbf{Z}) = \mathbf{I}$. This vector must undergo the following linear transformation:

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

According to Theorem 4.1, the vector $\mathbf{Y}$ has a multivariate normal distribution with mean $\mu_\mathbf{Y} = \mathbf{\Sigma}^{1/2}\mu_\mathbf{Z} + \mu = \mu$ and covariance matrix $\mathbf{\Sigma}^{1/2}\mathbf{I\Sigma}^{1/2} = \mathbf{\Sigma}$. This will be the method we use to obtain the p-dimensional sample of size n from a multivariate normal with mean μ and covariance $\mathbf{\Sigma}$. To obtain the square root matrix in Python, we can use the command to obtain the spectral decomposition

`np.linalg.svd()` or alternatively the command `linalg.cholesky(a,/,*,upper=False)`, which returns the Cholesky factor of a positive definite matrix, which is in fact a type of square root. The singular value decomposition in this case specializes to the spectral decomposition, since the matrix $\mathbf{\Sigma}$ is symmetric. We will generate a vector of independent standard normal random variables $\mathbf{Z}$ using the command `np.random.normal()`. Then transformation (4.2) is performed.

Let us illustrate and present the program for generating multivariate normal variables for a particular bivariate case ($p = 2$) with mean vector $\mu = [10, 50]^\top$ and covariance matrix given by:

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

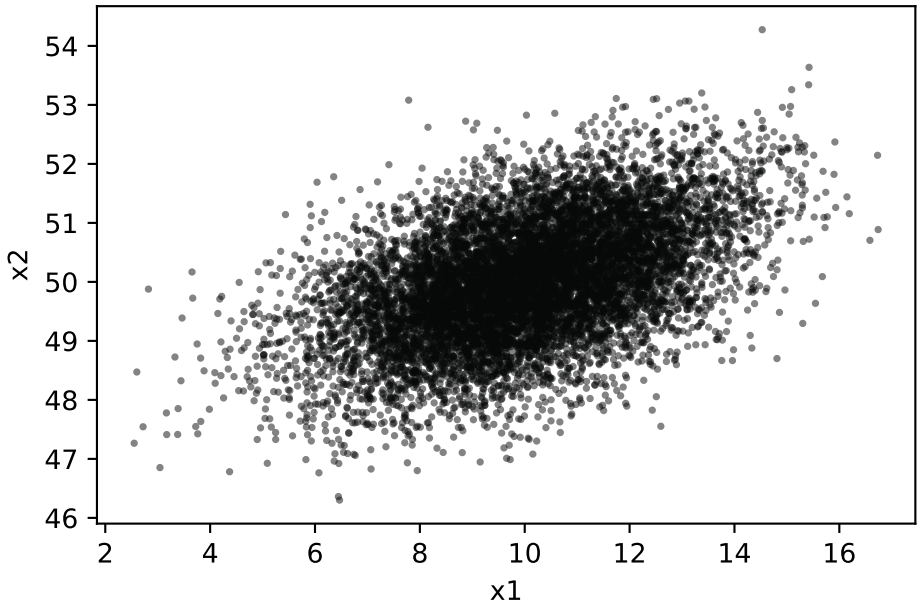
The resulting program is given by:

```
# Python function to generate n multivariate normal random vectors
# with mean vector mu and covariance sigma.
# The result is an n x p matrix, where p is the number of variables
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 @ (d[:, None]**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

# Example of use
n = 10000
sigma = np.array([[4,1],[1,1]])
mu = [10, 50]
x = rnormmv(n, mu, sigma)
print('Mean: \n', np.mean(x, axis = 0))
print('Covariance: \n', 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()
```

```
Mean:
 [10.01081487 50.00329797]
Covariance:
 [[3.95563953 0.99320992]
 [0.99320992 1.01206154]]
```



> **Note on Plots in the Browser**
>
> In some browsers (especially **Firefox**), displaying plots with `plt.show()` in the interactive environment of this page may fail. If the plot does not appear, open the material in **Google Chrome**, **Microsoft Edge**, or **Safari**.

In Python, we have the `numpy` generator `random.multivariate_normal(mean, cov, size=None, check_valid='warn', tol=1e-8)` for generating data from the multivariate normal distribution. In the following script, we apply both our function and the `numpy` function to generate a large number of observations and compare their performance in terms of processing time. The `numpy` function protects the process by, for example, checking whether the covariance matrix is positive definite. In our function, we do not use any protection, although it would be possible to do so.

```
# Performance comparison of the two multivariate normal
# generators
from datetime import datetime
import numpy as np

n = 60000
sigma = np.array([[4,1],[1,1]])
mu = [10, 50]
t1 = datetime.now()
x = rnormmv(n, mu, sigma)
t2 = datetime.now()
trnm = t2-t1
print('average time of rnormmv: ',trnm.microseconds / n)
t1 = datetime.now()
x = np.random.multivariate_normal(mu, sigma, size=n)
t2 = datetime.now()
tnp = t2-t1
print('average time of numpy: ',tnp.microseconds / n)
```

```
# relative time
print('numpy is faster by a factor of ',trnm.microseconds / tnp.microseconds)
```

```
average time of rnormmv:  11.489866666666666
average time of numpy:  0.0392
numpy is faster by a factor of  293.10884353741494
```

The `numpy` routine was many times faster than our implementation. This is partly due to the fact that it is compiled and, possibly, also to the method used to obtain the square root matrix. Swapping the `svd` square root matrix for `Cholesky` or the spectral decomposition could be done, and the average performance of the procedures evaluated, to define the best implementation strategy. We made a change to our implementation. We added an argument with one of three options so that the user can choose which method to use. Obtaining the square root was implemented in a separate function.

```
import numpy as np

# Python function to generate n multivariate normal random vectors
# with mean vector mu and covariance sigma.
# The result is an n x p matrix, where p is the number of variables
def sigmaroot(sigma, metodo='chol'):
  if metodo == 'svd':
    u, d, vt = np.linalg.svd(sigma)
    root = u @ (d[:, None]**0.5 * vt)
  elif metodo == 'eig':
    d, u = np.linalg.eigh(sigma)
    root = u @ (d[:, None]**0.5 * np.transpose(u))
  else:
    root = np.linalg.cholesky(sigma)
  return  root

# Valid methods are: 'chol', 'svd', and '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
```

The comparison between them was presented in the following `script`:

```
import numpy as np

# Performance comparison of the two multivariate normal
# generators
n = 60000
sigma = np.array([[4,1],[1,1]])
mu = [10, 50]
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('average time of svd: ',tsvd.microseconds / n)
print('average time of eig: ',teig.microseconds / n)
print('average time of chol: ',tchol.microseconds / n)
print('average time of numpy: ',tnp.microseconds / n)
# relative time
print('numpy is faster than svd by a factor of ',tsvd/tnp)
print('numpy is faster than eig by a factor of ',teig/tnp)
print('numpy is faster than chol by a factor of ',tchol/tnp)
```

```
average time of svd:  11.565883333333334
average time of eig:  11.938966666666667
average time of chol:  11.391283333333334
average time of numpy:  0.03436666666666666
numpy is faster than svd by a factor of  336.5436469447139
numpy is faster than eig by a factor of  347.39961202715807
numpy is faster than chol by a factor of  331.4631425800194
```

Although our function is relatively fast, taking between 2 and 16 microseconds to run each bivariate observation in this case, it was still far less efficient than the `numpy` routine. A more detailed and separate evaluation of the effect of using the three methods for obtaining the "square root" matrix is necessary, given that the dimension is small ($p = 2$) in the example and only one call is made to generate n multivariate random vectors. The main bottleneck in this case was the loop used, and not the method for obtaining the square root of the covariance matrix in the example. Compilation is fundamental in relation to interpretation. It is extremely simple to generate multivariate normal data using the `numpy` function, which frees us from having to program our own random generators. We reiterate that we did this because we want our reader and our student to be able to uncover what lies behind each method here and also to develop their programming skills by implementing sophisticated routines such as these.

4.3 Wishart and Inverse Wishart Distribution

The Wishart and inverse Wishart distributions are related to the distributions of matrices of uncorrected sums of squares and cross products $\mathbf{W}$ obtained from samples of size $n-1$ of the multivariate normal distribution with mean $\mathbf{0}$. Consider $\mathbf{X}_j = [X_1, X_2, \cdots, X_p]^\top$ the j-th vector ($j = 1, 2, \cdots, \nu$) of a random sample of size ν from a normal with mean $\mathbf{0}$ and covariance $\mathbf{\Sigma}$. Then the random matrix

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

has a Wishart distribution with $n-1$ degrees of freedom and parameter $\mathbf{\Sigma}$ (positive definite matrix).

Likewise, if we have a random sample of size n from a multivariate normal distribution with mean μ and covariance $\mathbf{\Sigma}$, the distribution of the random matrix

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

is Wishart with $\nu = n-1$ degrees of freedom and parameter $\mathbf{\Sigma}$.

The Wishart density function of a random matrix $\mathbf{W}$ of sums of squares and cross products, denoted by $W_p(\nu,\mathbf{\Sigma})$, is defined by:

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

where $\Gamma(x) = \int_0^\infty t^{x-1}e^{-x}dt$ is the gamma function.

Thus, to generate Wishart variables with parameters $n-1$ integer degrees of freedom and positive definite matrix $\mathbf{\Sigma}$, we can use a generator of multivariate normal random samples and obtain the sample matrix of sums of squares and cross products. This matrix will be a realization of a Wishart random variable, which is a matrix of dimension $p \times p$. The following function can be used to obtain random realizations from a Wishart:

```
import numpy as np

# Example for generating random matrices of sums of squares and cross products
# W with Wishart(nu, Sigma) distribution, nu = n - 1
# uses random.multivariate_normal to generate multivariate normals.
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

# Example of use
sigma = np.array([[4, 1], [1, 1]])
nu    = 5
w     = rwishart(nu, sigma)
print('Wishart realization: \n',w)
```

```
Wishart realization:
 [[15.08731559 10.26259309]
 [10.26259309  7.82653942]]
```

Another related distribution that frequently appears in multivariate inference is the inverse Wishart. Consider $\mathbf{W}$ a random matrix $W_p(\nu,\mathbf{\Sigma})$. Then the distribution of $\mathbf{S} = \mathbf{W}^{-1}$, given by the density function

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

is the inverse Wishart, which we will denote by $W_p^{-1}(\nu, \mathbf{\Sigma})$.

If we want to generate a random matrix from an inverse Wishart distribution instead of a Wishart, we simply need to perform the transformation $\mathbf{S} = \mathbf{W}^{-1}$. Thus, we will modify the previous program to generate realizations from both the Wishart and the inverse Wishart distributions. It is important to note that in our notation the parameters of the inverse Wishart are those of the Wishart. This is relevant, since some authors present the parameters of the inverse Wishart rather than those of the Wishart, and we must be attentive to this fact; otherwise, we will generate random variables with different densities.

```
import numpy as np


# Example for generating random matrices of sums of squares and cross products
# W with Wishart(nu, Sigma) distribution, nu = n - 1
# uses the mvtnorm package to generate normal samples.
# Example for generating random matrices of sums of squares and cross products
# W with Wishart(nu, Sigma) distribution, nu = n - 1
# and inverse Wishart
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:
    iw = np.linalg.pinv(w)
    res = {'W': w, 'WI': iw}
    return res
  else:
    return w

# Example of use
sigma = np.array([[4, 1], [1, 1]])
nu    = 5
wi = True
res     = rw_wi(nu, sigma, wi)
print('Both, W and WI \n', res)
wi = False
res     = rw_wi(nu, sigma, wi)
print('Only the Wishart \n', res)
```

```
Both, W and WI
 {'W': array([[15.79229631,  2.14163672],
       [ 2.14163672,  0.80564121]]), 'WI': array([[ 0.09901792, -0.26321943],
       [-0.26321943,  1.94096377]])}
Only the Wishart
 [[19.84067566  6.43800911]
 [ 6.43800911  2.8901032 ]]
```

An important aspect we need to mention concerns the need to generate Wishart or inverse Wishart variables with real degrees of freedom. For this, we can use the algorithm described by Smith and Hocking [1972]. Consider the decomposition of the matrix $\mathbf{\Sigma}$ given by $\mathbf{\Sigma} = \mathbf{\Sigma}^{1/2}\mathbf{\Sigma}^{1/2}$, which we will use to perform a transformation on the generated random matrix that, according to properties of Wishart matrices, will be Wishart.

We must first construct a lower triangular matrix $\mathbf{T} = [t_{ij}]$ (i,j = 1, 2, $\cdots$, p), with $t_{ii} \sim \sqrt{\chi^2_{\nu+1-i}}$ and $t_{ij} \sim N(0,1)$ if $i > j$, and $t_{ij} = 0$ if $i < j$. Then we must obtain the matrix $\mathbf{\Gamma} = \mathbf{T}\mathbf{T}^\top$, which has distribution $W_p(\nu, \mathbf{I})$. In this way, we obtain $\mathbf{W} = \mathbf{\Sigma}^{1/2}\mathbf{\Gamma}\mathbf{\Sigma}^{1/2}$ with the desired distribution, that is, with distribution $W_p(\nu,\mathbf{\Sigma})$. It is clear that with this algorithm we can generate Wishart variables with real or integer degrees of freedom.

```
import numpy as np

# More efficient function for generating random matrices of sums of
# squares and cross products W with Wishart(nu, Sigma) distribution and
# inverse Wishart WI(nu, Sigma)
def rwwi_sh(nu, sigma, wi = True):
  tip = type(sigma)
  if tip is int or tip is 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 is int or tip is float:
    w = s**2 * t**2
  else:
    w = s @ t @ np.transpose(t) @ np.transpose(s)
  if wi:
    if tip is int or tip is float:
      iw = 1 / w
    else:
      iw = np.linalg.pinv(w)
    res = {'W': w, 'WI': iw}
    return res
  else:
    return w

# Example of use
sigma = np.array([[4, 1], [1, 1]])
nu    = 5
wi = True
res     = rwwi_sh(nu, sigma, wi)
print('Both, W and WI \n', res)
wi = False
res     = rwwi_sh(nu, sigma, wi)
print('Only the Wishart \n', res)
```

```
Both, W and WI
 {'W': array([[60.78813332, 17.03935951],
       [17.03935951, 10.91763044]]), 'WI': array([[ 0.02924449, -0.04564245],
```

```
       [-0.04564245,  0.16283004]])}
Only the Wishart
 [[30.99024788  5.32527254]
 [ 5.32527254  2.09762949]]
```

We would like to draw attention to some facts about this function, since we used some commands and resources not mentioned up to this point. First, we used the command `np.diag(np.random.chisquare(df, p)**0.5)` to fill the diagonal of the matrix $\mathbf{T}$ with realizations of chi-square random variables and then the `diag` method to transform the vector into a diagonal matrix.

Another interesting aspect that deserves mention is the use of the Cholesky factor instead of obtaining the square root matrix of $\mathbf{\Sigma}$. The Cholesky factor uses the decomposition $\mathbf{\Sigma} = \mathbf{S}\mathbf{S}^\top$, where $\mathbf{S}$ is a lower triangular matrix. Python, through the function `np.linalg.cholesky(sigma)`, returns the matrix $\mathbf{S}$. Finally, if the number of variables is equal to 1, the Wishart distribution specializes to the chi-square and the inverse Wishart to the inverse chi-square. Thus, when $p = 1$, the algorithm will return variables $\sigma^2 X$ and $1/(\sigma^2 X)$ with distributions proportional to the chi-square and inverse chi-square distributions, respectively, where X is a chi-square variable with ν degrees of freedom.

The processing for the scalar case was controlled with the use of `if`, since matrix operations are not applicable if the arguments are integer scalars or `float` (real) numbers.

4.4 Multivariate Student's t Distribution

The family of elliptical distributions is very important in the multivariate setting and in Bayesian applications. The multivariate Student's t distribution is a particular case of this family. This distribution is of particular interest in procedures for multiple comparison of treatments against some control evaluated in the experiment. The density function of the random vector $\mathbf{X} = [X_1, X_2, \cdots, X_p]^\top \in \mathbb{R}^p$ with parameters given by the mean vector $\mu = [\mu_1, \mu_2, \cdots, \mu_p]^\top \in \mathbb{R}^p$ and symmetric positive definite matrix $\mathbf{\Sigma}$ $(p \times p)$ is

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

where the function g is given by

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

The random variable $\mathbf{X}$ has mean μ and covariance matrix $\nu\mathbf{\Sigma}/(\nu-2)$ in the case $\nu > 2$.

If we perform the transformation $\mathbf{Y} = \mathbf{\Sigma}^{-1/2}(\mathbf{X} - \mu)$, we obtain the symmetric spherical multivariate t distribution , whose density is given by:

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

The random variable $\mathbf{Y}$ will have a null mean vector and covariance $\nu\mathbf{I}/(\nu-2)$, and the density will have spherical contours of equal probability.

We will present the general form for generating p-dimensional multivariate t random variables with ν degrees of freedom and parameters μ and $\mathbf{\Sigma}$. Let $\mathbf{Z}$ be a random vector with distribution $N_p(\mathbf{0}, \mathbf{I})$ and U

a random variable with chi-square distribution with ν degrees of freedom. Then the random vector $\mathbf{Y}$, given by the transformation

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

has a spherical multivariate t distribution with ν degrees of freedom. The vector $\mathbf{X}$ obtained by the linear transformation

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

has an elliptical multivariate t distribution with ν degrees of freedom and parameters μ and $\mathbf{\Sigma}$.

We must apply transformation (4.7) n times to n different random vectors $\mathbf{Y}$ and variables U. At the end of this process, we will have a sample of size n from the desired multivariate t distribution with ν degrees of freedom. Thus, to generate data from a multivariate t with dimension p, degrees of freedom ν (not necessarily integer), any mean vector μ, and positive definite matrix $\mathbf{\Sigma}$, we can use the following Python function, replacing in expression (4.7) the square root matrix with the Cholesky factor $\mathbf{F}$ of $\mathbf{\Sigma}$:

```
#| fig-width: 10
#| fig-height: 5
#| fig-dpi: 120

import matplotlib.pyplot as plt
import numpy as np

# Function to generate multivariate t random variables
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

# Example of use
n = 3000
nu = 3
mu = [0,0]
sigma = np.identity(2)
x = rtmult(n, mu, sigma, nu)

# Create the two subplots side by side
fig, (ax1, ax2) = plt.subplots(1, 2)
fig.suptitle('Multivariate t distributions')

# First subplot
ax1.scatter(x[:,0], x[:,1], s = 3, c = '#070808', alpha = 0.5)
ax1.set_xlabel('x1')
ax1.set_ylabel('x2')

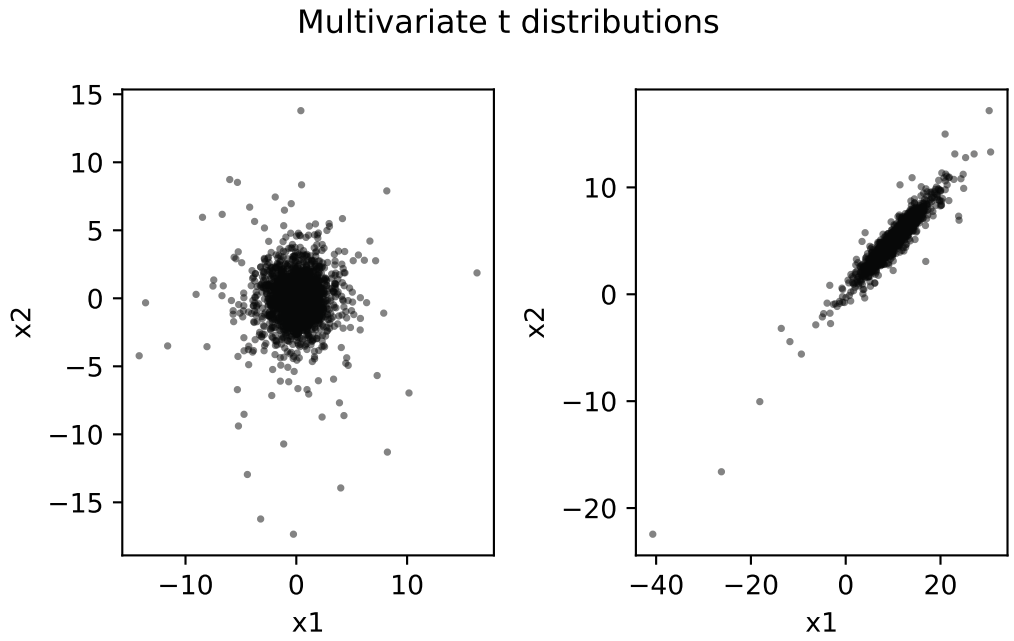
# Data for the second subplot
mu = [10, 5]
sigma = np.array([[4, 1.9], [1.9, 1]])
```

```
x = rtmult(n, mu, sigma, nu)

# Second subplot
ax2.scatter(x[:,0], x[:,1], s = 3, c = '#070808', alpha = 0.5)
ax2.set_xlabel('x1')
ax2.set_ylabel('x2')

# Adjust spacing so axes are not cut off and force rendering
plt.tight_layout()
plt.show()

# Text output that will appear just below the plot
print('Sample covariance: \n', np.cov(x, rowvar = False))
print('Expected covariance: \n', nu * sigma)
```




```
Sample covariance:
 [[11.39650964  5.46032385]
 [ 5.46032385  2.90304623]]
Expected covariance:
 [[12.   5.7]
 [ 5.7  3. ]]
```

Positive real degrees of freedom can be used as an argument to the function created. In the previous example, two random samples were generated for illustration. The values $[0,, 0, 0]^\top$, $\mathbf{I}_3$, and 3 were defined as `default` for the parameters μ, $\mathbf{\Sigma}$, and ν, respectively. We can also use the implementation provided by the function `multivariate_t.rvs(mu, sigma, df n)` from the `scipy` library to generate data from the multivariate Student's t distribution.

```
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)
```

```
[[ 3.0564157  -1.5193986 ]
 [-1.92349571  0.99893207]
 [ 6.22773886 -2.09945077]
 [-0.08442755 -2.4844584 ]
 [-1.2138771  -3.41383853]]
```

4.5 Other Multivariate Distributions

There are many other multivariate distributions. We will mention only two more of them: the multivariate log-normal and the multivariate contaminated normal. Generating a random vector of dimension p with a multivariate log-normal distribution is done by taking the following vector $\mathbf{Y} = [\exp(Z_1), \exp(Z_2), \cdots, \exp(Z_p)]^\top$, where $\mathbf{Z} \sim N_p(\mu, \mathbf{\Sigma})$.

```
import numpy as np

# generator of realizations of multivariate log-normal
# variables with parameters mu and sigma
def rlgnormalmv(n, mu=np.zeros(3), sigma=np.identity(3)):
  x = np.exp(np.random.multivariate_normal(mu,sigma,n))
  return x

# Example of use
n = 7
print(rlgnormalmv(n)) # default lgnormalmv
```

```
[[ 1.46442973  0.82070683  0.46427474]
 [ 2.38181525  0.68162771  0.34604334]
 [ 0.56320958  0.53713347  0.25180584]
 [ 1.14540077  0.34006953  0.76270396]
 [ 0.80621126 16.53633323  0.92121825]
 [ 2.56541548  0.48737716  0.56682442]
 [ 0.93716405  1.09642529  3.56302979]]
```

To generate realizations from the multivariate contaminated normal, we consider the simulation of a random vector $\mathbf{X}$ whose probability density function is given by:

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

where $\mathbf{\Sigma}_i$ is positive definite, $i = 1, 2$, and $0 \leq \delta \leq 1$.

If the mixture has the same mean $\mu_1 = \mu_2 = \mu$ and the covariances are merely multiples of one another $\mathbf{\Sigma}_2 = c\mathbf{\Sigma}_1$, for $c > 0$, then the contaminated multivariate normal distribution belongs to the elliptical family.

```
#| fig-width: 8
#| fig-height: 5
```

```
#| fig-dpi: 120

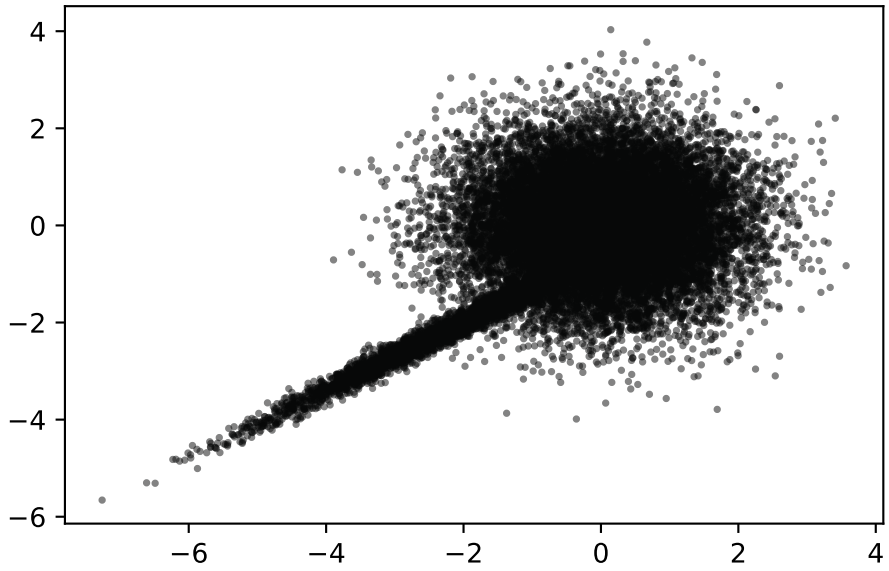
import matplotlib.pyplot as pplt
import numpy as np

# generator of multivariate contaminated normal variables
# with delta being the proportion of non-contaminants: (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

# Example
n = 20000
delta = 0.8
mu1 = np.zeros(2)
sig1 = np.identity(2)
mu2 = [-2, -2]
sig2 = np.array([[2, 1.4], [1.4, 1]])
x = rncm(n, mu1, mu2, sig1, sig2, delta)
# Create a new figure dedicated to this block
fig, ax = pplt.subplots()

# Plot using the created 'ax' object
_ = ax.scatter(x[:,0], x[:,1], s = 3, c = '#070808', alpha = 0.5)

pplt.show()
```



4.6 Exercises

1. Generate a sample of size ($n = 10$) from a trivariate normal distribution with mean vector $\mu = [5, 10, 15]^\top$ and covariance matrix

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

   Estimate the sample mean and covariance. Repeat this process 1,000 times and estimate the mean of the sample means and the mean of the sample covariance matrices. Do these values correspond exactly to the respective expected values? If not, present the main cause(s).

2. In the original text, there was a modification of the `rnormmv` function producing the `rnormmv_2` function, and then the average processing speeds were compared when using one of the following options for the "square root" matrix of $\mathbf{\Sigma}$: `svd`, `eig`, or `cholesky` from `linalg`. In this exercise, perform a further improvement of the `rnormmv_2` function by implementing a function to eliminate the `loop` through the use of vectorization. Verify whether there will be an improvement in performance relative to the average processing speeds for each generated random variable, which is the p-dimensional vector. Test this both on the vectorized version of `rnormmv_2` and on the original `rnormmv_2` version using new values of p, since in the original text only a single value of p was tested, and a small one at that, $p = 2$.

3. We know that Wishart variables have mean $\nu\mathbf{\Sigma}$. Present a program to verify whether the expected value, ignoring Monte Carlo error, is achieved using the functions presented in this Chapter to generate Wishart variables. Use 10,000 Monte Carlo replications. This would be a simple, though not conclusive, way of checking whether each function is performing the simulations correctly. It serves at least to indicate the presence of error, but does not guarantee correctness, which is obviously guaranteed by the underlying theory.

4. Implement Python functions to generate multivariate log-normal and elliptical contaminated-normal random variables.

Chapter 5

Algorithms for Means, Variances, and Covariances

Many algorithms for computing means, variances, and covariances are imprecise and may produce final results containing large errors. The need to use efficient algorithms for these simple tasks is evident, and they will be described in this chapter.

5.1 Introduction

Fortunately, Python uses accurate algorithms for computing the mean, variance, and covariance. We will seek to clarify how the use of inefficient algorithms can lead to inconsistent and imprecise results. Our objective in this chapter is to present efficient algorithms for estimating these parameters when the conventional formulas may fail if used directly in the algorithms.

These efficient algorithms are particularly useful when the data have large magnitude or are very close to zero. In this particular case, some spreadsheet programs, such as Excel in its older versions, could fail [McCullough and Wilson, 1999]. Knowledge of algorithms that lead to greater numerical precision can prevent the researcher from making the same mistakes found in some *software*.

5.2 Univariate Algorithms

The basic idea of using algorithms for the mean, for the sum of powers of deviations from the mean, or for the sum of products of deviations is to apply the existing formulas recursively. If we wish, for example, to obtain the mean of n observations, we can initially compute the mean of the first observation, which is the observation itself. In a second stage, we propose an expression to update the mean of the first observation, incorporating the second one, and so on. The same updating procedure is applied to variances or covariances when we have more than one variable.

For a sample of size n given by X_1, X_2, $\cdots$, X_n, the conventional sample mean and variance are obtained, respectively, by:

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

and

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

Some existing algorithms have sought to improve the textbook algorithms, which are the formulas in equations (5.1 and 5.2), by making adaptations that pass through the sample twice. These algorithms can be efficient from the standpoint of precision, but they are not fast, precisely because they pass through the data twice. West [1979] proposed using an algorithm that makes a single pass, updating the mean and variance at each new observation. We can easily show that for a sample of size n, the mean equals X_1 if $n = 1$, $(X_1 + X_2)/2$ if $n = 2$, and so on. At the $(k-1)$-th step we can specify the estimator of the mean as:

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

At the k-th step we will have observed X_k, and the updated mean is:

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

The question we ask is: "can we express the mean at the k-th step as a function of the mean at the $(k-1)$-th step?" The answer to this question is yes, and the result provides the desired algorithm. From equation (5.3) we obtain:

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

resulting in the final recursive equation

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

for $2 \leq k \leq n$, where $\bar{X}_1 = X_1$.

Likewise, if we define the corrected sum of squares of the first k sample observations $1 < k \leq n$ as:

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

we will see that the corresponding variance is given by $S_k^2 = W2_k/(k-1)$. If we expand this expression by isolating the k-th term and simplify the resulting expression, we obtain:

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

The expression we developed (5.5) is equivalent to the one presented by West [1979]. This can be easily demonstrated by substituting $\bar{X}_k$, obtained in equation (5.4), into equation (5.5), from which we obtain:

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

for $2 \leq k \leq n$, where $W2_1 = 0$. The variance is obtained by $S^2 = W2_n/(n-1)$.

To demonstrate expression (5.6) directly, we have:

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

resulting in

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

We can generalize this expression to compute the covariance $S_{(x,y)}$ between a variable X and another variable Y. The expression for the sum of products is given by:

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

for $2 \leq k \leq n$, where $W2_{1,(x,y)} = 0$. The covariance estimator is obtained by $S_{(x,y)} = W2_{n,(x,y)}/(n-1)$.

By analyzing all these expressions, we can observe that to compute the corrected sum of squares and sum of products, we need to compute the means of the variables at the k-th step or at the previous step. The advantage is that in a single pass through the data, we obtain all the estimates. We can further extend these results to obtain sums of the third and fourth powers of deviations from the mean. The expressions we derived for this are:

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

and

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

for $2 \leq k \leq n$, where $W3_1 = 0$ and $W4_1 = 0$.

Thus, we can implement the function `medsq()` that returns the mean, the sum of the square, cube, and fourth power of deviations from the mean, and the variance from a data vector $\mathbf{x}$ of dimension n.

```
# function to return the mean, sums of deviations from the mean
# squared, cubed and to the fourth power, and variance
def medsq(x):
  n = len(x)
  if (n <= 1):
    print('Vector must have more than 1 element!')
    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 = {'mean': xb, 'variance': s2, 'SQ2': w2, 'W3': w3, 'W4': w4}
  return res
# Example
x = [1, 2, 3, 4, 5, 7, 8]
res = medsq(x)
for key in res.keys():
    print(key, ":", res[key])
```

```
mean : 4.285714285714286
variance : 6.571428571428572
SQ2 : 39.42857142857143
W3 : 22.04081632653064
W4 : 391.4518950437319
```

5.3 Algorithms for Mean Vectors and Covariance Matrices

In this section, we present the multivariate extension for obtaining the mean vector and the matrices of sums of squares and products and of covariances. For this reason, we did not implement a specific function to obtain the covariance between two variables X and Y. Let a random sample in $\mathbb{R}^p$ be given by $\mathbf{X}_1, \mathbf{X}_2, \cdots, \mathbf{X}_j, \cdots, \mathbf{X}_n$, where these vectors will be arranged in a matrix $\mathbf{X}$ of dimensions $(n \times p)$. To extend the results of the previous section, we will use the same expressions, taking due care to adapt them to handle matrix and vector operations. Thus, to estimate the population mean vector, instead of using the classical textbook estimator given by

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

we will use the recursive expression given by

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

for $2 \leq k \leq n$, where $\bar{\mathbf{X}}_1 = \mathbf{X}_1$.

Likewise, the adaptation of expression (5.6) to p dimensions is direct, and the result obtained is:

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

for $2 \leq k \leq n$, where $\mathbf{W}_1 = \mathbf{0}$, a matrix of zeros of dimensions $(p \times p)$. The covariance matrix estimator is obtained by $\mathbf{S} = \mathbf{W}_n/(n-1)$.

We implemented the function `medcov()`, presented next, to obtain the mean vector, the matrix of sums of squares and products, and the covariance matrix. The argument of this function must be a matrix of multivariate data with n rows (observations) and p columns (variables). The resulting program and an example are presented below. We chose an example in which we generate a data matrix from a multivariate normal.

```
# function to return the mean vector, the matrix of
# sums of squares and products, and the covariance matrix
import numpy as np

def medcov(x):
  n = x.shape[0]
  p = x.shape[1]
  if (n <= 1):
    print('Matrix must have more than 1 row!')
    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 = {'mean': xb, 'covariance': s, 'SQP': w}
  return res
# Example
```

```
n = 1000
p = 5
x = np.random.multivariate_normal(np.zeros(p),np.identity(p),n)
res = medcov(x)
for keys,values in res.items():
    print(keys)
    print(values)
# compare with the result from our function
print('np covariance: ', np.cov(x, rowvar = False, ddof = 1))
print('np mean: ', np.mean(x, axis = 0))
```

```
mean
[-0.0383097  -0.04331253 -0.00208868 -0.00695855 -0.02672627]
covariance
[[ 0.99000853 -0.03824654 -0.00500518 -0.02360502 -0.02044487]
 [-0.03824654  1.04934623  0.04593988  0.01900796 -0.02128114]
 [-0.00500518  0.04593988  0.96151393  0.00797854 -0.03673669]
 [-0.02360502  0.01900796  0.00797854  1.02883542  0.015017  ]
 [-0.02044487 -0.02128114 -0.03673669  0.015017    0.91951825]]
SQP
[[ 989.01852046  -38.20829807   -5.00017811  -23.58141279  -20.42442684]
 [ -38.20829807 1048.29687912   45.89394281   18.9889527   -21.25985452]
 [  -5.00017811   45.89394281  960.55241454    7.97055899  -36.69994938]
 [ -23.58141279   18.9889527     7.97055899 1027.80658447   15.00198312]
 [ -20.42442684  -21.25985452  -36.69994938   15.00198312  918.59873008]]
np covariance:  [[ 0.99000853 -0.03824654 -0.00500518 -0.02360502 -0.02044487]
 [-0.03824654  1.04934623  0.04593988  0.01900796 -0.02128114]
 [-0.00500518  0.04593988  0.96151393  0.00797854 -0.03673669]
 [-0.02360502  0.01900796  0.00797854  1.02883542  0.015017  ]
 [-0.02044487 -0.02128114 -0.03673669  0.015017    0.91951825]]
np mean:  [-0.0383097  -0.04331253 -0.00208868 -0.00695855 -0.02672627]
```

Fortunately, because Python (`numpy`) uses double precision and employs high-quality algorithms for these tasks, we do not need to worry about implementing functions such as those presented in this chapter. But if we were to use a compiler for the Pascal, Fortran, or C and C^{++} languages, we should make use of these algorithms, since only in this way will we achieve high precision, especially when dealing with data of large magnitude or very close to zero.

5.4 Case Studies

We will consider two cases of instabilities in numerical computation with classical solutions, known as Log-Sum-Exp and Kahan Summation. The Log-Sum-Exp (LSE) trick is used mainly to avoid *overflow* and *underflow* (passively) when computing exponentials, while Kahan summation is used to reduce the accumulated rounding error when adding floating-point numbers.

The LSE trick solves the instability caused by "excessive dynamic range". When dealing with exponential functions, the difference in magnitude between inputs can be enormous, causing intermediate results to exceed the representable range of floating-point numbers.

Kahan summation, in turn, solves the instability caused by "error accumulation". Floating-point operations have rounding errors, and single computation errors may be negligible, but when accumulated over thousands of operations, these small errors amplify, causing the final result to lose all meaning. Algorithms such as Kahan's show that, through careful algorithm design, it is possible to control the error bound so

that it does not grow without limit with the data size.

5.4.1 Log-Sum-Exp (LSE)

Its main function is to safely compute $\log\left(\sum_{i=1}^{n} e^{x_i}\right)$. If computed directly, when x_i is very large, e^{x_i} will cause *overflow*, and when x_i is very small (negative), e^{x_i} will cause *underflow*, resulting in zero.

The central principle is based on exploiting the mathematical identity

$$\log\left(\sum_{i=1}^{n} e^{x_i}\right) = m + \log\left(\sum_{i=1}^{n} e^{x_i - m}\right),$$

where $m = \max(x_i)$. This amounts to a type of normalization of the values, subtracting the maximum value from all exponents, ensuring that the largest input value to the exponential function is 0, thereby completely avoiding *overflow*. LSE ensures that no term suffers *overflow*, and any *underflow* that occurs affects only terms whose contribution is numerically negligible.

This method is extremely common in probabilistic models and machine learning algorithms, especially when working with probabilities in logarithmic space. For example, in the *Softmax* function for classification in neural networks, in neural network loss functions, in Bayesian inference computations, and in the Viterbi algorithm (speech recognition, bioinformatics, natural language processing, digital communications, and GPS).

The stable solution, considering that $m = x_{\max} = \max(x_1, \ldots, x_n)$, is obtained by

$$\begin{aligned} y =& \log\left(\sum_{i=1}^{n} e^{x_i}\right) = \log\left(\sum_{i=1}^{n} e^{x_i - m + m}\right) \\ =& \log\left(e^{m}\right) + \log\left(\sum_{i=1}^{n} e^{x_i - m}\right) \\ =& m + \log\left(\sum_{i=1}^{n} e^{x_i - m}\right). \end{aligned}$$

5.4.2 Kahan Summation (Compensated Summation)

The goal is to reduce the accumulated rounding error when adding a large number of floating-point numbers. When you add a very small number to a very large number, the lower bits of the small number are "absorbed" (lost), which, when accumulated, produces large deviations.

The central principle is to maintain an additional "compensation" variable c. At each iteration, first subtract the compensation from the previous iteration from the value to be added, then perform the addition as usual, and then compute the low-order part lost in this addition (that is, the new compensation value) using the difference between the sum and the original value. In simple terms, it "recovers" and temporarily stores the small values that would be lost, in order to "try again" in the next addition.

The method is applied in scenarios that require high summation precision, such as scientific computing libraries, processing of large datasets (for example, Flink's CompensatedSum), and numerical simulation algorithms that require stable results.

Imagine that we want to add an element x to our current `sum`:

1. $y = x - c$
   $\rightarrow$ Subtracts the accumulated error compensation from the new element x.
2. $t = \text{sum} + y$
   $\rightarrow$ Performs the addition in the main accumulator. This is where precision is lost due to rounding.

3. $c = (t - \text{sum}) - y$
   → Recovers the error. Mathematically, $(t - \text{sum})$ should equal y, but numerically, due to rounding, the difference reveals exactly what was discarded. We store this in c with the sign inverted for the next iteration.

4. $\text{sum} = t$
   → Updates the main accumulator.

   Practical note: NumPy does **not** use Kahan summation in `np.mean`, `np.var`, `np.std`, or `np.cov`. Instead, it uses *pairwise summation* (pairwise/hierarchical summation), which reduces the rounding error from $O(n)$ to $O(\log n)$ and is almost as fast as naive summation. For most statistical cases, this is sufficient. Kahan summation offers even greater precision (error close to $O(1)$), but with a slightly higher computational cost.

5.4.3 Practical Demonstration in Python

The code below runs both scenarios, comparing the naive approach against the numerically stable approach.

```
import numpy as np

print("=== CASE 1: LOG-SUM-EXP ===")
# Large values that cause overflow in the direct computation
x = np.array([1000.0, 1001.0])

# 1. Naive approach
try:
    with np.errstate(over='ignore'):
        lse_naive = np.log(np.sum(np.exp(x)))
except Exception as e:
    lse_naive = "Error in the code"

# 2. Stable approach (LSE)
x_max = np.max(x)
lse_stable = x_max + np.log(np.sum(np.exp(x - x_max)))

print(f"Naive LSE: {lse_naive} (Invalid result due to overflow of e^1000)")
print(f"Stable LSE: {lse_stable}")

print("\n=== CASE 2: KAHAN SUMMATION ===")
# Creating 1,000,000 elements equal to 0.1 using float32
# The exact expected value of the sum is 100,000.00
values = np.ones(1_000_000, dtype=np.float32) * 0.1

# 1. Traditional sum
sum_traditional = np.float32(0.0)
for v in values:
    sum_traditional += v

# 2. Kahan summation
sum_kahan = np.float32(0.0)
c = np.float32(0.0)  # Error compensation variable
for v in values:
    y = v - c            # Subtracts the previous compensation from the current element
    t = sum_kahan + y    # Sums into the main accumulator (rounding occurs here)
```

```
    c = (t - sum_kahan) - y  # Recovers the actual loss of precision
    sum_kahan = t        # Updates the accumulator

print(f"Expected sum:      100000.00")
print(f"Traditional sum: {sum_traditional:.2f} (Erored due to accumulated rounding)")
print(f"Kahan sum:       {sum_kahan:.2f} (Hit the value with precision)")
```

```
=== CASE 1: LOG-SUM-EXP ===
Naive LSE: inf (Invalid result due to overflow of e^1000)
Stable LSE: 1001.3132616875182

=== CASE 2: KAHAN SUMMATION ===
Expected sum:      100000.00
Traditional sum: 100958.34 (Erored due to accumulated rounding)
Kahan sum:       100000.00 (Hit the value with precision)
```

5.5 Exercises

1. Show the numerical equivalence between the following expressions

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

   and

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

   To do this, implement both functions in `Python` and perform a numerical comparison (a single example is enough).

2. Implement a function in Python to obtain the sum of products, equation (5.7), and the covariance between the n observations of two variables whose values are arranged in two vectors $\mathbf{X}$ and $\mathbf{Y}$. Create a matrix with n rows and $p = 2$ columns from some example and use the function `medcov()` to compare the results obtained.

3. The sample skewness and kurtosis coefficients are functions of the sums of powers of deviations from the mean. The skewness coefficient is given by $\sqrt{b_1} = (W3_n/n)/(W2_n/n)^{3/2}$, and the kurtosis coefficient is $b_2 = (W4_n/n)/(W2_n/n)^2$. Implement a Python function using `medsq()` to estimate the univariate skewness and kurtosis coefficients.

4. Use a sample from an area of your specialty and determine the mean, variance, sum of squares, sum of cubed deviations, and sum of fourth-power deviations. Also determine the skewness and kurtosis coefficients.

Chapter 6

Approximation of Distributions

Algorithms for obtaining probabilities have been the subject of research for many decades and still are. The importance of having algorithms that are both fast and accurate is indescribable. The main difficulty is that most probabilistic models do not have an explicitly known distribution function. Thus, sophisticated numerical methods are required to compute probabilities. Another aspect is the need to invert distribution functions to obtain quantiles of the random variable for a known cumulative probability. In hypothesis tests and in interval and region estimation procedures, we almost always use these algorithms indirectly without realizing it.

In this chapter, we will introduce these concepts and present some basic ideas of general methods for performing the necessary quadratures. Particular numerical methods will be addressed for a few models. We will also address the discrete and continuous cases separately. To conclude, we will present the main pre-existing Python functions for a series of probabilistic models.

6.1 Introduction

We do not have many concerns about obtaining probabilities and quantiles when using Python, since most probabilistic models and special functions are already implemented. Our objective goes beyond this fact, as our intention is to seek the knowledge necessary to go further and to understand how a given function operates and what its capabilities and limitations are.

We will begin our discussion with the exponential distribution to draw attention to the main difficulty in this process. Thus, we chose this model precisely because it does not present such difficulties. Considering a random variable X with exponential distribution with parameter λ, we have the density function

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

and the distribution function

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

For this probabilistic model, we can compute probabilities using the distribution function (6.2) and obtain quantiles with the inverse distribution function given by:

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

We implemented, in Python, the density, distribution, and inverse distribution functions and named them `dexp`, `pexp`, and `qexp`, respectively. These functions are:

```
import numpy as np

# exponential density function
# f(x) = lamb * exp(-lamb * x)
def dexp(x, lamb = 1):
  x = np.atleast_1d(np.asarray(x, dtype=float))
  fx = np.zeros(len(x))
  if any(x < 0):
    fx[x >= 0] = lamb * np.exp(-lamb * x[x >= 0])
    return fx
  else:
    fx = lamb * np.exp(-lamb * x)
    return fx


# exponential distribution function
# F(x) = 1 - exp(-lamb * x)
def pexp(x, lamb = 1):
  x = np.atleast_1d(np.asarray(x, dtype=float))
  Fx = np.zeros(len(x))
  if any(x < 0):
    Fx[x >= 0] = 1 - np.exp(-lamb * x[x >= 0])
    return Fx
  else:
    Fx = 1.0 - np.exp(-lamb * x)
    return Fx

# inverse exponential distribution function
# q = -log(1 - p) / lamb
def qexp(p, lamb = 1):
  p = np.atleast_1d(np.asarray(p, dtype=float))
  q = np.zeros(len(p))
  if any(p >= 1) or any(p < 0):
    q[(p >= 0) & (p < 1)] = -np.log(1 - p[(p >= 0) & (p < 1)]) / lamb
    mask_invalid = (p >= 1) | (p < 0)
    q[mask_invalid] = np.nan
    print('NaN produced!')
    return q
  else:
    q = -np.log(1 - p) / lamb
    return q

# Example
lamb = 0.1
x = np.array([29.95732, -0.03])
print('pdf of negative values: ',dexp(x, lamb))
print('cdf of negative values: ',pexp(x, lamb))
p = np.array([-0.5, 0.95, 1.0])
print('icdf of improper values: ',qexp(p, lamb))
x = 29.95732
p  = 0.95
print('pdf: ',dexp(x, lamb))
```

```
print('cdf: ',pexp(x, lamb))
print('icdf: ',qexp(p, lamb))
```

```
pdf of negative values:  [0.005 0.   ]
cdf of negative values:  [0.94999999 0.        ]
NaN produced!
icdf of improper values:  [        nan 29.95732274         nan]
pdf:  [0.005]
cdf:  [0.94999999]
icdf:  [29.95732274]
```

In the case where the user enters scalar values (integers or *float*) or lists of values, the previous functions convert them into *NumPy* lists using the stacked methods `np.atleast_1d(np.asarray(x, dtype=float))` and `np.atleast_1d(np.asarray(p, dtype=float))`. If the argument is a scalar, the `np.asarray` method converts it into a `0`-dimensional array of *float* type, and the other method, `np.atleast_1d`, converts it into an array with at least `one` dimension (*NumPy* list). The advantage of this conversion method is that it does not create additional copies of the argument in memory if it is already a list. The other method, which we will present in the next function, is `np.array(x, ndmin=1, dtype=float)`, which performs the same conversion much more quickly but duplicates the variable `x` in memory. The `ndmin=1` option ensures that the number of dimensions of the created *float* array is at least `1`. In some cases, efficiency in terms of processing speed should be chosen; in others, memory efficiency is the right option.

Another concern we should have is regarding possible incorrect user inputs in the function arguments. We have not had many concerns so far about possible errors in calls to our functions with incorrect arguments. To illustrate, let us consider only the exponential probability density function. In this case, if $\lambda > 0$ and the values of x are positive or zero, we have no problem. If, on the other hand, the values of x are negative, then the density value should be zero. This kind of concern existed in the implementation of the previous *dexp* function. However, we may have zero or negative values of the parameter λ used incorrectly in the function call, which is not a valid procedure, and no protection against this action was made. Thus, it is prudent that some problems be anticipated and consistent results be returned. In this case, the result *nan* should be returned, since the density function is not valid under these conditions. Furthermore, the *lamb* argument of the function is considered a scalar, but the x argument can be a vector. We can also handle this case by treating both arguments as vectors, even if we do not change the scalar value of the default *lamb* argument, and by adjusting their dimensions if they have different sizes. The following program illustrates this and uses *np.resize* to perform this adjustment.

```
import numpy as np

def dexp(x, lamb=1):
    # Convert to arrays and ensure they are floats
    x_arr = np.array(x, ndmin=1, dtype=float)
    lamb_arr = np.array(lamb, ndmin=1, dtype=float)

    # Expand so that they have the same size
    size = max(len(x_arr), len(lamb_arr))

    x_expanded = np.resize(x_arr, size)
    lamb_expanded = np.resize(lamb_arr, size)

    # Initialize result with NaN to handle invalid cases
    result = np.full(size, np.nan, dtype=float)
```

```
    # Mask for valid lambda values (>= zero)
    valid_lambda = lamb_expanded >= 0

    # Mask for valid x values (non-negative)
    valid_x = x_expanded >= 0

    # Combined mask: x >= 0 AND lambda >= 0
    valid_mask = valid_x & valid_lambda

    # Compute the density only for valid cases
    result[valid_mask] = (lamb_expanded[valid_mask]*
        np.exp(-lamb_expanded[valid_mask]*x_expanded[valid_mask]))

    # For x < 0 WITH valid lambda: result is 0
    # For x < 0 WITH invalid lambda: keeps NaN (already initialized)
    result[(x_expanded < 0) & valid_lambda] = 0.0

    # Return appropriate format
    if np.isscalar(x) and np.isscalar(lamb):
        return float(result[0])
    else:
        return result

# Tests
print("=== Tests of the dexp function ===")
# Case 1: positive lambda (normal)
print("\n1. Positive lambda:")
print(f"dexp(1, 1) = {dexp(1, 1):.4f}")

# Case 2: lambda zero
print("\n2. Lambda zero:")
print(f"dexp(1, 0) = {dexp(1, 0)}")

# Case 3: negative lambda (invalid)
print("\n3. Negative lambda:")
print(f"dexp(1, -1) = {dexp(1, -1)}")

# Case 4: mixture of valid and invalid lambdas
print("\n4. Mixture of lambdas:")
x = [1, 2, 3]
lambdas = [0.5, 0, -1]
result = dexp(x, lambdas)
print(f"x = {x}")
print(f"lambdas = {lambdas}")
print(f"result = {result}")

# Case 5: negative x (returns 0, except with invalid lambda)
print("\n5. Negative x:")
print(f"dexp(-1, 0) = {dexp(-1, 0)}")
print(f"dexp(-1, -1) = {dexp(-1, -1)}")
```

```
# Case 6: vectors with recycling
print("\n6. Recycling with invalid lambda:")
x = [1, 2, -3, -4]
lamb = [0.5, -0.5]
result = dexp(x, lamb)
print(f"x = {x}")
print(f"lamb = {lamb}")
print(f"result = {result}")

# Case 7: check of correct calculation
print("\n7. Check of expected values:")
test_cases = [
        (0, 1, 1.0),           # f(0) = lambda
        (1, 1, np.exp(-1)),    # f(1) = e^-1
        (2, 0.5, 0.5 * np.exp(-1))  # f(2) = 0.5 * e^-1
    ]


for x_val, lamb_val, expected in test_cases:
    result = dexp(x_val, lamb_val)
    print(f"dexp({x_val}, {lamb_val}) = \
          {result:.4f} (expected: {expected:.4f})")
```

```
=== Tests of the dexp function ===

1. Positive lambda:
dexp(1, 1) = 0.3679

2. Lambda zero:
dexp(1, 0) = 0.0

3. Negative lambda:
dexp(1, -1) = nan

4. Mixture of lambdas:
x = [1, 2, 3]
lambdas = [0.5, 0, -1]
result = [0.30326533 0.                nan]

5. Negative x:
dexp(-1, 0) = 0.0
dexp(-1, -1) = nan

6. Recycling with invalid lambda:
x = [1, 2, -3, -4]
lamb = [0.5, -0.5]
result = [0.30326533        nan 0.                nan]

7. Check of expected values:
dexp(0, 1) =           1.0000 (expected: 1.0000)
dexp(1, 1) =           0.3679 (expected: 0.3679)
dexp(2, 0.5) =           0.1839 (expected: 0.1839)
```

In this case, this function automatically resizes the vectors by repetition to equalize their sizes before the element-wise computation. This occurs with the use of *np.resize*, after the maximum size relative to the sizes of both vectors is obtained. If the sizes are different, an incomplete recycling of the smaller vector is performed through cyclic repetition, truncating the result when it reaches the size of the larger vector, without completing the last cycle. This means that a cyclic recycling of the smaller vector is performed, interrupting the repetition when it reaches the exact size of the larger vector, even if this means leaving the last cycle incomplete. After adjusting the vector sizes, a *result* vector is created for the results with *nan*. Thus, if there are invalid values in the argument inputs, a combined *boolean* mask of non-negative x values and zero or positive *lamb* values is obtained. The density is computed only where these two conditions occur simultaneously, and the *nan* values of *result* are replaced by the density values. The problem is that if x is negative and *lamb* is positive, the density value should be zero and is not computed using the previous mask. But if x is negative and *lambda* is invalid, the previously defined *nan* values in *result* are kept. By definition, the probability density function exists for all reals, provided that the model parameters are correctly defined. The program makes a specific final adjustment for the case where x is negative but *lamb* is positive, with the command *result[(x_expanded < 0) & valid_lambda] = 0.0* (see case 6).

The *Python* functions for the exponential were easily implemented, since we can explicitly obtain the distribution function and its inverse. If, on the other hand, we had the normal model

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

we could not explicitly obtain the distribution function, much less the inverse of the probability distribution function. Since for the vast majority of probabilistic models we encounter the same problems, we need numerical methods to obtain these functions. For this reason, we will present some details in this chapter about some general and specific methods for certain models. In the discrete case, we can use sequential recursive algorithms. However, since most discrete probabilistic models have an exact relationship with continuous models, we encounter the same problems when using these relationships.

6.2 Discrete Probabilistic Models

We will present algorithms for obtaining the probability function, the probability distribution function, and its inverse for the discrete models we consider most important: the binomial and Poisson models. We will begin our discussion with the binomial model. Considering a random variable X with binomial distribution, its probability function is given by:

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

where n is the sample size or number of independent Bernoulli trials with success probability p, and x is the number of successes, which belongs to the finite set 0, 1, $\cdots$, n. The probability distribution function is given by:

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

We saw in chapter 3, equation (3.14), that we can obtain the cumulative probabilities recursively. Given $P(X = 0) = (1-p)^n$, we obtain the probabilities for the other values of X, that is, for $x = 1, \cdots, n$, recursively using the relation $P(X = x) = P(X = x-1)[(n-x+1)/x][p/(1-p)]$. We can summarize it as follows:

$$P(X = x) = P(X = x-1)\left(\frac{g}{x} - r\right),$$

where $r = p/(1-p)$ and $g = r(n+1)$, for $x = 2, 3, \cdots, n$.

This same algorithm, appropriately modified, is used to obtain the cumulative probabilities and the inverse of the distribution function. If the values of n and p are large, this algorithm may be inefficient. For the case where the parameter p is large (close to one), we can use the binomial property given by: if $X \sim Bin(n,p)$, then $Y = n - X \sim Bin(n, 1-p)$. Thus, we can, for example, obtain $P(X = x)$ equivalently by $P(Y = n - x)$ and $P(X \leq x) = P(Y \geq n - x)$, which can be rewritten as $F_X(x) = 1 - F_Y(n - x - 1)$, except for $x = n$, where $F_X(x) = 1$. In this way, we switch variables to perform the computation and then return the value corresponding to the original event.

```
# probability function and distribution
# of the binomial(n, p)
def dpbinom(x, n, p = 0.5):
  if p < 0 or p > 1:
    return {'pdf': float('nan'),'cdf': float('nan')}
  if p == 0:
    if x < 0:
      return {'pdf': 0,'cdf': 0}
    elif x >= 0:
      if x == 0:
        f = 1
      else:
        f = 0
      return {'pdf': f,'cdf': 1}
  elif p == 1:
    if x < n:
      return {'pdf': 0,'cdf': 0}
    elif x >= n:
      if x == n:
        f = 1
      else:
        f = 0
      return {'pdf': f,'cdf': 1}
  if p > 0.5:
    pp = 1 - p
    xx = n - x
  else:
    pp = p
    xx = x
  qq = 1 - pp
  if (xx < 0 or xx > n):
    f = 0
    if xx < 0:
      cdf = 1
    else:
      cdf = 0
    return {'pdf': f,'cdf': cdf}
  else:
    f  = qq**n
  r = pp / qq
  g = r * (n + 1)
  cdf = f
  u = 0
```

```
  while(u < xx):
    u += 1
    f *= (g / u - r)
    cdf += f
  if p > 0.5:
      cdf = 1 - cdf + f # 1-F(n-x) and not 1-F(n-x-1) was computed
  return {'pdf': f,'cdf': cdf}

# inverse of the distribution function
# binomial(n, p)
def qbinom(prob, n, p):
  if prob < 0 or prob > 1:
    return float("nan")
  if prob == 1:
    return n
  if prob == 0:
    return -1
  if p == 1:
    return n
  if p == 0:
    return 0
  q = 1 - p
  f = q**n
  r = p / q
  g = r * (n + 1)
  x = 0
  cdf = f
  while (prob - cdf) >= 1.e-14:
    x += 1
    f *= (g / x - r)
    cdf += f
  return x

# Example of use
p = 0.5713131
x = 5 # remove the tolerance, qbinom fails
n = 20
res = dpbinom(x, n, p)
for keys,values in res.items():
    print(keys)
    print(values)
prob = res['cdf']
print('inverse CDF',qbinom(prob, n, p))
```

```
pdf
0.0028636362867051975
cdf
0.0036697111708624904
inverse CDF 5
```

If we use the property that if X is binomial with parameters n and p, then $n - X$ is also binomial with parameters n and $1 - p$. Since the algorithm is more efficient when $p < 0.5$, because $q = 1 - p$ is closer

to 1 and the execution speed of the algorithm is proportional to np. Thus, we gain processing time and precision if $p > 0.5$ and we perform the switch to $n - X$. The following `script` presents this modification in the `qbinom` function:

```
# inverse of the distribution function
# binomial(n, p), with switch if p > 0.5
def qqbinom(prob, n, p):
  if prob < 0 or prob > 1:
    return float("nan")
  if prob == 1:
    return n
  if prob == 0:
    return -1
  if p == 1:
    return n
  if p == 0:
    return 0

  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

# Example of use
p = 0.5713131
x = 5 # remove the tolerance, qbinom fails
n = 20
res = dpbinom(x, n, p)
print(res)
prob = res['cdf']
print('inverse CDF',qqbinom(prob, n, p))
```

```
{'pdf': 0.0028636362867051975, 'cdf': 0.0036697111708624904}
inverse CDF 5
```

The binomial distribution function has an exact relationship with the beta distribution function. If we have algorithms to obtain the incomplete beta, we can use this relationship to obtain probabilities of the binomial distribution. The binomial distribution function, $F(x)$, relates to the incomplete beta function, $Ip(\alpha,\beta)$, as follows:

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

Thus, we can anticipate the importance of knowing numerical integration algorithms for continuous distributions. This becomes even more evident when we realize that for large values of n the recursive algorithms are inefficient and may become inaccurate. In Section 6.3 we will present some general integration methods for continuous functions. In the following `script`, we use relation (6.7) to obtain the binomial distribution function.

```
# probability function and distribution of the
# binomial(n, p) from the relationship with the
# incomplete beta function - beta cdf
import numpy as np
from scipy.stats import beta, binom

def pbinombeta(x, n, p = 0.5):
  if p < 0 or p > 1:
    print('p must be in the interval [0, 1]')
    return float("nan")
  x = np.array(x, ndmin=1, dtype=int) # converts to integers
  cdf = np.full(len(x), 0, dtype=float)
  x_0_n_1 = (x >= 0) & (x < n)
  x_maior_igual_n = x >= n
  a =  x[x_0_n_1] + 1
  b  = n - x[x_0_n_1]
  cdf[x_0_n_1] = 1 - beta.cdf(p, a, b)
  cdf[x_maior_igual_n] = 1
  return cdf

# Example
p = 0.5713131
n = 20
x = [-1, 0, 5, 7.7, 9, 20, 21]
print("Values:            ", x)
print("Relationship with beta: ", pbinombeta(x, n, p))
print("SciPy:              ", binom.cdf(x, n, p))
```

```
Values:             [-1, 0, 5, 7.7, 9, 20, 21]
Relationship with beta:  [0.00000000e+00 4.39343542e-08 3.66971117e-03 3.86412187e-02
 1.91577173e-01 1.00000000e+00 1.00000000e+00]
SciPy:               [0.00000000e+00 4.39343542e-08 3.66971117e-03 3.86412187e-02
 1.91577173e-01 1.00000000e+00 1.00000000e+00]
```

The Poisson distribution is the second one we will consider. There is a relationship between the Poisson distribution function and the incomplete gamma. If a discrete random variable X with values $x = 0, 1, 2, 3, \cdots$ has a Poisson distribution with parameter λ, then we can define the probability function as:

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

We can obtain probabilities from this distribution using the relationship of the incomplete gamma with the Poisson distribution function

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

where

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

is the incomplete gamma function, for values of $x \geq 0$. The `script` for this is

```
import numpy as np
from scipy.stats import gamma, poisson

# probability function and distribution of the
# Poisson(lamb) using the relationship with the
# incomplete gamma function - gamma cdf
def pgama(x, lamb = 1.0):
  if lamb < 0:
    print('lambda must be greater than or equal to 0')
    return float("nan")
  x = np.array(x, ndmin=1, dtype=int) # converts to integers
  x_maior_0 = x >= 0
  cdf = np.full(len(x), 0, dtype=float)
  a =  x[x_maior_0] + 1.0
  cdf[x_maior_0] = 1 - gamma.cdf(lamb, a)
  return cdf

# Example
lamb = 0.85
x = [-1, 0, 3.0, 100.0]
print("Values:            ", x)
print("Relationship with gamma: ",pgama(x, lamb))
print("SciPy:              ",poisson.cdf(x, lamb))
```

```
Values:             [-1, 0, 3.0, 100.0]
Relationship with gamma:  [0.         0.42741493 0.98886897 1.        ]
SciPy:               [0.         0.42741493 0.98886897 1.        ]
```

6.3 Continuous Probabilistic Models

To obtain the distribution function or the inverse of the distribution function of continuous probabilistic models, as a rule, we must use numerical methods. There are exceptions, such as the exponential model described at the beginning of this chapter. The vast majority of probabilistic models used today make use of algorithms specially developed to perform quadratures in each particular case. These algorithms are, in general, more precise than the quadrature methods, as numerical integration procedures are called. There are several numerical quadrature methods, such as Simpson's method, Gaussian quadratures, and Monte Carlo methods. We will first present the Simpson and Monte Carlo methods, which are simpler and less precise.

We will use the normal model to exemplify the use of these general integration methods in continuous probability functions. A random variable X with normal distribution with mean μ and variance σ^2 has density function given by:

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

The probability distribution function cannot be obtained explicitly, and the is defined by:

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

In general, we use the standard normal to apply the quadratures. In this case, the mean is $\mu = 0$ and the variance is $\sigma^2 = 1$. Thus, we frequently denote the density by $\phi(z)$, the distribution function by $\Phi(z)$, and its inverse by $\Phi^{-1}(p)$, where $0 < p < 1$. The variable Z is obtained from a linear transformation of a normal variable X by $Z = (X - \mu)/\sigma$.

We will present, very briefly, the extended trapezoidal rule for performing function quadratures. Let f_i be the value of the target function at the point x_i, that is, $f_i = f(x_i)$. Then the trapezoidal rule is given by:

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

where $h = x_n - x_1$ and the error term $O(h^3 f'')$ means that the true answer differs from the estimate by an amount that is the product of h^3 and the value of the second derivative of the function evaluated at some point in the integration interval determined by x_1 and x_n, with $x_1 < x_n$.

This equation returns very imprecise values for the quadratures of most functions of interest in statistics. But if we use this function $n - 1$ times to integrate over the intervals (x_1, x_2), (x_2, x_3), $\cdots$, (x_{n-1}, x_n) and sum the results, we obtain the composite formula or extended trapezoidal rule:

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

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

One of the best ways to implement the trapezoidal function is discussed and presented by Press et al. [1992]. In this implementation, the average of the function at its integration endpoints x_1 and x_n is initially taken. Successive refinements are performed. In the first stage, we must add the value of the function evaluated at the midpoint of the integration limits, and in the second stage, the intermediate points 1/4 and 3/4 are inserted, and so on.

We can implement this routine in an iterative, dynamic, and recursive manner:

1. We compute the initial value of the integral S considering the endpoints, as follows:

$$S = \frac{(b-a)}{2}\left(f(a) + f(b)\right).$$

2. At step $n = 2$, we update S by inserting a value $x = a + (b - a)/2$ at the midpoint between a and b. We compute $f(x)$ and store it in *sum*. We define *it* as the number of points to be inserted between a and b by $it = 2^{n-2}$, where $n = 2$ indicates that we are at the second step. We update S by

$$S = \frac{1}{2}\left(S + (b-a) \times sum/it\right). \tag{6.13}$$

3. We set $n = n + 1$ and repeat step 2, updating S from its previous value computed in the previous step using Equation 6.13. For each value of n, we must compare the values from the previous and current steps until a given precision is reached.

Remarks

- Note that sum in step 2 depends on it. For example, if $n = 3$, it $= 2^1 = 2$. We must compute $\Delta = (b - a)/\text{it}$, and the two points to be inserted are $x_1 = a + \Delta/2$ and $x_2 = x_1 + \Delta$. Thus, sum $= f(x_1) + f(x_2)$.
- At any point of the iterative process for a given n, we have it $= 2^{n-2}$ points, $\Delta = (b - a)/\text{it}$, $x_1 = a + \Delta/2$, and $x_k = x_{k-1} + \Delta$, $k = 2, 3, \ldots, \text{it}$. Thus, sum $= \sum_{k=1}^{\text{it}} f(x_k)$.

Let `func()` be the function of interest, a and b the integration limits, and n the previously defined number of integration intervals. Then we can obtain the function `trapzd()` by adapting the same function implemented in Fortran by Press et al. [1992] as follows:

```
# This routine computes the n-th refinement stage of the
# extended trapezoidal integration, where func is an
# external function of interest that must be called for
# n=1, 2, etc., and the value of s must be returned by
# the function on each new call.
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   # space between points
    x   = a + 0.5 * dl
    soma = 0.0
    for j in range(it):
      soma += func(x)
      x += dl
    # refines the value of s
    s = 0.5 * (s + (b - a) * soma / it)
  return s
# function to perform quadratures of
# functions defined in func()
# until a given precision
# has been reached
def qtrap(func, a, b):
  eps  = 1.0e-11
  nmax = 30
  olds = -1.e30 # impossible value for the initial quadrature
  n    = 1
  fim  = 0
  s = -1.0e15  # arbitrary value to start the loop
  while abs(s-olds) >= eps * abs(olds) and fim == 0:
    olds = s
    s = trapzd(func, a, b, s, n)
    n = n  + 1
    if n > nmax:
      fim  = 1
  if n > nmax:
```

```
    print('Step limit exceeded!')
    return
  return s
```

We must call the `qtrap()` function by specifying the function of interest `func()` and the integration limits. Thus, for the standard normal, we must use the following functions:

```
# pdf of the standard normal
import scipy.stats as sps

def dnorm(x):
  fx = (1.0 / (2.0 * np.pi)**0.5) * np.exp(-x**2 / 2)
  return fx

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

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

```
0.9750021048512256
0.9750021048517795
```

It is clear that we have general numerical methods more efficient than the one presented. The Gaussian quadratures represent a class of methods that are usually more efficient than the one presented. Eventually, there are specific methods for obtaining the integrals of the main probabilistic models that are more efficient. The greater efficiency of these specific methods is due to two aspects: processing speed and precision. If they exist, these specific routines are already implemented in Python. As an illustration, we can replace the function we implemented `pnorm()` with the pre-existing Python `normal.cdf()` from `scipy.stats`. We will illustrate one of these specialized algorithms for obtaining the standard normal distribution function. The Hasting algorithm has a maximum error of 1×10^{-6} and is given by:

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

where G is given by

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

where

$$\eta = \frac{1}{1 + 0.2316418|x|}$$

and $a_1 = 0.319381530$, $a_2 = -0.356563782$, $a_3 = 1.781477937$, $a_4 = -1.821255978$, and $a_5 = 1.330274429$.

The result of implementing this procedure is:

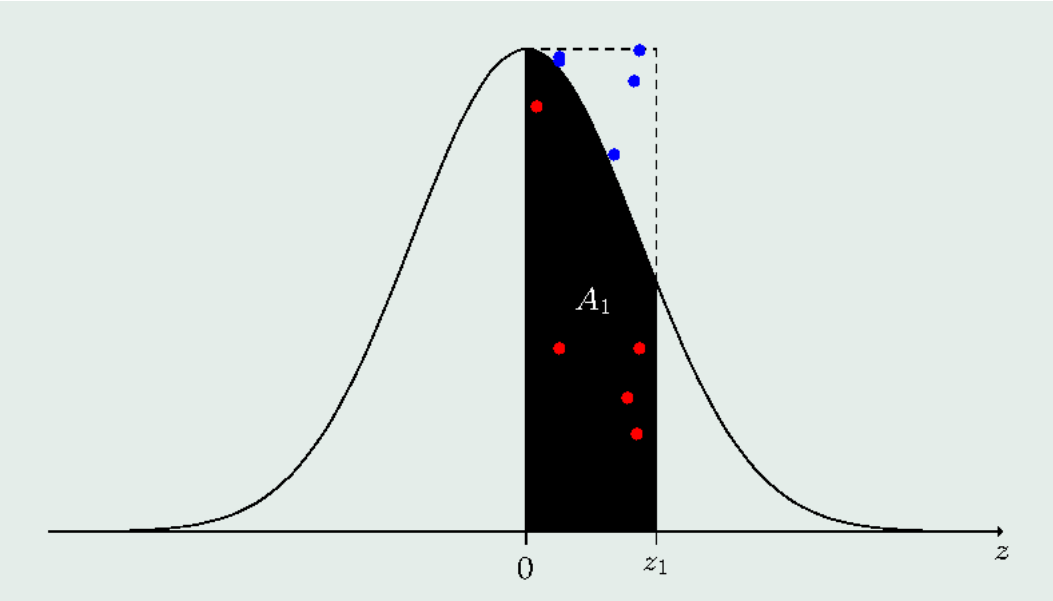
```
# CDF of the standard normal:
# Hasting approximation
def phnorm(x):
  eta = 1 / (1 + abs(x) * 0.2316418)
  a1  = 0.319381530; a2 = -0.356563782
  a3 = 1.781477937; a4 = -1.821255978
  a5  = 1.330274429
  phi = 1 / (2 * np.pi)**0.5 * np.exp(-x * x / 2)
  g = (a1*eta+a2*eta**2+a3*eta**3+a4*eta**4+a5*eta**5)*phi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

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

```
np.float64(0.9750021668514388)
```

With the use of this function, we gain in precision, especially for large absolute values of the upper integration limit, and especially we gain in processing time. We can also address the Monte Carlo methods, which are especially useful for integrating complex and multidimensional functions. We will present only a rather rudimentary version of this method. The idea is to determine a rectangle that encloses the function we wish to integrate and to bombard the region with random points (u_1, u_2) from the uniform distribution.

We count the number of points under the function and determine the corresponding area, based on the proportionality between this number of points and the total number of simulated points in relation to the area under the function in the region of interest relative to the total area of the rectangle. If we know the maximum of the function f_{max}, we can determine this rectangle completely. Thus, the rectangle of interest is defined by the base (value between 0 and z_1 in absolute value, with z_1 provided by the user) and by the height (value of the density at the maximum point). Thus, the area of this rectangle is $A = |z_1| f_{max}$. In the case of the standard normal, the maximum obtained for $z = 0$ is $f_{max} = 1/\sqrt{2\pi}$, and the area of the rectangle is $A = |z_1|/\sqrt{2\pi}$. If the area under the curve we wish to estimate is defined by A_1, we can generate uniform numbers u_1 between 0 and $|z_1|$ and uniform numbers u_2 between 0 and f_{max}. For each generated value u_1, we compute the density $f_1 = f(u_1)$. Thus, the ratio between the areas A_1/A is proportional to the ratio n/N, where n represents the number of points (u_1, u_2) for which $u_2 \leq f_1$ and N the total number of generated points. Hence, the integral is obtained by $A_1 = |z_1| \times f_{max} \times n/N$, where z_1 is the value of the standard normal for which we wish to compute the area between 0 and $|z_1|$, in order to obtain the distribution function at the point z_1, that is, to obtain $\Phi(z_1)$. Thus, if $z_1 \leq 0$, then $\Phi(z_1) = 0.5 - A_1$, and if $z_1 > 0$, then $\Phi(z_1) = 0.5 + A_1$. See the following illustrative figure:



For the particular case of the standard normal, we implemented the following function:

```
# Quadrature of the standard normal
# via Monte Carlo simulation
import numpy as np

def mcpnorm(x, N = 2000):
  max = 1 / (2 * np.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 * np.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)

# example
z = 1.96
N = 1500000
print('MC Prob: ', mcpnorm(z, N))
print('MC Error: ', 1 / N**0.5)
```

```
MC Prob:  0.9753119713540519
MC Error:  0.0008164965809277261
```

Another way to obtain an approximation of the integral

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

by Monte Carlo is to generate m uniform numbers between 0 and $abs(z)$, say z_1, z_2, ..., z_m, and obtain

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

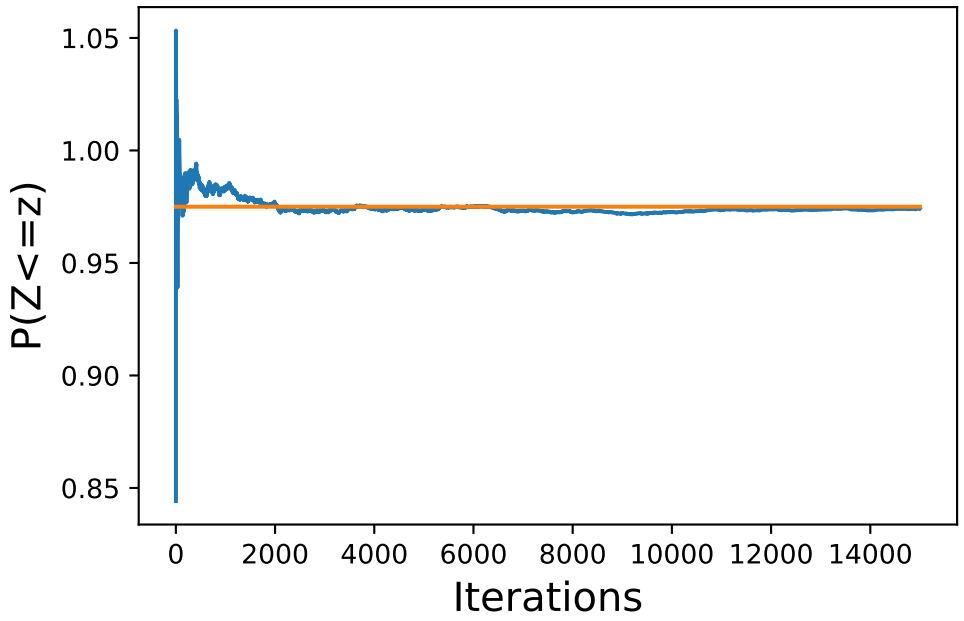
where $\phi(t) = 1/\sqrt{2\pi} \times \exp\{-t^2/2\}$ is the standard normal density function evaluated at the point t, and $\Delta z = abs(z) - 0$. The order of error of this process is given by $O(m^{-1/2})$. The Python program for obtaining the value of the standard normal distribution function $\Phi(z)$ using these ideas is presented below. By comparing this Monte Carlo alternative with the first one, we can verify that there was a great decrease in the Monte Carlo error in computing this integral in this new approach. Many variants and improvements in this process can be implemented, but we will not discuss them here.

```
# Quadrature of the standard normal via
# Monte Carlo simulation. Second way to
# obtain the integral: classical form
import numpy as np
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 * np.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

# Example
m = 15000   # number of points much smaller than in the previous case
z = 1.96
print('MC Prob: ', pmcnorm(z, m)) # MC estimate
p = sps.norm.cdf(z)       # true value
print('sps Prob: ', p)
# Order of error: O(m^(-1/2))
print('MC Error: ', 1/m**0.5)
# convergence check
np.random.seed(1000)
x = np.random.uniform(0, abs(z), m)
pdf = (1 / (2 * np.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('Iterations', fontsize=15)
plt.ylabel('P(Z<=z)', fontsize=15)
plt.show()
```

```
MC Prob:  0.9722940388350266
sps Prob:  0.9750021048517795
MC Error:  0.00816496580927726
```



6.4 Gaussian Quadratures

Gaussian quadratures play a preponderant role in statistics, since the main algorithms for obtaining probabilities and quantiles use such methods implicitly. In this section, we will present, in a rather simplified way and without delving into the more technical mathematical aspects, the main Gaussian quadrature method. Those interested in a more detailed treatment of these aspects can consult numerous specific books. We recommend, for example, reading Quarteroni et al. [2000].

The basis of these quadratures is interpolation based on orthogonal polynomials. Their use goes beyond the topic of numerical quadratures and can also be used to approximate least squares solutions and numerical differentiation. An n-point Gaussian quadrature is one that provides exact results for the integration of a polynomial of degree equal to or less than $2n-1$. The domain of the quadratures is, without loss of generality and by convention, assumed to be $[-1,1]$. The general rule for the n-point Gaussian quadrature of a function $f(x)$ on the domain $[-1,1]$ is given by

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

where x_i and w_i are called the nodes and weights, respectively, of the quadrature.

The nodes are values in the interval $[-1,1]$, and the weights are positive. We must choose them appropriately so that the result of the integral is accurately approximated by the sum on the right. We often use a weight function $w(x_i)$, such that the integral (6.15) can be represented by

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

where $g(x)$ is such that $f(x) = w(x)g(x)$, and w_i in this case are alternative weights.

For example, for the specific case of the function $f(x) = e^{2x}$, using weights $w_1 = w_2 = 1$ and nodes $x_1 = \sqrt{3}/3$ and $x_2 = -\sqrt{3}/3$, on the domain from -1 to 1, results in

$$\int_{-1}^{1} e^{2x} dx \approx f(\sqrt{3}/3) + f(-\sqrt{3}/3) = 3.4882.$$

The exact value of the integral is

$$\int_{-1}^{1} e^{2x} dx = \left[\frac{e^{2x}}{2}\right]_{-1}^{1} = 3.6269.$$

It is surprising that the sum $f(x_1) + f(x_2)$ yields exact values for polynomials of degree up to 3, $2n-1$, since $n = 2$. What we must do is find mechanisms to obtain the integration weights and nodes. For an integration interval different from $[-1, 1]$, that is, $[a, b]$, $b > a$, the following transformation does not alter the integration precision:

$$\begin{aligned} \int_a^b f(x)dx = & \quad \frac{b-a}{2} \int_{-1}^{1} f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right) dx \\ \approx & \quad \frac{b-a}{2} \sum_{i=1}^{n} w_i f\left(\frac{b-a}{2}x_i + \frac{a+b}{2}\right). \end{aligned}$$

It is worth emphasizing that the nodes used in the quadratures are the roots of the orthogonal polynomials of some of the commonly used Gaussian families. Among them, we can mention the Legendre, Laguerre, Hermite, Chebyshev, and Jacobi polynomials. We will describe only the Gauss-Legendre quadrature next.

The quadrature called Gauss-Legendre is used for integration intervals defined by $[-1, 1]$ and can be extended to more general intervals if we use expression (6.17). Thus, for the domain $[-1, 1]$, the Gauss-Legendre quadrature can be applied by

$$\int_a^b f(x)dx \approx \begin{cases} \sum_{i=1}^{n} w_i f(x_i) & \text{for } a = -1 \text{ and } b = 1 \\ \sum_{i=1}^{n} w_i g(y_i) & \text{for } a \text{ and } b \text{ any reals,} \end{cases} \tag{6.17}$$

where $g(y)$ is given by

$$g(y) = \frac{b-a}{2} f\left(\frac{b-a}{2}x + \frac{a+b}{2}\right), \tag{6.18}$$

where $y = (b-a)x/2 + (b+a)/2$.

The next step, in applying Gaussian quadratures, is to compute the nodes and weights for each of them. We will present this as a sequence of steps. We will use the following quantities for all methods. The scalar μ_0, defined by

$$\mu_0 = \int_a^b w(x)dx$$

and the coefficient vectors $\mathbf{a}$ $(n \times 1)$ and $\mathbf{b}$ $(n-1 \times 1)$, where n is the number of points of the quadrature to be performed.

Subsequently, we will also use a matrix denoted by $\mathbf{J}$, $(n \times n)$, for which we will obtain the eigenvalues and eigenvectors. Since we will use an algebra problem solved by obtaining eigenvalues and eigenvectors,

this method is known as the Golub–Welsch algorithm. We will initially describe how to obtain the vectors $\mathbf{a}$ and $\mathbf{b}$ and the constant μ_0 for the Gauss-Legendre quadrature method. Technical details of how this can be done and the theoretical foundations can be found in Quarteroni et al. [2000].

For the Gauss-Legendre quadrature we have:

1. set $\mu_0 = 2$;
2. $\mathbf{a} = \mathbf{0}$ $(n \times 1)$;
3. $\mathbf{b} = [b_j]$ $(n - 1 \times 1)$, such that

$$b_j = \frac{j}{\sqrt{4j^2 - 1}}, \qquad j = 1, 2, \ldots, n - 1.$$

Once the necessary quantities $\mathbf{a}$, $\mathbf{b}$, and μ_0 are defined, we must apply the second part of the algorithm to obtain the nodes x_i's and the weights w_i's, $i = 1, 2, \ldots, n$. For this, we use the Golub–Welsch algorithm, by which we determine the matrix $\mathbf{J}$ from these quantities and determine its eigenvalues and eigenvectors. The matrix $\mathbf{J}$ is a symmetric tridiagonal matrix defined by

$$\mathbf{J} = \begin{bmatrix} a_1 & b_1 & 0 & \ldots & \ldots & \ldots \\ b_1 & a_2 & b_2 & 0 & \ldots & \ldots \\ 0 & b_2 & a_3 & b_3 & 0 & \ldots \\ 0 & \ldots & \ldots & \ldots & \ldots & 0 \\ \ldots & \ldots & 0 & b_{n-2} & a_{n-1} & b_{n-1} \\ \ldots & \ldots & \ldots & 0 & b_{n-1} & a_n \end{bmatrix}.$$

We determine the eigenvalues and eigenvectors of $\mathbf{J}$ and denote them by

$$\mathbf{x}^* = \begin{bmatrix} x_1^* \\ x_2^* \\ \vdots \\ x_n^* \end{bmatrix} \qquad \text{and} \qquad \mathbf{P} = \begin{bmatrix} p_{11} & p_{21} & \cdots & p_{n1} \\ p_{12} & p_{22} & \cdots & p_{n2} \\ \vdots & \vdots & \ddots & \vdots \\ p_{1n} & p_{2n} & \cdots & p_{nn} \end{bmatrix},$$

respectively.

We modify the usual notation for representing the elements of a matrix. Thus, we verify that in matrix $\mathbf{P}$, the first index of the element p_{ij} refers to the index of the i-th eigenvector, and the second index refers to the j-th element of that eigenvector. This is why the unusual index notation in this matrix. Therefore, each column of $\mathbf{P}$ corresponds to one of the n eigenvectors associated with each of the eigenvalues x_i^* sorted in decreasing order.

Next, we can obtain the nodes (points) by simply taking the eigenvalues in increasing order, that is, the vector of nodes $\mathbf{x}$, with components x_i, is defined by

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} x_n^* \\ x_{n-1}^* \\ \vdots \\ x_1^* \end{bmatrix}, \tag{6.19}$$

where x_i^* is the i-th eigenvalue of the tridiagonal matrix $\mathbf{J}$.

We can verify that the nodes are the eigenvalues of this matrix presented in reverse order, that is, in increasing order rather than the usual way of presenting them, which is in decreasing order. Finally, we

can obtain the weights as a function of μ_0 and the first elements of each eigenvector of $\mathbf{J}$, also considered in reverse order, considering the classical ordering from largest to smallest eigenvalue. This function seeks to perform an appropriate normalization to ensure the validity of the quadrature.

Thus, we have that the quadrature weights are obtained by

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_{n-1} \\ w_n \end{bmatrix} = \mu_0 \begin{bmatrix} p_{n,1}^2 \\ p_{n-1,1}^2 \\ p_{n-2,1}^2 \\ \vdots \\ p_{2,1}^2 \\ p_{1,1}^2 \end{bmatrix}, \tag{6.20}$$

where p_{ij} is the j-th element of the i-th eigenvector of $\mathbf{J}$ corresponding to the i-th eigenvalue x_i^*.

We can observe that the values w_i's depend on p_{ij} in reverse order of importance, this order being determined by the eigenvalues of $\mathbf{J}$.

With the weights, nodes, and weight functions, when the latter is not unity, we can apply the quadrature methods presented earlier. Our next step is to implement functions to obtain the weights and nodes and to provide an example.

```
# Function to obtain nodes and weights
# of the Gauss-Legendre quadrature
import numpy as np
import scipy as sp


def gausslegendquad(n):
  n = int(n)
  if n < 0:
    print('A non-negative number of nodes is required!')
    return
  if n == 0:
    return {'xi':[], 'wi': []}
  i = np.arange(1, n)
  mu0 = 2
  b = i / np.sqrt(4 * i**2 - 1)
  J = np.zeros(n * n)
  J[(n + 1) * (i-1) + 1] = b
  J[(n + 1) * i - 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}

# generic function to obtain the
# quadrature on [-1,1]
def quadgl(func, n = 8):
  gl = gausslegendquad(n)
  res = np.sum(gl['wi'] * func(gl['xi']))
  return res
```

```
# example: f(x) = exp(2x)
# int_{-1}^{1} f(x) dx - vectorized
def e2x(x):
  return np.exp(2*x)

# polynomial function
def p3(x):
  return x**3-0.4*x**2+2.3*x+5
# Examples
n = 4
gl = gausslegendquad(n)
print('Nodes:   ', gl['xi'])
print('Weights: ', gl['wi'])
n = 8
print('e^(2x): ',quadgl(e2x, n))
print('Exact value: ',(e2x(1)-e2x(-1))/2)
print('p3: ',quadgl(p3, 2))
print('Exact value wolfram: ', 9.73333)
```

```
Nodes:    [-0.86113631 -0.33998104  0.33998104  0.86113631]
Weights:  [0.34785485 0.65214515 0.65214515 0.34785485]
e^(2x):  3.626860407846865
Exact value:  3.626860407847019
p3:  9.73333333333333
Exact value wolfram:  9.73333
```

In the previous example, we used the `sp.linalg.eigh()` function from `scipy.linalg`, since the `np.linalg.eig()` function does not necessarily return the eigenvalues in order. The `eigh` function, in turn, returns the eigenvalues sorted, but in increasing order rather than decreasing order, as is done in many other programs. We can apply the quadratures to the normal to illustrate the cases where the limits are not -1 and 1, including the previous examples.

```
import scipy as sp

# generic function to change the
# integration limits from -1 to 1
# to a and b (finite)
def hx(func, x, a = -1, b = 1):
  aux = (b - a) / 2
  h = aux * func(aux * x + (a + b) / 2)
  return h

# modification of quadGL to allow
# limits other than -1 and 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)
def pnormal(x, mu = 0, sigma = 1, n = 16):
  if np.isclose(x, mu):
```

```
    return 0.5
  else:
    if np.isclose(mu, 0) and np.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

# integrate e2x from 1 to 3
a = 1
b = 3
print('Quadrature y=e^(2x): ', quadglab(e2x, 8, a, b))
print('Exact value: ',(e2x(b)-e2x(a))/2)
print('Quad. p3: ', quadglab(p3, 2, a, b)) # exact value 35.73333
x = 1.96
print('Quad. normal: ', pnormal(x))
print('Exact value: ', sp.stats.norm.cdf(x))
```

```
Quadrature y=e^(2x):  198.01986869689384
Exact value:  198.01986869690222
Quad. p3:  35.73333333333333
Quad. normal:  0.9750021048517794
Exact value:  0.9750021048517795
```

In the previous quadratures, we can replace our function that computes the nodes and weights with another one that uses a more efficient method for obtaining eigenvalues and eigenvectors. This method is applicable to symmetric tridiagonal matrices. The following `script` presents this implementation. We only need to provide a vector with the diagonal of the matrix and another with the elements of the two secondary diagonals, which are identical. The function used was `sp.linalg.eigh_tridiagonal()` from the `scipy` library.

```
# Function to obtain nodes and weights
# of the Gauss-Legendre quadrature
# This version avoids creating
# the matrix J, by applying a
# process for obtaining
# eigenvalues and eigenvectors on
# tridiagonal matrices - faster
import numpy as np
import scipy as sp

def gausslegendquad2(n):
  n = int(n)
  if n < 0:
    print('A non-negative number of nodes is required!')
    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
gl = gausslegendquad2(n)
print('Nodes:   ', gl['xi'])
print('Weights: ', gl['wi'])
```

```
Nodes:    [-0.86113631 -0.33998104  0.33998104  0.86113631]
Weights:  [0.34785485 0.65214515 0.65214515 0.34785485]
```

6.5 Newton-Raphson

The Newton-Raphson method is used to obtain the solution of a linear or non-linear equation given by $f(x) = 0$. From an initial value x_0, we can draw a tangent line at the point $(x_0, f(x_0))$. The slope of this line is given by the derivative of f evaluated at x_0. Let $m = f'(x_0)$ be the slope of the tangent line at the point $(x_0, f(x_0))$.

Thus,

$$\begin{aligned} m =& \frac{\Delta y}{\Delta x} \\ \Delta y =& m\Delta x \\ y - y_0 =& f'(x_0)(x - x_0) \\ y =& y_0 + f'(x_0)(x - x_0). \end{aligned}$$

Hence,

$$f(x) = f(x_0) + f'(x_0)(x - x_0).$$

Solving the equation for $f(x) = 0$, we have:

$$\begin{aligned} f(x) =& 0 \\ f(x_0) + f'(x_0)(x - x_0) &= 0 \\ x &= x_0 - \frac{f(x_0)}{f'(x_0)} \end{aligned}$$

For any point x_n of the n-th repetition of this process, we have

$$x_{n+1} = x_n - \frac{f(x_n)}{ft(x_n)},$$

which is the expression for solving a linear or non-linear equation by Newton-Raphson. In Figure 6.1 we present an example in which we perform three iterations to illustrate, in search of the solution of the presented equation.

In our particular case, we will present the Newton-Raphson numerical method to obtain the solution of the equation:

$$z = \Phi^{-1}(p),$$

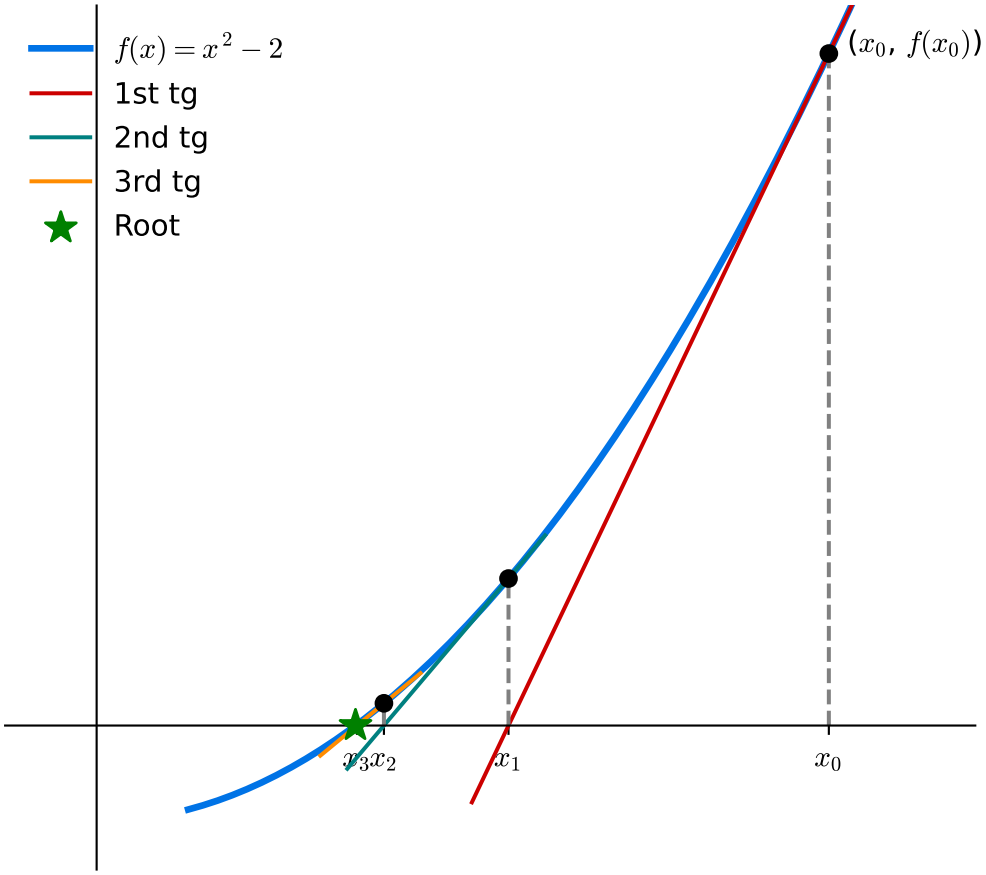


Figure 6.1: Construction of the tangent lines in the Newton-Raphson method for $f(x) = x^2 - 2$.

where $0 < p < 1$ is the value of the standard normal distribution function, $\Phi^{-1}(p)$ is the inverse of the standard normal distribution function at the point p, and z is the corresponding quantile we wish to find given a value of p. We can present this problem by means of the following equation

$$\Phi(z) - p = 0,$$

where $\Phi(z)$ is the standard normal distribution function evaluated at z.

Thus, our objective is to find the zeros of the function $f(z) = \Phi(z) - p$. In general, we can solve this equation numerically using the Newton-Raphson method, which is an iterative process. Thus, we must have an initial value for the quantile to start the process, and at the $(n+1)$-th step of the iterative process we can update the value of the quantile by

$$z_{n+1} = z_n - \frac{f(z_n)}{f'(z_n)}, \tag{6.21}$$

where $f'(z_n)$ is the first-order derivative of the function for which we wish to obtain the roots, evaluated at the point z_n. For this particular case, we have that

$$z_{n+1} = z_n - \frac{[\Phi(z_n) - p]}{\phi(z_n)}, \tag{6.22}$$

where $\phi(z_n)$ is the standard normal density function. As initial values, we will use a rough approximation, since our objective is only to demonstrate the method. Thus, if p is less than 0.5, we will use $z_0 = -0.1$; if $p > 0.5$, we will use $z_0 = 0.1$. Obviously, if $p = 0$, the function should return $z = 0$. The function *qPhi* below returns the normal quantiles, given the values of p between 0 and 1, the real mean μ, and the positive real variance σ^2, using the Newton-Raphson method and the Hasting `phnorm` function to obtain the value of the standard normal distribution function.

```
# auxiliary function to return the value of the
# standard normal density function
def phi(z):
  return (1 / (2 * np.pi)**0.5 * np.exp(-z * z / 2))

# Newton-Raphson method to obtain quantiles
# of the normal distribution with real mean mu and positive
# real standard deviation sig, given 0 < p < 1
# uses the functions phi(z) and  phnorm(z),
# presented earlier
def qphi(p, mu = 0, sig = 1, func = phnorm):
  eps = 1e-11
  if p <= 0 or p >= 1:
    print('The value of p must be between 0 and 1!')
    return
  if sig  <= 0:
    print('The standard deviation must be greater than 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}

# Example
p   = 0.975
mu  = 0
sig = 1
print('normal q Hasting: ', qphi(p, mu, sig, phnorm))
print('normal q Gauss-Legendre: ', qphi(p, mu, sig, pnormal))
print('Exact value: ', sps.norm.ppf(p, mu, sig)) #for comparison
```

```
normal q Hasting:  {'x': np.float64(1.9599629237446643), 'iter': 8}
normal q Gauss-Legendre:  {'x': np.float64(1.9599639845400563), 'iter': 8}
Exact value:  1.959963984540054
```

We could also have used the secant method, since it does not require the first-order derivative but needs two initial values to start the process and has slower convergence. The reader is encouraged to consult Press et al. [1992] for more details. The `pnorm` function, which uses the trapezoidal method, could also have been used. In that case, the precision would be higher, but the processing time would also be higher. To do this, it would suffice to use the `pnorm` function as the value of the `func` argument in the call. It would also be possible to call `mcpnorm` or `pmcnorm` or any other efficient implementation we have for the quadrature.

Fortunately, Python also has pre-programmed routines for this and many other probabilistic models, which relieves us of the need to program routines to obtain the distribution functions and inverse distribution functions of the most varied existing probabilistic models.

6.6 Pre-Existing Functions in Python

In Table Table 3.1 we present a good portion of the routines for generating data from the most varied probabilistic models covered by Python. Shortly after the mentioned table, we present the Python procedures from the `scipy.stats` library to obtain the probability or density function, the distribution function, and the quantile function. We should consult the resources of this library to learn about the most different methods associated with each of its objects.

6.7 Exercises

1. Compare the precision of the three algorithms for obtaining the normal distribution function presented in this chapter: `pnorm`, which uses the trapezoidal method; `phnorm`, which uses the Hasting method; and `pnormal`, based on the Gauss-Legendre Gaussian quadrature. Use the `norm.cdf()` function from `SciPy Stats` as a reference and compute the absolute and relative errors for a vector of supplied quantiles. percentiles.

2. Use different numbers of simulations in the two Monte Carlo methods presented to integrate the

normal distribution function and estimate the maximum relative and absolute Monte Carlo error over 30 replications for each size. Use the quantiles 1.00, 1.645, and 1.96 and the `norm.cdf()` function from `SciPy Stats` as a reference.

3. The Cauchy distribution is a particular case of Student's t with $\nu = 1$ degree of freedom. The Cauchy density is given by:

$$f(x) = \frac{1}{\pi(1 + x^2)}$$

Use the trapezoidal method presented here to implement the computation of the values of the Cauchy distribution function. We can also obtain the distribution function and its inverse analytically. Obtain such functions and implement them in Python. Use the pre-existing `scipy.stats` functions to check the results obtained, which are expected to be exact for the analytical resolution of the distribution function and its inverse. For the trapezoidal method, compute the absolute and relative errors for some quantiles.

4. Use the most efficient Monte Carlo method described in this chapter to obtain values of the Cauchy distribution function presented in exercise 3. Use some numerical values to illustrate and compare with functions implemented in Python for this case. What should be the minimum number of Monte Carlo simulations required to have reasonable precision?

Chapter 7

Sets and Elements of Combinatorial Analysis in Python

In this chapter, we intend to present the basic concepts of combinatorial analysis and sets in a very simple way. We will build some simple functions for computing probabilities computationally in a few particular cases. We saw in chapter 1 the Python `set` objects. The `set()` operator is used to create a set. The argument of the `set()` function is a list or a tuple. They are immutable, unordered, and have no duplicate elements.

Several functions can be used for this type of object in Python, such as membership (`in`), union (`union` or `|`), intersection (`intersection` or `&`), and symmetric difference (`symmetric_difference` or `^`), as illustrated in chapter 1. It is also possible to obtain differences, such as `A-B`, between two sets `A` and `B`.

Combinatorial analysis allows us to solve countless probability problems. Problems that commonly occur are listed below:

a. selecting x elements among n elements, $0 \leq x \leq n$, without repeating any element (sampling without replacement) where order does not matter: combination;
b. selecting x elements among n elements, $0 \leq x \leq n$, allowing repetition of the selected elements (sampling without replacement) where order matters: arrangements;
c. distributing n elements into x, $x = n$, positions in different ways: permutations;
d. different possible paths to follow with n_1 possibilities at the first stage, n_2 possibilities at the second stage, and so on up to n_k possibilities at the k-th stage: counting.

Thus, we want to obtain all possible combinations, permutations, arrangements, or paths (countings) possible in solving some probability problem or another situation in general.

7.1 Introduction to Combinatorial Analysis in Python

Combinatorial analysis and counting methods are essential to understanding probability. In Python we can compute the number of combinations given by

$$\binom{n}{x} = \frac{n!}{x!(n-x)!}, \text{ for } 0 \leq x \leq n,$$

by `math.comb(n, x)`. The listing of the combinations of n taken x at a time is obtained by the command `combinations(range(1,n+1), x)` from the `itertools` package, which must be imported. The first argument of the `combinations` function is a list, which in this case was a list ranging from 1 to n, where

we used the `range` function to obtain this list. We can, instead, use any other list. We can also use the `comb` function from `scipy.special` to obtain the number of combinations in an alternative way.

The command `combinations(n, x)` from the `itertools` package returns an object that must be converted to a list with $\binom{n}{x}$ elements, where each element of the list corresponds to a tuple of x elements (the combination).

The following `script` illustrates a particular case of these commands:

```
from itertools import combinations
from scipy import special as sps
import math

n = 4
x = 2
print('Comb. n, x (math):', math.comb(n, x))
print('Comb. n, x (sp):', sps.comb(n, x)) # result is float
comb = list(combinations(range(1,n+1), x))
print('Combinations: \n', comb)
# listing combination 0 of the list
print('Comb 0: ', comb[0])
print('N. comb.: ', len(comb)) # number of combinations
# returns a combination of letters
print(list(combinations(['a','b','c'], 2)))
```

```
Comb. n, x (math): 6
Comb. n, x (sp): 6.0
Combinations:
 [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
Comb 0:  (1, 2)
N. comb.:  6
[('a', 'b'), ('a', 'c'), ('b', 'c')]
```

7.2 Permutations and Arrangements

The total number of permutations of n objects is given by $n!$, and with the `itertools` package we can list them in a list of size $n!$, where each element is a tuple of size n, using the `permutations()` function;

We can also take the arrangements of n taken x at a time with the same command, whose number of arrangements is given by

$$A_{n,x} = \frac{n!}{(n-x)!},$$

where in Python this value is obtained through the function `math.perm(n, x)`.

The following `script` illustrates these commands:

```
# Obtain all permutations of n
# in n positions - permutation
# or in x positions x<n, arrangement
from itertools import permutations
import math

n = 3
# permutations
```

```
perm = list(permutations(range(1,n+1)))
print(perm)
# arrangements
x = 2 # x < n
print('N. combinations: ', math.perm(n, x))
arran = list(permutations(range(1,n+1), x))
print('Permutations\n', arran)
# Prints the permutations
for i in list(perm):
  print(i)
tipo = str(type(perm[0])).replace('<', '[').replace('>', ']')
print(tipo)
```

```
[(1, 2, 3), (1, 3, 2), (2, 1, 3), (2, 3, 1), (3, 1, 2), (3, 2, 1)]
N. combinations:  6
Permutations
 [(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)
[class 'tuple']
```

To randomly draw permutations or arrangements, the command `random.sample(range(1,n+1),n)` returns a permutation of n taken n at a time. If we want a draw of n taken x at a time, `random.sample(range(1,n+1),x)`, that is, an arrangement resulting in x elements without repetition (sample without replacement). It is not about the case in question (permutations and arrangements), but if we want sampling with replacement, as required in the *bootstrap* methods, we use the function `random.choices(range(1,n+1),x)`, where x is a value between `1` and `n`. Both commands require us to import the Python `random` library. The following `script` illustrates these commands:

```
# Obtain random samples of
# permutations or arrangements
# (without replacement), or samples
# with replacement
import random

n = 5
# samples of permutations
amostra = random.sample(range(1,n+1), n)
print('Sample: \n', amostra)
x = 3 # x < n
# samples of arrangements
amostra = random.sample(range(1,n+1), x)
print('Sample: \n', amostra)
# samples with replacement
x =  3
amos = random.choices(range(1,n+1), k=x)
print('Sample: \n', amos)
```

```
Sample:
```

```
 [5, 4, 1, 2, 3]
Sample:
 [1, 2, 4]
Sample:
 [1, 3, 4]
```

7.3 Counting

The fundamental counting principle, known as the multiplicative principle, is used to find the number of possibilities for an event consisting of k stages. The stages must be successive and independent. If an event has two ($k = 2$) stages, where the first has n_1 possibilities and the second has n_2 possibilities, then there are $n_1 \times n_2$ possibilities.

The fundamental counting principle is the multiplication of the options of each stage to determine the total number of possibilities. This concept is important for combinatorial analysis, which is an area of mathematics that brings together methods for solving problems including counting, and is therefore very useful in scientific research to determine the probabilities of natural phenomena.

To obtain all possibilities in k stages, each with the same number n of possibilities (n^k), we created a Python class with some functions. The function `contag(n, k)` returns all (n^k) countings from n possibilities in k stages, such as, for example, with $n = 2$ sexes of animals in $k = 3$ births, or $n = 2$ doors of a building in $k = 2$ possibilities (entrances and exits), corresponding to the entrances and exits of the building. We also did the same when we have a list (any iterator) of size k corresponding to the k stages in which each element refers to the number of possibilities of that stage. If x is the iterator, the total countings (possibilities) are $\prod_{i=0}^{k-1} x[i]$.

The question, for example, of in how many different ways a person can enter and leave a building with 2 entrances and two exits, is what we intend to list (enumerate). The proposed function uses an iterative process controlled by the `while`, which at each iteration updates the particular count. The `permuta` or `permutav` functions are called for the case of n quantities in k stages and for the case of a list x of stages, respectively.

```
import numpy as np

class Cont:

  def permuta(self, z, pos, permn, n, k, prim):
    while np.sum(z[1:(k+1)]) != n * k:
      if prim:
        z = np.full(k + 1, 1)
        prim = False
        permn = np.append(permn, [z[1:(k+1)]], axis=0)
      else:
        if z[pos] < n:
          z[pos] = z[pos] + 1
          permn = np.append(permn, [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 and i >= 1 and i <= k:
            if z[i] < n:
              z[i] = z[i] + 1
              if paratras:
                z[(i+1):(k+1)] = 1
              permn = np.append(permn, [z[1:(k+1)]], axis=0)
              pos = k
              nachei = False
            else:
              if pos == k:
                i = i - 1
              else:
                i = i + 1
  return permn

def permutav(self, z, pos, permn, x, k, prim):
  while np.sum(z[1:(k+1)]) != sum(x):
    if prim:
      k = len(x)
      pos = k
      z = np.full(k + 1, 1)
      prim = False
      permn = np.append(permn, [z[1:(k+1)]], axis=0)
    else:
      if z[pos] < x[pos-1]:
        z[pos] = z[pos] + 1
        permn = np.append(permn, [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 and i >= 1 and i <= k:
          if z[i] < x[i-1]:
            z[i] = z[i] + 1
            if paratras:
              z[(i+1):(k+1)] = 1
              permn = np.append(permn, [z[1:(k+1)]], axis=0)
              pos = k
              nachei = False
          else:
            if pos == k:
              i = i - 1
            else:
              i = i + 1
  return permn

def contag(self, *args):
  if len(args) == 2 and isinstance(args[0], (int, np.integer)) \
```

```
                        and isinstance(args[1], (int, np.integer)):
      x, k = args
      prim = True
      permn = np.empty((0, k), int)
      z = np.array([])
      pos = k
      n = x
      return self.permuta(z, pos, permn, n, k, prim)

    elif len(args) == 1 and isinstance(args[0], (list, tuple, np.ndarray)):
      x = args[0]
      k = len(x)
      pos = k
      prim = True
      permn = np.empty((0, k), int)
      z = np.array([])
      return self.permutav(z, pos, permn, x, k, prim)

    else:
      raise TypeError(
        "contag accepts (int, int) or (list). "
        f"Received: {[type(a).__name__ for a in args]}"
      )


# Example of use
res = Cont()
n = 3
k = 3
val = res.contag(n, k)
print('Arrangements or permutations of n taken k at a time.\n', val)
x = [3, 4, 2]
print('Countings of the levels of x.\n', res.contag(x))
```

```
Arrangements or permutations of n taken k at a time.
 [[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]]
Countings of the levels of x.
 [[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]]
```

In the current `script` we have several novelties. For the first time in the course, we created a Python class. In this class we implemented four functions. The first two are intended to perform the listings of all countings in an iterative process. The next two have the same name and represent versions of the same function with different arguments, denoted by `contag`. For this we used the `overloading` technique.

The overload was implemented manually, without external dependencies. Instead of decorators, the `contag` function receives a variable number of arguments (`*args`) and checks, at the beginning of the body, how many and of what type these arguments are. If it receives two integers, it executes the `permuta` method (countings of `n` taken `k` at a time); if it receives a list (or tuple/array), it executes the `permutav` method (countings of the levels of a vector `x`); otherwise, it raises an explanatory `TypeError`. Thus, the same calling interface is maintained — `contag(n, k)` and `contag(x)` — and the interpreter chooses the appropriate option at runtime.

To illustrate, we created an object of the `Cont` class, the object `res`, through which we call the `contag` method twice with different arguments — first with two integers and then with a list — which causes, when executed, the interpreter to select the appropriate version and produce the corresponding outputs.

To illustrate, we created an object of the `Cont` class, the object `res`, through which we call the `contag`

method twice with different arguments — first with two integers and then with a list — which causes, when executed, the interpreter to select the appropriate version and produce the corresponding outputs.

If, for example, we wanted to obtain all possible sex-of-birth possibilities for families with up to 4 children, we would have the following sample space for our experiment with $2^4 = 16$ elements. The subtraction of 1 from all elements of the result of the call to the `contag` method is because the `sexo` vector has two levels, and in `Python` lists start at zero rather than one. The elements of the `contag` result are in this case composed of numbers 1 and 2 (female and male), whose positions in the `sexo` vector are 0 and 1, respectively.

```
# Sample space of children
# by sex
# with n = 4 and 0 <= x <= n
filhos = Cont()
n = 4
res = filhos.contag(2, n) -  1 # -1 for compatibility
sexo = ['F','M']               # with the indexes of the sexo list
tipo = str(type(res[0])).replace('<', '[').replace('>', ']')
print('Type of res[0]: ', tipo)
print('Sample space of 4 births \
         by sex:\n', np.array(sexo)[res])
```

```
Type of res[0]:  [class 'numpy.ndarray']
Sample space of 4 births          by sex:
 [['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']]
```

Python already has these resources implemented for counting. The `itertools` library has the `product` function, which plays the same role as our functions implemented in the `cont` class.

```
import itertools

l1 = ['F','M']
print('Sample space of 4 births by sex:\n',
       np.asarray(list(itertools.product(l1, repeat=4))))
n = 3
l2 = list(range(1, n+1))
print('Permutations of n, n!\n',
      np.asarray(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())
print('List of vectors l3:\n', l3)
result = itertools.product(*l3) # *l3: takes the product of the components of l3
print('Countings from the \
         levels of array l3:\n',np.asarray(list(result)))
```

```
Sample space of 4 births by sex:
 [['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']]
Permutations of n, n!
 [[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]]
List of vectors l3:
 [[1, 2, 3], [1, 2, 3, 4], [1, 2]]
Countings from the          levels of array l3:
 [[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]]
```

7.4 Sets in Python

Let us recall the main functions of the `set` objects, sets in Python, so that we can perform some simple operations. Sets in Python are a collection of elements with no occurrence of values with multiplicities. Sets are created by the `set` method or by using braces. Thus, `x = set([1,2,3])` or `x={1,2,3}` is a set, but `y= [1,2,3,3]` is not, because it has repeated elements. If we consider the command `y= set([1,2,3,3])`, this object would be a set, since Python automatically eliminates the repeated elements when creating the set.

The arguments of the `set` function are the lists or the tuples. We can efficiently use sets to remove duplicate elements from a list or tuple. First, we apply the `set()` operator with the argument being a list or a tuple and then apply the `list()` or `tuple()` argument to the result. Another advantage of sets is that they can have elements of different types. Sets are immutable, because their elements cannot be exchanged, but we can add and remove elements from a set in Python. Furthermore, sets are unordered. Let us see an example of how to create a set.

```
# Create sets in Python
A = {'Laranja', 'Maçã', 'Pera'}
B = set(('Laranja'))
C = set(('Laranja',))
D = set((1,2,3,'Frutas'))
print('A: ', A)
print('B: ', B)
```

```
print('C: ', C)
print('D: ', D)
```

```
A:  {'Maçã', 'Pera', 'Laranja'}
B:  {'r', 'L', 'a', 'j', 'n'}
C:  {'Laranja'}
D:  {1, 2, 3, 'Frutas'}
```

We can add or remove elements from a set. The elements that are arguments of the `set.add()` or `set.update()` functions must be immutable, such as tuples or strings, for the first method, and any iterator object, such as lists, for the second. If the argument is a list, an error will occur in the `add` method. To remove an element from a set, we can use methods such as `set.remove()`, `set.discard()`, or `set.pop()`. The first command removes a specific element from the set and returns an error if the element does not exist. The second removes the element but does not raise an error if the element does not exist, leaving the set as it was before its application. The third removes an element randomly from the set and has no argument. The `pop` method returns the removed element and updates (modifies) the set to which it was applied. The `add` and `update` methods differ in that the first does not support adding a list to the set. The `set.clear()` method removes all elements from the set. See the `script` with some examples:

```
# adding and removing elements
# from a set
x = {1,2,3,4}
y = [4,5,6,7]
x.update(y)
print('original x: ', x)
x.add(8)
print('x with 8 added: ', x)
x.remove(8)
print('x with 8 removed: ', x)
x.discard(9) # does not raise an error because 9 is not in x
print('x unchanged because it has no 9: ', x)
x.pop()
print('x with one of the elements removed: ', x)
```

```
original x:  {1, 2, 3, 4, 5, 6, 7}
x with 8 added:  {1, 2, 3, 4, 5, 6, 7, 8}
x with 8 removed:  {1, 2, 3, 4, 5, 6, 7}
x unchanged because it has no 9:  {1, 2, 3, 4, 5, 6, 7}

1

x with one of the elements removed:  {2, 3, 4, 5, 6, 7}
```

Python includes some set operations, some of which are:

- `set.union(x, y)`: union of sets x and y, which corresponds to the set of all elements present in x and in y, considering only once those elements with multiplicity greater than 1;
- `set.intersect(x,y)`: intersection of sets x and y, which corresponds to the set of elements present simultaneously in x and y;
- `set.difference(x,y)`: set of the difference between sets x and y, consisting of the set of all elements of x that are not in y;
- `x == y` or `x != y`: tests whether two sets x and y are equal or different, respectively;

- `c in y`: membership, that is, tests whether c is an element of set y;
- `set.symmetric_difference(x, y)`: the set of elements that are either in x or in y, but not in both simultaneously.

```
# set operations
# illustrative
y = set([1,2,3,3])
x = {3,4,5}
print('x U y: ', set.union(x,y))
print('x U y: ', x.union(y))
print('original x: ',  x) # does not modify x
print('x & y: ', set.intersection(x,y))
print('x - y: ', set.difference(x,y))
print('x equals y: ', x == y)
print('x is different from y: ', x != y)
t = tuple(set((1,2,3,3))) # removing duplicates from a tuple
print('Tuple in set eliminates duplicates: ', t)
tipo = str(type(t)).replace('<', '[').replace('>', ']')
print('Type of t: ', tipo)
y = set([1,2,3])
x = {3,4,5}
print('(x-y) U (y-x): ', set.symmetric_difference(y,x))
```

```
x U y:  {1, 2, 3, 4, 5}
x U y:  {1, 2, 3, 4, 5}
original x:  {3, 4, 5}
x & y:  {3}
x - y:  {4, 5}
x equals y:  False
x is different from y:  True
Tuple in set eliminates duplicates:  (1, 2, 3)
Type of t:  [class 'tuple']
(x-y) U (y-x):  {1, 2, 4, 5}
```

Other basic set operations are the following:

- `y.issubset(x)`: returns `True` if y is contained in or equal to set x;
- `x.issuperset(y)`: returns `True` if x contains or is equal to set y;
- `x.isdisjoint(y)`: returns `True` if x and y have no elements in common, that is, if the intersection is an empty set.

```
# other set operations
x = {1,2,3,4,5}
y = {1, 3, 5}
# y is contained in x
print('y is contained in x: ', y.issubset(x))
# x contains y
print('x contains y: ', x.issuperset(y))
# whether x and y are disjoint
print('x and y are disjoint: ', x.isdisjoint(y))
```

```
y is contained in x:  True
x contains y:  True
x and y are disjoint:  False
```

7.5 Some Probability Problems

We will present some particular cases, such as the birthday coincidence case. For birthday coincidence, let us consider the coincidence of births of a group of n people on the same date and that the year has 365 days. We also assume that the distribution of births over the birthdays' years is uniformly random, which is a strong assumption. The probability that no two people have the same birth date (event A) in a group of n people is:

$$P(A) = \frac{A_{365,n}}{365^n} = \frac{365!}{(365-n)!365^n}.$$

We want the probability of the complementary event, which is given by $P(A^c) = 1 - P(A)$. Hence,

$$P(A^c) = 1 - \frac{A_{365,n}}{365^n} = 1 - \frac{365!}{(365-n)!365^n},$$

which is the probability of there being at least one birthday coincidence on the same date.

We implemented two ways. In the first, we use the complete formula. We take the logarithm (we use the `gammaln` function from `scipy.special`) and at the end recover it by taking the `exp` of the result. In the second alternative, we use the `math` library, where the total number of arrangements $A_{365,n}$ is obtained by the command `math.perm(365, n)`.

The program for the first alternative in Python is:

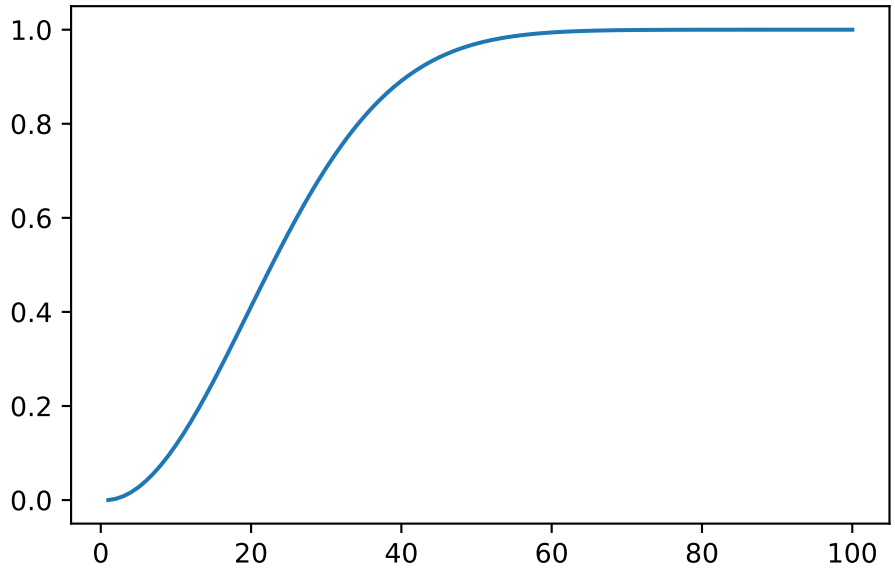
```
import numpy as np
import matplotlib.pyplot as plt
from numpy.lib.scimath import log
from numpy import exp
from scipy.special import gammaln

# Use of log of gamma to obtain the log of the factorial
def ca(n):
  # inverting the log (i.e. exp)
  return 1 - exp(gammaln(365+1) - gammaln(365-n+1) -n*log(365) )

n = np.arange(1, 100+1)
# Create an isolated space for the figure
fig, ax = plt.subplots()

# Draw the curve of ca(n)
_ = ax.plot(n, ca(n))

# Display the plot natively
plt.show()
```

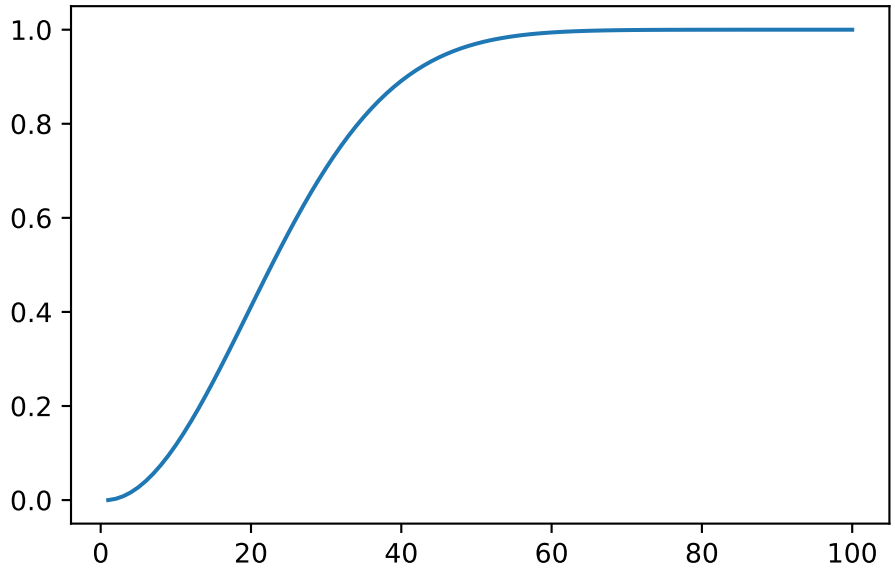


Our second implementation is:

```
# birthday coincidence
# probabilities
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)
# Create an isolated space for the figure
fig, ax = plt.subplots()

# Draw the curve of ca(n)
_ = ax.plot(n, pca(n))

# Display the plot natively
plt.show()
```



In the second example, let us consider the probability that a committee of size m, when drawn randomly from a group of size n, does not contain representatives of a specific existing group of size k among the n elements of the whole group. This probability is:

$$P(\text{Not represented}) = \frac{\binom{n-k}{m}}{\binom{n}{m}}.$$

The Python program was implemented and illustrated with the example of Brazilian senators, who have representatives from the different Brazilian federal units. In Brazil we have $n = 27 \times 3$ senators, that is, 81 senators in total, with 3 from each state or from the Federal District. If we are to form a random committee of $m = 10$ senators, what is the probability that any given federal unit containing $k = 3$ senators is not represented? We saw through the analysis of the previous formula that we have $\binom{n}{m}$ different ways of drawing committees of size m from n elements of the whole group. If we remove from the total n the k that will be excluded from the committee composition, $n - k$ remain, which can be sampled m at a time. Thus, the number of committees without the k of the considered group is $\binom{n-k}{m}$. The ratio between these two quantities gives us the desired probability. The `script` with the senators example is presented below.

```
# Probability that a committee of m people
# has no representative of a subgroup of size
# k when a committee of size m is drawn from
# a group containing n elements, k < n and 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
# Example
n = 27*3 # number of senators in Brazil, 81 (3 from each state or DF)
m = 10 # committee of m of them
k = 3 # any fixed state not being represented
pnr(n,m,k)
m = range(1, 81+1)
```

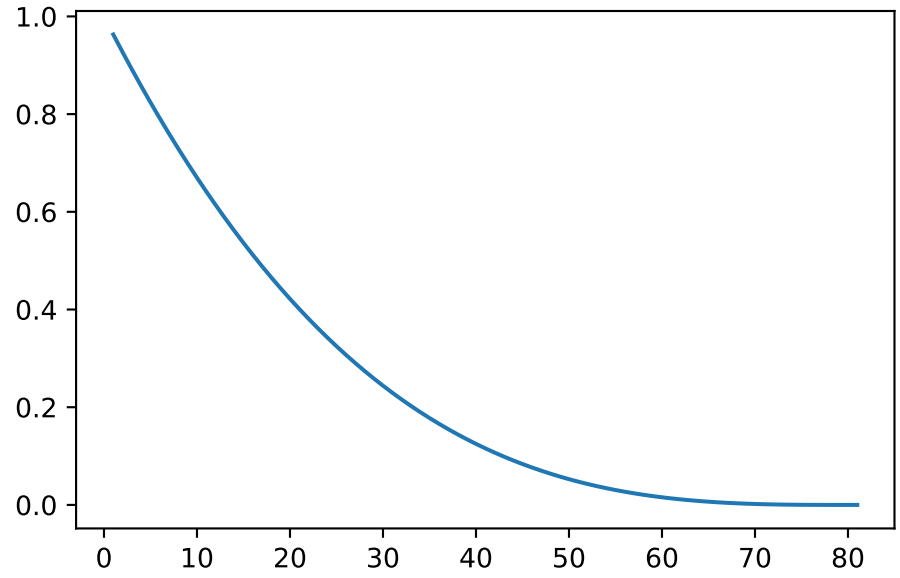
```
# Create an isolated space for the figure
fig, ax = plt.subplots()

# Draw the curve of ca(n)
_ = ax.plot(m, pnr(n,m,k))

# Display the plot natively
plt.show()

# probability that MG and DF
# are not represented on the committee
k = 6
m = 10
pnr(n,m,k)
```

```
np.float64(0.6698898265353962)
```




```
np.float64(0.44129815640475195)
```

Our next example refers to the probabilities in a poker hand, building the possibilities and probabilities. In this game we can have a hand with nothing (no pairs, no two pairs, no three of a kind, etc.), one with a pair, one with two pairs, one with three of a kind, one with a straight (any suit), one with a *flush* (different values that are not in sequence of the same suit), one with a *full house* (three of a kind and a pair), one with a *four* (four of a kind), one with a straight of the same suit, and a royal straight of the same suit (*royal flush* — from 10 to Ace).

The total number of possibilities of distributing 52 in a draw of 5 cards (one hand) is:

$$\binom{52}{5} = \frac{52!}{5!(52-5)!} = 2{,}598{,}960.$$

- To obtain a hand with nothing, we must understand that the deck is made of 52 cards, with 13 card values (Ace, 1, 2, $\cdots$, 10, Jack, Queen, and King) and 4 suits for each value (spades, clubs, diamonds, and hearts). The number of possibilities of a five-card hand (draw of 5 cards without

replacement) containing a hand with nothing, that is, with no pairs, no three of a kind, no four of a kind, no straights of different suits or no straights of the same suit, is:

$$\binom{13}{5}\left(\binom{4}{1}\right)^5 - 10\left(\binom{4}{1}\right)^5 - 4\binom{13}{5} + 4 \times 10 = = 1{,}302{,}540,$$

that is, it is the number of cards of different values with any of the 4 suits $\binom{13}{5}4^5$, subtracted from all possibilities of straights of different suits 10×4^5 and from the possibilities of having cards of different values with the same suit $4\binom{13}{5}$. This value must be added to the ten possible straights for each of the suits 4×10. This addition occurs because we removed the possibilities of having cards of different values with the same suit, which includes the 10 possible straights of the same suit, and again it was removed when we eliminated the possibilities of having 5 cards of different values with the same suit, which includes the 10 possible straights of the same suit. Hence we need to add them, since they were removed twice.

- For the single-pair hand we have:

$$\binom{13}{1}\binom{4}{2}\binom{12}{3}4^3,$$

where $\binom{13}{1}$ chooses the value of the pair, $\binom{4}{2}$ chooses the suits for the pair, $\binom{12}{3}$ chooses among the 12 remaining values (not chosen for the pair), and 4^3 chooses the suits of the 3 values different from the pair.

- For two pairs we have:

$$\binom{13}{1}\binom{4}{2}\binom{12}{1}\binom{4}{2}\binom{11}{1}4,$$

where $\binom{13}{1}$ chooses the value of the first pair, $\binom{4}{2}$ chooses the suits for the first pair, $\binom{12}{1}$ chooses the second pair among the 12 remaining values, $\binom{4}{2}$ chooses the suits for the second pair, and $\binom{11}{1}$ chooses the values among the 11 remaining values (not chosen for the 2 pairs), and 4 chooses the suits of the value different from the two pairs.

- For three of a kind we have:

$$\binom{13}{1}\binom{4}{3}\binom{12}{2}4^2,$$

where $\binom{13}{1}$ chooses the value of the three of a kind, $\binom{4}{3}$ chooses the suits for the three of a kind, $\binom{12}{2}$ chooses the 2 values among the 12 remaining values, and 4^2 chooses the suits for these values.

- For the straight:

$$10 \times 4^5 - 10 \times 4,$$

where 10 chooses among the 10 possible straights (from Ace to 5, from 2 to 6, up to from 10 to Ace), and 4^5 chooses the suits of each value. The result includes straights of the same suit, which must be removed, corresponding to 10×4 (straights of the same suit including the royals).

- For the *flush* (cards of the same suit):

$$\binom{13}{5} \times 4 - 10 \times 4,$$

where $\binom{13}{5}$ chooses the 5 different values among the 13, and 4 chooses one of the suits for the 5 values. The result includes straights of the same suit, which must be removed, corresponding to 10×4 (straights of the same suit including the royals).

- For the *full house* (three of a kind and a pair):

$$\binom{13}{1}\binom{4}{3}\binom{12}{1}\binom{4}{2}$$

where $\binom{13}{1}$ chooses the value for the three of a kind, $\binom{4}{3}$ chooses the suits of the three of a kind, $\binom{12}{1}$ chooses the value of the pair among the 12 remaining values, and $\binom{4}{2}$ chooses the suits of the pair.

- For the four of a kind (*four*):

$$\binom{13}{1}\binom{12}{1}4$$

where $\binom{13}{1}$ chooses the value for the four of a kind, which will necessarily have one of each suit, $\binom{12}{1}$ chooses a value for the remaining card among the 12 remaining values, and 4 chooses one of the suits of the remaining card, which is not the one of the four of a kind.

- For straights of the same suit (*straight flush*):

$$10 \times 4 - 4,$$

since there are 10 straights for one of the 4 suits, minus the 4 royal straights among them, from 10 to Ace of each of the four possible suits.

- The royal straight (*royal straight flush*): there is a single possible straight from 10 to Ace for each of the four suits, totaling 4 possible royal straights.

The following program creates a `dataframe` with all these countings and computes the probabilities by dividing them by the total number of combinations:

```
# Poker hand
# probabilities
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 }
print('Must be zero: ', sum(poker.values()) - sps.comb(52,5)) # check
print('Poker (dict)\n', poker)
mão = list(poker.keys())
possibilidades = list(poker.values())
datapoker = pd.DataFrame({'hand': mão, 'possibilities': possibilidades})
datapoker['probabilities'] = list(poker.values()) / sps.comb(52,5)
print('Poker hands \n', datapoker)
sum(datapoker['probabilities'])
```

```
Must be zero:  0.0
Poker (dict)
 {'none': np.float64(1302540.0), 'pair': np.float64(1098240.0), 'two.pairs': np.float64(123552.0), 't
Poker hands
              hand  possibilities  probabilities
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
```

Our next example refers to the probability of the maximum difference in an ordered enumerated list. Obtain the largest *gap* (jump) (maximum difference) between two consecutive numbers in the draw without replacement of m numbers among the first n integers from 1 to n, for $m < n$. In Python, the command `np.diff(x)` returns the differences between consecutive numbers of the list `x`.

The following program uses the `checkgap` function to return the maximum difference of any iterator. The `maxgap` function receives n, m, and k and computes the probabilities for each case. To do this, it traverses all combinations generated by `combinations`, returning the maximum of each combination obtained in `y`, and with the command `mean(y == k)` the exact probability of $P(X = k)$ is obtained, where X is the variable representing the largest difference between consecutive integers of a sample without replacement of size m obtained among n integers, for m from 1 to n, and k the value of the largest difference (value of the random variable) for which we want to compute the probability of occurrence. If $m = 2$, then the maximum difference will be considered the consecutive difference, because the difference in each outcome of the sample space is unique, since we have samples of size 2.

```
# Max gap probabilities
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 must be between 1 and 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)

# function for the whole support
def maxgapsx(n, m):
  if m <= 1 or m > n:
    print('m must be between 1 and 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)

# example of use
n = 6
m = 3
k = 2
print('max gap n, m, k: ', maxgap(n,m,k))
print('max gap n, m: ', maxgapsx(n,m))
```

```
max gap n, m, k:  0.4
max gap n, m:     x  P(X=x)
0  1     0.2
1  2     0.4
2  3     0.3
3  4     0.1
```

Our last example is for the probability of the sums of faces in the throw of n dice and of the difference in absolute value of the faces in the throw of 2 dice. For this case, we can use a similar version, either by applying the previously presented `contag` function or by applying the `product` function from the `intertools` library. We chose to use the `contag` function from the `cont` class.

With the use of some small additional details, we obtained the results for both cases. The second case, of the difference, for obvious reasons, results from the throw of only 2 dice. In both cases we also present the probability results by obtaining all possibilities that correspond to the sample space of the random experiment. We have a function to obtain a probability for a specific value of the random variable and another for all probability values relative to the support of the random variable, in both cases, that of the sum of n dice and that of the difference of two dice. The results are presented in the following `script`:

```
# Probabilities for the throw of
# n dice and obtaining the sum and also
# for the absolute difference of 2 dice
# somafaces: returns P(X=x)
# depends on class cont
def somafaces(n, x):
  if x<n or x>6*n:
    print('x must be between n and 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):
  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)

# Probabilities of the difference between
# two faces of two balanced dice
# in modulus - returns P(X=x), X=|i-j|
def difabs2faces(x):
  if x<0 or x>5:
    print('x must be between 0 and 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)

# Probabilities of the difference between
# two faces of two balanced dice
# in modulus - returns P(X=x), for the whole
# support of x
def difabs2facessx():
  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)
# Example
n = 4
x = 4
print('P(X =',x,') = ',somafaces(n, x))
dist = somafacessx(n)
print('dist: \n', dist)
print('Sum of all probabilities: \n', sum(dist['P(X=x)'])) # checking
print('P(X =',x,') = ',difabs2faces(x))
difabs2facessx()
```

```
P(X = 4 ) =  0.0007716049382716049
dist:
      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
Sum of all probabilities:
 1.0
P(X = 4 ) =  0.1111111111111111
```

	x	P(X=x)
0	0	0.166667
1	1	0.277778
2	2	0.222222
3	3	0.166667
4	4	0.111111
5	5	0.055556

7.6 Exercises

1. Obtain a function to compute, by Monte Carlo simulation, the probabilities of the throws of n dice and the sums obtained, as well as the absolute differences between faces in the independent throws of 2 dice. To do this, choose a large number of replications N and in each one simulate the outcomes of the dice throws. Obtain the results of the random variables and, at the end, compute the proportions of each occurrence, as we did exactly before. These proportions are empirical estimates of the desired probabilities.
2. Also obtain, by Monte Carlo, the empirical probabilities of a poker hand, simulating a large number of hands. For this you need to create a deck, which can be done with a dictionary, and then sample five-card hands and compute the empirical probabilities of each possible outcome. Compare with the exact probabilities obtained in this chapter. What do you expect to happen in this comparison when the number of simulations increases?

Chapter 8

Bootstrap Methods in Python

In this chapter, we intend to present the basic concepts of bootstrap in a very simple way. We will build some simple functions in a few particular cases.

8.1 Introduction

One of the main topics involving computationally intensive methods and, perhaps, the greatest responsible for the popularity of these methods is the bootstrap technique. The bootstrap method involves resampling with replacement from the original sample. Inspired by sampling theory, the bootstrap is used for hypothesis testing, parameter estimation by intervals, and estimation of standard errors.

The great advantage of using resampling methods, such as bootstrap, to perform inference arises when we do not know the probability distribution of the population and the normal model is not suitable for the data or residuals. The problem arises when we violate the conditions assumed for applying a test or obtaining a confidence interval or region.

In this chapter, we will study bootstrap methods for performing inference about parameters of interest.

8.2 Non-Parametric Bootstrap

Non-parametric bootstrap: the idea is to resample from the original sample. Let Y_1, Y_2, ..., Y_n be a random sample of size n from a population with unknown distribution, and let the parameter of interest be θ. We can obtain a series of resamples with replacement of size n and estimate the parameter of interest by a function $\hat{\theta}$ of the obtained samples.

This finite series of estimates represents a sample from the sampling distribution of the estimator, allowing us to make inference about θ. If we could obtain the sampling distribution of $\hat{\theta}$ directly from sampling theory, we could make inference about θ without needing bootstrap. For that, we would have to know the probability distribution of Y_j, $j = 1, 2, \ldots, n$, and, from it, deduce the sampling distribution of the function of interest $\hat{\theta}$.

This is not always possible, especially if we do not know the probability distribution of the random variable being sampled. For most probabilistic models, exact theories are not known or are analytically intractable. In that case, we use the empirical probability distribution, by which we assign to each of the sample observations the probability $1/n$.

The idea of the bootstrap is to replace the unknown distribution of the population by the empirical distribution. For this reason, we call this method the non-parametric bootstrap. By obtaining a finite

series of resamples of the same size as the original sample and building a sample from the distribution of the estimator, we are performing a mimicry process with respect to sampling theory.

This process is based on the fact that the sample obtained from the population contains all the available information of that underlying population. Hence, it comes to represent the "population" for the resampling process.

Efron (1979) was the researcher who organized the theories about this method and connected the non-parametric bootstrap with the statistical techniques for estimating errors, accepted since the 1930s, such as the jackknife and the delta method. Both the bootstrap technique and the permutation methods assume only that the random variables satisfy the principle of being exchangeable, which is a weaker assumption than being independent and identically distributed. By exchangeable we should understand that the probability distribution of any k consecutive observations ($k = 1, 2, \ldots, n$) does not change when the order of the observations is changed by some permutation.

The general algorithm of the bootstrap resamples, to estimate an unknown parameter θ, whose probability distribution is f (of Y), considering also that the estimator $\hat{\theta}$ is a function of the sample values, that is, $\hat{\theta} = g(Y_1, Y_2, \ldots, Y_n)$, is given by:

1. assign probability mass $1/n$ to each sample observation Y_j, $j = 1, 2, \ldots, n$, creating the empirical probability distribution $\hat{f}$;
2. generate random samples with replacement from the empirical probability distribution $\hat{f}$, called the bootstrap sample, denoted by $\tilde{Y}_1, \tilde{Y}_2, \ldots, \tilde{Y}_n$;
3. compute an estimate of $\hat{\theta}$ by $\tilde{\theta}$, using the bootstrap sample in place of the original sample, as follows: $\tilde{\theta} = g(\tilde{Y}_1, \tilde{Y}_2, \ldots, \tilde{Y}_n)$;
4. repeat steps 2 and 3 B times.

The number of resamples was denoted by B, and the choice of adequate values according to the situation will be discussed later. The inference we intend to make depends on the unknown distribution of $\hat{\theta} - \theta$. When we apply the previous algorithm, we obtain a Monte Carlo sample, from the bootstrap distribution, of the quantity $\tilde{\theta} - \hat{\theta}$. If n is sufficiently large, by the ideas of the bootstrap method, extremely proven and accepted today, we expect the two distributions to be approximately identical.

8.3 Estimation

To estimate parameters of a population, the bootstrap technique is an excellent alternative, especially if the distribution is not normal or another distribution whose estimation processes are well-founded theoretically, such as, for example, the estimation of binomial proportions p. If the bootstrap method is applied to these cases, it will not be a problem, since the results will be equivalent to those of classical theory.

However, if the assumptions of a classical estimation process are violated, in general, bootstrap methods will result in estimation processes with better properties. Initially, we must understand that bootstrap methods do not improve point estimates, but they provide appropriate mechanisms for estimating standard errors, confidence intervals, and the sampling distribution of the estimator, no matter how complex that estimator may be.

8.3.1 Standard Error

To estimate the standard error of an estimator $\hat{\theta}$ of a parameter θ, we must obtain the bootstrap distribution of that estimator, as described in the algorithm presented earlier. Thus, if we consider a random sample $Y_1, Y_2, \ldots, Y_n$ of size n from the population of interest, which depends on a parameter θ that we intend to estimate, then we apply the previous algorithm and obtain a bootstrap sample of size B from the distribution of that estimator.

If the sample is $\tilde{\theta}_1$, $\tilde{\theta}_2$, $\tilde{\theta}_3$, ..., $\tilde{\theta}_B$, we can estimate the standard error of $\hat{\theta}$ by

$$S_{\hat{\theta}} = \sqrt{\frac{1}{B-1}\sum_{j=1}^{B}\left(\tilde{\theta}_j - \bar{\tilde{\theta}}\right)^2} = \sqrt{\frac{1}{B-1}\left[\sum_{j=1}^{B}\tilde{\theta}_j^2 - \frac{\left(\sum_{j=1}^{B}\tilde{\theta}_j\right)^2}{B}\right]}, \tag{8.1}$$

where

$$\bar{\tilde{\theta}} = \frac{\sum_{j=1}^{B}\tilde{\theta}_j}{B}.$$

We infer from expression (8.1) that the standard error of the estimator of θ is the standard deviation of the bootstrap distribution of that estimator. If, however, the parameter θ is the mean μ of the sampled probability distribution, we can simplify the expression for the standard error estimator of the sample mean $\bar{Y}$, since, in that case, we do not need the bootstrap sample of the estimator.

From statistical theory, we know that the standard error of the sample mean $\bar{Y}$ is given by $S_{\bar{Y}} = (1/n)^{1/2}S$, where S is given by

$$S = \sqrt{\frac{1}{n-1}\sum_{j=1}^{n}(Y_j - \bar{Y})^2} = \sqrt{\frac{1}{n-1}\left[\sum_{j=1}^{n}Y_j^2 - \frac{\left(\sum_{j=1}^{n}Y_j\right)^2}{n}\right]}. \tag{8.2}$$

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

# function to return the median
def est_med(x):
    y = np.sort(x)
    n = len(x)
    if n%2 == 0:
        est = (y[n//2 - 1] + y[(n+2)//2 - 1]) / 2
    else:
        est = y[(n+1) // 2 - 1]
    return est

# function for the minimum
def est_min(x):
    return np.min(x)

# function for the maximum
def est_max(x):
    return np.max(x)

# function to perform one bootstrap resample
# and apply the generic estimator est, receiving
# x as a numpy list
def sampleOneBoot(x, est, *args):
    y = random.choices(x, k = len(x))
    estm = est(y, *args)
```

```
    return estm

# function to obtain the bootstrap distribution
def dist_boot(x, est, B = 2000, *args):
    dist = np.empty(0)
    for i in range(B):
        dist = np.append(dist, sampleOneBoot(x, est, *args))
    return dist

# function to obtain the standard error of the estimator
def stderr_est(x, est, B = 2000, *args):
    result = dist_boot(x, est, B, *args)
    se_boot = np.std(result)
    est_boot = est(x, *args)
    return {"Estimate": est_boot, "Standard error": se_boot}

# estimate the normal median
n = 300
B = 2000
x = np.random.normal(size=n)
dist = dist_boot(x, est_med)
plt.clf() # clear the plot

# Create an isolated space for the figure
fig, ax = plt.subplots()

# Draw the histogram
_ = ax.hist(dist,bins='auto',color='#14e8f3',rwidth=0.98,alpha=0.7)

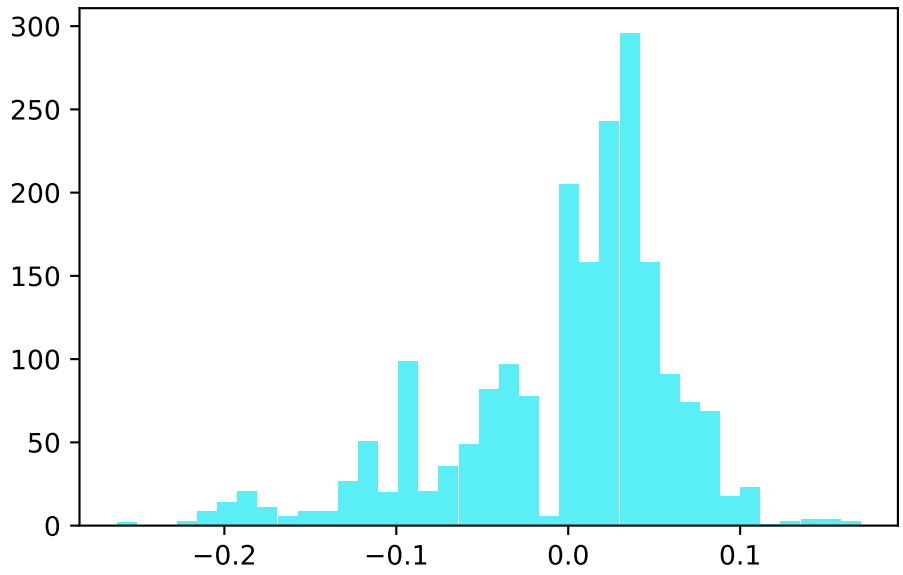
# Display the plot natively
plt.show()

print('Mean: ',np.mean(dist), 'and Standard deviation: ', np.std(dist, ddof=1))
res_med = stderr_est(x, est_med, B)
for key, value in res_med.items():
    print(f"{key}: {value}")
# theoretical value: approx. (pi/2)^0.5*sigma/n^0.5
print((np.pi/2)**0.5*1/n**0.5)

# minimum of a normal
res_min = stderr_est(x, est_min, B)
for key, value in res_min.items():
    print(f"{key}: {value}")

# maximum of a normal
res_max = stderr_est(x, est_max, B)
for key, value in res_max.items():
    print(f"{key}: {value}")
```

```
<Figure size 1650x1050 with 0 Axes>
```




```
Mean:  -0.0014359906114474116 and Standard deviation:  0.06580176216401669
Estimate: 0.0150549321507049
Standard error: 0.0656361196384175
0.07236012545582675
Estimate: -2.583042768467188
Standard error: 0.11934072315918523
Estimate: 2.811394894860523
Standard error: 0.14867392535754467
```

8.3.2 Bias Correction

We should explore in the bootstrap the possibility of bias correction for biased estimators. We know from classical theory that the bias δ of an estimator $\hat{\theta}$ is defined by

$$\delta_{\hat{\theta}} = E(\hat{\theta}) - \theta; \tag{8.3}$$

Since in the bootstrap distribution of an estimator $\hat{\theta}$, the mean represents the expectation in (8.3), and the estimator $\hat{\theta}$ in the original sample plays the same role as θ in the population, we can estimate the bias by

$$\tilde{\delta}_{\hat{\theta}} = \bar{\bar{\theta}} - \hat{\theta}. \tag{8.4}$$

It is common to think that $\bar{\bar{\theta}}$ would be the bias-corrected estimator, which is not true. We can obtain an adjusted estimator by

$$\begin{aligned}\hat{\theta}_{\text{adj.}} =& \hat{\theta} - \tilde{\delta}_{\hat{\theta}} = \hat{\theta} - \bar{\bar{\theta}} + \hat{\theta} \\ =& 2\hat{\theta} - \bar{\bar{\theta}};\end{aligned}$$

Next, we will address some of the different types of confidence intervals usually employed with the so-called non-parametric bootstrap method.

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

```
# function to obtain the standard error of the estimator
# and apply the bias correction
def stderr_vies_est(x, est, B = 2000, *args):
    result = dist_boot(x, est, B, *args)
    se_boot = np.std(result)
    est_boot = est(x, *args)
    mean_boot = np.mean(result)
    vies_boot = mean_boot - est_boot
    est_nvies = est_boot - vies_boot
    return {"Estimate": est_boot, "Standard error": se_boot, \
            "Bias ": vies_boot, "Unbiased est. ": est_nvies}

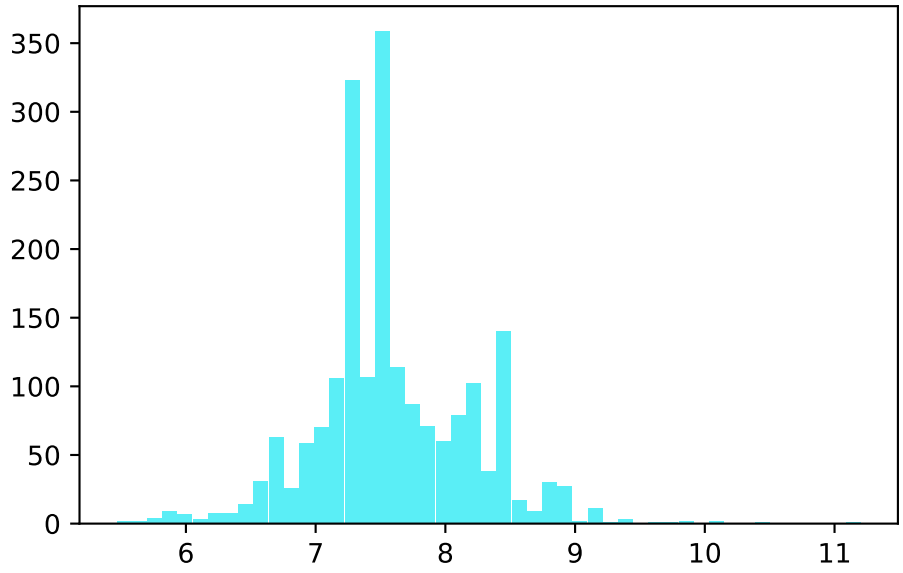
# estimate the exponential median and standard error
n = 300
B = 2000
lamb = 0.1
x = np.random.exponential(scale = 1 / lamb, size = n)
dist = dist_boot(x, est_med)
plt.clf() # clear the plot

# Create an isolated space for the figure
fig, ax = plt.subplots()

# Draw the histogram
_ = ax.hist(dist,bins='auto',color='#14e8f3',rwidth=0.98,alpha=0.7)

# Display the plot natively
plt.show()
res_med = stderr_vies_est(x, est_med, B)
for key, value in res_med.items():
    print(f"{key}: {value}")
# theoretical median
sp.stats.expon.ppf(0.5, scale = 1 / lamb)
```

```
<Figure size 1650x1050 with 0 Axes>
```




```
Estimate: 7.525538391538565
Standard error: 0.5888258414536909
Bias : 0.035515994078742175
Unbiased est. : 7.490022397459823
```

```
np.float64(6.931471805599453)
```

8.3.3 Standard Bootstrap Confidence Interval

In some situations, we can assume that the bootstrap distribution of the estimator $\tilde{\theta}$ is normal. This occurs in special sampling situations: such as the specific case where $\tilde{\theta}$ is the sample mean $\bar{Y}$ in sampling from the normal distribution, and also in the case of large samples n as a consequence of the central limit theorem.

We know that $\tilde{\theta}$ is a function of the sample elements, and as a consequence, for many of these functions, the normal distribution arises as a result of the central limit theorem. Unfortunately, experience has shown that this interval is, as a rule, very poor, in the sense that its properties are not preserved.

The main property required of a confidence interval is that it have coverage probability equal to the nominal confidence value $1-\alpha$ adopted, which, in general, is not met in this case. The idea is to admit the effect of the central limit theorem:

$$\tilde{\theta} \dot{\sim} N\left(\hat{\theta}, S^2_{\hat{\theta}}\right), \tag{8.5}$$

where the symbol $\dot{\sim}$ should be read as "is approximately distributed as", $\hat{\theta}$ is the value of the estimator in the original sample, and $S_{\hat{\theta}}$ is the estimator of the standard error of $\hat{\theta}$, obtained in the bootstrap distribution.

As a consequence of the result in (8.5), we can construct the standard bootstrap interval with confidence of approximately $1-\alpha$ by

$$IC_{1-\alpha}(\theta) : \left[\hat{\theta} - Z_{1-\alpha/2}S_{\hat{\theta}}; \hat{\theta} - Z_{\alpha/2}S_{\hat{\theta}}\right], \tag{8.6}$$

where $Z_{\alpha/2}$ and $Z_{1-\alpha/2}$ are the quantiles of the standard normal distribution. The confidence interval (8.6), as we have already commented, has a coverage probability, in general, below the nominal value of $100(1-\alpha)\%$.

We can improve this interval by obtaining the confidence interval limits using a bias-corrected estimator. The result for this case is:

$$IC_{1-\alpha}(\theta) : \left[\hat{\theta}_{\text{adj.}} - Z_{1-\alpha/2}S_{\hat{\theta}}; \hat{\theta}_{\text{adj.}} - Z_{\alpha/2}S_{\hat{\theta}}\right], \tag{8.7}$$

where $\hat{\theta}_{\text{adj.}}$ is given in (8.5).

```
import scipy as sp


# function to obtain the standard CI
# with bias correction
def ic_padrao_cv_boot(x, est, B = 2000, alpha = 0.05, *args):
    res = dist_boot(x, est, B, *args)
    est_boot = est(x, *args)
    se_boot = np.std(res)
    mean_boot = np.mean(res)
    vies_boot = mean_boot - est_boot
    me = sp.stats.norm.ppf(1 - alpha / 2) * se_boot
    li = est_boot - vies_boot - me
    ls = est_boot - vies_boot + me
    return {'estimate': est_boot, 'standard error': se_boot, \
            'bias': vies_boot, 'lower bound': li, 'upper bound': ls}

# example for the exponential median
n = 30
B = 2000
lamb = 0.1
alpha = 0.05
x = np.random.exponential(scale = 1 / lamb, size = n)
res_med = ic_padrao_cv_boot(x, est_med, B, alpha)
for key, value in res_med.items():
    print(f"{key}: {value}")
# theoretical median
sp.stats.expon.ppf(0.5, scale = 1 / lamb)
```

```
estimate: 10.170078872350903
standard error: 2.440295571102598
bias: -0.48976555147652867
lower bound: 5.876952992833737
upper bound: 15.442735854821127
```

```
np.float64(6.931471805599453)
```

Next, we present an example for the gamma distribution. In the subsequent examples, we will use the same sample from the same distribution.

```
import scipy as sp


# Example for the gamma median
n = 30
B = 2000
alpha = 0.05
a = 0.5
```

```
x = np.random.gamma(a, size = n)
res_med = ic_padrao_cv_boot(x, est_med, B, alpha)
for key, value in res_med.items():
    print(f"{key}: {value}")
# theoretical median
sp.stats.gamma.ppf(0.5, a)
```

```
estimate: 0.20319749915972624
standard error: 0.06702691596013077
bias: -0.0009762561184847307
lower bound: 0.0728034140015617
upper bound: 0.3355440965548602
```

```
np.float64(0.227468211559786)
```

8.3.4 Confidence Interval Based on Bootstrap Percentiles

It is one of the most interesting, since its derivation is, at the very least, curious. Assuming normality of the bootstrap distribution underuses the potential of the method. However, we can think that there exists a monotone function ϕ such that $\phi(\tilde{\theta})$ has a normal distribution.

If we obtain the bootstrap distribution of this function, obtained by applying it to each resample with replacement of the original sample, we can obtain its standard deviation $S_{\phi(\tilde{\theta})}$. Since this distribution, obtained by considering a finite number B, is a sample from a normal distribution, the intervals obtained by

$$IC_{1-\alpha}(\phi(\theta)) : \left[\phi(\hat{\theta}) - Z_{1-\alpha/2}S_{\phi(\tilde{\theta})}; \phi(\hat{\theta}) - Z_{\alpha/2}S_{\phi(\tilde{\theta})}\right]$$

or directly by sorting the sample of size B from the bootstrap distribution and taking its percentiles $100(\alpha/2)\%$ and $100(1-\alpha/2)\%$, should be equivalent as $B \to \infty$.

According to Efron and Tibshirani (1993), the transformation ϕ that leads to normality always exists. Thus, the confidence interval for θ, since ϕ is a monotone function, is given by

$$IC_{1-\alpha}(\theta) : \left[\phi^{-1}\left(\phi(\hat{\theta}) - Z_{1-\alpha/2}S_{\phi(\tilde{\theta})}\right); \phi^{-1}\left(\phi(\hat{\theta}) - Z_{\alpha/2}S_{\phi(\tilde{\theta})}\right)\right], \tag{8.8}$$

where ϕ^{-1} is the inverse function of ϕ.

We can verify from expression (8.8) that the confidence interval for the parameter on the original scale is obtained by the inverse transformation of the interval limits obtained on the transformed scale. We used in (8.8) the inverse transformation of the limits obtained from the standard normal percentiles and from the mean and variance of the bootstrap distribution to obtain the confidence interval limits on the original scale. However, we could have used a version that uses the inverse transformation of the percentiles $100(\alpha/2)\%$ and $100(1-\alpha/2)\%$ obtained directly from the bootstrap distribution on the transformed scale.

This can be done by virtue of what we have just commented, that is, that the bootstrap distribution obtained by the transformation, except for the fact that it is a finite sample, is normal, and the two intervals on the transformed scale are asymptotically equivalent.

Put this way, it seems easy to obtain an interval such as the one presented in (8.8). This is not true, since we would need to obtain the function ϕ, which would lead to a normal bootstrap distribution. Finding such a function for each real case is an infeasible task. On the other hand, we do not need to know such a function; it suffices to assume its existence.

Since the interval values obtained on the transformed scale, when mapped to the original scale, are equivalent to the percentiles $100(\alpha/2)\%$ and $100(1-\alpha/2)\%$ of the bootstrap distribution of $\tilde{\theta}$, the construction of the confidence interval by this method is trivial. This result is extremely interesting

and was preponderant for the success of bootstrap estimation methods. As Efron and Tibshirani (1993) state, we can understand the percentile method for obtaining the confidence interval as an automatic algorithm for incorporating such transformations that lead to normality. To make it clearer, we can say that the limits obtained from (8.8) correspond exactly to the interval based on percentiles of the bootstrap distribution on the original scale; therefore, we do not need to know the transformation that will lead the bootstrap distribution to normality, but only to assume its existence!

To apply the confidence interval based on bootstrap percentiles, we must apply the algorithm presented earlier. We obtain: $\tilde{\theta}_1$, $\tilde{\theta}_2$, $\tilde{\theta}_3$, ..., $\tilde{\theta}_B$.

We must sort these values, obtaining the order statistics by: $\tilde{\theta}_{(1)}$, $\tilde{\theta}_{(2)}$, $\tilde{\theta}_{(3)}$, ..., $\tilde{\theta}_{(B)}$. Then, we can obtain the confidence interval from this empirical distribution function, obtaining the percentiles $100(\alpha/2)\%$ and $100(1-\alpha/2)\%$. Thus, we construct the confidence interval based on bootstrap percentiles by

$$IC_{1-\alpha}(\theta) : \left[\tilde{\theta}_{(k_1)}; \tilde{\theta}_{(k_2)}\right], \tag{8.9}$$

where $k_1 = \lfloor (B+1)(\alpha/2) \rfloor$ and $k_2 = \lfloor (B+1)(1-\alpha/2) \rfloor$, that is, they are the largest integers that are not greater than $(B+1)(\alpha/2)$ and $(B+1)(1-\alpha/2)$, respectively; $\tilde{\theta}_{(k_1)}$ is the percentile $100(\alpha/2)\%$ of the empirical bootstrap distribution function; and $\tilde{\theta}_{(k_2)}$, the percentile $100(1-\alpha/2)\%$ of the empirical bootstrap distribution function.

This interval has better properties than the previous one and works well in most simpler cases. However, there are better alternatives. We will see below some other types of intervals that are better than the first two, although in some situations the percentile method is superior to some of them.

A great advantage of the percentile interval is that it has the property of respecting the transformation, that is, the limits of this interval obey the domain of the parameter being estimated, which does not happen in some methods.

```
import scipy as sp


# function to obtain the Percentile CI
def ic_perc_boot(x, est, B = 2000, alpha = 0.05, *args):
    res = dist_boot(x, est, B, *args)
    est_boot = est(x, *args)
    se_boot = np.std(res)
    mean_boot = np.mean(res)
    vies_boot = mean_boot - est_boot
    li, ls = np.percentile(res, [alpha/2 * 100, (1-alpha/2) * 100])
    return {'estimate': est_boot, 'standard error': se_boot, \
            'bias': vies_boot, 'lower bound': li, 'upper bound': ls}


# Example for the gamma median
B = 2000
alpha = 0.05
res_med = ic_perc_boot(x, est_med, B, alpha)
for key, value in res_med.items():
    print(f"{key}: {value}")
# theoretical median
sp.stats.gamma.ppf(0.5, a)
```

```
estimate: 0.20319749915972624
standard error: 0.06542438978229925
```

```
bias: 0.0013119764630146746
lower bound: 0.0985059996149705
upper bound: 0.3179907375430121
```

```
np.float64(0.227468211559786)
```

8.3.5 Basic Bootstrap Confidence Interval

A variation of the confidence interval based on bootstrap percentiles results from obtaining the distribution of $\tilde{\theta} - \hat{\theta}$, which simply represents the bootstrap version of the sampling distribution of $\hat{\theta} - \theta$. For large values of B, $\tilde{\theta} - \hat{\theta}$ converges in distribution to $\hat{\theta} - \theta$.

If we consider that there exists a monotone function $\phi(\tilde{\theta} - \hat{\theta})$ capable of leading to normality, then we would only need to obtain the percentiles $100(\alpha/2)\%$ and $100(1-\alpha/2)\%$ of the original distribution, since they map equivalent values on the transformed normal scale.

If we denote by $\tilde{\delta}^{(\alpha/2)}$ and $\tilde{\delta}^{(1-\alpha/2)}$ these percentiles on the original scale, then

$$P\left[\tilde{\delta}^{(\alpha/2)} \leq \tilde{\theta} - \hat{\theta} \leq \tilde{\delta}^{(1-\alpha/2)}\right] = 1 - \alpha P\left[\tilde{\theta}^{(\alpha/2)} - \hat{\theta} \leq \tilde{\theta} - \hat{\theta} \leq \tilde{\theta}^{(1-\alpha/2)} - \hat{\theta}\right] = 1 - \alpha.$$

Since

$$\tilde{\theta} - \hat{\theta} \xrightarrow{d} \hat{\theta} - \theta,$$

then, substituting into the previous expression, we have

$$P\left[\tilde{\theta}^{(\alpha/2)} - \hat{\theta} \leq \hat{\theta} - \theta \leq \tilde{\theta}^{(1-\alpha/2)} - \hat{\theta}\right] = 1 - \alpha.$$

If we isolate θ we have

$$P\left[2\hat{\theta} - \tilde{\theta}^{(1-\alpha/2)} \leq \theta \leq 2\hat{\theta} - \tilde{\theta}^{(\alpha/2)}\right] = 1 - \alpha. \tag{8.10}$$

The limits of the probabilistic statement in (8.10) are the limits of a basic bootstrap $100(1-\alpha)\%$ confidence interval, given by

$$IC_{1-\alpha}(\theta) : \left[2\hat{\theta} - \tilde{\theta}^{(1-\alpha/2)};\, 2\hat{\theta} - \tilde{\theta}^{(\alpha/2)}\right]. \tag{8.11}$$

This interval can also be seen as the bootstrap t confidence interval, which will be the next one to be studied, when the standard error in each resample is considered equal to unity. The quantities $\tilde{\theta}^{(1-\alpha/2)}$ and $\tilde{\theta}^{(\alpha/2)}$ of interval (8.11) are, respectively, the percentiles $\tilde{\theta}_{(k_1)}$ and $\tilde{\theta}_{(k_2)}$ of interval (8.9) based on percentiles. We observe that the present method is merely a variation of the percentile method.

```
import scipy as sp


# function to obtain the Basic Standard CI
def ic_basic_boot(x, est, B = 2000, alpha = 0.05, *args):
    res = dist_boot(x, est, B, *args)
    est_boot = est(x, *args)
    se_boot = np.std(res)
    mean_boot = np.mean(res)
    vies_boot = mean_boot - est_boot
    ls, li = np.percentile(res, [alpha/2 * 100, (1 - alpha / 2) * 100])
    li = 2 * est_boot - li
    ls = 2 * est_boot - ls
```

```
    return {'estimate': est_boot, 'standard error': se_boot, \
            'bias': vies_boot, 'lower bound': li, 'upper bound': ls}


# Example for the gamma median
B = 2000
alpha = 0.05
res_med = ic_basic_boot(x, est_med, B, alpha)
for key, value in res_med.items():
    print(f"{key}: {value}")
# theoretical median
sp.stats.gamma.ppf(0.5, a)
```

```
estimate: 0.20319749915972624
standard error: 0.0653215490203138
bias: 0.001435641993475123
lower bound: 0.08840426077644037
upper bound: 0.30788899870448194
```

```
np.float64(0.227468211559786)
```

8.3.6 Bias-Corrected and Accelerated Bootstrap Confidence Interval

The bootstrap confidence intervals previously presented have deficiencies relative to coverage probabilities, at least in part of practical situations. Another method for obtaining bootstrap intervals has better qualities than the previous intervals, although it has greater difficulty of derivation: bias-corrected and accelerated bootstrap, *bias-corrected and accelerated* (BC_a).

This is the ideal method to be used. The BC_a method is obtained similarly to the percentile bootstrap method, but when determining the percentiles of the empirical bootstrap distribution $\tilde{\theta}_{(1)}$, $\tilde{\theta}_{(2)}$, $\tilde{\theta}_{(3)}$, ..., $\tilde{\theta}_{(B)}$ of the estimator, two quantities $\hat{a}$ and $\hat{z}_0$ are also needed. The quantities $\hat{a}$ and $\hat{z}_0$ are defined as acceleration and bias correction.

The confidence interval based on bootstrap percentiles with bias-corrected and accelerated correction, BC_a, is given by

$$IC_{1-\alpha}(\theta) : \left[\tilde{\theta}_{(k_1)}; \tilde{\theta}_{(k_2)}\right], \tag{8.12}$$

where $k_1 = \lfloor (B+1)p_1 \rfloor$ and $k_2 = \lfloor (B+1)p_2 \rfloor$, that is, they are the largest integers that are not greater than $(B+1)p_1$ and $(B+1)p_2$, respectively; $\tilde{\theta}_{(k_1)}$ is the percentile $100(p_1)\%$ of the empirical bootstrap distribution function; and $\tilde{\theta}_{(k_2)}$, the percentile $100p_2\%$ of the empirical bootstrap distribution function.

For known values of $\hat{a}$ and $\hat{z}_0$, the values of p_1 and p_2, used to determine the orders k_1 and k_2 of the value of $\tilde{\theta}$ in the empirical bootstrap distribution, as presented in (8.12), are determined by

$$p_1 = \Phi\left(\hat{z}_0 + \frac{\hat{z}_0 + z_{\alpha/2}}{1 - \hat{a}\left[\hat{z}_0 + z_{\alpha/2}\right]}\right) \text{ and } p_2 = \Phi\left(\hat{z}_0 + \frac{\hat{z}_0 + z_{1-\alpha/2}}{1 - \hat{a}\left[\hat{z}_0 + z_{1-\alpha/2}\right]}\right), \tag{8.13}$$

where $\Phi(x)$ is the standard normal distribution function evaluated at the value x, and $z_{\alpha/2}$ and $z_{1-\alpha/2}$ are the quantiles $100(\alpha/2)\%$ and $100(1-\alpha/2)\%$ of that same distribution, respectively.

If both $\hat{a}$ and $\hat{z}_0$ are null, the values of p_1 and p_2 will be, respectively, $\alpha/2$ and $1-\alpha/2$, as can be easily deduced from (8.13). In that case, the intervals BC_a, expression (8.12), and percentile bootstrap, expression (8.9), are equivalent. Non-null values of these quantities modify the limits of interval (8.12) and reduce some deficiencies of the percentile interval (8.9), as stated by Efron and Tibshirani (1993).

The value of $\hat{z}_0$ is defined as the standard normal quantile whose cumulative probability is given by the proportion of estimates in the bootstrap distribution that is less than the estimate of θ in the original sample. This value reflects the discrepancy that exists between the median of the distribution of $\tilde{\theta}$ and the median of the distribution of $\hat{\theta}$, expressed in standard normal units.

Thus, we define $\hat{z}_0$ by

$$\hat{z}_0 = \Phi^{-1}\left(\frac{\sum_{i=1}^{B} I\left(\tilde{\theta}_i < \hat{\theta}\right)}{B}\right), \tag{8.14}$$

where $\Phi^{-1}(p)$ is the inverse of the standard normal distribution function evaluated at p between 0 and 1, and $I(\cdot)$ is the indicator function that returns 1 if the value of its argument is true and 0 if false.

The value $\hat{z}_0$ will be zero only if the proportion of values of the bootstrap distribution that are less than $\hat{\theta}$ equals 50%. The value of $\hat{a}$ can be computed using jackknife estimates of the parameter of interest.

Suppose the estimator $\hat{\theta}$ is obtained by a function of the vector of sample observations $\mathbf{Y} = [Y_1, Y_2, \ldots, Y_n]^\top$ by $\hat{\theta} = g(\mathbf{Y})$. If we remove the i-th observation Y_i, we have the vector $\mathbf{Y}_{(i)} = [Y_1, Y_2, \ldots, Y_{i-1}, Y_{i+1}, \ldots, Y_n]^\top$, which corresponds to the original vector without the i-th observation. The jackknife estimator is obtained by applying this same function to the resulting vector without the i-th observation, being given by $\hat{\theta}_{(i)} = g(\mathbf{Y}_{(i)})$.

If we compute the mean of the n jackknife estimates, obtained by removing each of the n original observations, by

$$\bar{\hat{\theta}}_{(.)} = \frac{\sum_{i=1}^{n} \hat{\theta}_{(i)}}{n},$$

the acceleration $\hat{a}$ can be obtained by

$$\hat{a} = \frac{\sum_{i=1}^{n} \left(\bar{\hat{\theta}}_{(.)} - \hat{\theta}_{(i)}\right)^3}{6\left[\sum_{i=1}^{n} \left(\bar{\hat{\theta}}_{(.)} - \hat{\theta}_{(i)}\right)^2\right]^{3/2}}. \tag{8.15}$$

The quantity $\hat{a}$ is called acceleration because it represents the rate of change of the standard error of $\hat{\theta}$ relative to the change of the values of the true parameter θ (EFRON; TIBSHIRANI, 1993). This confidence interval has two advantages: a) it yields estimates of the confidence interval limits within the parameter space, that is, it has the property of respecting the transformation; and b) it provides very precise results, in the sense that the coverage probabilities $P(\theta < \tilde{\theta}_{(k_1)}) = \alpha/2$ and $P(\theta > \tilde{\theta}_{(k_2)}) = \alpha/2$ are observed with second-order accuracy, whereas the percentile bootstrap method has only first-order accuracy.

```
import numpy as np
import scipy as sp


# auxiliary function to compute the acceleration
def aceler(x, est, *args):
    n = len(x)
    ai = np.empty(0)
    for i in range(n):
```

```
        yi = np.delete(x, i)
        ai = np.append(ai, est(yi, *args))
    ab = np.mean(ai)
    a = np.sum((ab - ai)**3)/(6*(np.sum((ab - ai)**2))**1.5)
    return a

# function to obtain the CI with bias-corrected and accelerated correction
def ic_cva_boot(x, est, B = 2000, alpha = 0.05, *args):
    res = dist_boot(x, est, B, *args)
    est_boot = est(x, *args)
    p = np.sum(res < est_boot) / B
    z0 = sp.stats.norm.ppf(p)
    z1 = sp.stats.norm.ppf(alpha / 2)
    z2 = sp.stats.norm.ppf(1 - alpha / 2)
    a = aceler(x, est, *args)
    aux = z0 + (z0+z1)/(1-a*(z0+z1))
    p1 = sp.stats.norm.cdf(aux)
    aux = z0 + (z0+z2)/(1-a*(z0+z2))
    p2 = sp.stats.norm.cdf(aux)
    se_boot = np.std(res)
    mean_boot = np.mean(res)
    vies_boot = mean_boot - est_boot
    li, ls = np.percentile(res, [p1 * 100, p2 * 100])
    return {'estimate': est_boot, 'standard error': se_boot, \
            'bias': vies_boot, 'lower bound': li, 'upper bound': ls}


# Example for the gamma median
B = 2000
alpha = 0.05
res_med = ic_cva_boot(x, est_med, B, alpha)
for key, value in res_med.items():
    print(f"{key}: {value}")
# theoretical median
sp.stats.gamma.ppf(0.5, a)
```

```
estimate: 0.20319749915972624
standard error: 0.06478227907247332
bias: -0.0002117816301100417
lower bound: 0.09378609665950804
upper bound: 0.317644635048805
```

```
np.float64(0.227468211559786)
```

8.3.7 Bootstrap Confidence Interval with Bias Correction

This method is a variation of the previous method, in which the acceleration $\hat{a}$ is considered null. In that case, the confidence interval is obtained by the same expression (8.13), although the values of p_1 and p_2 are computed by

$$p_1 = \Phi\left(2\hat{z}_0 + z_{\alpha/2}\right) \text{ and } p_2 = \Phi\left(2\hat{z}_0 + z_{1-\alpha/2}\right), \tag{8.16}$$

where $\hat{z}_0$ is obtained in the same way as described earlier, through equation (8.14).

This method is particularly interesting in some situations where the values of $\hat{a}$ are difficult to obtain. These cases are discussed in Efron and Tibshirani (1993). However, the BC_a method is, in general, more efficient and should be preferred.

```
import numpy as np
import scipy as sp


# function to obtain the CI with bias correction
def ic_cv_boot(x, est, B = 2000, alpha = 0.05, *args):
    res = dist_boot(x, est, B, *args)
    est_boot = est(x, *args)
    p = np.sum(res < est_boot) / B
    z0 = sp.stats.norm.ppf(p)
    z1 = sp.stats.norm.ppf(alpha / 2)
    z2 = sp.stats.norm.ppf(1 - alpha / 2)
    aux = 2 * z0 + z1
    p1 = sp.stats.norm.cdf(aux)
    aux = 2 * z0 + z2
    p2 = sp.stats.norm.cdf(aux)
    se_boot = np.std(res)
    mean_boot = np.mean(res)
    vies_boot = mean_boot - est_boot
    li, ls = np.percentile(res, [p1 * 100, p2 * 100])
    return {'estimate': est_boot, 'standard error': se_boot, \
            'bias': vies_boot, 'lower bound': li, 'upper bound': ls}


# Example for the gamma median
B = 2000
alpha = 0.05
res_med = ic_cv_boot(x, est_med, B, alpha)
for key, value in res_med.items():
    print(f"{key}: {value}")
# theoretical median
sp.stats.gamma.ppf(0.5, a)
```

```
estimate: 0.20319749915972624
standard error: 0.0659610034790913
bias: -0.0001514521672552338
lower bound: 0.09378609665950804
upper bound: 0.3169784850854408
```

```
np.float64(0.227468211559786)
```

8.3.8 Bootstrap in Python

Among some possibilities, we can use `scipy.stats.bootstrap` in Python to apply the bootstrap interval estimation methods. The `bootstrap` function from `scipy` is given by `bootstrap(data, statistic, *, n_resamples=9999, batch=None, vectorized=None, paired=False, axis=0, confidence_level=0.95, alternative='two-sided', method='BCa', bootstrap_result=None, rng=None, random_state=None)`. Three methods are available: `method{'percentile', 'basic', 'BCa'}`, with `BCa` being the default method. The following program shows how to obtain the confidence interval by this `scipy` procedure.

The data must be converted into a tuple before using the bootstrap method. See the example:

```
import scipy as sp


# Demonstrating scipy.stats.bootstrap
# Example for the gamma median
B = 2000
alpha = 0.05
x = (x,) # data must be in a tuple of arrays
res_med = sp.stats.bootstrap(x, est_med, n_resamples=B, \
    confidence_level=1-alpha, method='percentile')
print(res_med.confidence_interval)
res_med = sp.stats.bootstrap(x, est_med, n_resamples=B, \
    method='basic', confidence_level=1-alpha)
print(res_med.confidence_interval)
res_med = sp.stats.bootstrap(x, est_med, n_resamples=B, \
    method='BCa', confidence_level=1-alpha)
print(res_med.confidence_interval)
# theoretical median
sp.stats.gamma.ppf(0.5, a)
```

```
ConfidenceInterval(low=np.float64(0.0985059996149705), high=np.float64(0.3179907375430121))
ConfidenceInterval(low=np.float64(0.08840426077644037), high=np.float64(0.30788899870448194))
ConfidenceInterval(low=np.float64(0.0985059996149705), high=np.float64(0.3179907375430121))
```

```
np.float64(0.227468211559786)
```

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. *Estatística computacional: uma introdução em turbo Pascal*. LTC - Livros Técnicos e Científicos Editora, Rio de Janeiro, 1988.

L. Devroye. Generating the maximum of independent identically distributed random variables. *Computers and Mathematics with Applications*, 6:305–315, 1980.

L. Devroye. *Non-uniform random variate generation*. Springer-Verlag, New York, 1986.

Bradley Efron and Robert J. Tibshirani. *An Introduction to the bootstrap*. Chapman and Hall, New York, 1993.

Daniel Furtado 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, 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

algorithm
  Hasting, 110

beta
  incomplete, 106

class
  Python, 128

density
  exponential, 51
  log-normal, 53
  normal, 53
  Tukey-lambda, 56
distribution
  binomial, 59
  multivariate t
    spherical, 80
  of linear
    combinations, 72
distribution function
  binomial, 102
  exponential, 97
  normal, 108

equation
  recursive, 88

function
  density
    exponential, 97
    inverse Wishart, 77
    multivariate normal, 71
    normal, 102
    Wishart, 77
  inverse distribution
    binomial, 62
    exponential, 52
  probability mass
    binomial, 59
    geometric, 60
functions
  trigonometric, 54

generation
  of variables
    multivariate t, 80
generator
  minimal
    standard, 39

inverse distribution function
  exponential, 97

Jacobian
  transformation, 53

language
  high-level, 38
lemma
  sum of binomials, 60
  waiting time
    exponential, 60
    geometric, 60

matrix
  covariances, 91
  sum of squares and products, 91
Mersenne
  Twister, 41
method
  Box-Muller, 53
  congruential, 37
  inversion
    discrete, 62
methods
  lists, 13

numbers
  pseudo-random, 37
  random, 37, 39
  uniform, 37

precision
  double, 40
probability function
  binomial, 102
  Poisson, 106

quadratures

Gaussian, 110
Monte Carlo, 111
quantiles
exponential, 97

random, 37
rule
trapezoidal
extended, 108

sampling
rejection, 50
simulation
random matrices
inverse Wishart, 78
Wishart, 78
statistics
order, 158
sum
of products, 89

theorem
probability integral transform, 49
tuples, 15

Wishart
inverse, 77