

Maurizio Petrelli

Introduction to Python in Earth Science Data Analysis

From Descriptive Statistics to Machine
Learning

March 3, 2021

Springer Nature

*To Agata, Anna, Caterina,
Atomo, and Arianna*

Contents

Part I Python for Geologists, a Kick-off

1	Setting Up Your Python Environment, Easily	3
1.1	The Python Programming Language	3
1.2	Programming Paradigms	4
1.3	A Local Python Environment for Scientific Computing	5
1.4	Remote Python Environments	7
1.5	Python Packages for Scientific Applications	8
1.6	Python Packages Specifically Developed for Geologists	9
2	Python Essentials for a Geologist	11
2.1	Start Working with the IPython Console	11
2.2	Naming and Style Conventions	14
2.3	Working with Python Scripts	15
2.4	Conditional Statements, Indentation, Loops, and Functions	17
2.5	Importing External Libraries	21
2.6	Basic Operations and Mathematical Functions	22
3	Starting Solving Geological Problems Using Python	25
3.1	My First Binary Diagram Using Python	25
3.2	Start Making Models in Earth Science	32
3.3	Quick Intro to Spatial Data Representation	37

Part II Describing Geological Data

4	Graphical Visualization of a Geological Data Set	43
4.1	The Statistical Description of a Data Set, Key Concepts	43
4.2	Visualizing Univariate Sample Distributions	44
4.3	Preparing Publication Ready Binary Diagrams	47
4.4	Visualization of Multivariate Data: a First Attempt	68

5	Descriptive Statistics 1: Univariate Analysis	71
5.1	Basics of Descriptive Statistics	71
5.2	Location	71
5.3	Dispersion or Scale	76
5.4	Skewness	81
5.5	Descriptive Statistics in Pandas	83
5.6	Box Plots	85
6	Descriptive Statistics 2: Bivariate Analysis	87
6.1	Covariance and Correlation	87
6.2	Simple Linear Regression	91
6.3	Polynomial Regression	92
6.4	Non-Linear Regression	94
Part III Integrals and Differential Equations in Geology		
7	Numerical Integration	103
7.1	Definite Integrals	103
7.2	Basic Properties of Integrals	104
7.3	Analytical and Numerical Solutions of Definite Integrals	105
7.4	Fundamental Theorem of Calculus and Analytical Solutions	105
7.5	Numerical Solutions of Definite Integrals	107
7.6	Computing the Volume of Geological Structures	113
7.7	Computing the Lithostatic Pressure	114
8	Differential Equations	121
8.1	Introduction	121
8.2	Ordinary Differential Equation	122
8.3	Numerical Solutions of First Order ODEs	127
8.4	The Fick's law of diffusion, a Widely Used PDE	131
Part IV Probability Density Functions and Error Analysis		
9	Probability Density Functions and Their Use in Geology	143
9.1	Probability Distribution and Density Functions (PDF)	143
9.2	The Normal Distribution	144
9.3	The Log-Normal Distribution	149
9.4	Other Useful PDFs for Geological Applications	151
9.5	Density Estimation	151
9.6	The Central Limit Theorem and Normal Distributed Means	158
10	Error Analysis	161
10.1	Dealing with Errors in Geological Measurements	161
10.2	Reporting Uncertainties in Binary Diagrams	169
10.3	The Linearized Approach in Error Propagation	173
10.4	The Monte Carlo Approach in Error Propagation	180

Part V Robust Statistics and Machine Learning

11 Introduction to Robust Statistics	189
11.1 Classical and Robust Approaches to Statistics	189
11.2 Normality Tests	190
11.3 Robust Estimators for Location and Scale	196
11.4 Robust Statistics in Geochemistry	202
12 Machine Learning	205
12.1 Introduction to Machine Learning in Geology	205
12.2 Machine Learning in Python	207
12.3 A Study Case of Machine Learning in Geology	208

Part VI Appendices

A Python Packages Specifically Developed for Geologists	221
B Introduction to Object Oriented Programming	225
B.1 Object-oriented programming	225
B.2 Defining classes, attributes and methods in Python	225
C The Matplotlib Object Oriented API	229
C.1 Matplotlib Application Programming Interfaces	229
C.2 Matplotlib Object Oriented API	229
C.3 Fine Tuning of Geological Diagrams using the OO-style	231
D Working with pandas	235
D.1 How to perform common operations in pandas	235
References and Further Readings	239

Part I
Python for Geologists, a Kick-off

Chapter 1

Setting Up Your Python Environment, Easily

1.1 The Python Programming Language

Python is a high-level, modular, and interpreted programming language¹. What does it mean? A high-level programming language is characterized by a strong abstraction from the details of the computer. It means that the code is easy to understand for humans. Python is modular, i.e., it supports modules and packages, which allows program flexibility and code reuse. In detail, Python is composed of a "core" that allows for all basic operations plus a wide ecosystem of specialized packages to perform specific tasks. To better understand, a Python package or library is a reusable portion of code, a collection of functions and modules (i.e., a group of functions) allowing us to complete specialized tasks (e.g., reading an excel file or drawing a publication-ready diagram). Python is an interpreted language (like MATLAB, Mathematica, Maple, and R). On the contrary, C or FORTRAN are compiled languages. What is the difference between compiled and interpreted languages? Roughly speaking, in compiled languages, a translator compiles each code listing in executable files. Once compiled, the target machine can run the resulting executable files directly. Interpreted-languages compile codes in real-time, during each execution. The main difference for a novice programmer is that interpreted codes typically run slower than compiled executable ones. However, performances are not an issue in most of everyday operations. Performances start becoming significant in computing-intensive tasks like complex fluid dynamic simulations or 3D graphical applications. If needed, Python performances can be significantly improved with the support of specific packages, like Numba, which are able to compile Python codes, approaching the speed of C and FORTRAN.

Being an interpreted language, Python allows easy code exchanges over different platforms (i.e., cross-platform), fast prototyping, and great flexibility.

Possible convincing arguments for Earth scientists to start learning Python are: 1) an easy to learn syntax; 2) great flexibility; 3) the support of a large community

¹ <https://www.python.org>

of users and developers; 4) it is free and open-source; 5) it will improve your skills and proficiency.

1.2 Programming Paradigms

A programming paradigm is a style, or general approach to writing codes (Gabbrielli & Martini, 2010; Turbak & Gifford, 2008; Van Roy & Haridi, 2004). As a zero-order approximation, we can identify two archetype paradigms for programming: imperative and declarative. The first one mainly focuses on "how" to solve a problem, the latter on "what" to solve. Starting from these two archetypes, programmers developed many derived paradigms. Examples of derived programming paradigms are the procedural, object-oriented, functional, logic, aspect-oriented, just to cite a few. The selection of a specific programming paradigm to develop your code depends on the overall nature of your project and final scope of your work. For parallel computing, the functional approach provide a well established framework. An exhaustive documentation about programming paradigms is outside the scope of the present book. Here, I will limit to the illustration of those that are supported in Python.

The Python programming language is primary minded for the object-oriented programming style, but it also supports, sometime spuriously, a purely imperative, procedural, and functional paradigms (Gabbrielli & Martini, 2010; Turbak & Gifford, 2008; Van Roy & Haridi, 2004):

Imperative: The imperative approach is the oldest and simplest programming paradigm. It is simply based on providing a defined sequence of instructions to a computer.

Procedural: the procedural approach is a sub-set of the imperative programming. Instead of simply providing a sequence of instructions, it stores portions of code in one or more procedures (i.e., subroutines or functions). Any given procedure can be called at any point during a program execution, allowing code organization and reuse.

Object-oriented: like the procedural style, the object-oriented approach is a sub-set (i.e., an evolution) of the imperative programming. Here, objects are the key elements. One of the main benefits of the object-oriented approach is that it maintains a strong relation with real-world entities (e.g., shopping carts in websites, WYSIWYG environments, etc...).

Functional: the functional approach is a declarative type of programming. The purely functional paradigm bases the computation in evaluating mathematical functions and it is well suited for high-load and parallel computing applications.

In this introductory book where we will take advantage of Python flexibility without focusing too much on specific code styling or a particular paradigm. Specifically,

our codes will be mainly imperative for the easiest tasks and procedural for more advanced modelling. Also, we will take benefit from many libraries (e.g., pandas and matplotlib) developed in an object-oriented fashion.

1.3 A Local Python Environment for Scientific Computing

There are two main strategies to create a Python environment suitable for scientific computing on your personal computer: 1) installing the Python core and adding all the required scientific packages separately; 2) installing a "ready-to-use" Python environment, specifically developed for scientific purposes. You can try both options but I suggest you to start with the second one as it requires almost zero programming skills and you will be ready to start immediately your journey in the Python world nearly painlessly.

An Example of "ready-to-use" scientific Python environment is the Anaconda Python Distribution². Anaconda Inc (previously Continuum Analytics) develops and maintain the Anaconda Python distribution providing different solutions that include a free release and two charged versions. The Individual Edition is the free option (and our choice), it is easy to install and it offers a community driven support. **To install the Individual Edition of the Anaconda Python distribution, I suggest following the directives reported in the official documentation³.** They simply consist in downloading and running the last stable installer for your Operating System (i.e., Windows OS, Mac OS or Linux). In the case of Windows and Mac OS, a graphical installer is available. The installation procedure is the same as for any other software application. The Anaconda installer will automatically install the Python core, the Anaconda Navigator, plus about 250 packages defining a complete environment for scientific visualization, analysis, and modelling. Over 7,500 additional packages can be individually installed, if needed, from the Anaconda repository with the "conda"⁴ package management system.

The Anaconda Navigator is a desktop Graphical User Interface (GUI), i.e., a program, that allows you to launch applications, install packages and manage environments without using command-line instructions (Fig. 1.1).

From the Anaconda Navigator, we can launch two of the main applications that we will use to write codes, run the modelling, and visualize the results. They are the Spyder application and the JupyterLab.

Spyder⁵ is an Integrated Development Environment (IDE), i.e., a software application, providing a set of comprehensive facilities for software development and scientific programming. It combines a text editor to write codes, inspection tools

² <https://www.anaconda.com>

³ <https://www.anaconda.com/products/individual/>

⁴ <https://docs.conda.io/>

⁵ <https://www.spyder-ide.org>

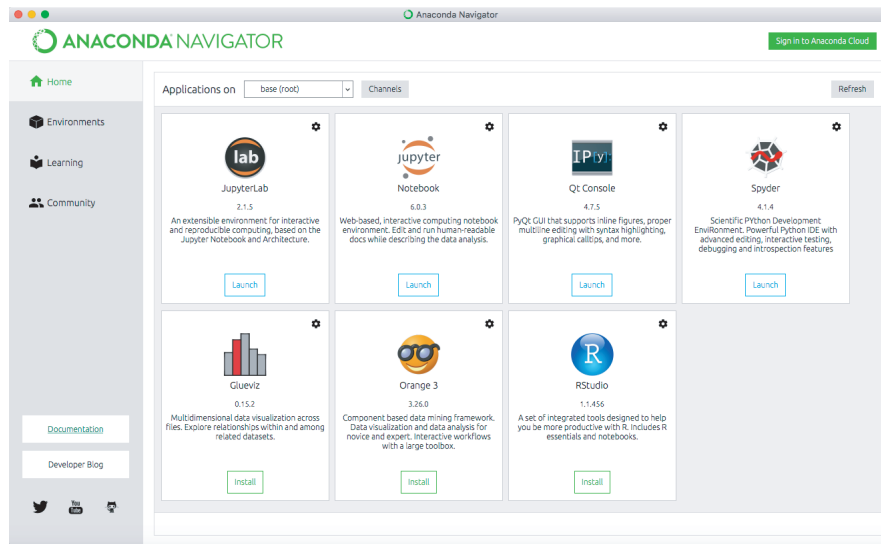


Fig. 1.1 Screenshot of the Anaconda Navigator.

for debugging, and an interactive Python console for code execution. We will spend most of our time using Spyder. Fig. 1.2 reports a screenshot of the Spider IDE.

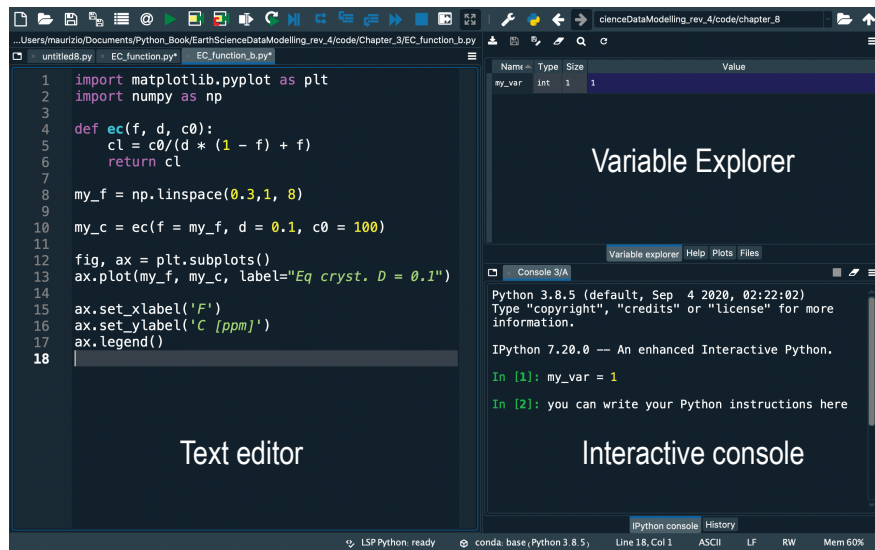


Fig. 1.2 Screenshot of the Spider IDE. On the left, we see the text editor to write our code. The bottom-right panel is the IPython interactive console. Finally, the top-right panel shows the Variable Explorer.

JupyterLab is a web-based development environment to manage Jupyter Notebooks. A Jupyter Notebook is a web application that allows creating and sharing documents containing live code, equations, visualizations and narrative text. Fig. 1.3 reports a screenshot of a Jupyter Notebook.

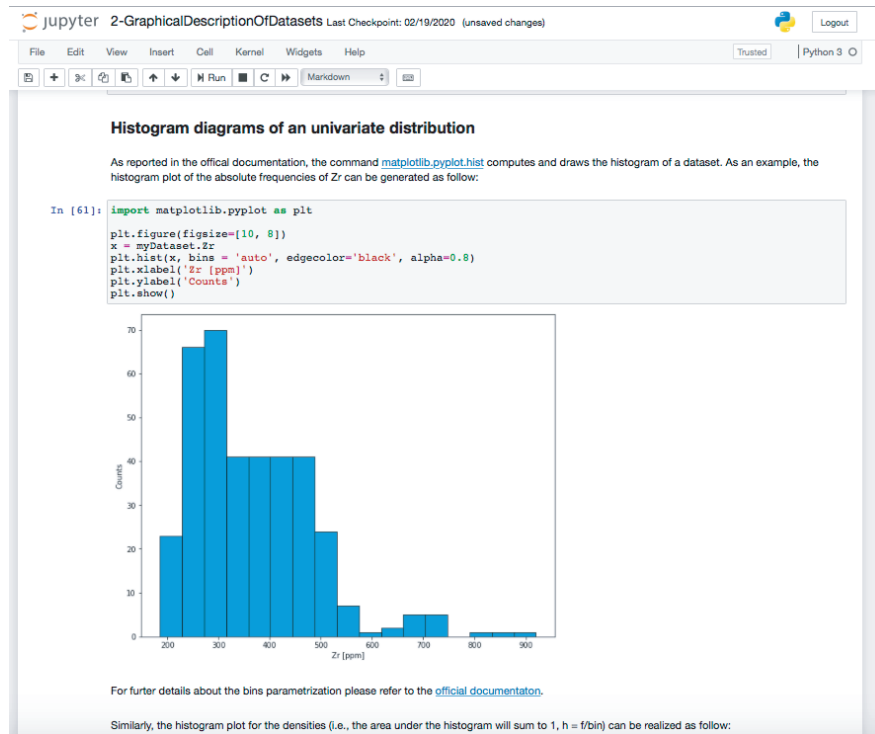


Fig. 1.3 Screenshot of a Jupyter Notebook. It combines narrative text, code, and visualizations.

Both Synder and JupyterLab allow writing code, perform the computing, and report the results. There is not a preferred choice. My personal choice was to use Synder and Jupyter Lab for research purposes and teaching, respectively.

1.4 Remote Python Environments

Remote Python environments are those running in a computer system or virtual machines that can be accessed online. As an example, Python environments can be installed on remote machines hosted by your academic institution (most of the universities manage centers of calculus offering this opportunity) or by commercial providers (often offering a basic free plan). The concepts and procedures I described for the installation of a local Python environment are still valid for remote machines.

However, working with remote machines will require additional skills to access and operate online (e.g, the knowledge of Secure Shell or Remote Desktop protocols for Linux and Windows-based machines respectively). Therefore, I suggest again starting with a local installation of the Anaconda Python distribution.

An alternative possibility to start working with Python online, without installing a local environment, could be the use of a remote IDE. As an example, commercial providers like Repl.it⁶ and PythonAnywhere⁷ offer a free and complete Python IDEs to start coding first, and develop advanced applications then. As a drawback, both the IDEs provided by Repl.it and PythonAnywhere are not specifically minded for scientific purposes. As a consequence, running the codes of this book will require the installation of additional libraries that are not included by default in the core distribution. Therefore, to easily replicate the codes and the examples reported in the present book, I suggest, one more time, the local installation of the most recent Anaconda Python distribution.

1.5 Python Packages for Scientific Applications

A key feature of Python is its modular nature. In this section, I list a few general purpose scientific packages that we will use widely in this book. For each library, I provide with a quick description taken from the official documentation, a link to the official website and, when possible, a reference for further readings.

NumPy is a Python library that provides a multidimensional array object and an assortment of routines for fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and other functionalities (Bressert, 2012)⁸.

Pandas is an open source library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language (Chen, 2017)⁹.

SciPy is a collection of mathematical algorithms and functions built on the NumPy extension of Python. It adds significant power to the interactive Python session by providing the user with high-level commands and classes for manipulating and visualizing data. With SciPy, an interactive Python session becomes a data-processing and system-prototyping environment rivaling systems such as MATLAB, IDL, Octave, R, and SciLab (Bressert, 2012)¹⁰.

⁶ <https://repl.it>

⁷ <https://www.pythonanywhere.com>

⁸ <https://numpy.org>

⁹ <https://pandas.pydata.org>

¹⁰ <https://scipy.org>

Matplotlib is a Python library for creating static, animated, and interactive data visualizations (Bisong, 2019)¹¹.

SymPy is a Python library for symbolic mathematics. Symbolic computation deals with the computation of mathematical objects symbolically. This means that the mathematical objects are represented exactly, not approximately, and mathematical expressions with unevaluated variables are left in symbolic form (Meurer et al., 2017)¹².

Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data pre-processing, model selection and evaluation, and many other utilities (Paper, 2020)¹³.

1.6 Python Packages Specifically Developed for Geologists

Many Python packages have been developed to solve geological problems. They form a wide, heterogeneous, and useful ecosystem allowing us to achieve specific geological tasks. Examples are Devito, ObsPy, and Pyrolyte to cite a few. Most of them can be installed easily by using the Conda package management system. Others requires a few additional steps and skills. The use of these specific packages is not covered in the present book, since they are typically developed to solve very specific geological problems. However, a novice to Python will benefit and probably require the notions reported in this book to approach these packages successfully. A comprehensive list of Python packages developed to solve geological tasks is given in the Appendix A.

¹¹ <https://matplotlib.org>

¹² <https://www.sympy.org>

¹³ <https://scikit-learn.org>

Chapter 2

Python Essentials for a Geologist

2.1 Start Working with the IPython Console

The IPython Console allows us to execute single instructions, multiple lines of code, and scripts receiving an output from Python (Rossant, 2018).

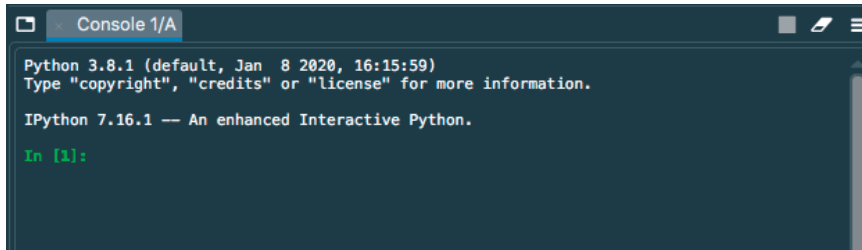
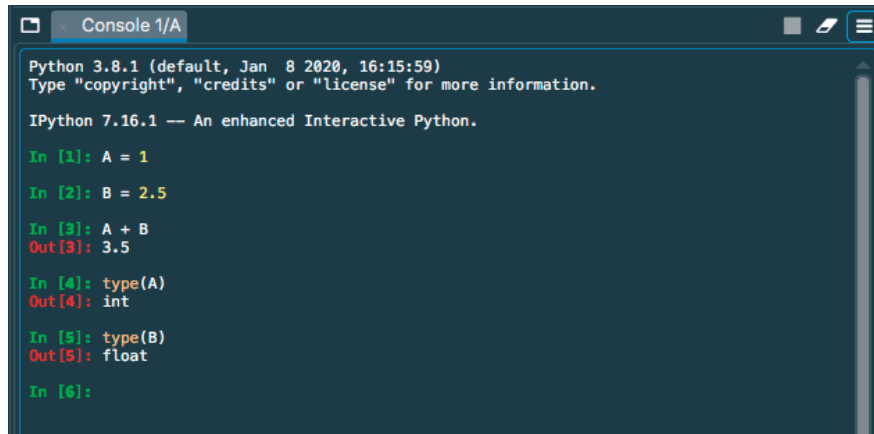


Fig. 2.1 The IPython console.

To start working with the IPython Console, see Fig. 2.2 where the first two instructions are $A = 1$ and $B = 2.5$. The meaning of these two commands is straightforward: they simply assign a value equal to 1 and 2.5 to the variables A and B , respectively. The third instruction is $A + B$ that sums the two variables A and B , obtaining the result 3.5.

Also, Fig. 2.2 provides information about the type of variables in Python (Fig. 2.3). Regarding numbers, Python supports integers, floating-point, and complex numbers. Integers and floating-point numbers differ by the presence or absence of decimals. In our case, A is an integer and B is a floating-point number. Complex numbers include a real part and an imaginary part and they are not discussed in this book. Operations like additions or subtractions automatically force integers to float-point numbers if one of the operands (in our case B) is of float type. The `type()` function returns the type of a variable. Additional data types that are relevant for the present book are: a) Boolean, i.e., True or False, b) Sequences, and c) Dictionaries.



```
Python 3.8.1 (default, Jan 8 2020, 16:15:59)
Type "copyright", "credits" or "license" for more information.

IPython 7.16.1 -- An enhanced Interactive Python.

In [1]: A = 1
In [2]: B = 2.5
In [3]: A + B
Out[3]: 3.5
In [4]: type(A)
Out[4]: int
In [5]: type(B)
Out[5]: float
In [6]:
```

Fig. 2.2 Start working with the IPython console.

In Python, a Sequence Type is an ordered collection of elements. Examples of sequences are: Strings, Lists, and Tuples. Strings are sequences of characters, Lists are ordered collections of data, and Tuples are similar to Lists, but they cannot be modified after their creation. Fig. 2.4 shows how to define and access Strings, Lists, and Tuples.

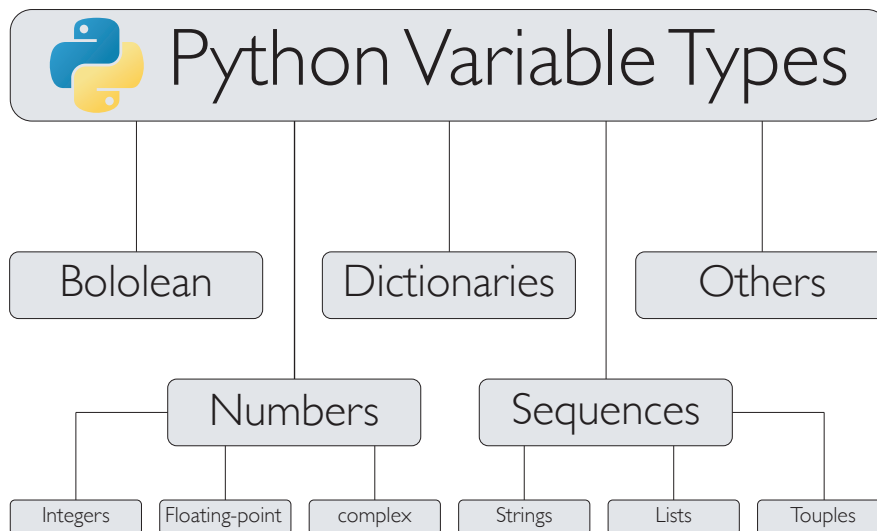
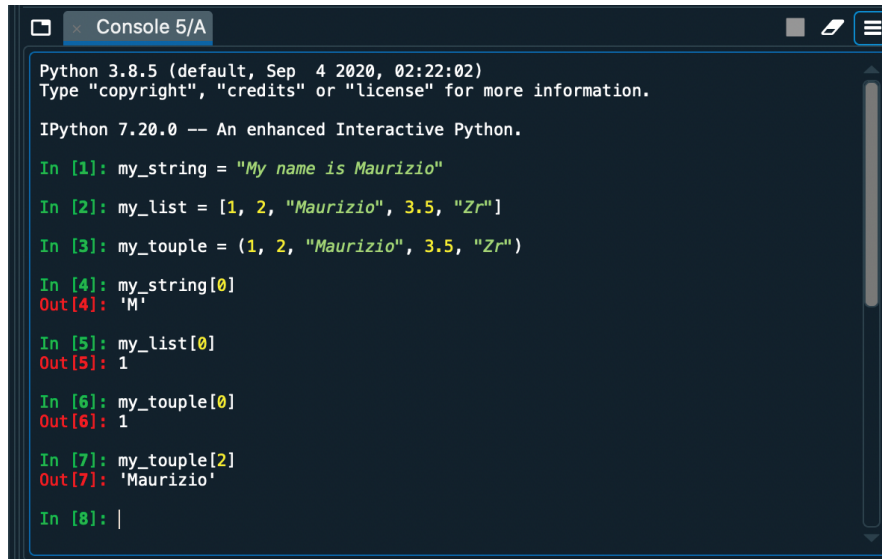


Fig. 2.3 Variable data types in Python.

The image shows a terminal window titled "Console 5/A" with a dark background. It displays the output of a Python 3.8.5 interpreter running IPython 7.20.0. The user has defined three variables: a string 'my_string' with the value "My name is Maurizio", a list 'my_list' with elements [1, 2, "Maurizio", 3.5, "Zr"], and a tuple 'my_touple' with elements (1, 2, "Maurizio", 3.5, "Zr"). The user then performs several indexing operations: my_string[0] returns 'M', my_list[0] returns 1, my_touple[0] returns 1, and my_touple[2] returns 'Maurizio'. The prompt 'In [8]: |' is visible at the bottom.

```
Python 3.8.5 (default, Sep 4 2020, 02:22:02)
Type "copyright", "credits" or "license" for more information.

IPython 7.20.0 -- An enhanced Interactive Python.

In [1]: my_string = "My name is Maurizio"

In [2]: my_list = [1, 2, "Maurizio", 3.5, "Zr"]

In [3]: my_touple = (1, 2, "Maurizio", 3.5, "Zr")

In [4]: my_string[0]
Out[4]: 'M'

In [5]: my_list[0]
Out[5]: 1

In [6]: my_touple[0]
Out[6]: 1

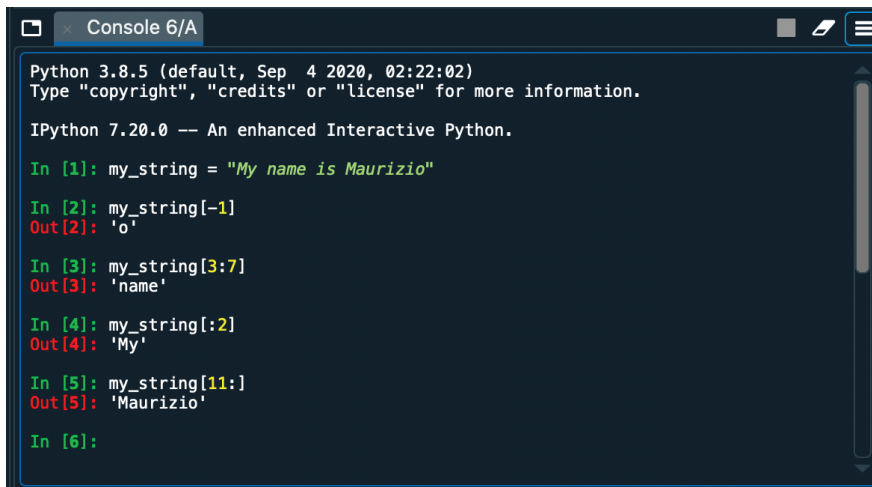
In [7]: my_touple[2]
Out[7]: 'Maurizio'

In [8]: |
```

Fig. 2.4 Defining and working with sequences.

The elements of a sequence can be accessed using indexes. In Python the first index of a sequence is always 0. As an example, the instruction `my_string[0]` returns the first element (i.e., "M") of the object `my_string` defined in Fig. 2.4. Similarly, `my_touple[2]` returns the third element of `my_touple` (i.e., "Maurizio"). Additional examples on how to access a sequence are reported in Fig. 2.5. Using negative numbers (e.g., `my_string[-1]`), the indexing of the sequences starts from the last element and proceeds in reverse mode. Two numbers separated by a colon (e.g. `[3:7]`) define an index range, sampling the sequence from the lower to the upper bounds, respectively, excluding the upper bound. In the case of the statement `my_string[3:7]`, the interpreter samples `my_string` from the third to the seventh indexes (i.e., 'name'). Finally, commands like `my_string[:2]` and `my_string[11:]` sample `my_string` from the beginning to the index 2 (excluded) and from the index 11 to the last element, respectively.

Dictionaries are data types consisting of a collection of key-value pairs. A dictionary can be defined by enclosing a comma-separated list of key-value pairs in curly braces, a colon separates each key from the associated value (Fig. 2.6). In a dictionary, a value is retrieved by specifying the corresponding key in square brackets (Fig. 2.6).



```

Python 3.8.5 (default, Sep 4 2020, 02:22:02)
Type "copyright", "credits" or "license" for more information.

IPython 7.20.0 -- An enhanced Interactive Python.

In [1]: my_string = "My name is Maurizio"

In [2]: my_string[-1]
Out[2]: 'o'

In [3]: my_string[3:7]
Out[3]: 'name'

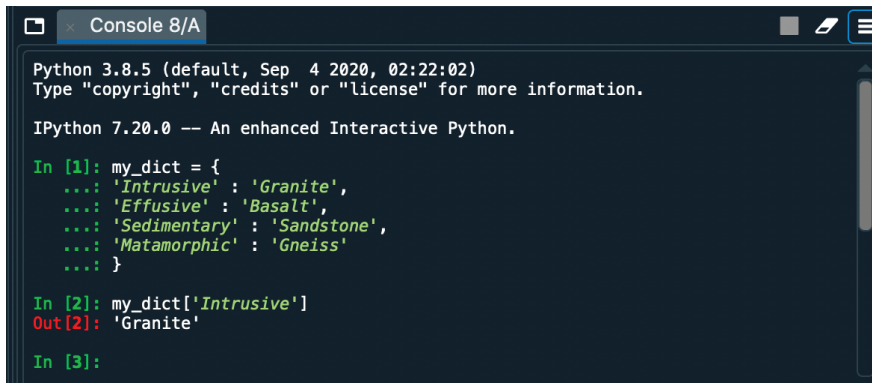
In [4]: my_string[:2]
Out[4]: 'My'

In [5]: my_string[11:]
Out[5]: 'Maurizio'

In [6]:

```

Fig. 2.5 Accessing Sequences.



```

Python 3.8.5 (default, Sep 4 2020, 02:22:02)
Type "copyright", "credits" or "license" for more information.

IPython 7.20.0 -- An enhanced Interactive Python.

In [1]: my_dict = {
...: 'Intrusive' : 'Granite',
...: 'Effusive' : 'Basalt',
...: 'Sedimentary' : 'Sandstone',
...: 'Matamorphic' : 'Gneiss'
...: }

In [2]: my_dict['Intrusive']
Out[2]: 'Granite'

In [3]:

```

Fig. 2.6 Defining and accessing Dictionaries.

2.2 Naming and Style Conventions

The main aim of using conventions in programming is to improve the readability of codes to facilitate the collaboration among different programmers. In Python the "PEP 8 – Style Guide for Python Code" gives coding conventions for the Python code, comprising the standard library in the main Python distribution¹.

Writing readable code here is meaningful for several reasons. The main one is to allow others to understand your code easily. This is crucial when you will start working on collaborative projects. Having common guidelines, it will allow the group to write consistent and elegant codes.

¹ <https://www.python.org/dev/peps/pep-0008/>

Within the book, I will try to follow the main rules defined by the "PEP 8 – Style Guide for Python Code". Please apologize me when I will take some poetic, i.e., coding, licenses. Table 4 reports few main coding conventions reported in the "PEP 8 – Style Guide for Python Code".

Table 2.1 Styling and Naming conventions in Python.

Type	Style or Naming Convention	Action
Function	Function names should be lowercase, with words separated by underscores as necessary to improve readability (cf. section 2.4).	function, my_function
Variable	Variable names follow the same convention as function names.	x, my_dataset
Constant	Constants are usually written in all capital letters with underscores separating words.	A, GREEK_P
Class	Start each word with a capital letter (CapWords convention). Do not use underscores to separate subsequent words (cf. appendix B).	Circle, MyClass
Method	Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability (cf. appendix B).	method, my_method
Names to avoid	Never use the characters 'l' (lowercase letter el), 'O' (uppercase letter oh), or 'I' (uppercase letter eye) as single character variable names.	
Indentation	the PEP 8 recommends the use of 4 spaces per indentation level (cf. section 2.4).	

2.3 Working with Python Scripts

A script is a sequence of code instructions used to automate processes (e.g., making a diagram, or a geological model) that would otherwise need to be executed step-by-step (e.g., in the IPython console). In detail, Python scripts are text files typically characterized by a .py extension and containing a sequence of Python instructions. To write and modify Python scripts, we only require a text editor. Spyder incorporates a text editor with advanced features (e.g., code completion and syntax inspection). In Spyder, the text editor is usually positioned in a panel on the left portion of the screen. During the execution of a Python script, the interpreter reads each instruction sequentially, starting from the first line . To execute a Python script in the active IPython console of Spyder, we can click the play button as reported in Fig. 2.7 or use the F5 keyboard shortcut. Keyboard shortcuts helps us in being more proficient. Few additional Keyboard shortcuts are shown in Table 2.2.

The script listing 2.1 reports the Python script of Fig. 2.7 with the output obtained in the IPython console incorporated at the end (lines 5 to 10).

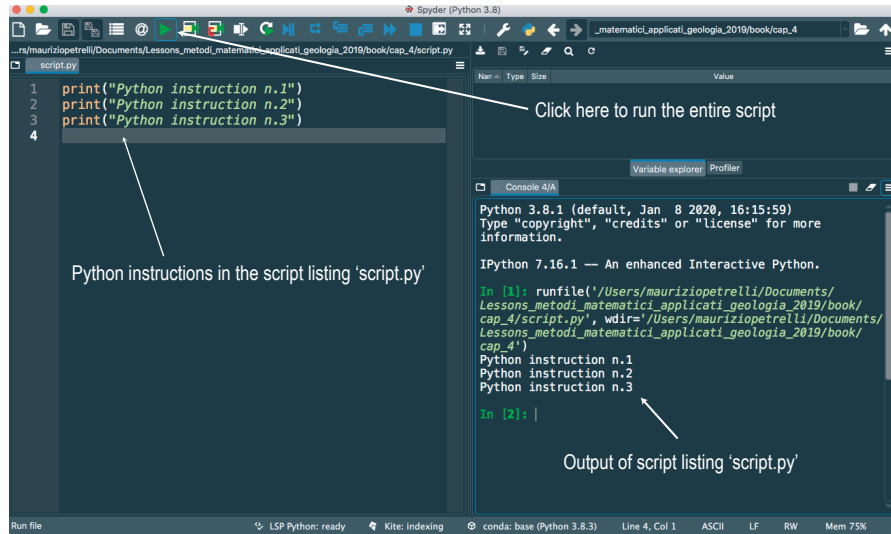


Fig. 2.7 Running a Python script.

The three single quotation marks (i.e., `'''`) positioned at lines 5 and 10 of the script listing 2.1 open and close a multi-line comment (i.e., lines of code which will not be considered by the interpreter). The symbol `#` define a single line comment. Comments are a fundamental part of Python codes since they help you and future users to clarify the code workflow. Take in mind that you might spend an entire day in developing a very proficient script; you wake up the day after without remembering how does the script work. Comments help a lot in these situations.

Table 2.2 Selected Spyder Keyboard Shortcuts.

Windows OS	Mac OS	Action
F5	F5	Run file (complete script)
F9	F9	Run selection (or current line)
Ctrl + T	Cmd + T	Open an IPython console
Ctrl + space	Ctrl + space	Code completion
Tab	Tab	Indent selected line(s)
Shift + Tab	Shift + Tab	Unindent selected line(s)
Ctrl + Q	Cmd + Q	Quit Spyder

As a matter of facts you don't necessary need Spyder to write a .py script. As stated above, Python scripts can be written using any text editor. The *python* instruction will run your scripts in the command line or terminal applications.

```
1 print("Python instruction n.1")
2 print("Python instruction n.2")
3 print("Python instruction n.3")
4
5 '''
6 Output:
7 Python instruction n.1
8 Python instruction n.2
9 Python instruction n.3
10 '''
```

Listing 2.1 Iterate a series of numbers.

2.4 Conditional Statements, Indentation, Loops, and Functions

Conditional Statements

In Python, the *if* statement allows for the conditional execution of a single or multiple instructions based on the value of an expression. To understand, consider the script listing 2.2. At line 1, we define the `MyVar` variable assigning to it a value equal to 2. At line 3, the *if* statement starts evaluating `MyVar`. In the specific case it executes the instruction at line 4 only in the case when `MyVar` is larger than 2. As it is not our case, the interpreter jumps to line 5 evaluating if `MyVar` is equal to 2. Please note that when we assign a value to a variable, we use `=`, whereas `==` is used to make comparisons. Being `MyVar` equal to 2, the interpreter executes the instructions from line 6 to 8. Finally, the instruction at line 10 is executed in all the remaining cases, i.e., `MyVar` less than 2.

```
1 MyVar = 2
2
3 if MyVar > 2:
4     print('MyVar is greater than 2')
5 elif MyVar == 2:
6     print('MyVar is equal to 1')
7     # more instructions could be added
8     # using the same indentation
9 else:
10    print('MyVar is less than 2')
11
12 '''
13 Output:
14 MyVar is equal to 2
15 '''
```

Listing 2.2 If, elif, else statements.

Indentation and Blocks

The term indentation refers to adding one or more white spaces before an instruction. In a Python script, contiguous instructions (e.g., lines from 6 to 8 of the script listing 2.2) that are indented to the same level are considered to be part of the same block. A block is considered by the interpreter as a single entity and it allows to structure our Python scripts. As an example, the blocks after the `if`, `elif`, and `else` statements in the script listing 2.2 are executed in agreement with the conditions reported at lines 3, 5, and 9, respectively. To better understand how the indentation works in Python, consider the code listing 2.3. The instructions from line 1 to 3 and at line 12 are always executed each time we run the script. The interpreter executes the instructions at lines 5, 9, 10, and 11 if, and only if, the variable `A` is equal to 1. Finally, the interpreter executes lines 7 and 8 when `A` and `B` are equal to 1 and 3, respectively.

```
1 # this instruction is always executed
2 # this instruction is always executed
3 # this instruction is always executed
4 if A==1:
5     # this instruction is executed if A = 1
6     if B = 3:
7         # this instruction is executed if A = 1 and B = 3
8         # this instruction is executed if A = 1 and B = 3
9     # this instruction is executed if A = 1
10    # this instruction is executed if A = 1
11    # this instruction is executed if A = 1
12 # this instruction is always executed
```

Listing 2.3 Iterate a series of numbers.

Take in mind that the indentation is a fundamental concept in Python, allowing the definition of simple operations like conditional statements, loops, and functions, but also more complex structures like modules, and packages.

For Loops

The *for* loop in Python iterates over a sequence (i.e, lists, tuples, and strings) or other iterable objects. As an example, the code listing 2.4 displays an iteration over the list named `rocks`. In detail, at line 1 we define a list (i.e., `rocks`), at line 3 we implement the iteration, and at line 4 we print on the screen the result of the iterations, i.e., each element of the sequence.

Often, we perform iterations using `range()`. The command `range()` is a Python function which returns a sequence of integers.

```
1 rocks = ['sedimentary', 'igneous intrusive', 'igneous effusive',  
          'methamorphic']  
2  
3 for rock in rocks:  
4     print(rock)  
5  
6 '''  
7 Output:  
8 sedimentary  
9 igneous intrusive  
10 igneous effusive  
11 methamorphic  
12 '''
```

Listing 2.4 Iterate over a list.

```
1 print('a sequence from 0 to 2')  
2 for i in range(3):  
3     print(i)  
4  
5 print('-----')  
6 print('a sequence from 2 to 4')  
7 for i in range(2,5):  
8     print(i)  
9  
10 print('-----')  
11 print('a sequence from 2 to 8 with a step of 2')  
12 for i in range(2,9,2):  
13     print(i)  
14  
15 '''  
16 Output:  
17 a sequence from 0 to 2  
18 0  
19 1  
20 2  
21 -----  
22 a sequence from 2 to 4  
23 2  
24 3  
25 4  
26 -----  
27 a sequence from 2 to 8 with a step of 2  
28 2  
29 4  
30 6  
31 8  
32 '''
```

Listing 2.5 Plotting an Histogram distribution as probability density in Python.

The range syntax is `range(start, stop, step)` where `start`, `stop`, and `step` parameters are the initial, the final, and the step values of the sequence, respectively. Note that the upper limit (i.e., `stop`) is not included in the sequence. If we pass only one argument to the range function [e.g., `range(6)`], it is interpreted as the stop parameter, with the sequence starting from 0. The code listing 2.5 reports some examples of iterations over a sequence of numbers generated using the `range()` function.

While Loops

The `while` loop checks a test-condition. In detail, the loop starts only if the test-condition is `True`. After each iteration, the test-condition is checked again and the loop continues until the test-condition is `False`. To better understand, consider the code listing 2.6. At line 1, we define the object `MyVar` and we assign to it a value equal to 0. At line 3, we start evaluating the test-condition `MyVar < 5`. Being `MyVar` equal to 0, test-condition is `True` and the interpreter enters the loop. At line 4, it prints `MyVar` (i.e., 0), and at line 5, `MyVar` is updated to a value equal to 1. Now, the test-condition is evaluated again, and the loop goes ahead until it remains `True` (i.e., `MyVar < 5`). As a consequence, the interpreter repeats the instructions at lines 4 and 5 (i.e., the block of code with the same indentation after the test condition), until `MyVar` reaches the value of 5.

```
1 MyVar = 0
2
3 while MyVar < 5:
4     print(MyVar)
5     MyVar = MyVar + 1
6
7 '''
8 Output:
9 0
10 1
11 2
12 3
13 4
14 '''
```

Listing 2.6 Iterate a series of numbers.

Functions

A function is a block of reusable code that is developed to complete a specific task. Function blocks begin with the keyword `def` followed by the name of the function and parentheses (e.g., code listing 2.7). Input parameters or arguments should be placed within these parentheses. The code block of a function starts after a colon

(:) and, therefore, it has to be indented. Using the optional statement `return`, we can pass back to the caller a single or multiple answers (for example some variables computed within the function). The code listing 2.7 shows how to define and use a simple function. At line 1, we define a function, named `sum`, accepting two input parameters: `a` and `b`. At line 3, the function computes the variable `c` by summing `a` and `b`. Finally, the function ends at line 3, returning `c` to the caller. At line 5, we define the variable `res` by calling the function `sum` with `a=2` and `b=3` as input parameters. At line 6, we print a string containing the value of `res`. Note that the `str()` function converts numbers to strings.

```
1 def sum(a, b):
2     c = a + b
3     return c
4
5 res = sum(a=2, b=3)
6 print('the result is ' + str(res))
7
8 '''
9 Output:
10 the result is 5
11 '''
```

Listing 2.7 Defining a function.

2.5 Importing External Libraries

The Anaconda Python distribution includes almost all the packages that are used for the most common operations in Data Science. Examples are NumPy, SciPy, pandas, matplotlib, seaborn, and scikit-learn, briefly described in section 1.5, just to cite a few. The `import` and `from` statements allow us to import entire modules, packages, or single functions in our scripts. The code listing 2.8 reports examples of using the `import` and `from` statements. Note that at line 1 of the code listing 2.8, we are importing the entire pandas package in an object named `pd`. At line 2, we import the matplotlib module `pyplot` as `plt`. At line 3, the `random()` function is imported from the NumPy package. Finally, at line 4, we import all the functions in the SymPy package.

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from numpy import random
4 from sympy import *
```

Listing 2.8 Use of `import` and `from` statements.

2.6 Basic Operations and Mathematical Functions

Basic mathematical operators like the sum or the multiplications are always available in Python and listed in Table 2.3.

Additional trigonometric and arithmetic functions can be accessed using the math and NumPy libraries. Also, these libraries contain relevant constants like the π (Archimede's constant) and e (Euler's number). The main difference between the math of NumPy libraries relies in the fact that the first is minded to work with scalars and the second is optimized to operate with arrays. However, NumPy works well with scalars too. Being NumPy more flexible than math, in the present book I will provide examples using the NumPy library only. Tables 2.4 and 2.5 report some relevant NumPy constants and functions, respectively. Also, the code listing 2.9 report some introductory examples on how to use NumPy constants and mathematical functions.

Table 2.3 Basic mathematical operations in Python.

Operator	Description	Example	Operator	Description	Example
+	Addition	$3 + 2 = 5$	-	Subtraction	$3 - 2 = 1$
*	Multiplication	$3 * 2 = 6$	/	Division	$6 / 2 = 3$
**	Power	$3 ** 2 = 9$	%	Modulus	$2 \% 2 = 0$

Table 2.4 Relevant constants in NumPy.

Numpy	Description	Value	Numpy	Description	Value
e	Euler's number (e)	2.718...	pi	Archimedes' const. (π)	3.141...
euler_gamma	Euler's constant (γ)	0.577...	inf	positive infinity	∞

Table 2.5 Introducing exponents, logarithms, and trigonometric, functions in NumPy.

Numpy	Description	Numpy	Description	Numpy	Description
sin()	Trigonom. sine	cos()	Trigonom. cosine	tan()	Trigonom. tangent
arcsin()	Inverse sine	arccos()	Inverse cosine	arctan()	Inverse tangent
exp()	Exponential	log()	Natural logarithm	log10()	Base 10 logarithm
log2()	Base-2 logarithm	sqrt()	Square-root	abs()	Absolute value

```
1 import numpy as np #import numpy
2
3 # relevant constants
4 greek_p = np.pi
5 euler_number = np.e
6
7 # print greek_p and euler_number on the screen
8 print("The Archimedes' constant is " + str(greek_p))
9 print("The Euler's number is " + str(euler_number))
10
11 # trigonometric functions
12 x = np.sin(greek_p / 2) # x = 1 expected
13
14 # print the result on the screen
15 print("the sine of a quarter of radiant is " + str(x))
16
17
18 # defining a 1D array in numpy
19 myArray = np.array([4, 8, 27])
20 # print myArray on the screen
21 print("myArray is equal to " + str(myArray))
22
23 log10_myArray = np.log10(myArray)
24
25 # print the result on the screen
26 print("The the base 10 logarithm of the elements in myArray is")
27 print(log10_myArray)
28
29 '''
30 Output:
31 The Archimedes' constant is 3.141592653589793
32 The Euler's number is 2.718281828459045
33 the sine of a quarter of radiant is 1.0
34 myArray is equal to [ 4  8 27]
35 The the base 10 logarithm of the elements in myArray is
36 [0.60205999 0.90308999 1.43136376]
37 '''
```

Listing 2.9 Our first approach with NumPy.

Now, we are now ready to begin learning how to solve geological problems using Python.

Chapter 3

Starting Solving Geological Problems Using Python

3.1 My First Binary Diagram Using Python

We start learning Python for the analysis of geological data performing two basic operations: importing data set using the pandas library and their representation in binary diagrams. Let's start with importing the data set using the pandas library. As introduced in section 1.5, pandas is a Python library (i.e., a tool) designed to help in working with structured data. In the practice, it provides several, ready to use, functions to work with scientific data. As an example, we can easily use pandas to import a data set stored in an Excel spreadsheet or a text file using a single row of code. To understand, look at the code listing 3.1.

```
1 import pandas as pd
2
3 #Example 1
4 myDataset1 = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5                             sheet_name='Supp_traces')
```

Listing 3.1 Importing an Excel file in Python.

At line 1, we import the pandas library by creating an object named `pd`. Now, the `pd` object stores all pandas functionalities.

At line 4, we define a pandas DataFrame (i.e., `myDataset1`) reading an Excel file named `'Smith_glass_post_NYT_data.xlsx'`. Also, being an Excel file potentially made of several spreadsheets, we point to a specific spreadsheet: `Supp_traces`. The imported data set contains trace element chemical concentrations of volcanic tephra published by Smith et al. (2011). It will be used as a representative proxy of a scientific geological data set. In detail, it consists of major (`Supp_majors`) and trace element (`Supp_traces`) analyses of tephra samples belonging to the recent activity (last 15 ky) of the Campi Flegrei Caldera (Italy).

The instruction `pd.read_excel()` accepts many input parameters allowing great flexibility. Two of the most important are: 1) a valid string path and 2) the

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Analysis no.	Strat. Pos.	Eruption	controlcode	Sample	Epoch	Crater size	Date of analysis	Cs	Ba	La	Ce	Pr	Nd	
298	2211	19	Soccavo 4	8	4-1	one	20	040810am	17.36	1365	70.67	119.50	13.26	49.72	
299	2212	19	Soccavo 4	8	4-1	one	20	040810am	19.29	1300	62.17	111.87	12.46	41.37	
300	2213	19	Soccavo 4	8	4-1	one	20	040810am	17.54	1401	69.18	118.96	13.96	50.49	
301	2214	19	Soccavo 4	8	4-1	one	20	040810am	18.01	1268	59.81	110.25	12.31	44.40	
302	2215	19	Soccavo 4	8	4-1	one	20	040810am	17.12	1356	60.52	114.70	12.69	47.94	
303	2216	16	Paleopisani 2	7	23	one	20	030810pm	16.04	1409	52.69	97.20	11.43	43.99	
304	2217	16	Paleopisani 2	7	23	one	20	030810pm	14.00	1344	56.04	100.41	11.54	43.07	
305	2218	16	Paleopisani 2	7	23	one	20	030810pm	14.25	1225	50.78	90.90	10.23	38.90	
306	2219	16	Paleopisani 2	7	23	one	20	030810pm	14.14	1207	48.33	87.64	9.99	42.95	
307	2220	16	Paleopisani 2	7	23	one	20	030810pm	15.83	1520	58.01	109.36	12.96	46.77	
308	2221	16	Paleopisani 2	7	23	one	20	030810pm	13.84	1355	54.99	98.57	11.84	42.18	
309	2222	16	Paleopisani 2	7	23	one	20	030810pm	14.87	1274	51.95	96.76	11.18	43.73	
310	2223	16	Paleopisani 2	7	23	one	20	030810pm	14.30	1307	53.83	96.29	11.73	42.71	
311	2224	16	Paleopisani 2	7	23	one	20	030810pm	13.31	1118	45.32	81.61	9.72	36.67	
312	2225	16	Paleopisani 2	7	23	one	20	030810pm	14.86	1268	52.49	97.48	11.32	42.25	
313	2226	16	Paleopisani 2	7	23	one	20	030810pm	14.87	1329	55.61	94.22	11.81	42.29	
314	2227	16	Paleopisani 2	7	23	one	20	030810pm	13.19	1320	53.71	98.70	10.81	42.15	
315	2228	16	Paleopisani 2	7	23	one	20	030810pm	12.41	1476	52.59	100.95	11.97	45.62	
316	2229	16	Paleopisani 2	7	23	one	20	030810pm	13.82	1317	54.07	96.72	12.06	46.10	

Fig. 3.1 The Smith_glass_post_NYT_data.xlsx Excel file.

sheet_name. In our case, the string path is the name of the Excel file (i.e., 'Smith_glass_post_NYT_data.xlsx'). If we provide the file name only as string path, the Python script and the Excel file must be placed in the same folder. Additional allowed string paths are local file addresses (e.g., '/Users/mauriziopetrelli/Documents/file.xlsx') or valid URL schemes, including http, ftp, and s3. Regarding the *sheet_name* parameter, it can be a string, an integer, a list, or None. The default value is 0, meaning that it opens the first sheet of the Excel file. In detail, integers and strings indicate sheet positions starting from 0 and the sheet names, respectively. Finally, lists of strings or integers are used to request multiple sheets.

As stated before, at line 4 of the code listing 3.1 we defined a DataFrame. What is a DataFrame? A DataFrame "is a 2-dimensional labeled data structure with columns of potentially different types"¹. What does it mean? We can imagine a DataFrame as a simple table, where Python has the full control.

To start plotting, we introduce an additional library named matplotlib. Matplotlib is "a comprehensive library for creating static, animated, and interactive visualizations in Python. It is a Python 2D plotting library which produces publication quality figures in a variety of hardcopy formats and interactive environments across platforms"². It generates plots, histograms, power spectra, bar charts, scatter-plots, etc..., with just a few lines of code. Matplotlib allows two different styles of writing: a) the pyplot and the object oriented Application Programming Interfaces (APIs). In matplotlib.pyplot (i.e., pyplot-style), each function operates a change to the active

¹ https://pandas.pydata.org/pandas-docs/stable/user_guide/dsintro.html

² <https://matplotlib.org>

figure. In the practice, to each command corresponds an effect on your diagram and it can be easily minded, organized, and managed in the imperative, i.e., the most basic and easiest (cf. section 1.2), way of coding. As drawback, `matplotlib.pyplot` is less-flexible and powerful than the matplotlib object-oriented interface (i.e., the OO-style). To note, learning the OO-style is not more difficult than `pyplot`. As a consequence, my idea is to start familiarizing with the OO-style directly starting with easy examples, then going in deeper details (see Appendix C).

As an example, the code listing 3.2 shows how to make a simple binary diagram using the OO-style. In detail, the code listing 3.2 highlights how to plot the elements Th against Zr in a scatter diagram. The workflow is simple: at lines 1 and 2, we import the pandas library and `matplotlib.pyplot` module, respectively. We know already the meaning of line 4, i.e., importing the 'Smith_glass_post_NYT_data.xlsx' Excel file into a DataFrame named `myDataset1`. At lines 6 and 7, we define two sequences of data selecting the columns Zr and Th, respectively, from `myDataset1`. At line 9, we generate a figure (i.e., the object `fig`) containing only one Axes (`ax`). Now, the terms Figure and Axes need additional clarifications. In matplotlib, the Figure object represent the whole diagram whereas the Axes are what you typically think when using the word 'plot' (Appendix C). A given figure can host a single (i.e., a simple diagram) or many Axes (i.e., a figure containing two or more sub-plots).

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset1 = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5                             sheet_name='Supp_traces')
6
7 x = myDataset1.Zr
8 y = myDataset1.Th
9
10 fig, ax = plt.subplots() # Create a figure containing one axes
11 ax.scatter(x, y)
```

Listing 3.2 First attempt in making a binary diagram in Python.

Although the diagram reported in Fig. 3.2 is a good start for a novice, it misses many mandatory information (e.g., axis label). So, let's start adding features to the diagram. As an example, `ax.set_title()`, `ax.set_xlabel()`, and `ax.set_ylabel()` add a title and labels to the x- and y-axis, respectively. Figure 3.3 displays the diagram of Fig. 3.2, updated with a new title and axes labels. To improve our skills about the use of Python in the visualization of scientific data look at the code listing 3.4 showing a procedure to slice a data set. In detail, lines 2 and 3 (code listing 3.4) describe how to divide the original data set (i.e., `myDataset1`) into two sub data sets (i.e., `mySubDataset1` and `mySubDataset2`) characterized by Zr contents above and below 450 ppm, respectively.

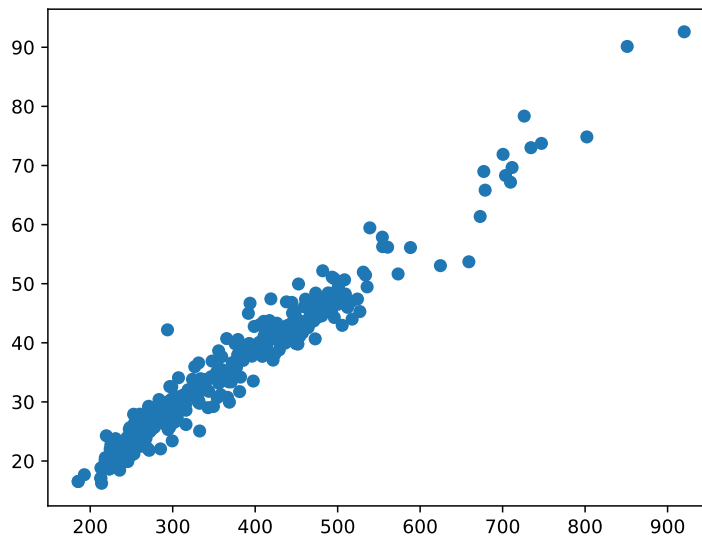


Fig. 3.2 The result of our first attempt in making a binary diagram in Python.

```

1 fig, ax = plt.subplots()
2 ax.scatter(x, y)
3 ax.set_title("My First Diagram")
4 ax.set_xlabel("Zr [ppm]")
5 ax.set_ylabel("Th [ppm]")

```

Listing 3.3 Second attempt: a binary diagram in Python including a title and axis labels.

Then, `mySubDataset1` and `mySubDataset2` are plotted at lines 11 and 16, respectively. Note that all the plotting instances (i.e., lines 11 and 16) after the command `plt.subplots()`, display the results in the same figure. Fig. 3.4 shows the result of code listing 3.4.

We now continue with an additional example of DataFrame slicing. In detail, the code listing 3.5, shows how to filter the original data set using the labels reported in 'Epoch' column. These labels divide the eruptions in four different periods, i.e., one, two, three, and three-b. After the slicing in sub data sets (lines 3, 6, 9, and 12), samples belonging to different Epochs are plotted using unique labels (lines 4, 7, 10, and 13, respectively).

The reader already familiar with the Python programming language may suggest a way to compress the code reported above making it concise, and more elegant using a loop (code listing 3.6).

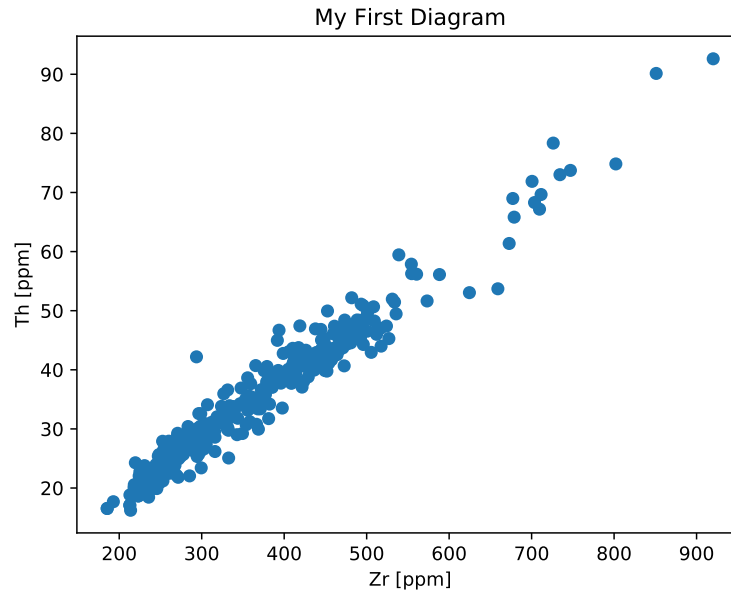


Fig. 3.3 The result of our second attempt in making a binary diagram in Python; it now includes a title and axis labels.

```

1 # Define two sub-dataset for Zr>450 and Zr<450 respectively
2 mySubDataset1= myDataset1[myDataset1.Zr> 450]
3 mySubDataset2= myDataset1[myDataset1.Zr< 450]
4
5 #generate a new picture
6 fig, ax = plt.subplots()
7 # Generate the scatter Zr Vs Th diagram for Zr > 450
8 # in blue also defining the legend caption as "Zr > 450 [ppm]"
9 x1 = mySubDataset1.Zr
10 y1 = mySubDataset1.Th
11 ax.scatter(x1, y1, color='blue', label= "Zr > 450 [ppm]")
12 # Generate the scatter Zr Vs Th diagram for Zr < 450
13 # in red also defining the legend caption as "Zr < 450 [ppm]"
14 x2 = mySubDataset2.Zr
15 y2 = mySubDataset2.Th
16 ax.scatter(x2, y2, color='red', label= "Zr < 450 [ppm]")
17
18 ax.set_title("My Second Diagram")
19 ax.set_xlabel("Zr [ppm]")
20 ax.set_ylabel("Th [ppm]")
21 # generate the legend
22 ax.legend()

```

Listing 3.4 Making a binary diagram with a sub-sampling (i.e., Zr>450 and Zr<450 ppb, respectively) of the original data set.

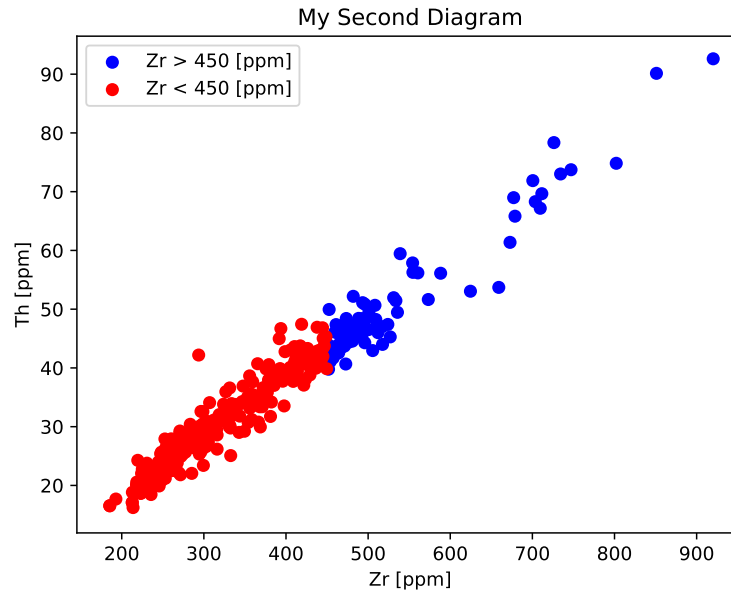


Fig. 3.4 The result of the code reported in the listing 3.4.

```

1 fig, ax = plt.subplots()
2
3 myData1 = myDataset1[(myDataset1.Epoch == 'one')]
4 ax.scatter(myData1.Zr, myData1.Th, label='Epoch 1')
5
6 myData2 = myDataset1[(myDataset1.Epoch == 'two')]
7 ax.scatter(myData2.Zr, myData2.Th, label='Epoch 2')
8
9 myData3 = myDataset1[(myDataset1.Epoch == 'three')]
10 ax.scatter(myData3.Zr, myData3.Th, label='Epoch 3')
11
12 myData4 = myDataset1[(myDataset1.Epoch == 'three-b')]
13 ax.scatter(myData4.Zr, myData4.Th, label='Epoch 3b')
14
15 ax.set_title("My Third Diagram")
16 ax.set_xlabel("Zr [ppm]")
17 ax.set_ylabel("Th [ppm]")
18 ax.legend()

```

Listing 3.5 Binary diagram with a sub-sampling (i.e., using the labels of the 'Epoch' column) of the original data set.

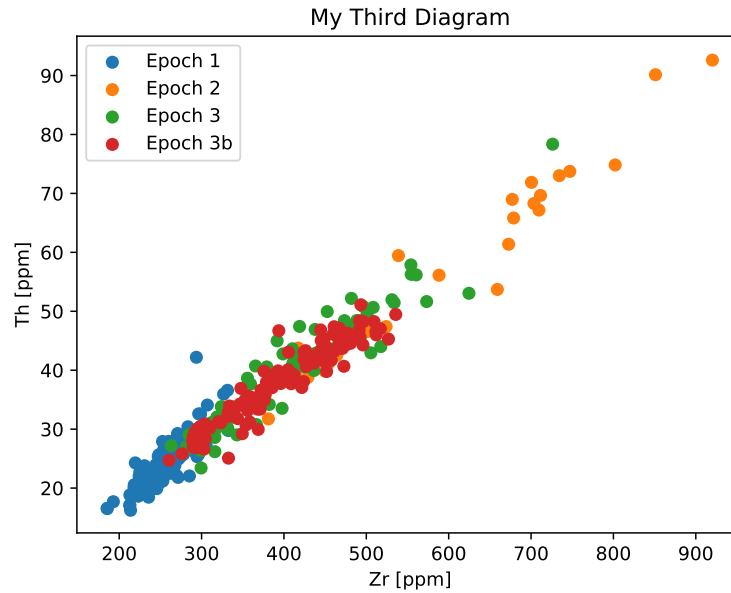


Fig. 3.5 The result of the code reported in the listing 3.5.

```

1 epochs = ['one', 'two', 'three', 'three-b']
2
3 fig, ax = plt.subplots()
4 for epoch in epochs:
5     myData = myDataset1[(myDataset1.Epoch == epoch)]
6     ax.scatter(myData.Zr, myData.Th, label="Epoch " + epoch)
7
8 ax.set_title("My Third Diagram again")
9 ax.set_xlabel("Zr [ppm]")
10 ax.set_ylabel("Th [ppm]")
11 ax.legend()

```

Listing 3.6 Re-writing the code reported in the listing 3.5 using a *for* loop.

As learned in the section 2.4 the *for* loop is utilized in Python to make iterations. You should become proficient in the use of loops, conditional statements, and functions (section 2.4). However, many everyday operations and tasks can be completed successfully without a deep knowledge of the syntax and “core semantics” of the Python language. So let’s solve our first geological problems.

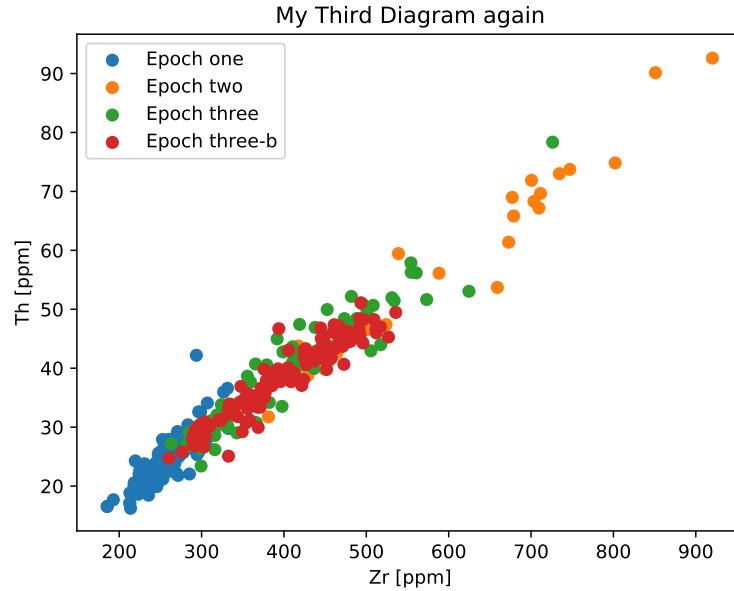


Fig. 3.6 The result of the code reported in the listing 3.6

3.2 Start Making Models in Earth Science

The development of a simple model in the field of Earth Sciences can provide useful information regarding Python syntax, workflow and way of thinking. This section shows how to develop a simple function (section 2.4) describing the evolution of trace elements in a magmatic system. In detail, the relation reported in Eq. 3.1 describes the evolution of the concentration (C) of a trace element in the liquid phase of a magmatic system during the process of crystallization at the thermodynamic equilibrium (Rollinson, 1993):

$$C = \frac{C_0}{D(1 - F) + F} \quad (3.1)$$

Where C_0 , D , and F are the initial concentration, the bulk partition coefficient of the trace element between melt and crystal, and the relative amount of melt in the system, respectively. The code listing 3.7 describes how to develop a function to solve the relation reported in Eq. 3.1.

In the code listing 3.7, at line 1 we define a function named `EC` accepting F , D , and $C0$ as input parameters. Now, moving to line 2, we observe that the code is indented. Remind that the indentation refers to the spaces that are used at the beginning of a row (section 2.4). All consecutive rows characterized by the same

indentation belong to the same block of code. In our case, indented lines 2 and 3 are part of the function EC.

```

1 def EC(F, D, C0):
2     CL = C0/(D*(1-F)+F)
3     return CL
4
5 MyC = EC(F=0.5, D=0.1, C0=100)
6
7 print('RESULT: ' + str(int(MyC)) + ' ppm')
8
9 '''
10 Output:
11 RESULT: 181 ppm
12 '''

```

Listing 3.7 Defining a function in python to model the Eq. 3.1.

In detail, we perform the computation and provide the results at lines 2 and 3, respectively. At line 5, we recall the function EC and we compute the concentration of a trace element in melt phase of a system characterized by F , D , and C_0 equal to 0.5, 0.1 and 100 ppm, respectively. The result is 181 ppm, printed on the screen at line 7. The line 7 of the code listing 3.7 requires further explanations. In detail, the statement *print()* reports the content within brackets on the screen, whereas the functions *str()* and *int()* convert a number to a string and a decimal number to an integer, respectively.

Code listing 3.8 reports a more exhaustive investigation of Eq. 3.1 for F values ranging from 1.0 to 0.3. For the readers that are not familiar with the Eq. 3.1, the Figure 3.7 shows the behaviour of an incompatible elements ($D < 1$, i.e., those elements that not easily enters in the crystals that are growing in the system and prefers remaining in the melt phase) from a completely molten system (i.e. $F = 1$) to a magmatic mush characterized by relative amount of melt in the order of 0.3.

The meaning of the statement at line 1 of code listing 3.8 is now straightforward: it imports *matplotlib.pyplot* functionalities in our script. At line 2, we import the NumPy library. As introduced in section 1.5, NumPy is a package for scientific computing, able to manage N-dimensional arrays, linear algebra, Fourier transform, and random numbers. The meaning of lines 4-6 is also straightforward: they define the EC function as in the code listing 3.7.

The use of NumPy starts at line 8 with the statement *np.linspace(0.3, 1, 8)*. It generates a 1D array, made of 8 elements, starting at 0.3 and ending at 1.0. The code listing 3.9 shows the result of printing *myF* on the screen.

Moving back to code listing 3.8, at line 10 we call the EC function using *myF* (i.e., a 1D array of 8 elements), 0.1, and 100 ppm as input parameters for F , D , and C_0 , respectively. The result is *MyC*, a 1D array of 8 element, one for each element of the array *MyF*. At line 13, we plot the results (i.e. *MyF* vs. *MyC*) using the instruction *ax.plot()*. By default, it plots a binary diagram connecting successive points using a line. Fig. 3.7 displays the result of the code listing 3.8.

```
1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 def EC(F, D, C0):
5     CL = C0/(D*(1-F)+F)
6     return CL
7
8 myF = np.linspace(0.3,1, 8)
9
10 myC = EC(F=myF,D=0.1,C0=100)
11
12 fig, ax = plt.subplots()
13 ax.plot(myF, myC, label="Eq cryst. D = 0.1")
14
15 ax.set_xlabel('F')
16 ax.set_ylabel('C [ppm]')
17 ax.legend()
```

Listing 3.8 Exploring the Eq. 3.1 in the F range from 0.3 to 1 and plot the results.

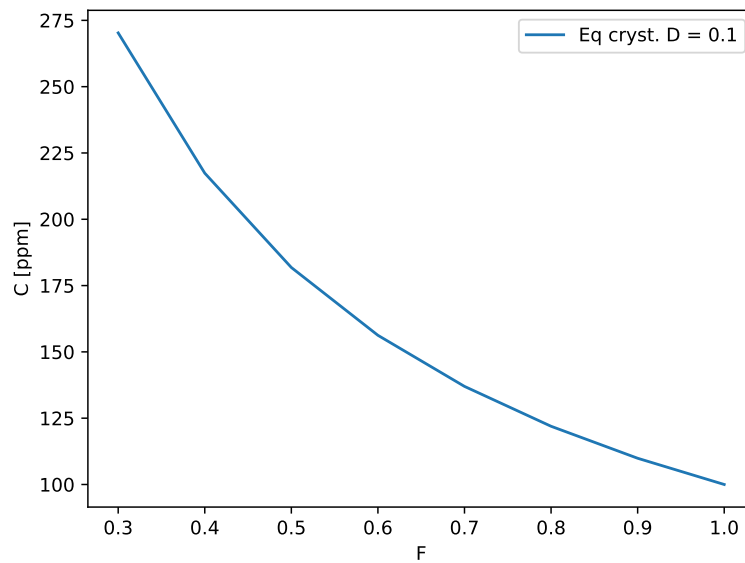


Fig. 3.7 The result of the code reported in the code listing 3.8

```

1 myF = np.linspace(0.3,1, 8)
2
3 print(myF)
4
5 '''
6 Output:
7 [0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
8 '''

```

Listing 3.9 The np.linspace() statement.

```

1 myF = np.arange(0,10, 1)
2
3 print(myF)
4
5 '''
6 Output:
7 [0 1 2 3 4 5 6 7 8 9]
8 '''

```

Listing 3.10 The np.arange() statement.

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 def EC(F, D, C0):
5     CL = C0/(D*(1-F)+F)
6     return CL
7
8 myF = np.linspace(0.3,1, 8)
9
10 myC1 = EC(F=myF,D=0.1,C0=100)
11 myC2 = EC(F=myF,D=1,C0=100)
12 myC3 = EC(F=myF,D=2,C0=100)
13
14 fig, ax = plt.subplots()
15 ax.plot(myF, myC1, label="Eq. 3.1 for D = 0.1")
16 ax.plot(myF, myC2, label="Eq. 3.1 for D = 1")
17 ax.plot(myF, myC3, label="Eq. 3.1 for D = 2")
18
19 ax.set_xlabel('F')
20 ax.set_ylabel('C [ppm]')
21 ax.legend()

```

Listing 3.11 Exploring the Eq. 3.1 for different values of D .

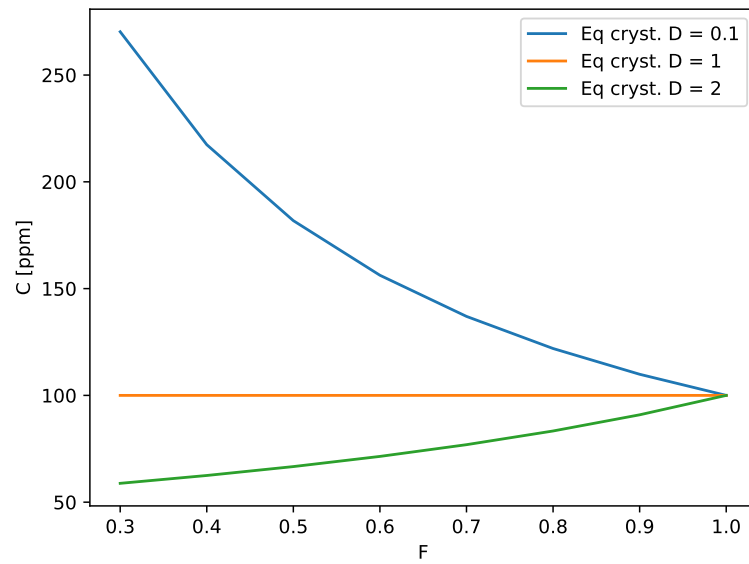


Fig. 3.8 The result of the code reported in the code listing 3.11 and 3.12.

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 def EC(F, D, C0):
5     CL = C0/(D*(1-F)+F)
6     return CL
7
8 myF = np.linspace(0.3,1, 8)
9
10 D = [0.1, 1, 2]
11
12 fig, ax = plt.subplots()
13
14 for myD in D:
15     myC = EC(F=myF, D=myD, C0=100)
16     ax.plot(myF, myC, label='Eq cryst. D = ' + str(myD))
17
18
19 ax.set_xlabel('F')
20 ax.set_ylabel('C [ppm]')
21 ax.legend()

```

Listing 3.12 Exploring the Eq. 3.1 for different values of D , using a loop.

There are many other ways to define a 1D array in NumPy. As an example, the `np.arange(start, stop, step)` functionality provide a similar way to obtain a 1D array (code listing 3.10).

To investigate the behaviour of the Eq. 3.1 for different D values, we could proceed as reported in code listing 3.11. Here, we define three models for different D values (lines 10, 11, and 12). Then, we plot the results in a single diagram (lines 15, 16, and 17) generated at line 14.

Again, the code listing 3.11, although easy to understand for a novice, it is not elegant, neither efficient. The code listing 3.12, shows how to obtain the same results of the code listing 3.11, but using a loop instead of defining each model separately.

3.3 Quick Intro to Spatial Data Representation

Visualizing spatial data is a fundamental task in geology. It has application in many fields like geomorphology, hydrology, volcanology, and geochemistry to cite just a few.

In this section, we will perform a simple task to start familiarizing with spatial data: import a data elevation model (DEM) stored in a .csv file and display each point using a color proportional to the elevation value. A .csv file is a text-file containing data separated by a delimiter like a comma, a tab, or a semicolon. To begin, let's start evaluating the data-set stored in the DEM.csv file (Fig. 3.9). It consists of four columns: an unique index, the elevation, the x-coordinates, and the y-coordinates, respectively (3.9). The data set refers to the Umbria region in Italy, where I currently live.

```

1 POINTID,ELEVATION,X_LOC,Y_LOC
2 1,594.9099731,2306636.664,4832459.757
3 2,1041.869995,2294636.664,4831959.757
4 3,1022.51001,2295136.664,4831959.757
5 4,647.7299805,2305136.664,4831959.757
6 5,634.6699829,2306636.664,4831959.757
7 6,649.2800293,2307136.664,4831959.757
8 7,909.7600098,2294136.664,4831459.757
9 8,945.9000244,2294636.664,4831459.757
10 9,979.1699829,2295136.664,4831459.757
11 10,715.2000122,2304636.664,4831459.757
12 11,623.0599976,2305136.664,4831459.757
13 12,640.5,2305636.664,4831459.757
14 13,565.4099731,2306136.664,4831459.757
15 14,660.9500122,2306636.664,4831459.757
16 15,623.1799927,2307136.664,4831459.757

```

Fig. 3.9 The DEM.csv comma delimited file.

Now, look at the code listing 3.13. At line 1 and 2, we import the pandas library and the matplotlib.pyplot subpackage. As you already know, they are collections of functions and methods to manage and plot scientific data.

The command `pd.read_csv()` at line 4 imports the .csv file named DEM.csv creating a new Dataframe (i.e., a table) named MyData. In detail, MyData now contains four columns named POINTID, ELEVATION, X_LOC, and Y_LOC, respectively. They refer to a unique identifier (POINTID), the elevation value (ELEVATION), and the (x,y) coordinates. The lines 6, 7, and 8 define three 1D arrays selecting the columns X_LOC, Y_LOC, and ELEVATION to be plotted successively. At line 10 the command `plt.subplots()` generates Figure containing a single Axes. Finally, the command `ax.scatter()` at line 11 creates a scatter plot filling each X_LOC and Y_LOC coordinate with a color proportional to the ELEVATION value (Fig. 3.10). The parameter `cmap='hot'` within the introduction `ax.scatter()` at line 11 of code listing 3.13 sets the colorbar to 'hot'. In this case, the lowest and the highest values of the colorbar, set at line 14, are characterized by a black and white color respectively. Intermediate colors follow the sequence of optical emissions of a dark body becoming progressively hotter (Fig. 3.10). Lines from 15 to 17 provide instructions to plot the colorbar (line 15), set the colorbar label (line 16), and the color of colorbar edges (line 17). Figure 3.11 show the result of code listing 3.13, setting `cmap` equal to 'hot'. Figure 3.12 reports most of colormaps available in matplotlib.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from matplotlib import cm
4
5 MyData = pd.read_csv('DEM.csv')
6 fig, ax = plt.subplots()
7 ax.scatter(x = MyData.X_LOC.values,
8           y = MyData.Y_LOC.values,
9           c=MyData.ELEVATION.values,
10          s=2, cmap='hot', linewidth=0, marker='o')
11 ax.axis('equal')
12 ax.axis('off')
13 colorbar = fig.colorbar(cm.ScalarMappable(cmap='hot'), extend='
    max', ax=ax)
14 colorbar.set_label('Elevation [m]', rotation=270, labelpad=20)
15 colorbar.outline.set_edgecolor('Grey')
```

Listing 3.13 Importing a data elevation model (DEM) stored in a .csv file and displaying the data as a scatter plot.

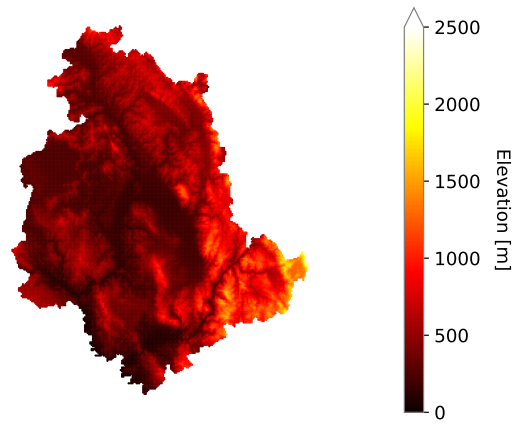


Fig. 3.10 The result of the code reported in the code 3.13.

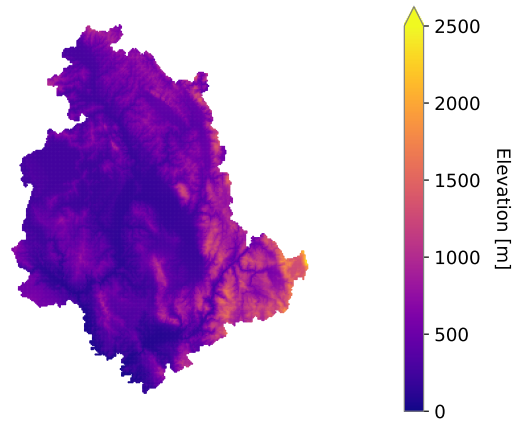


Fig. 3.11 The same of Fig. 3.10, but setting cmap='plasma'.

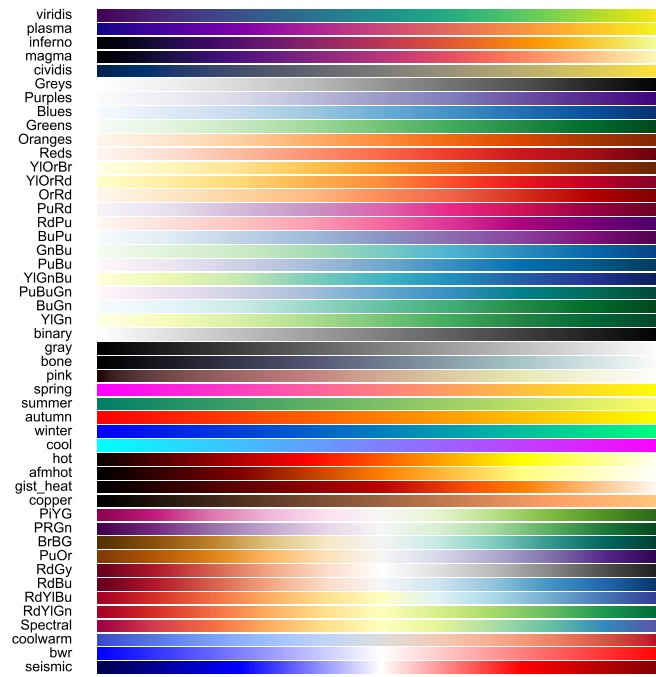


Fig. 3.12 Examples of colormaps potentially used in matplotlib.

Part II
Describing Geological Data

Chapter 4

Graphical Visualization of a Geological Data Set

4.1 The Statistical Description of a Data Set, Key Concepts

As reported by Ross (2017), "Statistics is the art of learning from data. It is concerned with the collection of data, their subsequent description, and their analysis, which often leads to the drawing of conclusions." In this section, we are going to provide some basic definitions for a proficient description of a geological data set. Note that data visualization is of paramount importance to understand data (Tufte, 2001). As a consequence, visualizing data should always come before any advanced statistical modelling (Healy, 2019; Tufte, 2001).

Population: The population is the set of all the elements of our interest. As an example, suppose to collect the strikes and dips of planar features (e.g., bedding planes, foliation planes, fold axial planes, fault planes, and joints) in a selected area. The population of the strikes is the set of all strikes, e.g., for a specific feature. Typically, the whole population cannot be measured, so we are forced to analyze a restricted sample of the population (Ross, 2017).

Sample: a subgroup of the population that will be studied in detail is called a sample (Ross, 2017). Examples are set of measurements of strikes, spring discharges rates or the acquisition of CO_2 flow rates for selected locations in volcanic areas. In geology, a piece of rock to be analysed is also called 'sample'. This is because it derives from the sampling of a specific rock formation (i.e., the population).

Discrete and continuous data: discrete data can only assume specific values. An example of discrete data is the number of springs in a specific area. Data are continuous when they can take any value within a range. The results of whole rock analyses and the measurements of flow discharge rates for springs are examples of continuous data (Ross, 2017).

Frequency distribution of a sample: a frequency distribution of a sample is a representation displaying the number of observations within a given interval. It could be either in tabular or graphical form (Ross, 2017).

4.2 Visualizing Univariate Sample Distributions

Histograms

A histogram is a bar-graph diagram where bars are placed adjacent to each other. It provides a qualitative description of an univariate sample distribution. The vertical axis of a histogram diagram can represent either absolute class frequencies, relative class frequencies, or probability densities. The intervals (i.e., bins) are contiguous and are often, but they are not required to be, of equal size.

The visual inspection of a histogram diagram provide many important information, including: 1) the degree of symmetry of the distribution; 2) its spread; 3) the presence of one or more classes characterized by high frequencies; 4) the occurrence of gaps; 5) the presence of outliers.

In Python, the instruction `matplotlib.axes.Axes.hist()` generates and draws histograms with great flexibility. As an example, consider the code listing 4.1. At lines 1 and 2, the pandas library and matplotlib.pyplot module are imported, respectively. At line 4, we define a DataFrame (i.e., myDataset) by importing the 'Supp_traces' spreadsheet of the 'Smith_glass_post_NYT_data.xlsx' file. At line 6, we generate a new Figure containing a single Axes. At line 7, we plot the histogram for the Zr column in myDataset.

The arguments *bins* define: 1) as a integer, the number of bins; 2) as a sequence, the edges of bins. In the specific case, *bins = 'auto'* uses a matplotlib internal method to estimate the optimal number of bins¹. The arguments *color* and *edgecolor* define the color of bar filling and bar edges, respectively. Finally, the argument *alpha* define the transparency. More details on how to customize an histogram diagram in matplotlib can be found in the official documentation²

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5                           sheet_name='Supp_traces')
6
7 fig, ax = plt.subplots()
8 ax.hist(myDataset.Zr, bins = 'auto', edgecolor='black', color='
9         tab:blue', alpha=0.8)
10 ax.set_xlabel('Zr [ppm]')
11 ax.set_ylabel('Counts')
```

Listing 4.1 Plotting an Histogram distribution using absolute frequencies in Python.

The code listing 4.2 performs the same operations of the code listing 4.1, but adding the instruction *density = True* at line 7. Using *density = True*, the y axes is

¹ https://numpy.org/doc/stable/reference/generated/numpy.histogram_bin_edges.html

² https://matplotlib.org/stable/api/_as_gen/matplotlib.axes.Axes.hist.html

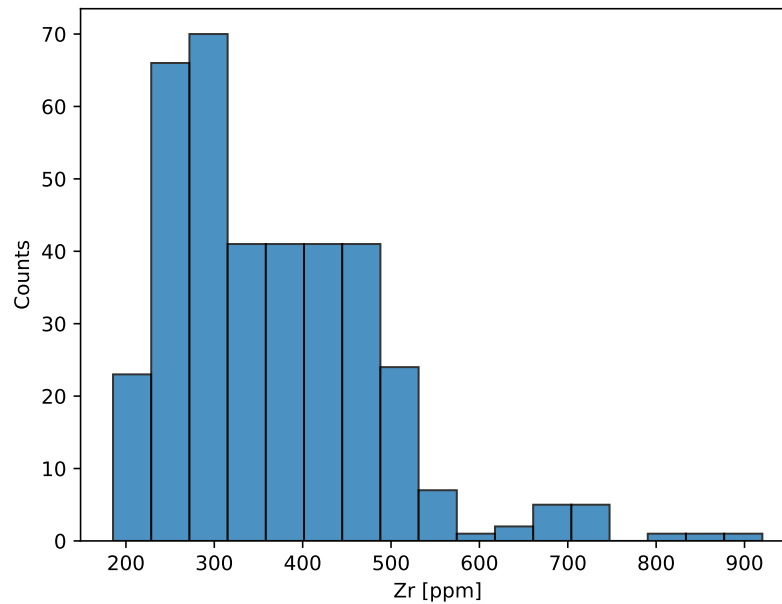


Fig. 4.1 The result of the code listing 4.1.

reported as probability density. In this case, the area under the whole histogram, i.e., the integral, will sum to 1). This is achieved by dividing the absolute frequencies by bin widths. The use of probability densities correspond to a first attempt in approximating a probability distribution, described in chapter 9.

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5                           sheet_name='Supp_traces')
6
7 fig, ax = plt.subplots()
8 ax.hist(myDataset.Zr, bins = 'auto', edgecolor='black', color=
9         'tab:blue', alpha=0.8, density = True)
10 ax.set_xlabel('Zr [ppm]')
11 ax.set_ylabel('Counts')
```

Listing 4.2 Plotting a histogram distribution as probability density in Python.

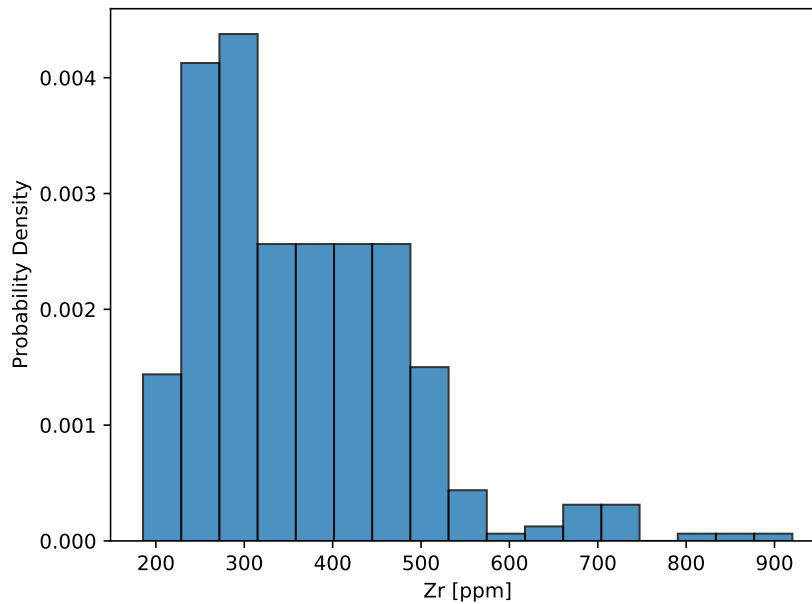


Fig. 4.2 The result of the code listing 4.2.

Plot of a cumulative distribution

A cumulative distribution function (CDF, also known as cumulative density function) of a distribution, evaluated at the value x , tells us the probability to get values less than or equal to x . The script listing 4.3 displays how to plot a cumulative distribution using `hist()`. It consists in adding the argument `cumulative = 1` or `cumulative = True` to the `hist()` instruction. The parameter `histtype='step'` avoid the filling of the area below the cumulative distribution. Finally the parameters `linewidth` and `color` define the line width and color, respectively.

```

1 fig, ax = plt.subplots()
2 ax.hist(myDataset.Zr, bins='auto', density=True, histtype='step',
3         linewidth=2, cumulative=1, color='tab:blue')
4 ax.set_xlabel('Zr [ppm]')
5 ax.set_ylabel('Likelihood of occurrence')
```

Listing 4.3 Plotting a cumulative distribution in Python.

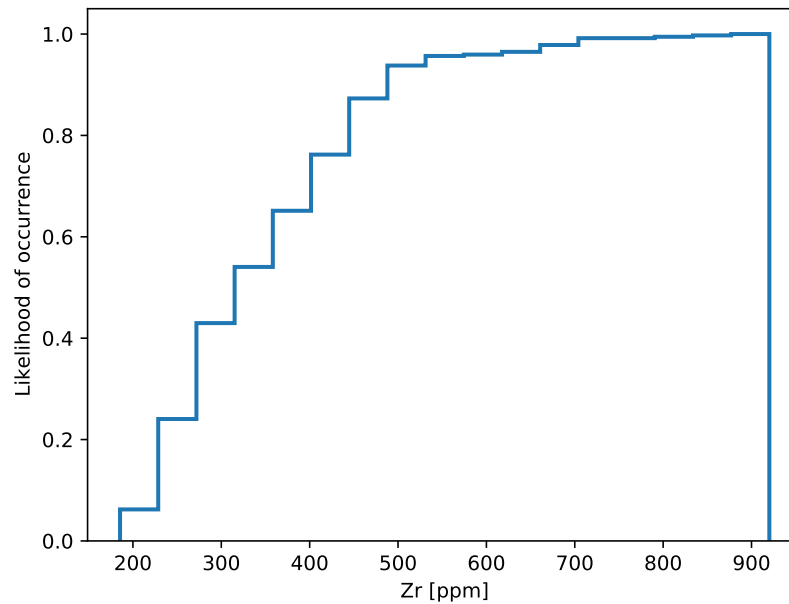


Fig. 4.3 The result of the code listing 4.3.

4.3 Preparing Publication Ready Binary Diagrams

Subplots

There are many options to create multiple subplots in matplotlib. In my opinion, the easiest is to create an empty figure, i.e., `fig = plt.figure()`, then adding multiple Axes (i.e., subplots), using the method `fig.add_subplot(nrows, ncols, index)`. The parameters (`nrows`), (`ncols`), and `index` indicate the numbers of rows, the numbers of columns (`ncols`) and the positional index, respectively. In details, the `index` starts at 1 in the upper left corner and increases to the right. To better understand, consider the code listing 4.4.

At line 1, we import the matplotlib.pyplot module. At line 3, we generate a new empty figure (i.e., `fig`). From line 6, we start creating and plotting a grid of diagrams (i.e., three columns and two rows) using `fig.add_subplot()` (i.e., lines 6, 10, 14, 18, 22, and 26). In the middle of each diagram, we plot a text highlighting `nrows`, `ncols`, and the `index`, respectively, using the command `text()`. Finally, the `tight_layout()` automatically adjusts subplot parameters so that the subplot(s) fits into the figure area. Avoiding using the `tight_layout()`, some overlaps among the elements of the diagram may occur.

```
1 import matplotlib.pyplot as plt
2
3
4 fig = plt.figure()
5 # index 1
6 ax1 = fig.add_subplot(2, 3, 1)
7 ax1.text(0.5, 0.5, str((2, 3, 1)), fontsize=18, ha='center')
8
9 # index 2
10 ax1 = fig.add_subplot(2, 3, 2)
11 ax1.text(0.5, 0.5, str((2, 3, 2)), fontsize=18, ha='center')
12
13 # index 3
14 ax1 = fig.add_subplot(2, 3, 3)
15 ax1.text(0.5, 0.5, str((2, 3, 3)), fontsize=18, ha='center')
16
17 # index 4
18 ax1 = fig.add_subplot(2, 3, 4)
19 ax1.text(0.5, 0.5, str((2, 3, 4)), fontsize=18, ha='center')
20
21 # index 5
22 ax1 = fig.add_subplot(2, 3, 5)
23 ax1.text(0.5, 0.5, str((2, 3, 5)), fontsize=18, ha='center')
24
25 # index6
26 ax1 = fig.add_subplot(2, 3, 6)
27 ax1.text(0.5, 0.5, str((2, 3, 6)), fontsize=18, ha='center')
28
29 plt.tight_layout()
```

Listing 4.4 Subplots with matplotlib.

As already pointed, the code listing 4.4, although easy to understand for a novice to Python, is nor elegant neither efficient. We can improve it, obtaining the same results using a for loop (code listing 4.5).

```
1 import matplotlib.pyplot as plt
2
3 fig = plt.figure()
4
5 for i in range(1, 7):
6     ax = fig.add_subplot(2,3,i)
7     plt.text(0.5, 0.5, str((2, 3, i)), fontsize=18, ha='center')
8
9 plt.tight_layout()
```

Listing 4.5 Subplots with matplotlib using a loop

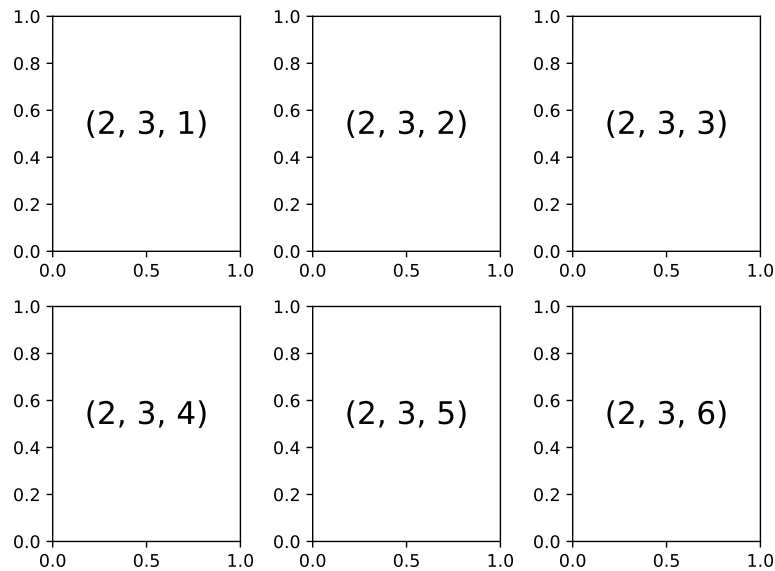


Fig. 4.4 The result of the code listing 4.4.

Markers

The option `markers` in scatter diagrams or other plots defines the shape of the symbol utilized to identify samples within the diagram. The code listing 4.6 and Fig. 4.5 show how to manage the marker parameter. Also, Table 4.1 reports an almost complete list of markers available in Python.

Table 4.1 Marker codes in matplotlib scatter and plot diagrams.

marker	Symbol	marker	Symbol	marker	Symbol
.	•	o	●	v	▼
^	▲	<	◀	>	▶
1	⌵	2	⌴	4	↵
4	⌶	8	●	s	■
p	⬠	h	●	H	●
+	+	x	×	D	◆
d	◆			-	-

```
1 fig = plt.figure()
2
3 ax1 = fig.add_subplot(2, 2, 1)
4 ax1.scatter(myDataset.Zr, myDataset.Th, marker='x', label="
5     cross")
6 ax1.set_xlabel("Zr [ppm]")
7 ax1.set_ylabel("Th [ppm]")
8 ax1.set_xlim([100, 1000])
9 ax1.set_ylim([0, 100])
10 ax1.legend()
11
12 ax2 = fig.add_subplot(2, 2, 2)
13 ax2.scatter(myDataset.Zr, myDataset.Th, marker='o', label="
14     circle")
15 ax2.set_xlabel("Zr [ppm]")
16 ax2.set_ylabel("Th [ppm]")
17 ax2.set_xlim([100, 1000])
18 ax2.set_ylim([0, 100])
19 ax2.legend()
20
21 ax3 = fig.add_subplot(2, 2, 3)
22 ax3.scatter(myDataset.Zr, myDataset.Th, marker='^', label="
23     triangle")
24 ax3.set_xlabel("Zr [ppm]")
25 ax3.set_ylabel("Th [ppm]")
26 ax3.set_xlim([100, 1000])
27 ax3.set_ylim([0, 100])
28 ax3.legend()
29
30 ax4 = fig.add_subplot(2, 2, 4)
31 ax4.scatter(myDataset.Zr, myDataset.Th, marker='d', label="
32     diamond")
33 ax4.set_xlabel("Zr [ppm]")
34 ax4.set_ylabel("Th [ppm]")
35 ax4.set_xlim([100, 1000])
36 ax4.set_ylim([0, 100])
37 ax4.legend()
38
39 fig.tight_layout()
```

Listing 4.6 Setting markers in scatter diagrams.

Marker dimensions

The dimension of markers can be defined by the option *s* in scatter diagrams. The code listing 4.7 and Fig. 4.6 show how to control the marker dimension.

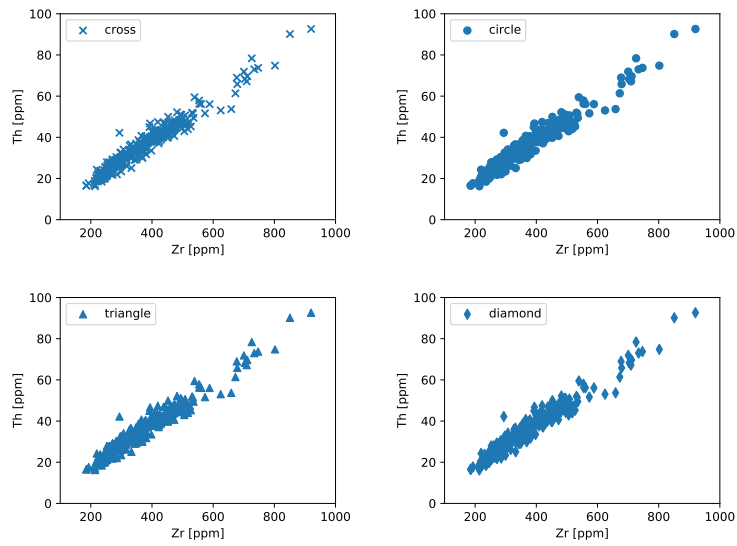


Fig. 4.5 The result of the code listing 4.6. The codes to change marker shapes are reported in 4.1.

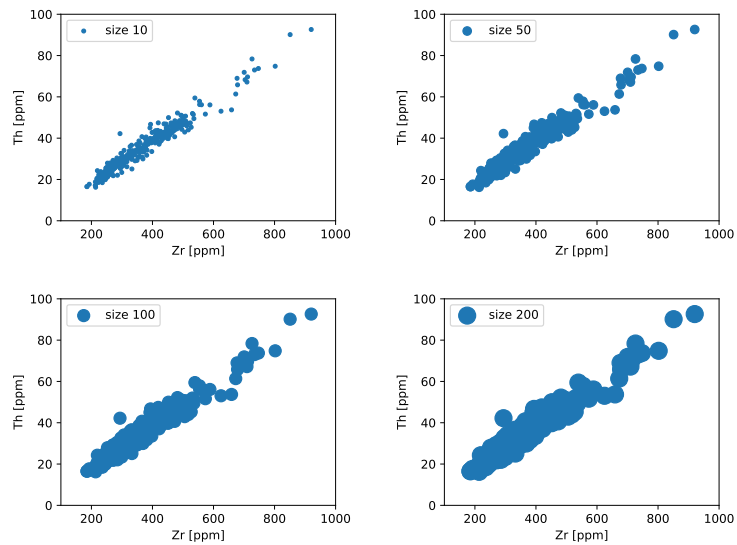


Fig. 4.6 The result of the code listing 4.7.

```
1 fig = plt.figure()
2
3 ax1 = fig.add_subplot(2, 2, 1)
4 plt.scatter(myDataset.Zr, myDataset.Th, marker='o', s = 10,
5             label="size 10")
6 ax1.set_xlabel("Zr [ppm]")
7 ax1.set_ylabel("Th [ppm]")
8 ax1.set_xlim([100, 1000])
9 ax1.set_ylim([0, 100])
10 ax1.legend()
11
12 ax2 = fig.add_subplot(2, 2, 2)
13 ax2.scatter(myDataset.Zr, myDataset.Th, marker='o', s = 50,
14            label="size 50")
15 ax2.set_xlabel("Zr [ppm]")
16 ax2.set_ylabel("Th [ppm]")
17 ax2.set_xlim([100, 1000])
18 ax2.set_ylim([0, 100])
19 ax2.legend()
20
21 ax3 = fig.add_subplot(2, 2, 3)
22 ax3.scatter(myDataset.Zr, myDataset.Th, marker='o', s = 100,
23            label="size 100")
24 ax3.set_xlabel("Zr [ppm]")
25 ax3.set_ylabel("Th [ppm]")
26 ax3.set_xlim([100, 1000])
27 ax3.set_ylim([0, 100])
28 ax3.legend()
29
30 ax4 = fig.add_subplot(2, 2, 4)
31 ax4.scatter(myDataset.Zr, myDataset.Th, marker='o', s = 200,
32            label="size 200")
33 ax4.set_xlabel("Zr [ppm]")
34 ax4.set_ylabel("Th [ppm]")
35 ax4.set_xlim([100, 1000])
36 ax4.set_ylim([0, 100])
37 ax4.legend()
38
39 fig.tight_layout()
```

Listing 4.7 Plotting a histogram distribution as probability density in Python.

Marker colors

It is possible to define the color of both the edge and the body of markers in scatter diagrams using the *edgecolor* and *c* options respectively (code listing 4.8 and Fig. 4.9). The *c* and *edgecolor* parameters can be a sequence of colors (i.e., one for each symbol of the diagram) or a single value. In the latter case, they specify the same color for all the symbols in the diagram. Color values can be specified in different way.

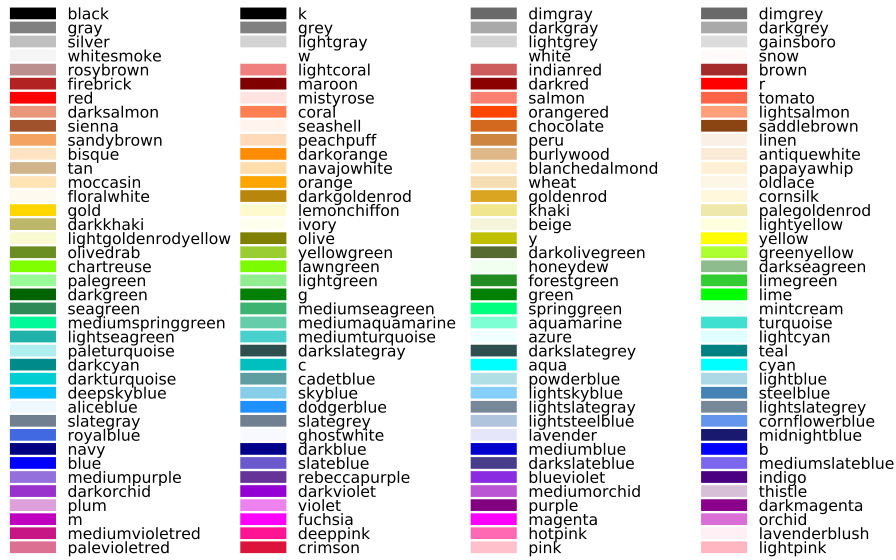


Fig. 4.7 Named colors, taken for the official documentation of matplotlib.

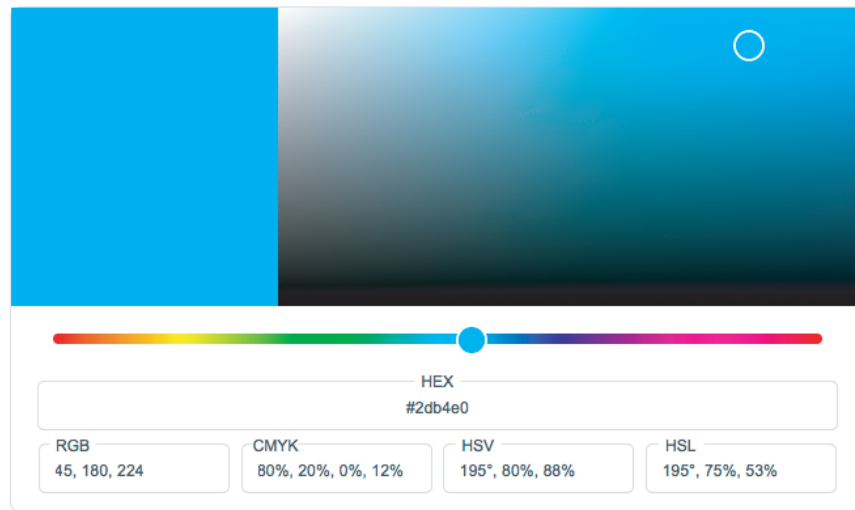


Fig. 4.8 An example of color picker. To get the hexadecimal RGB value, you simply need to copy the code in the HEX box.

Examples are hexadecimal RGB values (e.g., '#8B0000'), letters or names (Fig. 4.7, taken from the official documentation of matplotlib³), and gray scale levels (i.e., a value from 0 to 1, where 0 is black and 1 is white). To achieve the best flexibility with colors, I suggest using the hexadecimal RGB values, also known as HEX codes. An HEX code starts with the # symbol followed by six-digits, a combination three hexadecimal values ranging from 00 to FF (i.e., from 0 to 255 if reported to decimal notation). The first, second and third values represent the red, green and blue components of the color, respectively. At a first sight, the HEX notation could appear hard to use and not straightforward. However, you only need to use a "color picker" to select the color of your choice, and get the relative HEX code. Figure 4.8 displays the color picker provided by Google.

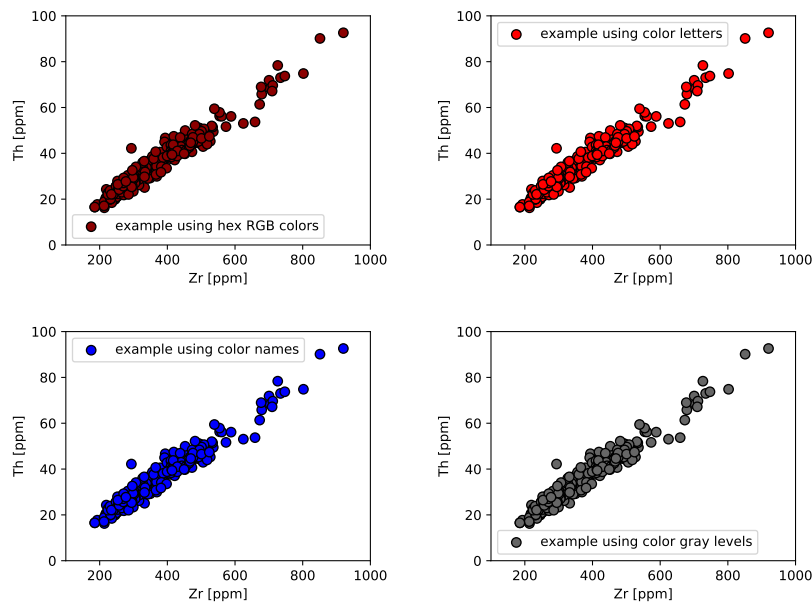


Fig. 4.9 The result of the code listing 4.8.

³ https://matplotlib.org/examples/color/named_colors.html

```
1 fig = plt.figure()
2
3 ax1= fig.add_subplot(2, 2, 1)
4 ax1.scatter(myDataset.Zr, myDataset.Th, marker='o', s = 60, c=
5     '#8B0000', edgecolor='#000000', label="example using hex
6     RGB colors")
7 ax1.set_xlabel("Zr [ppm]")
8 ax1.set_ylabel("Th [ppm]")
9 ax1.set_xlim([100, 1000])
10 ax1.set_ylim([0, 100])
11 ax1.legend()
12
13 ax2= fig.add_subplot(2, 2, 2)
14 ax2.scatter(myDataset.Zr, myDataset.Th, marker='o', s = 60, c
15     ='r', edgecolor='k', label="example using color letters"
16     )
17 ax2.set_xlabel("Zr [ppm]")
18 ax2.set_ylabel("Th [ppm]")
19 ax2.set_xlim([100, 1000])
20 ax2.set_ylim([0, 100])
21 ax2.legend()
22
23 ax3= fig.add_subplot(2, 2, 3)
24 ax3.scatter(myDataset.Zr, myDataset.Th, marker='o', s = 60, c
25     ='blue', edgecolor='black', label="example using color
26     names")
27 ax3.set_xlabel("Zr [ppm]")
28 ax3.set_ylabel("Th [ppm]")
29 ax3.set_xlim([100, 1000])
30 ax3.set_ylim([0, 100])
31 ax3.legend()
32
33 ax4= fig.add_subplot(2, 2, 4)
34 ax4.scatter(myDataset.Zr, myDataset.Th, marker='o', s = 60, c=
35     '0.4', edgecolor='0', label="example using color gray
36     levels")
37 ax4.set_xlabel("Zr [ppm]")
38 ax4.set_ylabel("Th [ppm]")
39 ax4.set_xlim([100, 1000])
40 ax4.set_ylim([0, 100])
41 ax4.legend()
42
43 fig.tight_layout()
```

Listing 4.8 Plotting a histogram distribution as probability density in Python.

Managing legends

The legend is a fundamental element of diagrams, often providing key notation to decipher the information presented in a plot. I already introduced how to add a legend in a plot using the `ax.legend()` command. It automatically creates a legend entry for each labeled element in the diagram.

We will see now how to customize your legend. In detail I will show you how to set the position and add a title to your legend.

The `loc` parameter sets the position of a legend within the diagram. Allowed entries for `loc` are: 'best', 'upper right', 'upper left', 'lower left', 'lower right', 'center left', 'center right', 'lower center', 'upper center', and 'center'. The `loc` parameter can also be expressed giving the coordinates of the lower-left corner of the legend. Some examples are reported in the code listing 4.9 and displayed in Fig 4.10. If not specified, the `loc` parameter assumes the 'best' option, meaning that it will attempt to achieve the minimum overlap with other drawn elements.

The title parameter adds a title to a legend with `title_fontsize` defining its font dimension.

Also, `frameon` (i.e. True or False), `ncol` (i.e. an integer), and `framealpha` (i.e. from 0 to 1) define the presence of a frame, its transparency, and the numbers of columns, respectively (code listing 4.10, and Fig. 4.11).

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset1 = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5                             sheet_name='Supp_traces')
6
7 x = myDataset1.Zr
8 y = myDataset1.Th
9
10 loc_parameters = ['upper right', 'upper left', 'lower left',
11                  'lower right', 'center', 'center left']
12
13 fig = plt.figure()
14 for i in range(len(loc_parameters)):
15     ax = fig.add_subplot(3,2,i+1)
16     ax.scatter(x, y, marker = 's', color = '#c7ddf4',
17               edgecolor = '#000000', label="loc = " + loc_parameters[i])
18     ax.set_xlabel("Zr [ppm]")
19     ax.set_ylabel("Th [ppm]")
20     ax.legend(loc=loc_parameters[i])
21 fig.tight_layout()

```

Listing 4.9 Customizing legend position using the `loc` parameter.

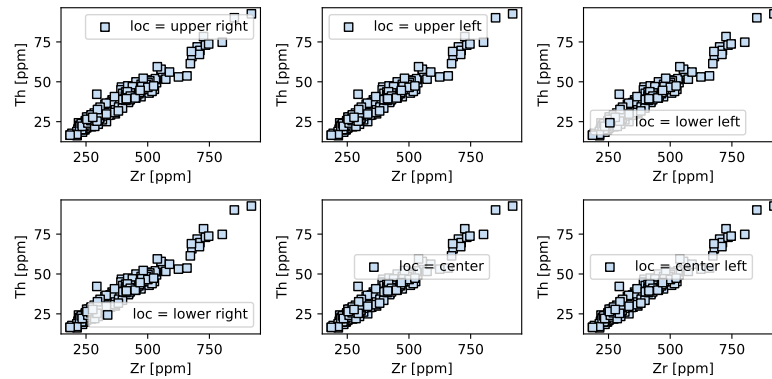


Fig. 4.10 The result of the code listing 4.9.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5     sheet_name='Supp_traces')
6
7 myDataset1= myDataset[myDataset.Epoch == 'one']
8 myDataset2 = myDataset[myDataset.Epoch == 'two']
9
10 fig = plt.figure()
11 ax1 = fig.add_subplot(2,1,1)
12 ax1.scatter(myDataset1.Zr, myDataset1.Th, marker = 's', color
13     = '#c7ddf4', edgecolor = '#000000', label="First Epoch")
14 ax1.scatter(myDataset2.Zr, myDataset2.Th, marker = 'o', color
15     = '#ff464a', edgecolor = '#000000', label="Second Epoch")
16 ax1.set_xlabel("Zr [ppm]")
17 ax1.set_ylabel("Th [ppm]")
18 ax1.legend(loc='upper left', framealpha=1, frameon=True, title=
19     "Age < 15 ky", title_fontsize=10)
20
21 ax2 = fig.add_subplot(2,1,2)
22 ax2.scatter(myDataset1.Zr, myDataset1.Th, marker = 's', color
23     = '#c7ddf4', edgecolor = '#000000', label="First Epoch")
24 ax2.scatter(myDataset2.Zr, myDataset2.Th, marker = 'o', color
25     = '#ff464a', edgecolor = '#000000', label="Second Epoch")
26 ax2.set_xlabel("Zr [ppm]")
27 ax2.set_ylabel("Th [ppm]")
28 ax2.legend(frameon=False, loc='lower right', ncol=2, title="
29     Age < 15 ky", title_fontsize=10)
30
31 fig.tight_layout()

```

Listing 4.10 Customizing legend parameters.

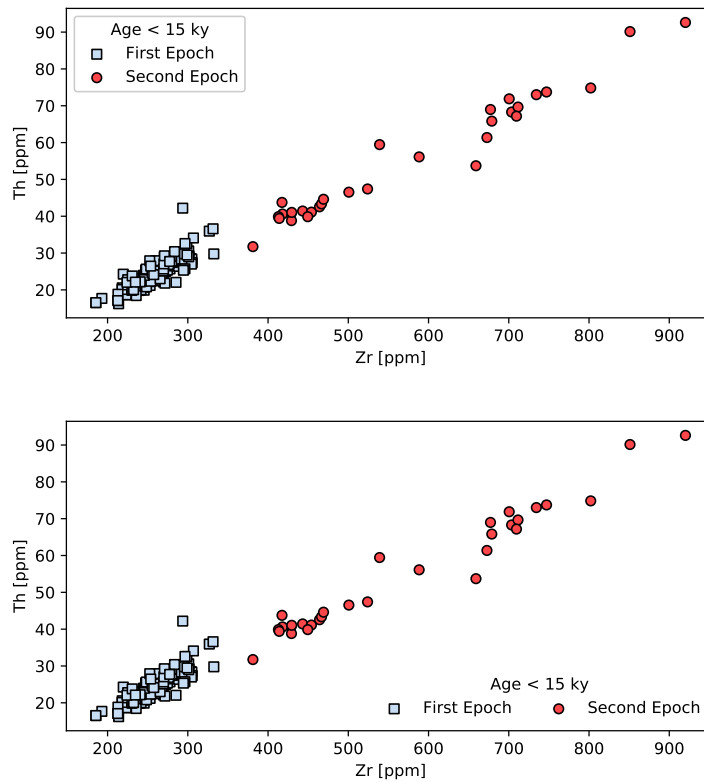


Fig. 4.11 The result of the code listing 4.10.

Rounding decimals, text formatting, symbols, and special characters

To reports data in diagrams (e.g., in the legend or as annotation), the rounding of a number or a string formatting is often required. In these cases, the `.format()` method is a flexible and useful tool. In detail, it allows positional injection of variables (i.e., numbers or strings) within strings and value formatting. To place a variable within a string, it is used a placeholder (i.e., `{}`). Also, it allows you to format date, times, or numbers and round decimals. To better understand, please consider the code listing 4.12. It reports some examples for the use of `.format()` in practical cases. In detail, at lines 5 and 7 it is used to insert two variables (i.e., name and surname) at specific positions of the text. Also, at lines 12 to 15, it shows how to insert a value (i.e., the Archimedeóconstant) and round it to a specific number of digits.

Also, the code listing 4.13 provide additional examples on the use of `.format()` for the use of plus and minus (lines 5-6), the reporting as percent (line 12), and the scientific notation (lines 18-19).

In addition, to insert characters that are illegal in a string (e.g. " or ' when they define the string), or do a specific action (e.g., go to a new line) you should use the escape character `\`. The table 4.2 and the code listing 4.11, provide you with some useful examples.

Table 4.2 Special characters and specific actions in strings using the escape character `\`.

Command	Result	Command	Result	Command	Result
<code>\n</code>	New Line	<code>\'</code>	Single Quote	<code>\"</code>	Double Quote
<code>\textbackslash</code>	<code>\</code>	<code>\ooo</code>	octal value	<code>\xhh</code>	hex value

```

1 # Go to new line using \n
2 print('-----
   ')
3 print("My name is\nMaurizio Petrelli")
4
5 # Inserting characters using octal values
6 print('-----
   ')
7 print("\100 \136 \137 \077 \176")
8
9 # Inserting characters using hex values
10 print('-----
   ')
11 print("\x23 \x24 \x25 \x26 \x2A")
12 print('-----
   ')
13
14 '''Output:
15 -----
16 My name is
17 Maurizio Petrelli
18 -----
19 @ ^ _ ? ~
20 -----
21 # $ % & *
22 -----
23 '''

```

Listing 4.11 Using the escape character `\`.

Our next challenge is: how to insert symbols and equations in diagrams? In my opinion, the simplest and most direct way to apply text formatting (e.g., apex and subscripts), insert symbols (i.e., μ or η), and introduce special characters (e.g., \pm) in matplotlib is the \TeX markup. Shortly, \TeX provides the foundations for \LaTeX , a high-quality typesetting system. In the practice, we can refer to \LaTeX as the de facto standard for the communication and publication of scientific documents⁴.

⁴ <https://www.latex-project.org>

Table 4.3 Introducing T_EX notation in matplotlib. Example: r'\$x^2\$' \rightarrow x^2 .

T _E X	Result	T _E X	Result	T _E X	Result
x ²	x^2	x_2	x_2	\pm	\pm
\alpha	α	\beta	β	\gamma	γ
\rho	ρ	\sigma	σ	\delta	δ
\pi	π	\eta	η	\mu	μ
\int	\int	\sum	Σ	\prod	\prod
\leftarrow	\leftarrow	\rightarrow	\rightarrow	\uparrow	\uparrow
\Leftarrow	\Leftarrow	\Rightarrow	\Rightarrow	\Uparrow	\Uparrow
\infty	∞	\nabla	∇	\partial	∂
\neq	\neq	\simeq	\simeq	\approx	\approx

```

1 # Introductory examples
2 name = 'Maurizio'
3 surname = 'Petrelli'
4 print('-----')
5 print('My name is {}'.format(name))
6 print('-----')
7 print('My name is {} and my surname is {}'.format(name,
8           surname))
9 print('-----')
10 # Decimal Number formatting
11 pi = 3.14159265358979323846
12 print('-----')
13 print("The 2 digit Archimedes' constant is equal to {:.2f}".
14       format(pi))
15 print("The 3 digit Archimedes' constant is equal to {:.3f}".
16       format(pi))
17 print("The 4 digit Archimedes' constant is equal to {:.4f}".
18       format(pi))
19 print("The 5 digit Archimedes' constant is equal to {:.5f}".
20       format(pi))
21 print('-----')
22 '''Results
23 -----
24 My name is Maurizio
25 -----
26 My name is Maurizio and my surname is Petrelli
27 -----
28 -----
29 The 2 digit Archimedes' constant is equal to 3.14
30 The 3 digit Archimedes' constant is equal to 3.142
31 The 4 digit Archimedes' constant is equal to 3.1416
32 The 5 digit Archimedes' constant is equal to 3.14159
33 -----
34 '''

```

Listing 4.12 Familiarizing with `.format()`.


```

1 # Explicit positive and negative reporting
2 a = +5.34352
3 b = -6.3421245
4 print('-----')
5 print("The plus symbol is not reported: {:.2f} | {:.2f}".format
      (+5.34352, -6.3421245))
6 print("The plus symbol is reported: {:.2f} | {:.2f}".format(a,
      b))
7 print('-----')
8
9 # Reporting as percent
10 c = 0.1558
11 print('-----')
12 print("Reporting as percent: {:.1%}".format(c))
13 print('-----')
14
15 # Scientific notation
16 d = 6580000000000
17 print('-----')
18 print("Scientific notation using e: {:.1e}".format(d))
19 print("Scientific notation using E: {:.1E}".format(d))
20 print('-----')
21
22 '''Results
23 -----
24 The plus symbol is not reported: 5.34 | -6.34
25 The plus symbol is reported: +5.34 | -6.34
26 -----
27 -----
28 Reporting as percent: 15.6%
29 -----
30 -----
31 Scientific notation using e: 6.6e+12
32 Scientific notation using E: 6.6E+12
33 -----
34 '''

```

Listing 4.13 More about number reporting using `.format()`.

Teaching \TeX and \LaTeX is far behind the scope of the present book but the reader is invited to refer to specialised books (Kopka & Daly, 2003; Lamport, 1994; Mittelbach et al., 2004). However, knowing a few specific rules and notations will greatly help us in upgrade the quality of our diagrams significantly. Note that any text element in `matplotlib` can use advanced formatting, mathematical elements, and symbols. To start using \TeX in `matplotlib`, we have to precede the quotes defining a string with an `r` (i.e. `r'this is my string'`), and surround the math text using the dollar symbol (`$`). The code listing 4.14 shows an example on how to use the \TeX notation to improve the quality of our diagrams (Fig. 4.12).

Also, table 4.3, provides some common \TeX instructions such as how to apply apex and subscripts, insert Greek letters like μ , η , and π , using special characters (e.g., \pm , ∞), or mathematical expressions (e.g., \int_a^b).

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5
6 def MyLine(x,m,q):
7     y = m * x + q
8     return y
9
10
11 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
12     sheet_name='Supp_majors', engine='openpyxl')
13 myDataset1= myDataset[myDataset.Epoch == 'one']
14 myDataset2 = myDataset[myDataset.Epoch == 'two']
15
16
17 x = np.linspace(52.5, 62, 100)
18 y = MyLine(x,m=0.3, q= -10.3)
19
20
21 fig, ax = plt.subplots()
22
23 ax.scatter(myDataset1.SiO2, myDataset1.K2O, marker = 's',
24     color = '#c7ddf4', edgecolor = '#000000', label=r'$1^{st}$
25     Epoch')
26 ax.scatter(myDataset2.SiO2, myDataset2.K2O, marker = 's',
27     color = '#ff464a', edgecolor = '#000000', label=r'$2^{nd}$
28     Epoch')
29 ax.plot(x,y, color = '#342a77')
30
31 ax.annotate(r'What is the 1$\sigma$ for this point?', xy
32     =(47.6, 6.6), xytext=(47, 8.8), arrowprops=dict(arrowstyle
33     ="->", connectionstyle="arc3" ) )
34 ax.text(52.4, 5.6, r'$ Na_{20} = 0.3 \cdot SiO_2 - 10.3$', dict(
35     size=10,rotation=33))
36
37 ax.text(53.5, 5.1, r'$ \mu_{SiO_2} = \frac {a_{1}+a_{2}+\cdots
38     +a_{n}}{n}$ = ' + '{:.1f} [wt.%]'.format(57.721) , dict(
39     size=11.5))
40
41 ax.xlabel(r'SiO$_2$ [wt%]')
42 ax.ylabel(r'K$_2$O [wt%]')
43
44 ax.legend()

```

Listing 4.14 Using TeX notation in matplotlib.

As drawback, adding r before the string precludes the use of `.format()` and `\` as escape character. To overcome this problem, I suggest to split the string into sub-

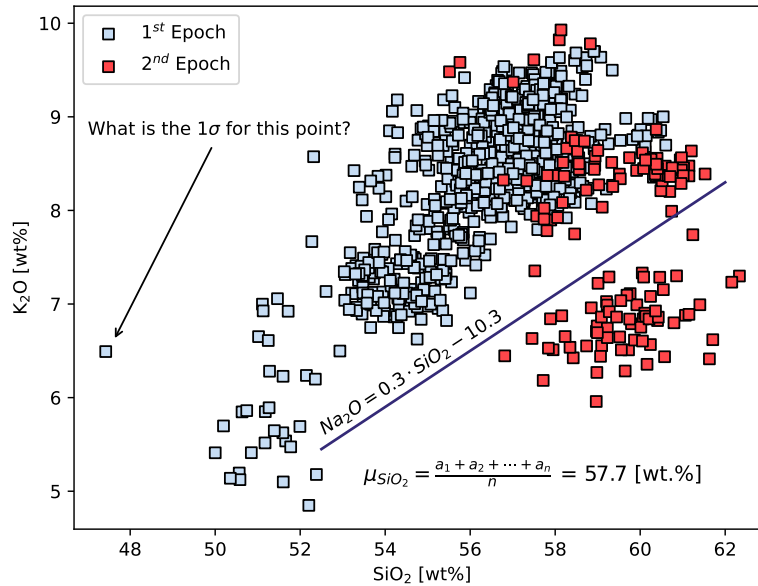


Fig. 4.12 The result of the code listing 4.14.

strings and then concatenate them using the + symbol, as reported in the code listing 4.12 (line 29).

Binary diagrams: `plot()` vs `scatter()`

In the previous sections, we introduced two different methods to visualize geological data in binary diagrams: the `plot()` and `scatter()` functions implemented in the matplotlib sub-package named pyplot. The two methods share many functionalities and can be often used indifferently. As an example, look at the code listing 4.15, showing how to plot a binary diagram with square markers.

Nevertheless, `plot()` and `scatter()` have some distinctions. As an example, the `plot()` function only can connect with a line the points defined by a sequence of (x,y) coordinates (Fig 4.14. Table 4.4, shows the main parameters that can be used to personalize the aesthetics of a `plt.plot()` diagram. However, the `plot()` function is less flexible than `scatter()` for marker sizing, and coloring. For each `plot()` declaration, all symbols must be of the same size, and color. On the contrary, using `scatter()` you can set different colors, and sizes for each marker. As an example, Fig 4.14 shows symbols with size proportional to the F parameter and color defined by the color sequence. Sometime, you will need to combine them. As an example, if you would like to plot and connect a sequence of samples with different colors and dimensions, you could use `scatter` for symbols and `plot` for the connecting line. The

zorder parameter is an integer number defining the stratigraphy of different layers in the diagram. In the specific case of code listing 4.16 (lines 34 and 35), it places symbols above the line.

Table 4.4 Parameters allowing the personalisation of a plot() diagram.

Parameter	Values	Description
alpha	[0,1]	Set the transparency
color, c	a color value (e.g., Fig. 4.8 and 4.7)	Set the color of the line
fillstyle	{'full', 'left', 'right', 'bottom', 'top', 'none'}	Set the marker fill style
linestyle, ls	{'-' , '-' , '-.' , ':' , ", (offset, on-off-seq), ...}	Set the style of the line
linewidth, lw	a float number	Set the line width in points
marker	a marker style (e.g., Tab. 4.1)	Set the marker style
markeredgecolor, mec	a color value	Set the marker edge color
markeredgewidth, mew	a float number	Set the marker edge width
markerfacecolor, mfc	a color value	Set the marker face color
markersize, ms	float number	Set the marker size in points

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset1 = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5                             sheet_name='Supp_traces')
6
7 x = myDataset1.Zr
8 y = myDataset1.Th
9
10 fig = plt.figure()
11 ax1 = fig.add_subplot(1,2,1)
12 ax1.scatter(x, y, marker = 's', color = '#ff464a', edgecolor = '#000000')
13 ax1.set_title("using scatter()")
14 ax1.set_xlabel("Zr [ppm]")
15 ax1.set_ylabel("Th [ppm]")
16 ax2 = fig.add_subplot(1,2,2)
17 ax2.plot(x, y, marker = 's', linestyle = '', color = '#ff464a',
18           markeredgecolor = '#000000')
19 ax2.set_title("using plot()")
20 ax2.set_xlabel("Zr [ppm]")
21 ax2.set_ylabel("Th [ppm]")
22 fig.tight_layout()

```

Listing 4.15 Often plot() and scatter() can be used to solve the same tasks.

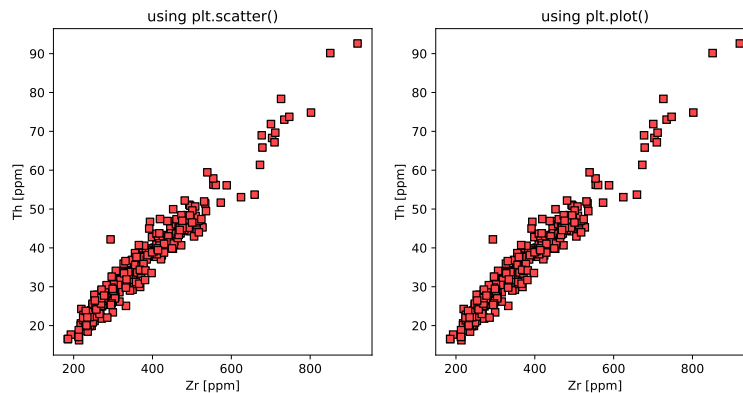


Fig. 4.13 The result of the code listing 4.15.

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 def EC(F, D, C0):
5     CL = C0 / (D * (1 - F) + F)
6     return CL
7
8 myF = np.linspace(0.1, 1, 10)
9
10 myC1 = EC(F=myF, D=0.1, C0=100)
11
12 colors = ['#ff9494', '#cbeaa2', '#d1a396', '#828fc3', '#95b2e5', '#
13         e9b8f4', '#f4b8e5', '#b8f4f2', '#c5f4b8', '#f9ca78']
14
15 fig = plt.figure()
16 ax1 = fig.add_subplot(2,2,1)
17 ax1.plot(myF, myC1, marker = 'o', linestyle = '-', markersize
18         = 5)
19
20 ax2 = fig.add_subplot(2,2,2)
21 ax2.scatter(myF, myC1, marker = 'o', s = myF*150)
22 ax2.set_xlabel('F')
23 ax2.set_ylabel('C [ppm]')
24
25 ax3 = fig.add_subplot(2,2,3)
26 ax3.scatter(myF, myC1, marker = 'o', c = colors, s = myF*150)
27 ax3.set_xlabel('F')
28 ax3.set_ylabel('C [ppm]')
29
30 ax4 = fig.add_subplot(2,2,4)
31 ax4.plot(myF, myC1, marker = '', linestyle='-', zorder = 0)
32 ax4.scatter(myF, myC1, marker = 'o', c = colors, s = myF*150,
33            zorder = 1)
34 ax4.set_xlabel('F')
35 ax4.set_ylabel('C [ppm]')
36 fig.tight_layout()

```

Listing 4.16 Main differences between plot() and scatter().

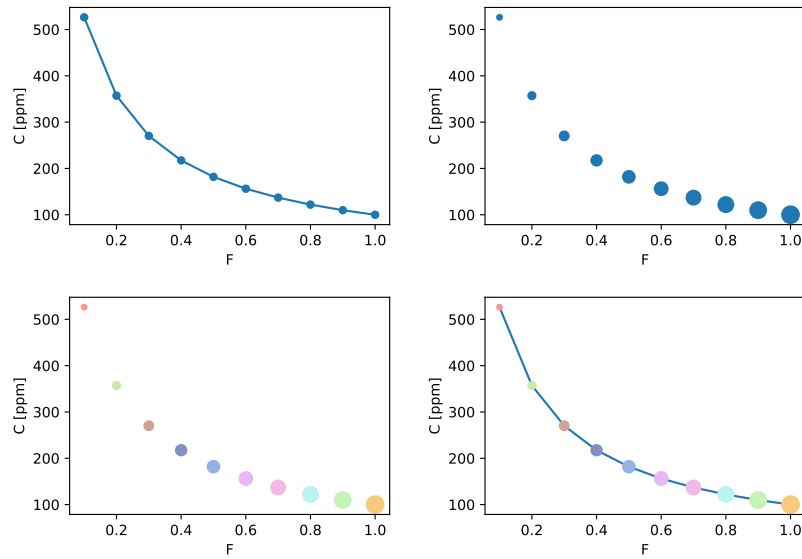


Fig. 4.14 The result of the code listing 4.16

An Example of publication-ready diagram

As a final task, we are going to prepare a publication-ready diagram (code listing 4.17). In code listing 4.17, lines 1 and 2 import the pandas library and the matplotlib.pyplot module, respectively. At line 4, the Excel file is imported in a DataFrame named myDataset. Lines 6, 7, and 8 define three sequences named epochs, colors, and markers, respectively. At line 10, we generate a new empty figure. At line 11 a loop iterate over the sequences of epochs, colors, and markers using the `zip()` function. This enables us to iterate over two or more lists at the same time. Then, i.e., line 12, a new DataFrame named myData is defined by filtering myDataset using the labels the Epoch column. At line 13 we add a Zr vs. Th scatter diagram of the resulting myData to the figure generated at line 10. Finally, we add axis labels (lines 15 and 16), and a legend with a title (line 17). Note that `\n` simply defines a new line in legend title.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5     sheet_name='Supp_traces')
6
7 epochs = ['one', 'two', 'three', 'three-b']
8 colors = ['#c8b4ba', '#f3ddb3', '#c1cd97', '#e18d96']
9 markers = ['o', 's', 'd', 'v']
10
11 fig, ax = plt.subplots()
12 for (epoch,color,marker) in zip(epochs, colors, markers):
13     myData = myDataset[(myDataset.Epoch == epoch)]
14     ax.scatter(myData.Zr, myData.Th, marker=marker, s = 50, c=
15         color , edgecolor='0', label="Epoch " + epoch)
16
17 ax.set_xlabel("Zr [ppm]")
18 ax.set_ylabel("Th [ppm]")
19 ax.legend(title="Phlegraean Fields \n Age < 15 ky")

```

Listing 4.17 Plotting a histogram distribution as probability density in Python.

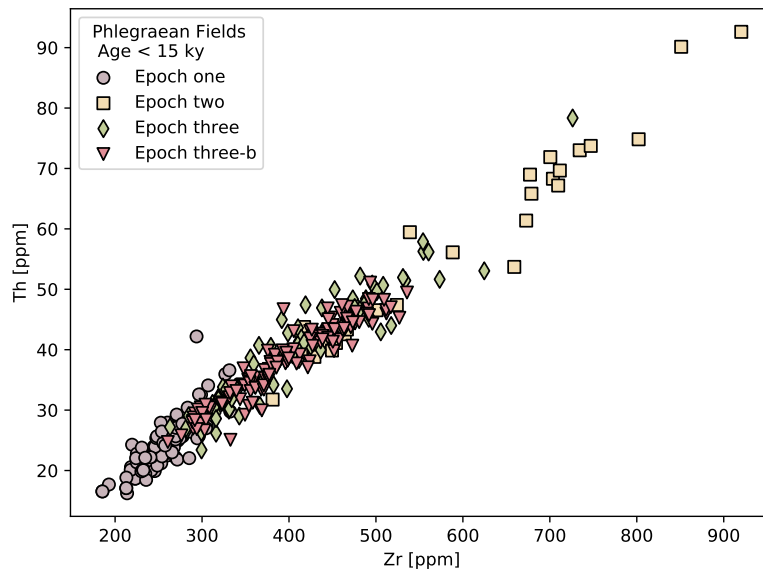


Fig. 4.15 The result of the code listing 4.17.

4.4 Visualization of Multivariate Data: a First Attempt

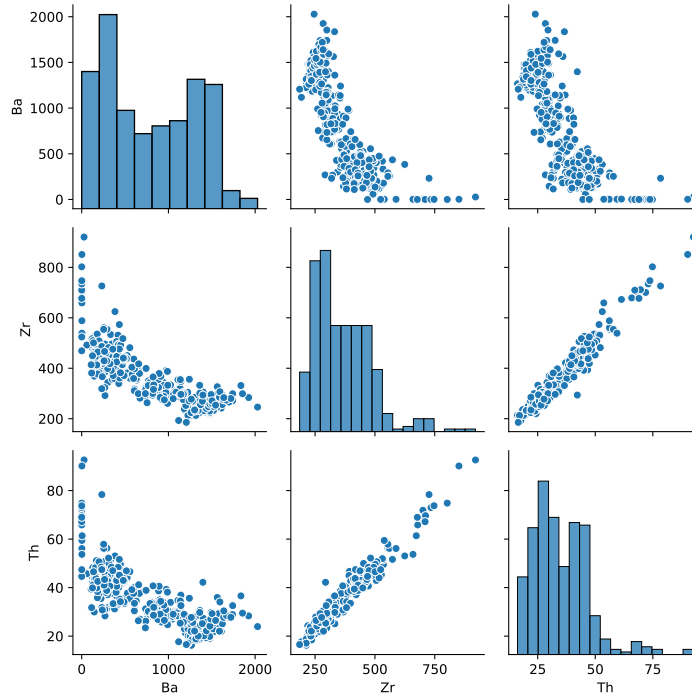


Fig. 4.16 The result of the code listing 4.18.

The `seaborn.pairplot()` function plots pairwise relationships in a data set. By default, this function creates a grid of diagrams where each variable in the data set is shared in the y-axis across a single row and in the x-axis across a single column. In diagonal diagrams, the `pairplot()` function draws a plot showing the univariate distribution for the variable in that column⁵. The code listing 4.18 shows how to generate a `pairplot()` diagram using Ba, Zr, and Th. In detail, at line 1 and 2, we import the pandas and seaborn libraries, respectively. We know already the meaning of line 4, i.e., importing an Excel file in a DataFrame named `myDataset`. At line 6, we generate a new DataFrame (i.e., `myDataset1`) by filtering `myDataset` for Ba, Zr, and Th columns. More details about the filtering and slicing of a DataFrame are reported in Appendix D. Finally, at line 7 we generate a pairplot diagram. Figure 4.16 displays the pairplot generated using the code listing 4.18

⁵ <https://seaborn.pydata.org/generated/seaborn.pairplot.html>


```
1 import pandas as pd
2 import seaborn as sns
3
4 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5                           sheet_name='Supp_traces')
6
7 myDataset1 = myDataset[['Ba', 'Zr', 'Th']]
8 sns.pairplot(myDataset1)
```

Listing 4.18 A first attempt in visualizing multivariate data using *sns.pairplot()*.

Chapter 5

Descriptive Statistics 1: Univariate Analysis

5.1 Basics of Descriptive Statistics

Descriptive statistics deals with measures, tools, and strategies that can be used to summarize a data set. These measures are quantities extracted from the data providing information about 1) the location of a data set, sometime defined as central tendency; 2) amount of data variation (i.e., the dispersion), and 3) the degree of symmetry (i.e., the skewness). Measures of the location of a data set are the arithmetic, geometric, and harmonic means. The median and the modal value of mono-modal distributions are also measures of the location of a data set. The total spread of a data set is a rough estimation of dispersion. More accurate estimations of the dispersion of a data set are the variance, the standard deviation, and the inter-quartile range. The skewness of a data set can be measured by parameters such as the Pearson's first coefficient of skewness or the Fischer-Pearson's coefficient of skewness.

5.2 Location

In descriptive statistics, it is useful to represent an entire data set with a single value describing its location or position. That single value is defined as the central tendency. The mean, the median, and the mode fall in this category.

Means

The arithmetic mean μ_A is the average of all numbers and is defined as:

$$\mu_A = \bar{z} = \frac{1}{n} \sum_{i=1}^n z_i = \frac{z_1 + z_2 + \cdots + z_n}{n} \quad (5.1)$$

The geometric mean μ_G is a type of mean, which indicates the location of a data set using the product of their values:

$$\mu_G = (z_1 z_2 \cdots z_n)^{\frac{1}{n}} \quad (5.2)$$

Finally, the harmonic mean μ_H can be expressed as:

$$\mu_H = \frac{n}{\frac{1}{z_1} + \frac{1}{z_2} + \cdots + \frac{1}{z_n}} \quad (5.3)$$

In the following, when not explicitly specified, I will refer to the symbol μ to claim the arithmetic mean. One of the ways (remember, there are also many ways to get the solution of a problem in Python) for getting the different means for a specific feature (in our case the concentration of a chemical element like Zirconium, Zr) in the imported data set is reported (code listing 5.1 and Fig. 5.1).

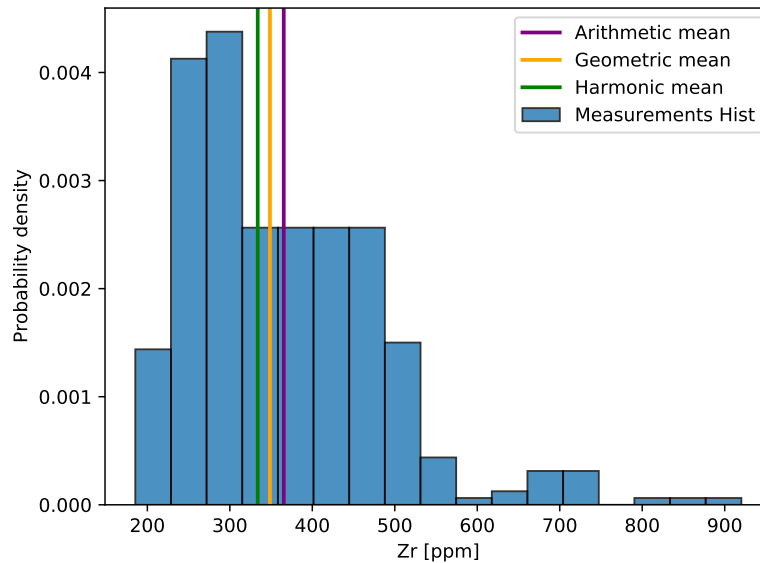


Fig. 5.1 The result of the code reported in the listing 5.1.

```

1 import pandas as pd
2 from scipy.stats.mstats import gmean, hmean
3 import matplotlib.pyplot as plt
4
5 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
6     sheet_name='Supp_traces')
7
8 a_mean = myDataset.Zr.mean()
9 g_mean = gmean(myDataset['Zr'])
10 h_mean = hmean(myDataset['Zr'])
11
12 print ('-----')
13 print ('arithmetic mean')
14 print ("{:0:1f} [ppm]".format(a_mean))
15 print ('-----')
16
17 print ('geometric mean')
18 print ("{:0:1f} [ppm]".format(g_mean))
19 print ('-----')
20
21 print ('harmonic mean')
22 print ("{:0:1f} [ppm]".format(h_mean))
23 print ('-----')
24
25 fig, ax = plt.subplots()
26 ax.hist(myDataset.Zr, bins='auto', density=True, edgecolor='k',
27     label='Measurements Hist', alpha=0.8)
28 ax.axvline(a_mean, color='purple', label='Arithmetic mean',
29     linewidth=2)
30 ax.axvline(g_mean, color='orange', label='Geometric mean',
31     linewidth=2)
32 ax.axvline(h_mean, color='green', label='Harmonic mean',
33     linewidth=2)
34 ax.set_xlabel('Zr [ppm]')
35 ax.set_ylabel('Probability density')
36 ax.legend()
37
38 '''
39 Output:
40 -----
41 arithmetic mean
42 365.4 [ppm]
43 -----
44 geometric mean
45 348.6 [ppm]
46 -----
47 harmonic mean
48 333.8 [ppm]
49 -----
50 '''

```

Listing 5.1 Measuring and plotting the average values of a data set.

Median

The median, Me , is the number at the middle of a data set after a sorting from the lower to the higher value (code listing 5.2 and Fig. 5.2). As a consequence, to obtain the median value of a data set, data values must be ordered from the smallest to the largest. If the number of data values is odd, then the sample median is the middle value in the ordered list; if it is even, then the sample median is the average of the two middle values (Ross, 2010).

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5     sheet_name='Supp_traces')
6
7 median = myDataset.Zr.median()
8
9 print ('-----')
10 print ('median')
11 print ("{:0:.1f} [ppm]".format(median))
12 print ('-----')
13
14 fig, ax = plt.subplots()
15 ax.hist(myDataset.Zr, bins=20, density = True, edgecolor='k',
16     label="Measurements Hist", alpha=0.8)
17 ax.axvline(median, color="orange", label="Median", linewidth=2)
18 ax.set_xlabel('Zr [ppm]')
19 ax.set_ylabel('Probability density')
20 ax.legend()
21
22 '''
23 Output:
24 -----
25 median
26 339.4 [ppm]
27 -----
28 '''

```

Listing 5.2 Measuring and plotting the median of a data set.

Mode

The mode, Mo , of a data set is the value that appears most frequently in the data set (Ross, 2010). In python, it is possible to retrieve the modal value using the instructions reported in the code listing 5.3.

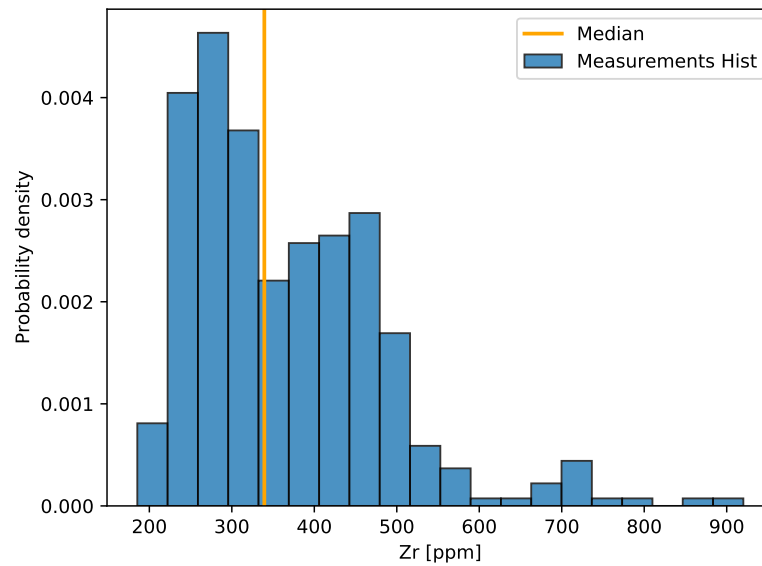


Fig. 5.2 The result of the code reported in the listing 5.2.

```

1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
6     sheet_name='Supp_traces')
7
8 hist, bin_edges = np.histogram(myDataset['Zr'], bins=20,
9     density=True)
10 modal_value = (bin_edges[hist.argmax()] + bin_edges[hist.
11     argmax()+1])/2
12
13 print ('modal value: {:.0f} [ppm]'.format(modal_value))
14
15 fig, ax = plt.subplots()
16 ax.hist(myDataset.Zr, bins=20, density = True, edgecolor='k',
17     label="Measurements Hist", alpha=0.8)
18 ax.axvline(modal_value, color="orange", label="Modal value",
19     linewidth=2)
20 ax.set_xlabel('Zr [ppm]')
21 ax.set_ylabel('Probability density')
22 ax.legend()
23
24 '''
25 Output: modal value: 277 [ppm]
26 '''

```

Listing 5.3 Measuring and plotting the modal value of a data set.

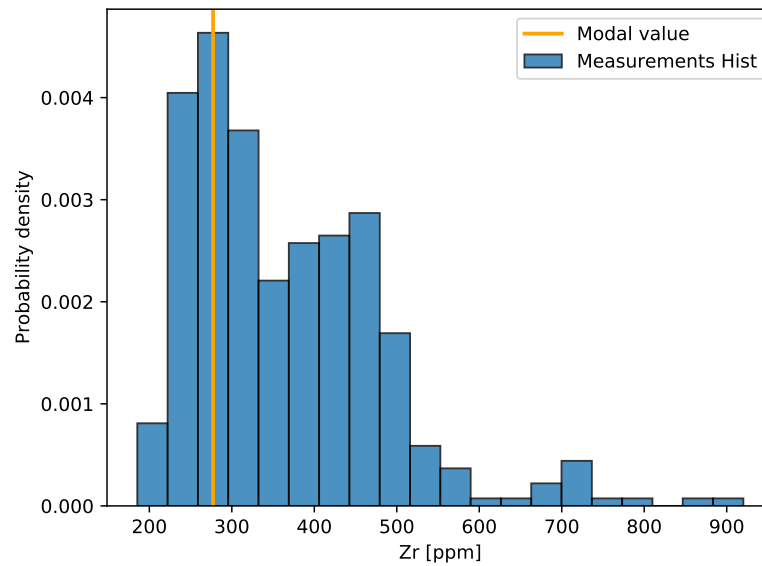


Fig. 5.3 The result of the code reported in the listing 5.3.

5.3 Dispersion or Scale

We have just introduced several estimators of the central tendency of a data set. However we have not yet considered any measure of its variability. The range, the variance and the standard deviation are all estimators of the dispersion (i.e., variability) of a data set.

In pandas, we can estimate the range as follow:

Range

A first gross estimator of the variability of a data set is provided by the range. The range, R , is the difference between the highest and lowest values in the the data set:

$$R = (z_{max} - z_{min}) \quad (5.4)$$


```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5     sheet_name='Supp_traces')
6
7 R = myDataset['Zr'].max()- myDataset['Zr'].min()
8
9 print ('-----')
10 print ('Range')
11 print ("{:0:.0f}".format(R))
12 print ('-----')
13
14 fig, ax = plt.subplots()
15 ax.hist(myDataset.Zr, bins= 20, density = True, edgecolor='k',
16     label='Measurements Hist')
17 ax.axvline(myDataset['Zr'].max(), color='purple', label='Max
18     value', linewidth=2)
19 ax.axvline(myDataset['Zr'].min(), color='green', label='Min
20     value', linewidth=2)
21 ax.axvspan(myDataset['Zr'].min(), myDataset['Zr'].max(), alpha
22     =0.1, color='orange', label='Range = ' + "{:0:.0f}".format(
23     R) + ' ppm')
24 ax.set_xlabel('Zr [ppm]')
25 ax.set_ylabel('Probability density')
26 ax.legend()

```

Listing 5.4 Measuring and plotting the range of a data set.

Variance and standard deviation

The variances, for the population (σ_p^2) and the sample (σ_s^2) distributions, are defined as:

$$\sigma_p^2 = \frac{\sum_{i=1}^n (z_i - \mu)^2}{n} \quad (5.5)$$

$$\sigma_s^2 = \frac{\sum_{i=1}^n (z_i - \mu)^2}{n - 1} \quad (5.6)$$

The standard deviation, σ , is the square root of the variance:

$$\sigma_p = \sqrt{\sigma_p^2} = \sqrt{\frac{\sum_{i=1}^n (z_i - \mu)^2}{n}} \quad (5.7)$$

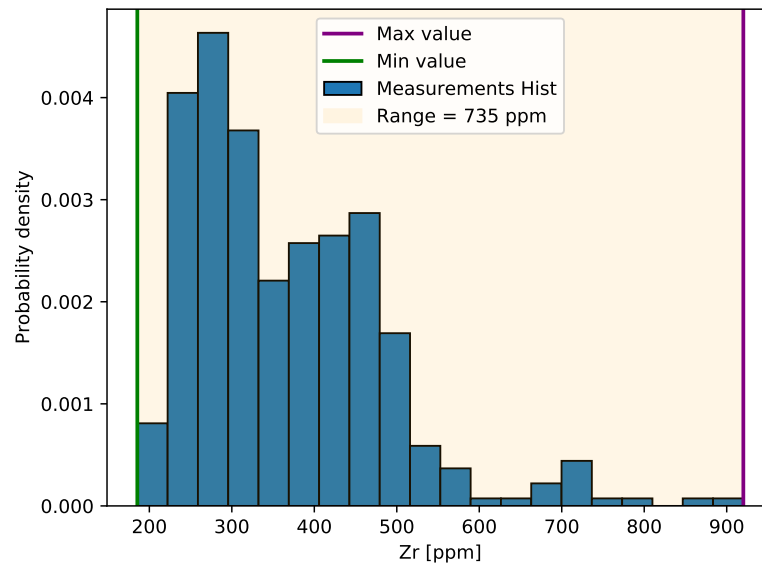


Fig. 5.4 The result of the code reported in the listing 5.4.

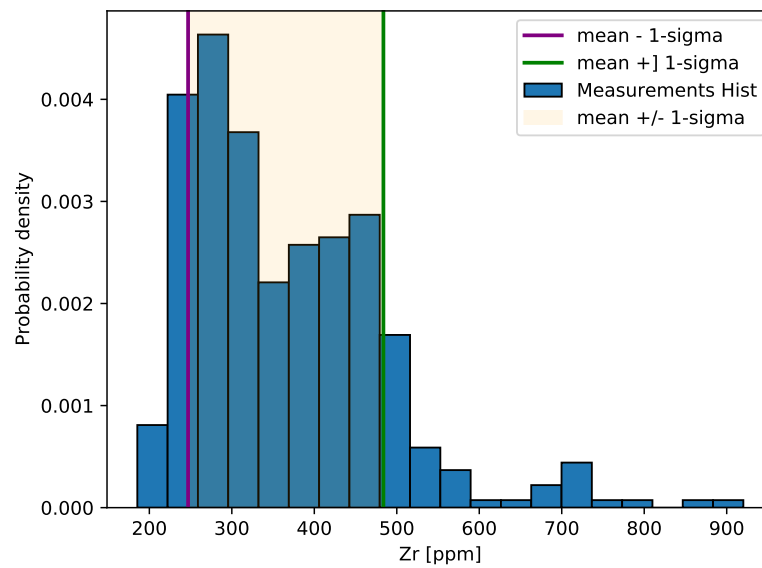


Fig. 5.5 The result of the code reported in the listing 5.5.

$$\sigma_s = \sqrt{\sigma_s^2} = \sqrt{\frac{\sum_{i=1}^n (z_i - \mu)^2}{n - 1}} \quad (5.8)$$

The variance (σ_s^2) and the standard deviation (σ_s) of a sample distribution can be estimated in pandas using the instructions reported in the code listing 5.5.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5     sheet_name='Supp_traces')
6
7 variance = myDataset['Zr'].var()
8 stddev = myDataset['Zr'].std()
9
10 print ('-----')
11 print ('Variance')
12 print ("{:0:.0f} [square ppm]".format(variance))
13 print ('-----')
14 print ('Standard Deviation')
15 print ("{:0:.0f} [ppm]".format(stddev))
16 print ('-----')
17
18 fig, ax = plt.subplots()
19 ax.hist(myDataset.Zr, bins=20, density=True, edgecolor='k',
20     label='Measurements Hist')
21 ax.axvline(myDataset['Zr'].mean() - stddev, color='purple',
22     label='mean - 1-sigma', linewidth=2)
23 ax.axvline(myDataset['Zr'].mean() + stddev, color='green',
24     label='mean + 1-sigma', linewidth=2)
25 ax.axvspan(myDataset['Zr'].mean() - stddev, myDataset['Zr'].
26     mean() + stddev, alpha=0.1, color='orange', label='mean
27     +/- 1-sigma')
28 ax.set_xlabel('Zr [ppm]')
29 ax.set_ylabel('Probability density')
30 ax.legend()
31
32 '''
33 Output:
34 -----
35 Variance
36 14021 [square ppm]
37 -----
38 Standard Deviation
39 118 [ppm]
40 -----
41 '''

```

Listing 5.5 Measuring and plotting the variance and the standard deviation of a data set.

In pandas, to estimate the variance and the standard deviation for an entire population, you need to change the Delta Degrees of Freedom (*ddof*). By default, the pandas commands `.var()` and `.std()` use `ddof = 1`, normalizing the measurements by $(n-1)$. Setting `.var(ddof = 0)` and `.std(ddof = 0)`, pandas calculates the σ_p^2 and σ_p , respectively. Variances and standard deviations can be also estimates for NumPy arrays using the same `.var()` and `.std()` commands. Differently from pandas, NumPy `.var()` and `.std()` set `ddof=0` as default parameter, therefore estimating the population variance (σ_p^2) and standard deviation (σ_p), respectively.

Inter quantile range

In descriptive statistics, the inter-quartile range (IQR) is equal to the difference between 75th and 25th percentiles, or between upper and lower quartiles (code listing 5.6).

```

1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
6     sheet_name='Supp_traces')
7
8 Q1 = np.percentile(myDataset.Zr, 25, interpolation = 'midpoint')
9 Q3 = np.percentile(myDataset.Zr, 75, interpolation = 'midpoint')
10 IQR = Q3 - Q1 # Interquartile range (IQR)
11
12 print ('-----')
13 print ('Interquartile range (IQR): {0:.0f} [ppm]'.format(IQR))
14 print ('-----')
15
16 fig, ax = plt.subplots()
17 ax.hist(myDataset.Zr, bins= 'auto', density = True, edgecolor='k',
18     label='Measurements Hist')
19 ax.axvline(Q1, color='purple', label='Q1', linewidth=2)
20 ax.axvline(Q3, color='green', label='Q3', linewidth=2)
21 ax.axvspan(Q1, Q3, alpha=0.1, color='orange', label='
22     Interquartile range (IQR)')
23 ax.set_xlabel('Zr [ppm]')
24 ax.set_ylabel('Probability density')
25 ax.legend()
26
27 '''
28 Output:
29 -----
30 Interquartile range (IQR): 164 [ppm]
31 -----
32 '''

```

Listing 5.6 Measuring and plotting the inter-quartile range (IQR) of a data set.

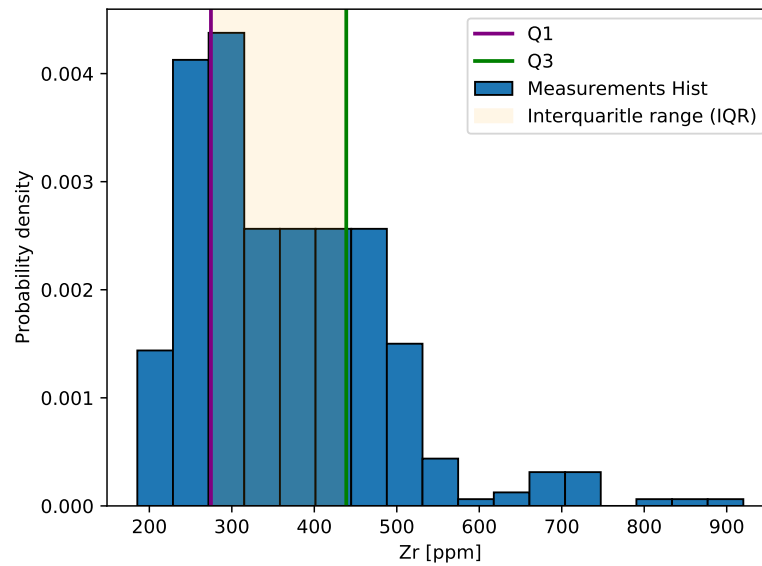


Fig. 5.6 The result of the code reported in the listing 5.6.

5.4 Skewness

After the introduction of some parameters providing information about the central tendency and the variability of a data set, we can start analyzing an index giving some constraints about the shape of a distribution: the skewness.

The skewness is a statistical parameter providing information about the symmetry in a distribution of values. In the case of a symmetric distribution, the arithmetic mean, the median and the mode express the same value. As a consequence, $Mo = Me = \mu_A$. Please note that the coincidence of these three values, although being a necessary condition for symmetric distributions, does not guarantee the symmetry of a distribution. On the contrary, the non-coincidence of these three parameters points to a skewed distribution. In particular, in the cases where $Mo < Me < \mu_A$ and $\mu_A < Me < Mo$ the distribution is characterized by a tail on the right and left side, respectively.

In the specific case of Zr concentration distribution, where $Mo < Me < \mu_A$, a tail on the right side is present, as expected.

```

1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
6     sheet_name='Supp_traces')
7
8 a_mean = myDataset.Zr.mean()
9
10 median = myDataset.Zr.median()
11
12 hist, bin_edges = np.histogram(myDataset['Zr'], bins= 20,
13     density=True)
14 modal_value = (bin_edges[hist.argmax()] + bin_edges[hist.
15     argmax()+1])/2
16
17 fig, ax = plt.subplots()
18 ax.hist(myDataset.Zr, bins= 20, density = True, edgecolor='k',
19     label="Measurements Hist")
20 ax.axvline(modal_value, color='orange', label='Modal Value',
21     linewidth=2)
22 ax.axvline(median, color='purple', label='Median Value',
23     linewidth=2)
24 ax.axvline(a_mean, color='green', label='Arithmetic mean',
25     linewidth=2)
26 ax.set_xlabel('Zr [ppm]')
27 ax.set_ylabel('Probability density')
28 ax.legend()

```

Listing 5.7 Providing a qualitative test of the skewness of a data set.

A parameter providing information about the skewness of a sample distribution is the Pearson's first coefficient of skewness (Eq. 5.9):

$$\alpha_1 = \frac{(\mu - Mo)}{\sigma_s} \quad (5.9)$$

A second parameter is the Pearson's second moment of skewness (Eq. 5.10):

$$\alpha_2 = \frac{3(\mu - Me)}{\sigma_s} \quad (5.10)$$

An additional parameter providing information about the sample skewness is the Fisher-Pearson's coefficient of skewness (Eq. 5.11):

$$g_1 = \frac{m_3}{m_2^{3/2}} \quad (5.11)$$

where

$$m_i = \frac{1}{N} \sum_{n=1}^N (x[n] - \mu)^i \quad (5.12)$$

In Python, the α_1 , α_2 , and g_1 parameters can be determined as follow as reported in the code listing 5.8.

```

1 import numpy as np
2 from scipy.stats import skew
3
4 a_mean = myDataset.Zr.mean()
5 median = myDataset.Zr.median()
6 hist, bin_edges = np.histogram(myDataset['Zr'], bins= 20,
7     density=True)
8 modal_value = (bin_edges[hist.argmax()] + bin_edges[hist.
9     argmax()+1])/2
10 standard_deviation = myDataset['Zr'].std()
11
12 a1 = (a_mean - modal_value) / standard_deviation
13 a2 = 3 * (a_mean - median) / standard_deviation
14 g1 = skew(myDataset['Zr'])
15
16 print ('-----')
17 print ("Pearson's first coefficient of skewness: {:.2f}".
18     format(a1))
19 print ("Pearson's 2nd moment of skewness: {:.2f}".format(a2))
20 print ("Fisher-Pearson's coefficient of skewness: {:.2f}".
21     format(g1))
22 print ('-----')
23
24 '''
25 Output:
26 -----
27 Pearson's first coefficient of skewness: 0.74
28 Pearson's 2nd moment of skewness: 0.66
29 Fisher-Pearson's coefficient of skewness: 1.26
30 -----
31 '''

```

Listing 5.8 Measuring the skewness of a data set.

5.5 Descriptive Statistics in Pandas

As reported in the official documentation of pandas, the command `describe()` "generates descriptive statistics that summarize the central tendency, dispersion and shape of a data set's distribution, excluding NaN (i.e., Not a Number) values."

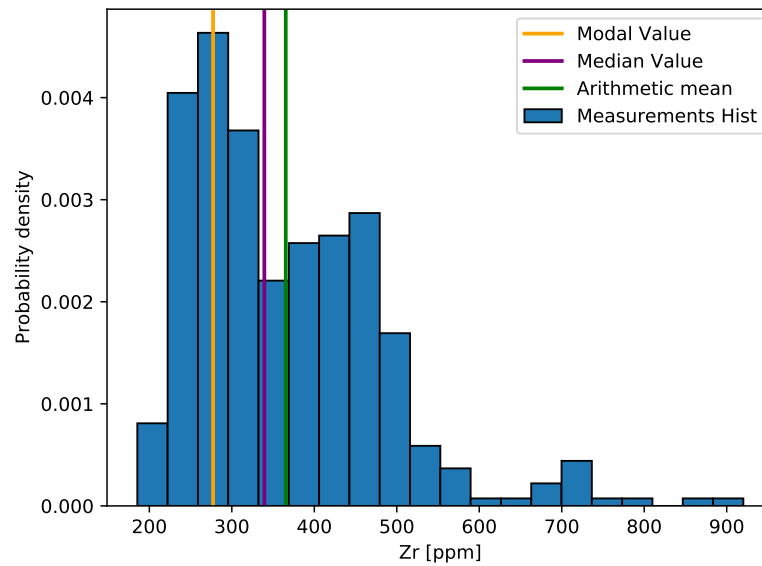


Fig. 5.7 The result of the code reported in the listing 5.7.

```

1 import pandas as pd
2
3 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
4     sheet_name='Supp_traces')
5
6 statistics = myDataset[['Ba', 'Sr', 'Zr', 'La']].describe()
7
8 print(statistics)
9
10 Output:
11
12      Ba      Sr      Zr      La
13 count  370.000000  369.000000  370.000000  370.000000
14 mean   789.733259  516.422115  365.377397   74.861088
15 std    523.974960  241.922439  118.409962   18.256772
16 min     0.000000    9.541958  185.416567   45.323289
17 25%    297.402777  319.667551  274.660242   61.745228
18 50%    768.562055  490.111131  339.412064   71.642167
19 75%   1278.422645  728.726116  438.847368   83.670805
20 max    2028.221963 1056.132069  920.174406  169.550008

```

Listing 5.9 Computing descriptive statistics in pandas.

5.6 Box Plots

A box plot or boxplot describe groups of numerical data using the inter-quartile distance. Also, box lines extending from the boxes, i.e., whiskers, indicate the variability outside the upper and lower quartiles. The outliers are sometime plotted as individual symbols. In detail, the bottom and the top of a box are always the first and third quartiles. A line is always reported inside the box and it represents the second quartile (i.e., the median). Regarding the whisker length, by default matplotlib uses a value equal to 1.5 multiplied by the inter-quartile distance. Any data not included between the whiskers is considered as an outlier. Using the matplotlib library, a boxplot can be defined as follow:

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5     sheet_name='Supp_traces')
6
7 fig, ax = plt.subplots()
8 my_flierprops = dict(markerfacecolor='#f8e9a1',
9     markeredgecolor='#24305e', marker='o')
10 my_medianprops = dict(color='#f76c6c', linewidth = 2)
11 my_boxprops = dict(facecolor='#a8d0e6', edgecolor='#24305e')
12 ax.boxplot(myDataset.Zr, patch_artist = True, notch=True,
13     flierprops = my_flierprops, medianprops = my_medianprops ,
14     boxprops = my_boxprops)
15 ax.set_ylabel('Zr [ppm]')
16 ax.set_xticks([1])
17 ax.set_xticklabels(['all Epochs'])
18 plt.show()

```

Listing 5.10 Providing a qualitative check of the skewness of a data set.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4
5 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
6     sheet_name='Supp_traces')
7
8 fig, ax = plt.subplots()
9 ax = sns.boxplot(x="Epoch", y="Zr", data=myDataset, palette="
10     Set3")

```

Listing 5.11 Providing a qualitative check of the skewness of a data set.

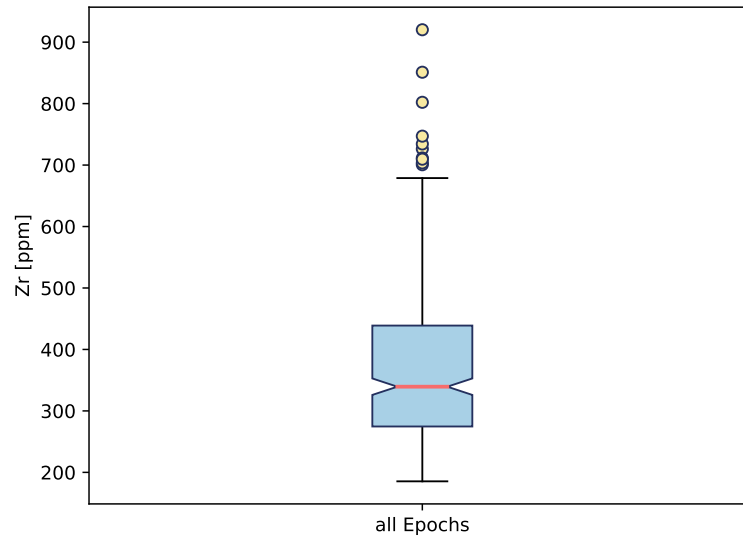


Fig. 5.8 The result of the code reported in the listing 5.10.

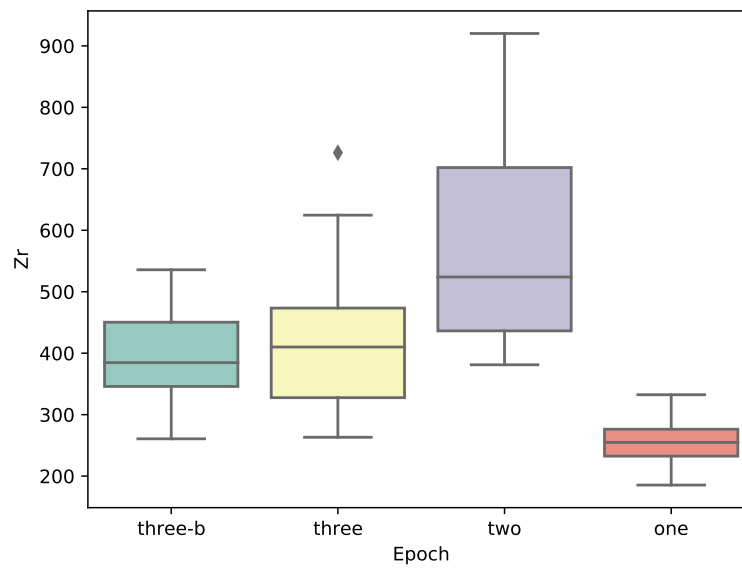


Fig. 5.9 The result of the code reported in the listing 5.11.

Chapter 6

Descriptive Statistics 2: Bivariate Analysis

6.1 Covariance and Correlation

In the present chapter, we start investigating how to capture the relationships between two variables, i.e., bivariate statistics. To begin, consider Fig. 6.1, resulting from the code listing 6.1. As you can infer, the two diagrams reported in Fig. 6.1 are different. From the previous chapter, we know how to describe each variable (i.e., La, Ce, Sc, and U) appearing in Fig. 6.1 using indexes of location (e.g., the arithmetic mean), dispersion (e.g., the standard deviation) and shape (e.g., the skewness).

```
1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5     sheet_name='Supp_traces')
6
7 fig = plt.figure()
8 ax1 = fig.add_subplot(2,1,1)
9 ax1.scatter(myDataset.La, myDataset.Ce, marker='o', edgecolor='k',
10     color='#c7ddf4', label='CFC recent Activity')
11 ax1.set_xlabel('La [ppm]')
12 ax1.set_ylabel('Ce [ppm]')
13 ax1.legend()
14
15 ax2 = fig.add_subplot(2,1,2)
16 ax2.scatter(myDataset.Sc, myDataset.U, marker='o', edgecolor='k',
17     color='#c7ddf4', label='CFC recent Activity')
18 ax2.set_xlabel('Sc [ppm]')
19 ax2.set_ylabel('U [ppm]')
20 ax2.legend()
```

Listing 6.1 Linear relation between two variables.

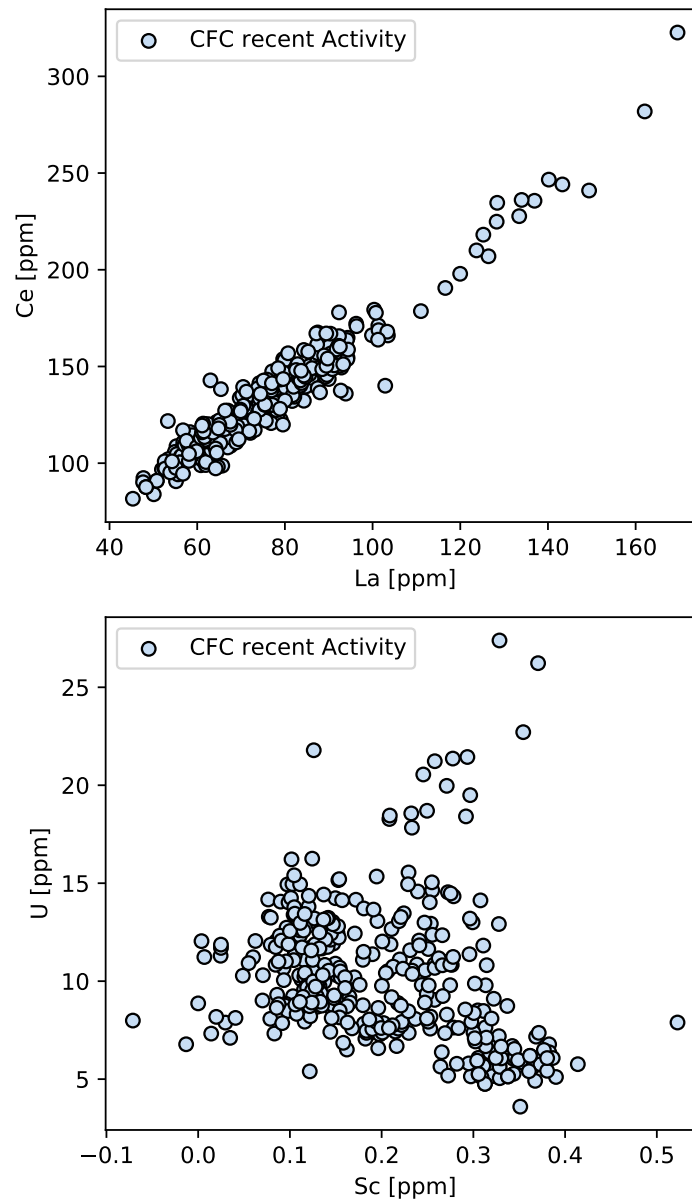


Fig. 6.1 The result of the code reported in the listing 6.1.

However, these indexes, although useful to describe a single variable, are not able to capture the relationships among them. As an example, the diagram La Vs. Ce clearly shows an increase in Ce as La increases and vice-versa. Mathematicians describe this using the concepts of covariance and correlation. On the contrary, it is not possible to define any simple relation between Sc and U.

Definition: The covariance of two sets of univariate samples x and yy , deriving from two random variables X and Y , is a measure of their joint variability, or their degree of correlation (Chatterjee & Hadi, 2013; Montgomery et al., 2012):

$$Cov_{xy} = \frac{\sum_{i=1}^n (y_i - \bar{y})(x_i - \bar{x})}{n - 1}. \quad (6.1)$$

A $Cov_{xy} > 0$ indicates a positive relationship between Y and X . On the contrary, if $Cov_{xy} < 0$, the relationship is negative (Chatterjee & Hadi, 2013; Montgomery et al., 2012). If X and Y are statistically independent, then $Cov_{xy} = 0$. Please note that while statistically independent variables are always uncorrelated, the converse is not necessarily true.

I'd like to stress that the covariance depends on the magnitudes of the two inspected variables. As a consequence, it does not tell us much about the strength of such a relationship (Chatterjee & Hadi, 2013; Montgomery et al., 2012). The normalized version of the covariance, i.e., the correlation coefficient, allows us to overcome this limitation showing, by its magnitude, the strength of the linear relation.

The Eq. 6.2 define the correlation coefficient, r_{xy} , for two joined univariate sets of data X and Y characterized by a covariance Cov_{xy} and standard deviations σ_{sx} and σ_{sx} , respectively (Chatterjee & Hadi, 2013; Montgomery et al., 2012):

$$r_{xy} = \frac{Cov_{xy}}{\sigma_{sx} \cdot \sigma_{sy}} = \frac{\sum_{i=1}^n (y_i - \bar{y})(x_i - \bar{x})}{\sqrt{\sum_{i=1}^n (y_i - \bar{y})^2 \sum_{i=1}^n (x_i - \bar{x})^2}}. \quad (6.2)$$

By definition, r_{xy} is scale invariant, i.e., it does not depend on the magnitude of considered values. Also, r_{xy} satisfies the following relation (Chatterjee & Hadi, 2013; Montgomery et al., 2012):

$$-1 \leq r_{xy} \leq 1. \quad (6.3)$$

For a pandas DataFrame, the covariance and the correlation can be readily computed using the `cov()` and `corr()` functions. These functions calculate the covariance and the correlation matrices for a DataFrame, respectively. A covariance matrix is a table showing the covariances Cov_{xy} between the variables in the DataFrame. Each cell in the table shows the covariance between two variables. The correlation matrix follows the same logic as the covariance matrix but reporting the correlation coefficients. In the latter, the diagonal is characterized by values equal to one, corresponding to the self-correlation coefficients.

The code listing 6.2 and Fig 6.2 report the computation and the subsequent representation of the covariance and the correlation matrices for the elements reported in Fig. 6.1, i.e., Ce, La, U, and Sc.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4
5 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
6     sheet_name='Supp_traces')
7
8 mySubDataset = myDataset[['Ce', 'La', 'U', 'Sc']]
9
10 cov = mySubDataset.cov()
11 cor = mySubDataset.corr()
12
13 fig = plt.figure(figsize=(11,5))
14
15 ax1 = fig.add_subplot(1,2,1)
16 ax1.set_title('Covariance Matrix')
17 sns.heatmap(cov, annot=True, cmap='cividis', ax=ax1)
18
19 ax2 = fig.add_subplot(1,2,2)
20 ax2.set_title('Correlation Matrix')
21 sns.heatmap(cor, annot=True, vmin=-1, vmax=1, cmap='coolwarm',
22     ax=ax2)
23 fig.tight_layout()

```

Listing 6.2 Estimating the covariance and the correlation matrix.

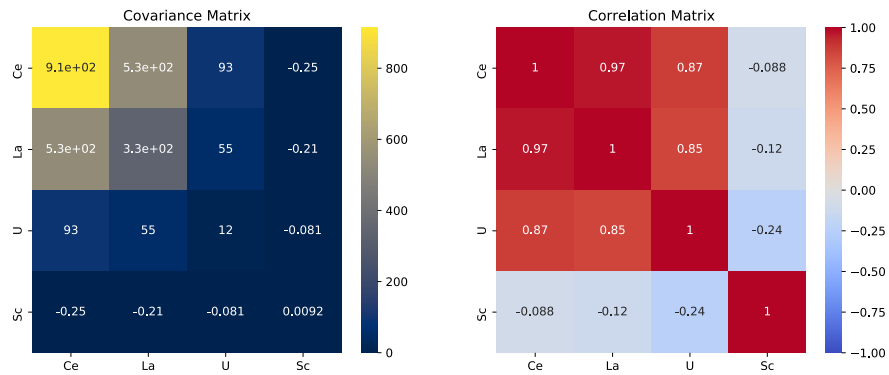


Fig. 6.2 The result of the code reported in the listing 6.2.

Please note that a value of r_{xy} close to 0 only means that X and Y are not linearly related, not excluding other relationships (Chatterjee & Hadi, 2013; Montgomery et al., 2012).

To evaluate non-linear relationships, other parameters should be used. As an example, the Spearman rank-order correlation coefficient is a non-parametric measure of the monotonicity of the relationship between two data sets. As in the case of the Pearson correlation coefficient, the Spearman rank-order correlation coefficient varies between -1 and +1, with 0 implying no correlation. Correlations of -1 or +1 imply an exact monotonic relationship. A positive correlation implies that as X increases, so does Y. On the contrary, negative correlations imply that as X increases, Y decreases. In python, the `scipy.stats.spearmanr()` function calculates the Spearman correlation coefficient together with the associated confidence (i.e., p-values).

6.2 Simple Linear Regression

Considering a response variable Y and a predictor X, we can define a linear model using the Eq. 6.4 (Chatterjee & Hadi, 2013; Montgomery et al., 2012):

$$Y = \beta_0 + \beta_1 X + \epsilon \quad (6.4)$$

where β_0 and β_1 are coefficients named intercept (i.e., the predicted value of Y at X=0) and slope (i.e., the change in Y for unit variation in X), respectively. Also, ϵ is the residual error (Chatterjee & Hadi, 2013; Montgomery et al., 2012). Using the least squares method, i.e., minimizing the sum of squares of the vertical distances from each point to the Eq. 6.4, β_1 and β_0 are estimated using the Eq. 6.5 and 6.6, respectively:

$$\beta_1 = \frac{\sum_{i=1}^n (y_i - \bar{y})(x_i - \bar{x})}{\sum_{i=1}^n (x_i - \bar{x})^2} = \frac{Cov_{xy}}{\sigma_{sx}^2} = r_{xy} \frac{\sigma_{sy}}{\sigma_{sx}} \quad (6.5)$$

$$\beta_0 = \bar{y} - \beta_1 \bar{x} \quad (6.6)$$

The square of the correlation coefficient (r_{xy}^2) with $0 \leq r_{xy}^2 \leq 1$, is typically used to provide a preliminary estimation of the goodness of the regression models.

A more exhaustive evaluation of the model requires a detailed analysis of the errors, i.e., error analysis, a topic that we will discuss in the Chapter 10.

In Python, there are many implementations of the least squares method for first order linear regression. Examples are the `linregress()` function (code listing 6.3 and Fig. 6.3) in the statistical module of Scipy and the linear regression module in statsmodels¹.

¹ <https://www.statsmodels.org>

```

1 import pandas as pd
2 import scipy.stats as st
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
7     sheet_name='Supp_traces')
8 fig = plt.figure()
9 ax1= fig.add_subplot(2,1,1)
10 ax1.scatter(myDataset.La, myDataset.Ce, marker='o', edgecolor=
11     'k', color='#c7ddf4', label='CFC recent Activity')
12 b1, b0, rho_value, p_value, std_err = st.linregress(myDataset.
13     La, myDataset.Ce)
14 x = np.linspace(myDataset.La.min(),myDataset.La.max())
15 y = b0 + b1*x
16 ax1.plot(x, y, linewidth=1, color='#ff464a', linestyle='--',
17     label = r"fit param.: $\beta_0$ = " + str(round(b0,1)) + r
18     " - $\beta_1$ = " + str(round(b1,1)) + r" - $r_{xy}^{2}$
19     = " + str(round(rho_value**2,2)))
20 ax1.set_xlabel('La [ppm]')
21 ax1.set_ylabel('Ce [ppm]')
22 ax1.legend(loc= 'upper left')
23
24 ax2 = fig.add_subplot(2,1,2)
25 ax2.scatter(myDataset.Sc, myDataset.U, marker='o', edgecolor='
26     k', color='#c7ddf4', label='CFC recent Activity')
27 b1, b0, rho_value, p_value, std_err = st.linregress(myDataset.
28     Sc, myDataset.U)
29 x = np.linspace(myDataset.Sc.min(),myDataset.Sc.max())
30 y = b0 + b1*x
31 ax2.plot(x, y, linewidth=1, color='#ff464a', linestyle='--',
32     label = r"fit param.: $\beta_0$ = " + str(round(b0,1)) + r
33     " - $\beta_1$ = " + str(round(b1,1)) + r" - $r_{xy}^{2}$
34     = " + str(round(rho_value**2,2)))
35 ax2.set_xlabel('Sc [ppm]')
36 ax2.set_ylabel('U [ppm]')
37 ax2.legend(loc= 'upper left')

```

Listing 6.3 Least square linear regression applied to the data of Fig. 6.1.

6.3 Polynomial Regression

The linear model defined in Eq. 6.4 can be easily generalized to a polynomial of degree n (Eq. 6.7):

$$Y = \beta_0 + \beta_1 X + \beta_2 X^2 + \beta_3 X^3 + \dots + \beta_n X^n + \epsilon \quad (6.7)$$

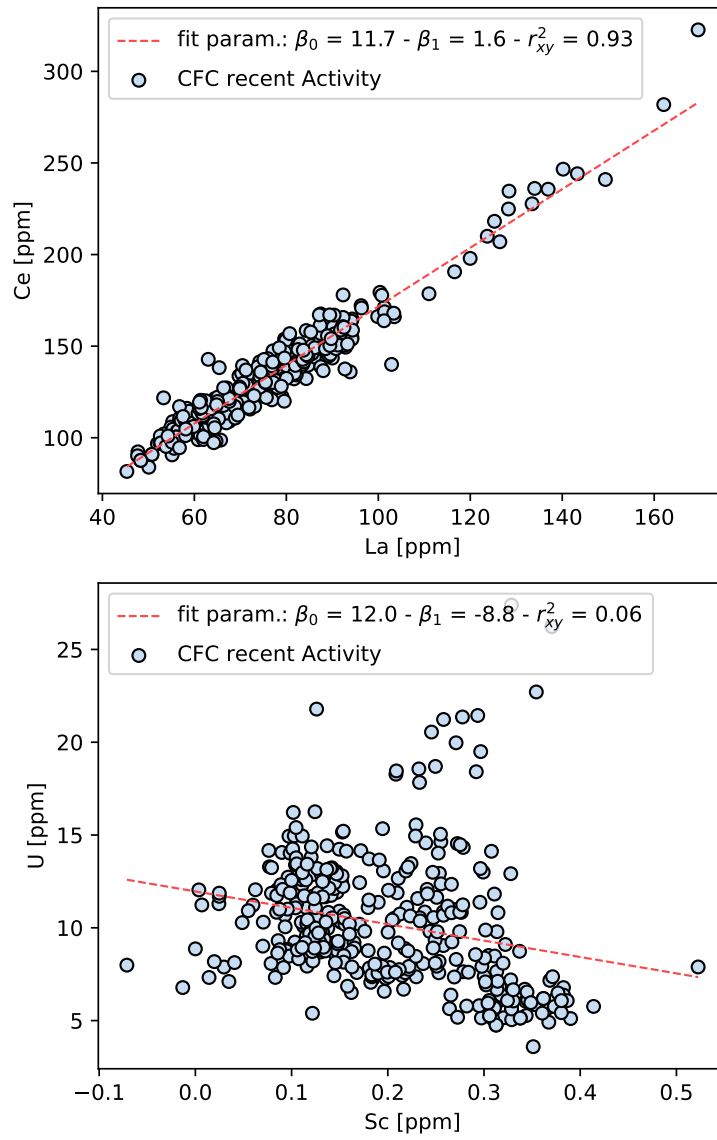


Fig. 6.3 The result of the code reported in the listing 6.3.

If $n > 1$, the function $Y(X)$ is not linear, but the regression model is still linear, since the regression parameters $\beta_0, \beta_1, \beta_2, \dots, \beta_n$, enter in Eq. 6.7 as linear terms (Chatterjee & Hadi, 2013; Montgomery et al., 2012).

Now, suppose having collected a geological quantity (e.g., the flow rate of a spring-water) at selected time intervals and you wish to fit your data with polynomial models of order 2, 3, and 4, respectively. The code listing 6.4 shows how to perform this task in Python using the `numpy.polyfit()` function (code listing 6.4 and Fig. 6.4).

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.arange(1,6)
5 y = np.array([0,1,2,9,9])
6
7 fig, ax = plt.subplots()
8 ax.scatter(x, y, marker = 'o', s = 100, color = '#c7ddf4',
9           edgecolor = 'k')
10
11 orders = np.array([2,3,4])
12 colors = ['#ff464a', '#342a77', '#4881e9']
13 linestyles = ['-', '--', '-.']
14
15 for order, color, linestyle in zip(orders, colors, linestyles):
16     betas = np.polyfit(x, y, order)
17     func = np.poly1d(betas)
18     x1 = np.linspace(0.5, 5.5, 1000)
19     y1 = func(x1)
20     ax.plot(x1, y1, color = color, linestyle = linestyle, label =
21           "Linear model of order " + str(order))
22
23 ax.legend()
24 ax.set_xlabel('A quantity relevant in geology\n(e.g., time)')
25 ax.set_ylabel('A quantity relevant in geology\n(e.g., spring flow
26           rate)')
27 fig.tight_layout()

```

Listing 6.4 Least square linear regression of n-order.

6.4 Non-Linear Regression

In regression analysis, the terms linear and non-linear does not describe the relationship between Y and X . Instead they are related to regression parameters entering the equation linearly or non-linearly (Chatterjee & Hadi, 2013; Montgomery et al., 2012). As an example, equations 6.4 and 6.7 are both linear. Also, the regression model for the Eq. 6.8 is linear. This is because they are linear natively (Eq. 6.4)

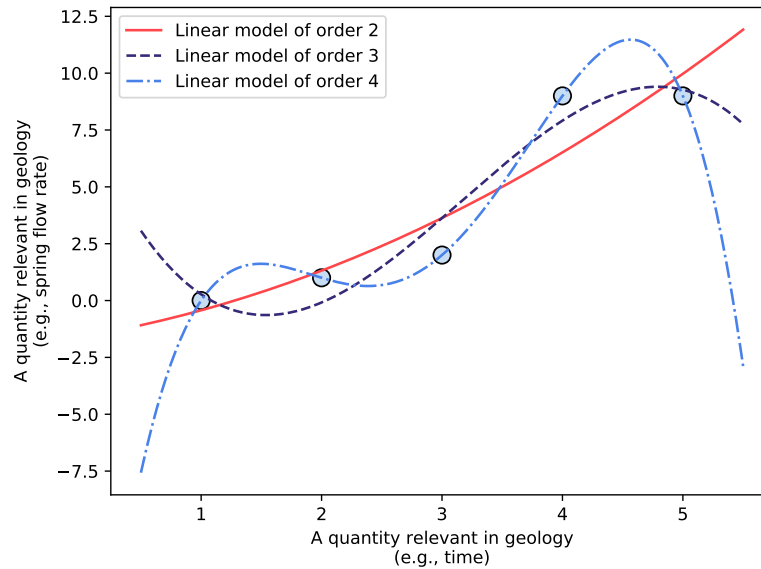


Fig. 6.4 The result of the code reported in the listing 6.4.

or they can be reported to a linear form after a transformation (Chatterjee & Hadi, 2013).

$$Y = \beta_0 + \beta_1 \log(X) + \epsilon \quad (6.8)$$

As an example, we can set $X^2 = X_2, X^3 = X_3, \dots, X^n = X_n$ in Eq. 6.7 and $X_1 = \log(X)$ in Eq. 6.8. The resulting Eq. 6.9 and Eq. 6.10 are both linear:

$$Y = \beta_0 + \beta_1 X + \beta_2 X_2 + \beta_3 X_3 + \dots + \beta_n X_n + \epsilon, \quad (6.9)$$

$$Y = \beta_0 + \beta_1 X_1 + \epsilon \quad (6.10)$$

As a general definition, in linear models all regression parameters enter the equation linearly, possibly after transformation of the data (Chatterjee & Hadi, 2013). On the contrary, in non-linear models the relationship between Y and some of the predictors is non-linear or some of the parameters appear non-linearly, but no transformation is possible to make the parameters appear linearly (Chatterjee & Hadi, 2013). Table 6.1 reports a checklist, modified from Motulsky and Christopoulos (2004), to evaluate if the linear regression is an appropriate approach for your data set.

An example of non-linear regression in petrology occurs during the application of the the crystal lattice-strain model (Blundy & Wood, 1994) for the interpretation of experimental data. In detail, the crystal lattice-strain model provides a conceptual

Table 6.1 Is the linear regression appropriate for your geological data set? Modified from Motulsky and Christopoulos (2004).

Question	Discussion
Can the relationship between X and Y be described by a straight line?	For many geological applications, the relationship between X and Y is not linear, making linear regression inappropriate. The suggestion is to either transform the data, or perform nonlinear curve fitting.
Is the scattering of data around the line Normally distributed? Is the variability the same everywhere?	Please note that linear regression analysis assumes that the scatter is Gaussian. Linear regression assumes that the scattering around the best-fit line has the same standard deviation all along the curve. The assumption is violated if the points with high or low X values tend to be farther from the best-fit line (i.e., homoscedasticity).
Do you know the X values precisely?	The least square linear regression model assumes that X values are exactly correct, i.e., X is very small compared to the variability in Y, and that experimental error or geological variability only affects the Y values.
Are the data points independent?	Whether one point is above or below the line is a matter of chance, and does not influence whether another point is above or below the line.
Are the X and Y values intertwined?	If the value of X is used to calculate Y (or the value of Y is used to calculate X), then linear regression calculations are invalid.

framework for quantifying partition coefficients (D_i) in magmatic systems (Blundy & Wood, 1994; Meltzer & Kessel, 2020):

$$D_i = D_o \cdot \exp \left\{ \frac{-4 \cdot \pi \cdot E \cdot N_A \cdot \left[\frac{r_o}{2} (r_i - r_o)^2 + \frac{1}{3} (r_i - r_o)^3 \right]}{R \cdot T} \right\} \quad (6.11)$$

where T is the temperature, r_i is the radius the a trace element i belonging to an isovalent set of elements, r_o is the radius of the ideal element that minimally strains the crystal lattice (i.e., characterized by the largest D_i), D_o is the partition coefficient for the ideal element characterized by a radius equal to r_o , and E is the apparent Young's modulus of the site. Finally, N_A and R are the Avogadro's number and the gas constant, respectively (Blundy & Wood, 1994; Meltzer & Kessel, 2020).

The exponential equation 6.11 defines a parabolic behaviour in a diagram with r_i and $\log_{10}(D_i)$ on the x and y axes, respectively.

Typically, the r_o , D_o , and E parameters are estimated by fitting the Eq. 6.11 to experimentally determined D_i , by non-linear regression.

In Python, the function `scipy.optimize.curve_fit()` applies the non-linear least squares approach to fit a function to data, and can be used to extract r_o , D_o , and E from experimental D_i . In detail, `curve_fit()` bases on three algorithms: the Trust Region Reflective algorithm (Branch et al., 1999), The Dogleg algorithm with rectangular


```

33 x1 = np.linspace(0.85,1.2,1000)
34 y1 = func(x1,popt1[0],popt1[1], popt1[2])
35 ax1.plot(x1,y1, color='#ff464a', linewidth=2, linestyle='--',
          label=r'$r_0$ = ' + str(round(popt1[0],3)) + r', $D_0$ = ' +
          str(round(popt1[1],0)) + ', E = ' + str(round(popt1[2],0)))
36 add_elements(ax = ax1)
37 ax1.set_yscale('log')
38 ax1.set_xlabel(r'Ionic Radius ($\AA$)')
39 ax1.set_ylabel(r'$D_i$')
40 ax1.set_ylim(0.005,3000)
41 ax1.legend()
42
43 # Levenberg-Marquardt algorithm
44 ax2 = fig.add_subplot(1,2,2)
45 ax2.set_title("Levenberg-Marquardt algorithm")
46 ax2.scatter(I_r, Di, s=80, color='#c7ddf4', edgecolors='k', label
            ='4 GPa - 1073 K, Kessel et al., 2005')
47
48 popt2, pcov2 = curve_fit(func, I_r, Di, method='lm', p0
            =(1.1,100,100))
49
50 x2 = np.linspace(0.85,1.2,1000)
51 y2 = func(x2,popt2[0],popt2[1], popt2[2])
52 ax2.plot(x2,y2, color='#4881e9', linewidth=2, linestyle='--',
          label=r'$r_0$ = ' + str(round(popt2[0],3)) + r', $D_0$ = ' +
          str(round(popt2[1],0)) + ', E = ' + str(round(popt2[2],0)))
53 add_elements(ax = ax2)
54 ax2.set_yscale('log')
55 ax2.set_xlabel(r'Ionic Radius ($\AA$)')
56 ax2.set_ylabel(r'$D_i$')
57 ax2.set_ylim(0.005,3000)
58 ax2.legend()
59
60 fig.tight_layout()

```

Listing 6.5 Least square non-linear regression to extract r_0 , D_0 , and E from an experimental set of D_i in the framework of the crystal lattice-strain model (Blundy & Wood, 1994).

Figure 6.5 shows the best fitting of the Eq. 6.11 using two different algorithms. They are the Trust Region Reflective algorithm (Branch et al., 1999) with bounds for the r_0 , D_0 , and E parameters (line 31 of code listing 6.5) and the Levenberg-Marquardt algorithm (Moré, 1978) provided with an initial guess of r_0 , D_0 , and E (p_0 , line 48 of code listing 6.5), respectively. The two algorithms return the same best fit parameters (Fig. 6.5).

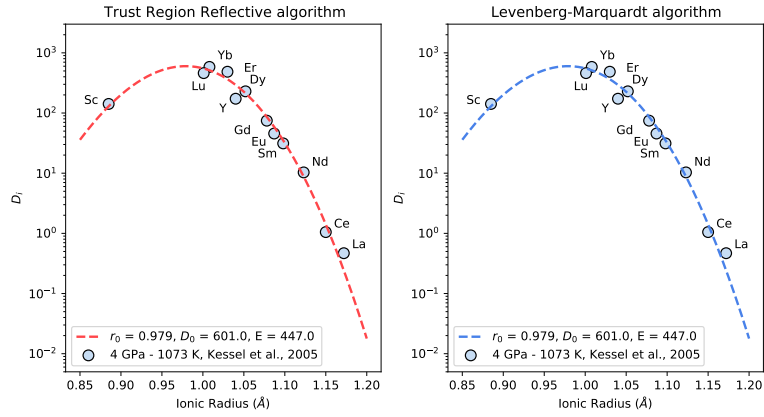


Fig. 6.5 The result of the code reported in the listing 6.5.

Part III
Integrals and Differential Equations in
Geology

Chapter 7

Numerical Integration

7.1 Definite Integrals

From the operational point of view, the integration mainly involves problems of two different classes (Priestley, 1997). The ones belonging to the first class, i.e., indefinite integrals, are those in which we know the derivative of a function and we aim at finding the function (Priestley, 1997). The problems of the second class, i.e., definite integrals, consist of adding up a large amount of extremely small quantities to find areas, volumes, centers of gravity, and many other applications (Priestley, 1997).

For most geological applications, the knowledge about integrals can be reduced to definite integrals.

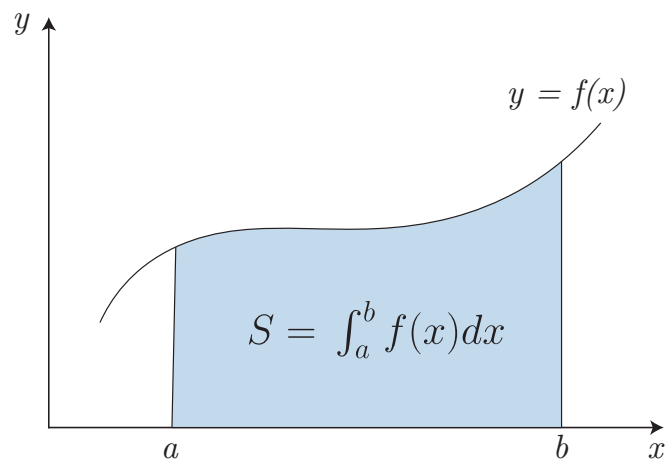


Fig. 7.1 Definite Integral.

Informal definition: Given a function f of a real variable x , the definite integral (S) of $f(x)$ in an interval of real numbers $[a, b]$ can be expressed as the area that is bounded by $f(x)$, the x -axis, and the vertical lines at x equal to a and b , respectively (Fig. 7.1).

Please note that the regions above and below the x -axis add and subtract from the total, respectively (Fig. 7.2).

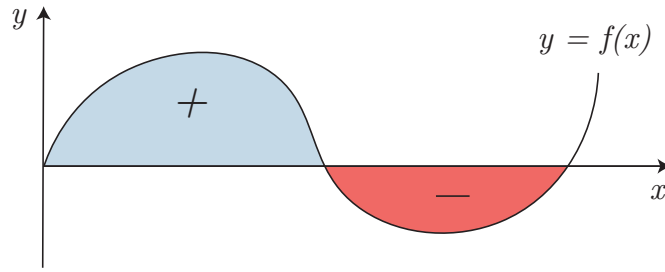


Fig. 7.2 Sign of definite integrals.

7.2 Basic Properties of Integrals

Definite integrals are subjected to some interesting properties, often useful during the solution of complex problems, reducing them to simpler ones.

Additive properties

$$\int_a^b f(x)dx + \int_b^c f(x)dx = \int_a^c f(x)dx \quad (7.1)$$

$$\int_a^a f(x)dx = 0 \quad (7.2)$$

$$\int_a^b f(x)dx = - \int_b^a f(x)dx \quad (7.3)$$

Scaling by a constant

$$\int_a^b c \cdot f(x)dx = c \int_a^b f(x)dx \quad (7.4)$$

Integral of a sum

$$\int_a^b [f(x) + g(x)]dx = \int_a^b f(x)dx + \int_a^b g(x)dx \quad (7.5)$$

7.3 Analytical and Numerical Solutions of Definite Integrals

As a general definition, analytical methods give exact solutions, but sometimes they are impossible to achieve. On the contrary, numerical methods give approximate solutions with allowable tolerances (i.e., an error characterized by a known confidence limit). Also, the use of numerical methods is mandatory when the function is only empirically estimated at discrete points, as in most cases dealing with geological sampling (e.g., volatile fluxes at volcanic areas).

A detailed description of the analytical solutions of definite integrals is behind the scope of this book and, in the following, I will provide a the definition for the 'Fundamental Theorem of Calculus' and few simple examples based on the symbolic approach in Python.

On the contrary, I will discuss the numerical methods in detail, mainly focusing on the algorithms allowing the solution of definite integrals even when $f(x)$ is not mathematically defined (i.e., we know some given fixed occurrences only) as in the case of the sampling in many geological fields.

7.4 Fundamental Theorem of Calculus and Analytical Solutions

Fundamental Theorem of Calculus

The 'Fundamental Theorem of Calculus' formulates an analytical link between differentiation and integration. The theorem is constituted of two parts. The first part establishes the relationship between differentiation and integration (Priestley, 1997; Strang et al., 2016).

Part 1: If $F(x)$ is continuous over an interval $[a, b]$ and the function $F(x)$ is defined by:

$$F(x) = \int_a^x f(t)dt \quad (7.6)$$

then $F'(x) = f(x)$ over $[a, b]$, and we define $F(x)$ as antiderivative of $f(x)$.

The second part affirms that if we can determine an antiderivative for the integrand, then we can assess the definite integral by evaluating the antiderivative at the extreme points of the interval and subtracting.

Part 2: If $f(x)$ is continuous over the interval $[a, b]$ and $F(x)$ is any antiderivative of $f(x)$, then

$$\int_a^b f(x)dx = F(b) - F(a) \quad (7.7)$$

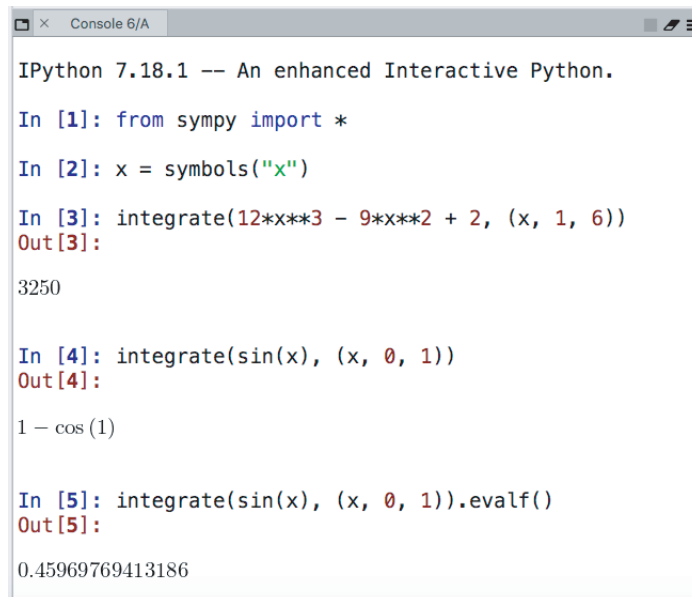
Analytical Solutions: the Symbolic Approach in Python

Symbolic computation deals with the manipulation and solution of mathematical expressions symbolically (Meurer et al., 2017). In the symbolic computation, mathematical objects are represented exactly and not approximately as in the case of numerical solutions (Meurer et al., 2017). Also, mathematical expressions with un-evaluated variables are left in the symbolic form (Meurer et al., 2017). In detail, the SymPy package uses the symbolic approach to simplify expressions, compute derivatives, integrals, and limits, solve equations, work with matrices, etc (Meurer et al., 2017).

As a simple example consider Fig. 7.3. It shows the use of SimPy to solve, analytically, the two definite integrals reported in Eq. 7.8 and Eq. 7.9.

$$\int_1^6 12x^3 - 9x^2 + 2dx = [3x^4 - 3x^3 + 2x]_1^6 = (3252 - 2) = 3250 \quad (7.8)$$

$$\int_0^1 \sin(x)dx = [-\cos(x)]_0^1 = 1 - \cos(1) \simeq 0.46 \quad (7.9)$$



```

IPython 7.18.1 -- An enhanced Interactive Python.

In [1]: from sympy import *
In [2]: x = symbols("x")
In [3]: integrate(12*x**3 - 9*x**2 + 2, (x, 1, 6))
Out[3]:
3250

In [4]: integrate(sin(x), (x, 0, 1))
Out[4]:
1 - cos(1)

In [5]: integrate(sin(x), (x, 0, 1)).evalf()
Out[5]:
0.45969769413186

```

Fig. 7.3 Symbolic integration using SimPy.

7.5 Numerical Solutions of Definite Integrals

Rectangles method

The simplest method to numerically approximate the solution of a definite integral is to divide the area of interest with many rectangles of equal width and variable height, then summing up all the areas of rectangles (Fig 7.4):

$$\int_a^b f(x)dx \approx h \sum_{i=0}^{n-1} f(x_i) \quad (7.10)$$

where n is the number of rectangles, $x_0 = a$, $x_n = b$, and h is defined as:

$$h = \frac{b - a}{n} \quad (7.11)$$

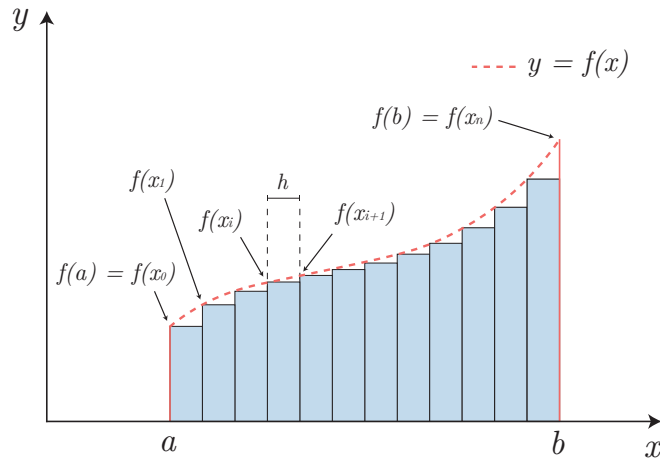


Fig. 7.4 The 'left-rectangles' method to solve definite integrals.

The procedure reported in Eq. 7.10 and Fig. 7.4 is the so called 'left-rectangular' approximation. Additional options are the 'right-' (Eq. 7.12) and 'midpoint-rectangular' approximations (Eq. 7.13), respectively (Fig. 7.5).

$$\int_a^b f(x)dx \approx h \sum_{i=1}^n f(x_i) \quad (7.12)$$

$$\int_a^b f(x)dx \approx h \sum_{i=0}^{n-1} \frac{(f(x_i) + f(x_{i+1}))}{2} \quad (7.13)$$

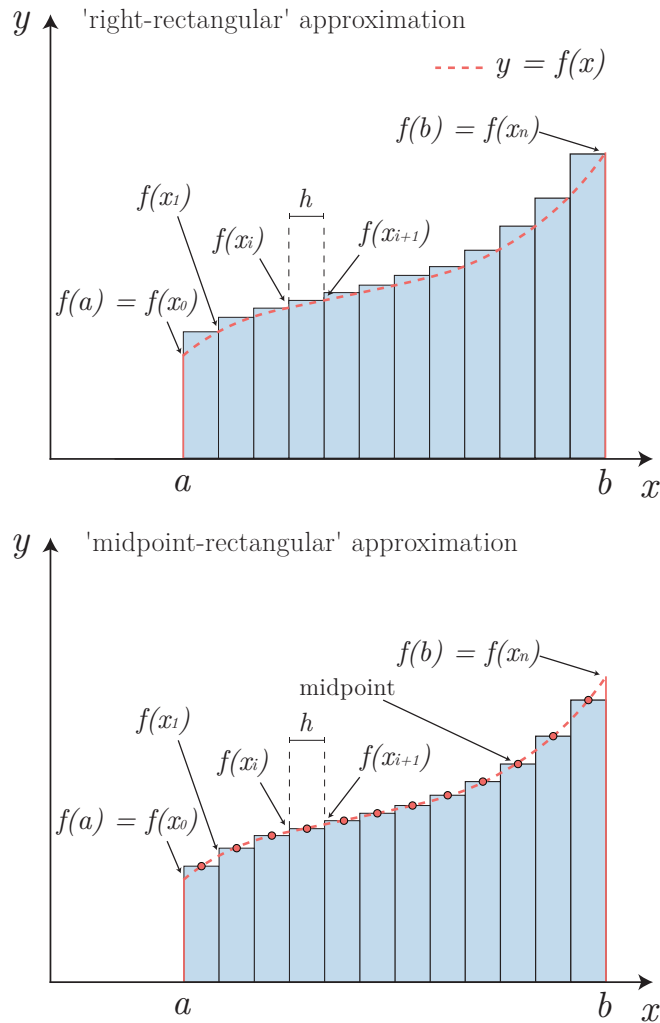


Fig. 7.5 The 'right-' and 'midpoint-rectangles' approximations to solve definite integrals.

Filling $f(x)$ using rectangles will roughly approximate the area of interest. However, the more rectangles you insert in between the boundaries (a and b), the more accurate the approximation will be since the untouched regions become more sparse.

In python, we can write a simple function to implement the rectangle method (code listing 7.1).


```

1 import numpy as np
2
3 def integrate_rec(f, a, b, n):
4     # Implementation of the rectangle method
5     h = (b-a)/n
6     x = np.linspace(a, b, n+1)
7     i=0
8     area=0
9     while i<n:
10         Sup_rect = f(x[i])*h
11         area += Sup_rect
12         i += 1
13     return area
14 '''
15 We test the Rectangle method on the sine function where the
16     definite integral in the interval [0, π/2] is equal to 1.
17 '''
18 S_5 = integrate_rec(np.sin, 0, np.pi/2, 5)
19 S_10 = integrate_rec(np.sin, 0, np.pi/2, 10)
20 S_100 = integrate_rec(np.sin, 0, np.pi/2, 100)
21
22 print('Using n=5, the rectangle method returns a value of {:.2
23     f}'.format(S_5))
24 print('Using n=10, the rectangle method returns a value of
25     {:.2f}'.format(S_10))
26 print('Using n=100, the rectangle method returns a value of
27     {:.2f}'.format(S_100))
28 '''
29 Output:
30 Using n=5, the rectangle method returns a value of 0.83
31 Using n=10, the rectangle method returns a value of 0.92
32 Using n=100, the rectangle method returns a value of 0.99
33 '''

```

Listing 7.1 Rectangles rule to solve definite integrals.

Trapezoidal rule

The trapezoidal rule is a technique similar to the rectangles method. Instead of rectangles, the trapezoidal rule uses trapezoids to fill the area under $f(x)$ (Fig. 7.6). The Eq. 7.14 and code listing 7.2 report the mathematical formulation of the trapezoidal rule and its implementation in Python, respectively.

$$S = \int_a^b f(x)dx \approx h \cdot \left[\frac{f(x_0) + f(x_n)}{2} \right] \cdot \sum_{i=1}^{n-1} f(x_i) \quad (7.14)$$

```

1 import numpy as np
2
3 def integrate_trap(f, a, b, n):
4     # Implementation of the trapezoidal rule
5     h = (b-a)/n
6     x = np.linspace(a, b, n+1)
7     i=1
8     area = h*(f(x[0]) + f(x[n]))/2
9     while i<n:
10        Sup_rect = f(x[i])*h
11        area += Sup_rect
12        i += 1
13    return area
14
15 '''
16 We test the trapezoidal rule on the known sine function where
17 the definite integral in the interval [0, π/2] is equal to 1.
18 '''
19
20 S_5 = integrate_trap(np.sin, 0, np.pi/2, 5)
21 S_10 = integrate_trap(np.sin, 0, np.pi/2, 10)
22
23 print('Using n=5, the trapezoidal rule returns a value of {:.2
24       f}'.format(S_5))
25 print('Using n=10, the trapezoidal rule returns a value of
26       {:.2f}'.format(S_10))
27
28 '''
29 Output:
30 Using n=5, the trapezoidal rule returns a value of 0.99
31 Using n=10, the trapezoidal rule returns a value of 1.00
32 '''

```

Listing 7.2 Trapezoidal rule to solve definite integrals.

Trapezoidal and composite Simpson rules using scipy

The `scipy.integrate` sub-package implements many techniques for the solution of definite integrals. These methods include the trapezoidal rule reported in the previous paragraph and the composite Simpson rule.

The composite Simpson rule consists of a technique that approximates the integral over each pair of consecutive sub-intervals using quadratic functions (Fig. 7.7). The resulting formula to calculate a definite integral is given in Eq. 7.15.

$$S = \int_a^b f(x)dx \approx \frac{h}{3} \sum_{i=1}^{n/2} [f(x_{2i-2}) + 4f(x_{2i-1}) + f(x_{2i})] \quad (7.15)$$

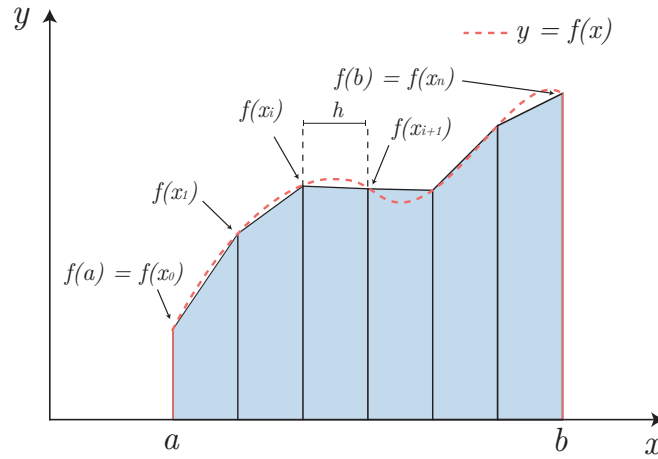


Fig. 7.6 Trapezoidal rule to solve definite integrals.

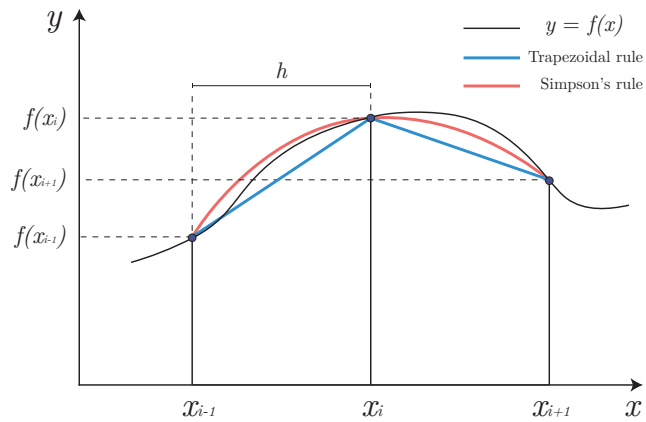


Fig. 7.7 Exemplification of the composite Simpson's rule, and comparison with the Trapezoidal rule.

Where n is an even number describing the number of sub-intervals of $[a, b]$, as in the case of the rectangles method and the trapezoidal rule.

The code listing 7.3 describes the application of the trapezoidal and the composite Simpson's rules to the equation $y = x^2$ using the `scipy.integrate` sub-package.

```

1 import numpy as np
2 from scipy import integrate
3
4
5 x = np.linspace(0,9, 3) # 3 divisions [x0,x1,x2], n=2
6 y = x**2
7
8 S_trapz = integrate.trapz(y,x)
9 S_simps = integrate.simps(y,x)
10
11
12 print('Using n=2, the trapezoidal rule returns a value of {:.0
13       f}'.format(S_trapz))
14 print('Using n=2, the composite Simpson rule returns a value
15       of {:.0f}'.format(S_simps))
16
17 '''
18 Output:
19 Using n=2, the trapezoidal rule returns a value of 273
20 Using n=2, the composite Simpson rule returns a value of 243
21 '''

```

Listing 7.3 Application of the trapezoidal and composite Simpson's rules to the equation $y = x^2$.

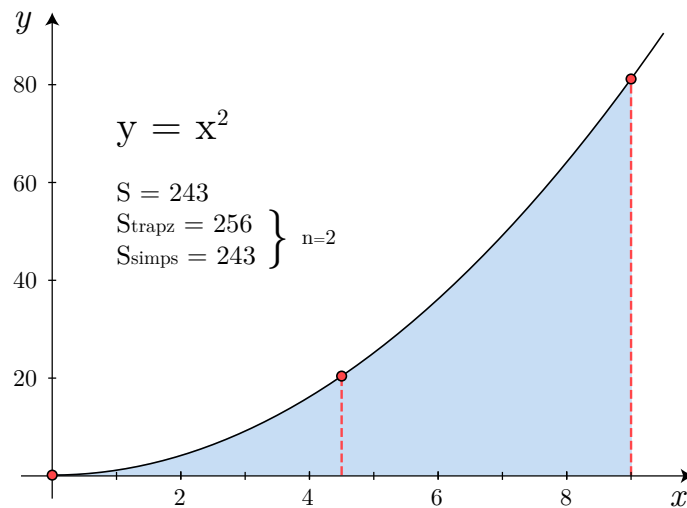


Fig. 7.8 Application of the trapezoidal and composite Simpson's rules to the equation $y = x^2$ in the interval $[0, 9]$. The analytical results is 243. Please note that being $y = x^2$ a quadratic function, it is perfectly fitted by the Simpson's rules.

7.6 Computing the Volume of Geological Structures

An example application in geology of definite integrals is the estimation of volume of structures that cannot be approximated by simple geometries. For example, the estimation of volumes is one of the most basic and widely applied tasks in hydrocarbon exploration and production (Slavinić & Cvetković Marko, 2016).

Qualitatively, to approximate the volume of a solid, we can slice it into many portions. Then, we estimate the volume of each single portion using quantifiable geometries (e.g., trapezoidal prisms). Finally, we sum all the obtained volume estimates (Slavinić & Cvetković Marko, 2016; Strang et al., 2016).

Quantitatively, if the distance between two successive slicing planes is infinitesimal, we can mathematically express the procedure using a definite integral (Eq. 7.16):

$$V = \int_a^b A(x) dx \quad (7.16)$$

where V is the volume of a solid extending from $x = a$ to $x = b$, and $A(x)$ is the area of its intersection with a plane passing through the point $(x,0,0)$, and parallel to yz (Fig. 7.9).

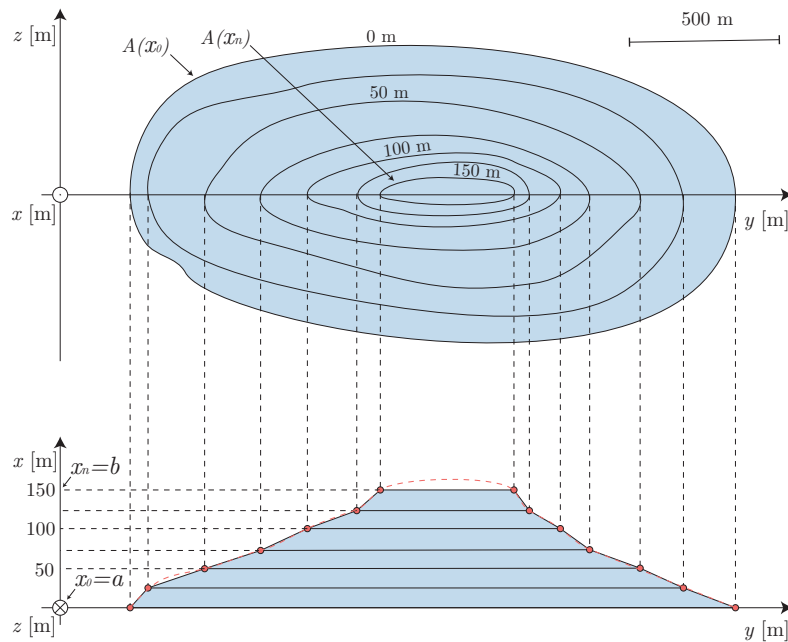


Fig. 7.9 Calculating the volume of geological structures.

The code listing 7.4 displays how to apply the Eq. 7.16 to the structure reported in Fig. 7.9.

```

1 import numpy as np
2 from scipy import integrate
3
4 contours_areas = np.array([194135, 136366, 79745, 38335, 18450,
5     9635, 3895])
6
7 x = np.array([0,25,50,75,100,125,150])
8
9 V_traps = integrate.trapz(contours_areas, x)
10 V_simps = integrate.simps(contours_areas, x)
11
12 print('The trapezoidal rule returns a volume of {:.0f} cubic
13     meters'.format(V_traps))
14 print('The composite Simpson rule returns a volume of {:.0f}
15     cubic meters'.format(V_simps))
16
17 '''
18 Output:
19 The trapezoidal rule returns a volume of 9538650 cubic meters
20 The composite Simpson rule returns a volume of 9431367 cubic
21     meters
22 '''

```

Listing 7.4 Application of the Eq. 7.16 to estimate the volume of the geological structure described in Fig. 7.9.

7.7 Computing the Lithostatic Pressure

We define the lithostatic pressure as the vertical pressure due to the weight of a column of rock at a specific depth. The pressure applied by a resting rock mass (this includes the fluids within the rock's pore space) under the acceleration of gravity is related to the rock's mass density by the formula (Eq. 7.17):

$$p(z) = p_0 + \int_0^z \rho(z) g(z) dz \quad (7.17)$$

where $p(z)$ is pressure at depth z , p_0 is the pressure at the surface, $\rho(z)$ is the bulk density for the rock mass as a function of depth, and $g(z)$ is the acceleration due to gravity.

As a zero-order approximation, we can assume p_0 equal to 0 and both $\rho(z)$ and $g(z)$ constant reducing the Eq. 7.17 to the Eq. 7.18. The code listing 7.5 reports the implementation of a the 7.18 in Python.

$$p(z) = \rho g z \quad (7.18)$$

```

1 def simple_lithopress(z,ro=2900,g=9.8):
2     p_MPa = z*g*ro/1e6 # return the pressure in MPa
3     return p_MPa
4
5 my_pressure = simple_lithopress(z=2000)
6
7 print('The pressure at 2000 meters is {:.0f} MPa'.format(
8     my_pressure))
9 '''
10 Output:
11 The pressure at 2000 meters is 57 MPa
12 '''

```

Listing 7.5 Simple listing showing the implementation of the Eq. 7.18 in Python.

Table 7.1 Earth's shells and relative densities.

Layer	r From [km]	r to [km]	Thickness [km]	Bottom Density [kg/m ³]	Top Density [kg/m ³]
Inner Core	1	1220	1220	13100	12800
Outer Core	1221	3479	2259	12200	9900
Lower Mantle	3480	5650	2171	5600	4400
Upper Mantle	5651	6370	720	4130	3400
Crust	6371	6400	30	3100	2700

Now, we move to the full implementation of the Eq. 7.17. Using the data reported by Dziewonski and Anderson (1981) and Anderson (1989), and assuming a linear variations for ρ from the upper and lower limit of each shell (i.e. crust, upper mantle, lower mantle, outer core and inner core) we can create an array defining a first-order approximation of $\rho(z)$ (Tab 7.1). To simplify data presentation, we start defining a new variable r (i.e., the distance from the Earth centre, with the Earth approximated to a sphere), as $r = R - z$, where R is the Earth radius ($R \approx 6400$ km).

```

1 import numpy as np
2 from scipy.integrate import trapz
3 import matplotlib.pyplot as plt
4
5 r = np.linspace(1,6400,6400)
6
7 def density():
8     ro_inner_core = np.linspace(13100, 12800, 1220)
9     ro_outer_core = np.linspace(12200, 9900, 2259)
10    ro_lower_mantle = np.linspace(5600,4400,2171)
11    ro_upper_mantle = np.linspace(4130,3400,720)
12    ro_crust = np.linspace(3100,2700,30)
13

```

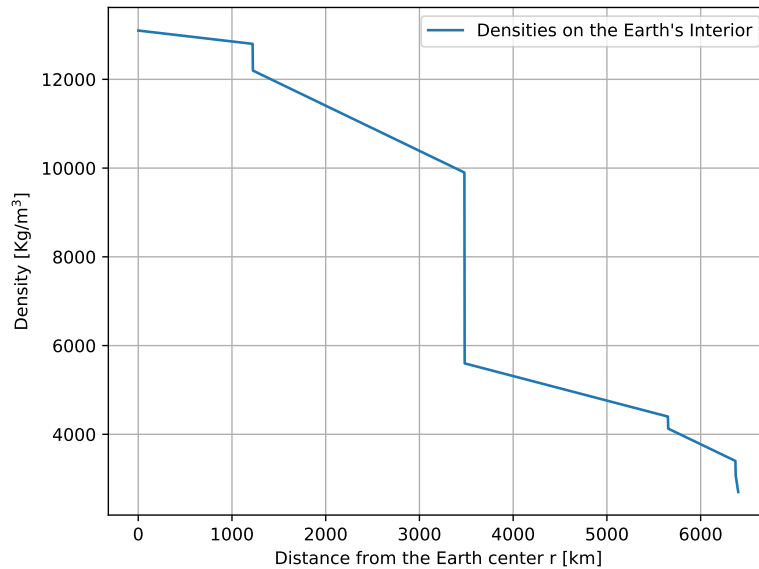


Fig. 7.10 The result of code listing 7.6.

```

14     ro_final = np.concatenate((ro_inner_core, ro_outer_core,
15                               ro_lower_mantle, ro_upper_mantle, ro_crust))
16     return ro_final
17
18 ro = density()
19
20 fig, ax = plt.subplots()
21 ax.plot(r,ro, label="Densities on the Earth's Interior")
22 ax.set_ylabel(r"Density [Kg/m$^3$]")
23 ax.set_xlabel("Distance from the Earth center r [km]")
24 ax.legend()
25 ax.grid()

```

Listing 7.6 Defining the density values on the Earth's interior.

The acceleration $g(r)$, at a distance r from the Earth center can be estimated using the following integral:

$$g(r) = \frac{4\pi G}{r^2} \int_{r_1=0}^r \rho(r_1) r_1^2 dr_1 \quad (7.19)$$

where G is the "universal gravitational constant" with an accepted value of $6.67408 \pm 0.0031 \cdot 10^{-11} m^3 Kg^{-1} s^{-2}$.


```

1 def gravity(r):
2
3     g = np.zeros(len(r))
4
5     Gr = 6.67408e-11
6     r = r * 1000 # from Km to m
7
8     for i in range(1,len(r)):
9
10        r1 = r[0:i]
11        ro1 = ro[0:i]
12        r2 = r1[i-1]
13
14        y = ro1*r1**2
15        y_int = trapz(y,r1)
16
17        g1 = ((4 * np.pi*Gr)/(r2**2)) * y_int
18        g[i] = g1
19
20    return g
21
22 g = gravity(r)
23
24 fig, ax = plt.subplots()
25 ax.plot(r,g)
26 ax.grid()
27 ax.set_ylabel('g [m/s^2]')
28 ax.set_xlabel('Distance from the Earth center r [km]')

```

Listing 7.7 Defining the acceleration due to gravity on the Earth's interior.

and, finally, $p(z)$

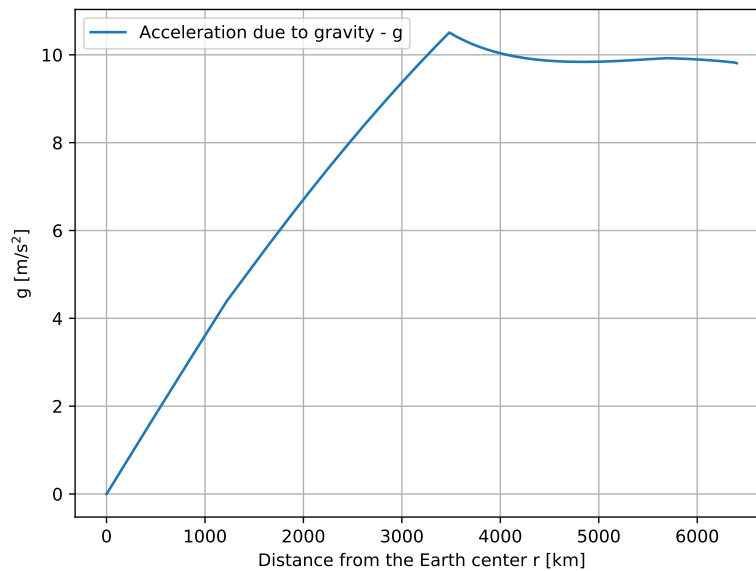


Fig. 7.11 The result of code listing 7.7.

```

1 p = np.zeros(len(r))
2
3 def pressure(r,ro,g):
4
5     r = r *1000
6
7     for i in range(0,len(r)):
8         r1 = r[i:len(r)]
9         ro1 = ro[i:len(r)]
10        g1 = g[i:len(r)]
11        y = ro1*g1
12        p1 = trapz(y,r1)
13        p[i] = p1
14    return p
15
16 p = pressure(r,ro,g)/1e9 # expressed in GPa
17
18 z = np.linspace(6400, 1, 6400)
19
20 fig, ax = plt.subplots()
21 ax.plot(z,p)
22 ax.grid()
23 ax.set_ylabel('P [GPa]')
24 ax.set_xlabel('Depth z from the Earth Surface [km]')

```

Listing 7.8 Computing the pressure on the Earth's interior.

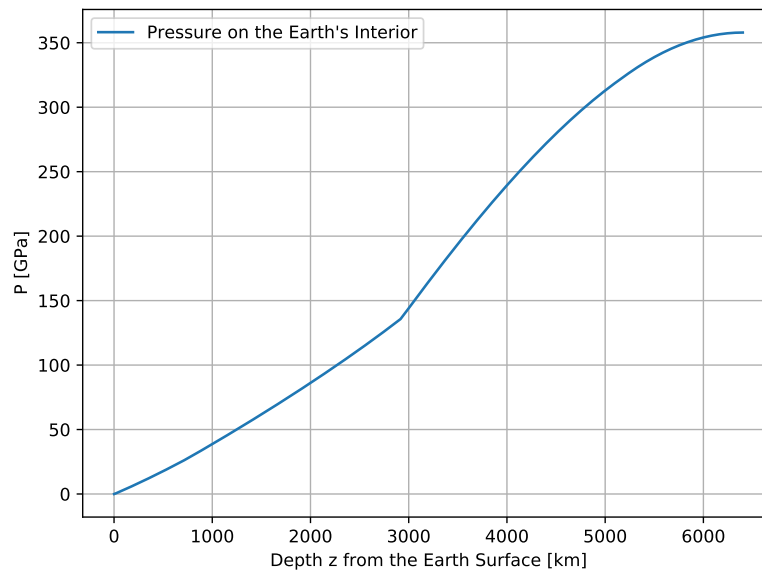


Fig. 7.12 The result of code listing 7.8.

Chapter 8

Differential Equations

8.1 Introduction

Differential equations govern many problems in physics, engineering, biology, and Earth Sciences.

Definition: A differential equation is an equation that relates one or more functions and their derivatives. (Zill, 2012).

Qualitatively, they describe the rate at which one variable is changing with respect to another. Examples are the rate of change in the number of atoms of a radioactive material over time and the change in temperature of magmas during cooling (Burd, 2019). An equation is called ordinary differential equation (ODE) if it contains ordinary derivatives only (Zill, 2012). In other words, an ode depends on one independent variable only. To make the concept clearer, in ordinary differential equations you will see one independent variable (e.g., t), one dependent variable (e.g., $y = N(t)$), and its derivatives (e.g., dN/dt). Everything else apart from the independent variable, the dependent variable, and its derivatives will be constants (King et al., 2003). The law of radioactive decay (i.e., the amount of radioactive material changes in time) is an example of ODE:

$$\frac{dN}{dt} = -\lambda \cdot N(t) \quad (8.1)$$

where N and λ are the number of radioactive nuclei and the probability of decay per nucleus per unit of time, respectively.

On the contrary, a partial differential equation (PDE) also contain partial derivatives. To better explain the concept, a PDE is a differential equation in which the dependent variable depends on two or more independent variables (King et al., 2003). The Fick's second law of diffusion is an example of PDE:

$$\frac{\partial C}{\partial t} = D \frac{\partial^2 C}{\partial x^2} \quad (8.2)$$

where C and D are the concentration of the investigated chemical element and a positive constant named 'diffusion coefficient', respectively.

To solve a differential equation, we need finding an expression for the dependent variable (e.g., $N(t)$ in 8.1) in terms of the independent ones (e.g., t in 8.1), which satisfies the investigated relation. By definition, a differential equation contains derivatives, so the achievement of a solution requires an integration. The general solution of an ODE is the one achieved by solving a differential equation in the absence of any initial conditions. It is a combination of functions and one or more constants. A Particular Solution of an ODE is the one obtained from the general solution by assigning specific values to the arbitrary constants. ODEs and PDEs can be solved using both analytical and numerical methods. In the present chapter, I will focus on the numerical solutions of differential equations using Python, but taking in mind that analytical solutions should be always explored, when possible (Burd, 2019).

8.2 Ordinary Differential Equation

As stated in the previous paragraph, ODEs contains ordinary derivatives only. The order of an ODE is the order of the highest derivative that appears in the equation. The explicit form of a n-order ode can be written as (Agarwal & O'Regan, 2008):

$$\frac{d^n y}{dx^n} = y^{(n)} = f(x, y, y', y'', \dots, y^{(n-1)}) \quad (8.3)$$

where f is a known function.

An ODE is linear if the unknown function appears linearly in the equation, otherwise it is nonlinear (Agarwal & O'Regan, 2008; Burd, 2019; King et al., 2003; Zill, 2012).

Direction fields of first order ODEs

Direction fields provide an overview of first order ODE solutions without actually solving the equation. Recall that first order ODEs are the ones that can be written in the following form (Agarwal & O'Regan, 2008):

$$y' = f(x, y) \quad (8.4)$$

A direction field is a set of small line segments passing through various, typically grid shaped, points having a slope that satisfy the investigated differential equation at that point.

In my knowledge, there is not a straightforward method in Python to plot a simple direction fields. However, we can easily develop a function for the scope (code listing

8.1). Figure 8.1 highlights the application of the code listing 8.1 to the following ODE:

$$y' = \frac{x^2}{1 - x^2 - y^2} \quad (8.5)$$

```

1 import numpy as np
2 from matplotlib import pyplot as plt
3
4 # Direction Field
5 def DirectionField(x_min, x_max, y_min, y_max, n_step, lenght,
6                   fun, ax):
7     # this is to avoid RuntimeError: divide by zero
8     np.seterr(divide='ignore', invalid='ignore')
9
10    x = np.linspace(x_min, x_max, n_step)
11    y = np.linspace(y_min, y_max, n_step)
12    X, Y = np.meshgrid(x, y)
13    slope = fun(X,Y)
14    slope = np.where(slope == np.inf, 10**3, slope)
15    slope = np.where(slope == -np.inf, -10**3, slope)
16    delta = lenght * np.cos(np.arctan(slope))
17    X1 = X - delta
18    X2 = X + delta
19    Y1 = slope*(X1-X)+Y
20    Y2 = slope*(X2-X)+Y
21    ax.plot([X1.ravel(), X2.ravel()], [Y1.ravel(), Y2.ravel()],
22           'k-', linewidth=1)
23
24 # Differential equations
25 def myODE(x,y):
26     dy_dx = x**2 / (1 - x**2 - y**2)
27     return dy_dx
28
29 # Make the plot
30 fig, ax1 = plt.subplots()
31 DirectionField(x_min = -2, x_max= 2, y_min= -2, y_max= 2, n_step=
32               30, lenght=0.05, fun=myODE, ax=ax1)
33 ax1.set_xlabel('x')
34 ax1.set_ylabel('y')
35 ax1.axis('square')
36 ax1.set_title(r"$ {y}' = - \frac{x^2}{1 - x^2 - y^2} $" )

```

Listing 8.1 Defining a function to develop a Direction Field.

At this point, you should already know the meaning of most instructions in the code listing 8.1. Exceptions are the statement at lines 8 and 12. In detail, the command line 8 simply avoid to display a warning when divisions by 0 occurs during the evaluation of the function at line 13. When a division by 0 occur, the returned value

could be inf , $-\text{inf}$, or NaN (i.e., not a number). In the first two cases, the slope is then 'adjusted' to a 'large' number at lines 14 and 15 (i.e. 1000 and -1000, respectively) to plot a vertical segment in the direction field. In the case of NaN , nothing is plotted at the corresponding node of the grid. At line 12, the command `np.meshgrid(x,y)` returns two coordinate matrices from two coordinate vectors. In detail, it creates two rectangular arrays: one of x values and one of y values. Combining the resulting matrices, we obtain a rectangular grid of coordinates. This approach is particularly useful when dealing with spatial data.

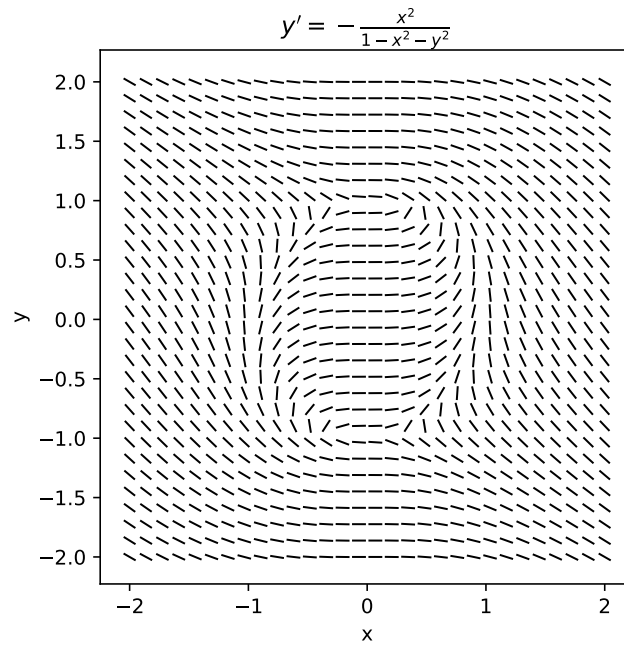


Fig. 8.1 Result of the code listing 8.1.

The `quiver()` method of the matplotlib provides an alternative method to investigate the behaviour of first order ODEs. In detail, it uses the formulation reported at the Eq. 8.6 :

$$\begin{aligned} \frac{dx}{dt} &= B(x, y) \\ \frac{dy}{dt} &= A(x, y) \end{aligned} \tag{8.6}$$

The advantage of Eq. 8.6 relies in the fact that the `quiver()` function can be used directly to displays velocity vectors in the $[x,y]$ space. Similarly, the `streamplot()`

function visualizes ODE solutions as streamlines. As an example, the code listing 8.2 implements the direction fields and the streamlines of the velocity field of the Eq. 8.7. Fig. 8.2 shows the results of the *quiver()* and *streamplot()* functions on the left and right panels, respectively.

$$\frac{dx}{dt} = x + 2y, \frac{dy}{dt} = -2x \quad (8.7)$$

The *quiver()* and *streamplot()* functions can be also used to investigate first order ODEs in the canonical form $y' = f(x, y)$. This is because $A(x, y)$ and $B(x, y)$ simply derive from the transformation reported in Eq 8.8.

$$\frac{dy}{dx} = \frac{A(x, y)}{B(x, y)} \quad (8.8)$$

As a consequence, if we assume that $A(x, y) = f(x, y)$ and $B(x, y) = 1$, the Eq. 8.8 reduces to the form $y' = f(x, y)$. As an example, the code listing 8.3 shows the application of the *quiver()* and *streamplot()* functions to the Eq. 8.9:

$$\frac{dy}{dx} = -y - 2 \cdot x^2 \quad (8.9)$$

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(-3, 3, 10)
5 y = x
6 X, Y = np.meshgrid(x, y)
7
8 dx_dt = X + 2*Y
9 dy_dt = - 2*X
10
11 fig = plt.figure()
12 ax1 = fig.add_subplot(1, 2, 1)
13 ax1.quiver(X, Y, dx_dt, dy_dt)
14 ax1.set_title('using quiver()')
15 ax1.set_xlabel('x')
16 ax1.set_ylabel('y')
17 ax1.axis('square')
18
19 ax2 = fig.add_subplot(1, 2, 2)
20 ax2.streamplot(X, Y, dx_dt, dy_dt)
21 ax2.set_title('using streamplot()')
22 ax2.set_xlabel('x')
23 ax2.set_ylabel('y')
24 ax2.axis('square')
25
26 fig.tight_layout()

```

Listing 8.2 Working with the *quiver()* and *streamplot()* methods in first order ODEs.

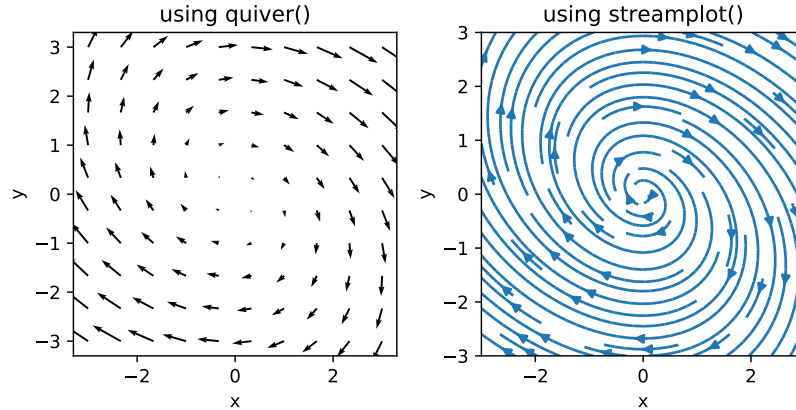


Fig. 8.2 Result of the code listing 8.2.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.linspace(-1, 1, 10)
5 y = x
6
7 X, Y = np.meshgrid(x, y)
8
9 dx_dt = np.ones_like(X)
10 dy_dt = - Y - 2 * X**2
11
12 # Making plot
13 fig = plt.figure()
14 ax1 = fig.add_subplot(1, 2, 1)
15 ax1.quiver(X, Y, dx_dt, dy_dt)
16 ax1.set_title('using quiver()')
17 ax1.set_xlabel('x')
18 ax1.set_ylabel('y')
19 ax1.axis('square')
20
21 ax2 = fig.add_subplot(1, 2, 2)
22 ax2.streamplot(X, Y, dx_dt, dy_dt, linewidth=0.5)
23 ax2.set_title('using streamplot()')
24 ax2.set_xlabel('x')
25 ax2.set_ylabel('y')
26 ax2.axis('square')
27
28 fig.tight_layout()

```

Listing 8.3 Working with the *quiver()* and *streamplot()* functions.

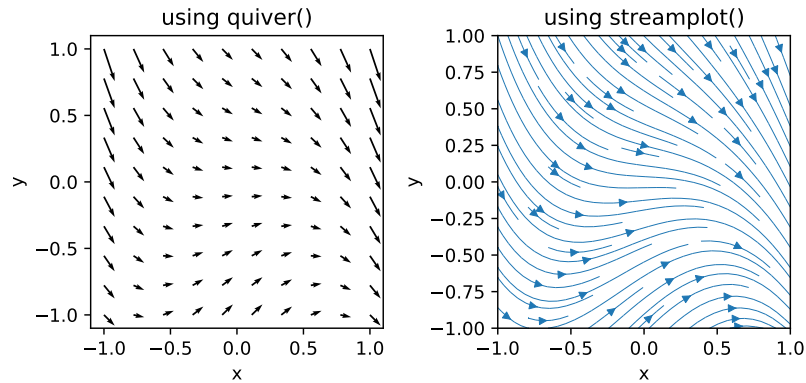


Fig. 8.3 Result of the code listing 8.3.

8.3 Numerical Solutions of First Order ODEs

The equation of radioactive decay (8.1) has an analytical solution with the form:

$$N(t) = N_0 \cdot e^{-\lambda t} = N_0 \cdot e^{-\frac{t}{\tau}} \quad (8.10)$$

where $N(t)$, N_0 , λ , and τ are the quantity N at time t , the quantity N at time $t = 0$, the exponential decay constant, and the mean lifetime, respectively. The radioactive decay represents a suitable example to illustrate some of the numerical techniques utilized for the solution of both ordinary and partial differential equations.

The Euler's method

The Euler's Method consists of a finite difference approximation to numerically solve differential equations by taking small finite steps Δt in the parameter t and approximating the function $N(t)$ with the first two terms in its Taylor expansion:

$$\frac{dN}{dt} \approx \frac{N(t + \Delta t) - N(t)}{\Delta t} \quad (8.11)$$

resulting in:

$$N(t + \Delta t) \approx \frac{dN}{dt} \cdot \Delta t + N(t) = -\lambda \cdot N(t) \cdot \Delta t + N(t) = N(t) \cdot (1 - \lambda \cdot \Delta t) \quad (8.12)$$

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3
4 # Euler Method
5 def EulerMethod(N0, decay_const, t_final, n_t_steps):
6     iterations = n_t_steps
7     delta_t = t_final/n_t_steps
8     t1 = np.linspace(0,iterations*delta_t, iterations)
9     N1 = np.zeros(t1.shape,np.float)
10    N1[0]=N0
11    for i in range(0,len(t1)-1):
12        N1[i+1] = N1[i] *(1 - decay_const * delta_t )
13    N1r=N1/N0
14    return N1,N1r,t1
15
16 Ne, Ner, te = EulerMethod(N0=10000, decay_const=1.54e-1,
17    t_final=20, n_t_steps=10)
18 #Analytical solution ...in the same points of the Euler method
19 def AnalyticalSolution(N0, decay_const, t_final, n_t_steps):
20
21     intermediate_points = n_t_steps
22     delta_t = t_final/n_t_steps
23     t2 = np.linspace(0,intermediate_points*delta_t,
24     intermediate_points)
25     N2 = N0 * np.exp(- decay_const * t2 )
26     N2r = N2/N0
27     return N2,N2r,t2
28 Na, Nar, ta = AnalyticalSolution(N0=10000, decay_const=1.54e
29    -1, t_final=20, n_t_steps=10)
30 EulerRelError = 100*(Ne-Na)/Na
31
32 fig = plt.figure()
33 ax1 = fig.add_subplot(1, 2, 1)
34 ax1.plot(te,Ner, linestyle="--", linewidth=2, label='Euler
35    method')
36 ax1.plot(ta,Nar, linestyle="--", linewidth=2, label='
37    Analytical Solution')
38 ax1.set_ylabel('Relative Number of  $^{238}\text{U}$  atoms')
39 ax1.set_xlabel('time in bilion years')
40 ax1.legend()
41
42 ax2 = fig.add_subplot(1, 2, 2)
43 ax2.plot(te,EulerRelError, linestyle="--", linewidth=2, label='
44    Deviation formthe \nextpected value')
45 ax2.set_ylabel('Relative Error, in %')
46 ax2.set_xlabel('time in bilion years')
47 ax2.legend()

```

Listing 8.4 Implementing the Euler's method in Python.

Now, assuming a decay constant (λ) of $1.54 \cdot 10^{-1}$ per billion-year ($1.54 \cdot 10^{-10}$ per year) as in the case of the uranium series from ^{238}U to ^{206}Pb , we can define a Python script to solve the Eq. 8.1 (code listing 8.4), and compare the results provided by the analytical and numerical solutions (Fig. 8.4). The deviation of the Euler's method (i.e., the error) from the expected value is a function of Δt .

Also, Euler's method is affected by an intrinsic issue, potentially leading to a progressive amplification of the error since it evaluates the derivatives at the beginning of the investigated interval (e.g., Δt) only. If the derivative at the beginning of the step is systematically incorrect, either too high or too low, then the numerical solution will be similarly systematically incorrect (Fig. 8.4).

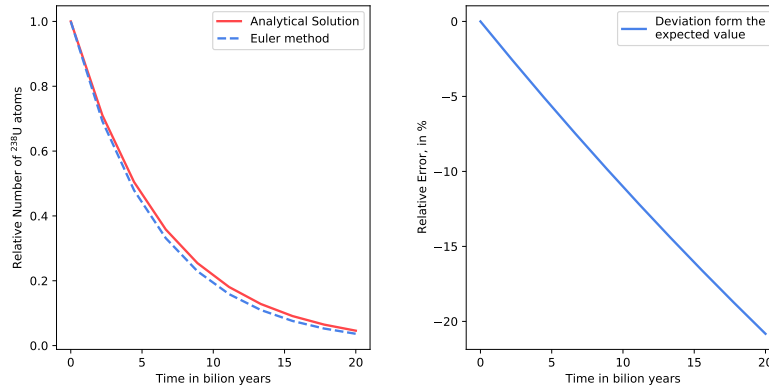


Fig. 8.4 Result of the code listing 8.4.

Note that, in this specific case, we leave the Δt value quite large to highlight the limits of the Euler's method. A reduction of Δt will improve the accuracy, significantly. However, as a general case, to improve the accuracy we need to estimate the derivative function at more than one point in the investigated step interval.

The `scipy.integrate.ode` class

The `scipy.integrate.ode` is generic interface class to numeric solution of ode. Available integrators are: Real-valued Variable-coefficient Ordinary Differential Equation solver (i.e., `vode`), Complex-valued Variable-coefficient Ordinary Differential Equation solver with fixed-leading-coefficient implementation (i.e., `zvode`), Real-valued Variable-coefficient Ordinary Differential Equation solver with fixed-leading-coefficient implementation (i.e., `lsoda`), an explicit Runge-Kutta method of order (4)5 (i.e., `dopri5`), and an explicit runge-kutta method of order 8(5,3) (i.e., `dopri853`). Please refer to a more specialised books for a detailed description of these methods (Atkinson et al., 2009; Griffiths & Higham, 2010; Li et al., 2017).

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 from scipy.integrate import ode
4
5 # using scipy.integrate.ode
6 def ode_sol(N0, decay_const, t_final, n_t_steps):
7     intermediate_points = n_t_steps
8     t3 = np.linspace(0,t_final, intermediate_points)
9     N3 = np.zeros(t3.shape,np.float)
10    def f(t, y, decay_const):
11        return - decay_const * y
12    solver = ode(f).set_integrator('dopri5') # runge-kutta of
13        order (4)5
14    y0 = N0
15    t0 = 0
16    solver.set_initial_value(y0, t0)
17    solver.set_f_params(decay_const)
18    k=1
19    N3[0] = N0
20    while solver.successful() and solver.t < t_final:
21        N3[k] = solver.integrate(t3[k])[0]
22        k += 1 # k = k + 1
23    N3r = N3 / N0
24    return N3,N3r,t3
25
26 # Analytical solution
27 Na, Nar, ta = AnalyticalSolution(N0=10000, decay_const=1.54e-1,
28     t_final=20, n_t_steps=10)
29
30 # Euler method
31 Ne, Ner, te = EulerMethod(N0=10000, decay_const=1.54e-1, t_final
32     =20, n_t_steps=10)
33 EulerRelError = 100*(Ne-Na)/Na
34
35 # runge-kutta of order (4)5
36 Node, Noder, tode = ode_sol(N0=10000, decay_const=1.54e-1,
37     t_final=20, n_t_steps=10)
38 odeRelError = 100*(Node-Na)/Na
39
40 # Make the plot
41 fig = plt.figure(figsize=(8,5))
42 ax1 = fig.add_subplot(1, 2, 1)
43 ax1.plot(ta,Nar, linestyle="-", linewidth=2, label='Analytical
44     Solution', c='#ff464a')
45 ax1.plot(te,Ner, linestyle="--", linewidth=2, label='Euler method
46     ', c='#4881e9')
47 ax1.plot(tode,Noder, linestyle="--", linewidth=2, label='Runge-
48     Kutta of order (4)5', c='#342a77')
49 ax1.set_ylabel('Relative Number of  $^{238}\text{U}$  atoms')
50 ax1.set_xlabel('Time in bilion years')
51 ax1.legend()
52
53
54 ax2 = fig.add_subplot(1, 2, 2)
55 ax2.plot(te,EulerRelError, linestyle="-", linewidth=2, c='#4881e9
56     ', label='Euler method')

```

```

46 ax2.plot(tode,odeRelError, linestyle="-", linewidth=2, c='#342a77
    ', label='Runge-Kutta of order (4)5')
47 ax2.set_ylabel('Relative Error, in %')
48 ax2.set_xlabel('Time in bilion years')
49 ax2.legend()
50
51 fig.tight_layout()

```

Listing 8.5 Euler's method vs. Runge-Kutta of order (4)5.

Here I will show how to apply the `scipy.integrate.ode` class to the real study case of radioactive decay. In detail the code listing 8.5 concerns the application of the explicit Runge-Kutta method of order (4)5 to the investigated equations comparing the obtained results with the ones resulting from the Euler's method (8.5).

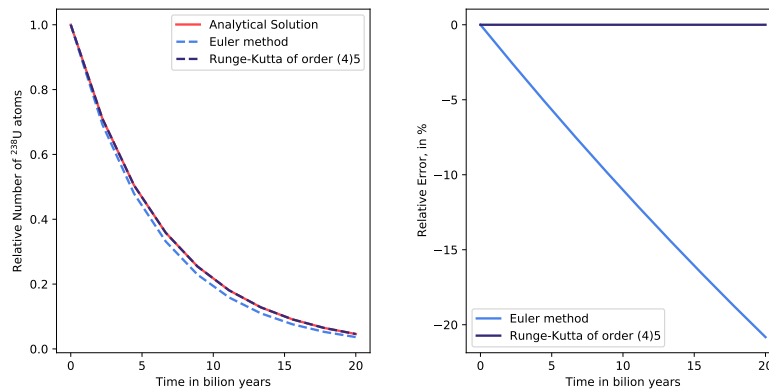


Fig. 8.5 Result of the code listing 8.5.

8.4 The Fick's law of diffusion, a Widely Used PDE

As originally reported by Fick (1855), the rate of transfer of diffusing substance through unit area of a section is proportional to the concentration gradient measured normal to the section, i.e.,

$$F = -D \frac{\partial C}{\partial x} \quad (8.13)$$

where F is the rate of transfer per unit area of section, C the concentration of the diffusing substance, x the space coordinate measured normal to the section, and D is called the diffusion coefficient. In some cases, e.g. diffusion in dilute solutions, D can reasonably be considered as a constant, whereas in others, e.g. diffusion in high

polymers, it depends very markedly on concentration (Crank, 1975). Equation 8.13 is universally known as the first Fick's law or first law of diffusion (Crank, 1975). The unit of C , F , x , and D is concentration (e.g., gram, gram moles or wt.%,), a concentration divided by time (e.g., g/s , note that the concentration must have the same unit of C), a length (e.g., meters), and a squared length divided by a time unit (e.g., m^2/s) respectively. In the case of one-dimensional processes characterized by a constant D , the equation 8.13 can be written as follow (i.e., one-dimensional second Fick's law or second law of diffusion; Crank, 1975):

$$\frac{\partial C}{\partial t} = D \frac{\partial^2 C}{\partial x^2} \quad (8.14)$$

For constant D and specific geometries, equation 8.2 can be solved analytically Crank. In all the other cases the problem requires a numerical solution.

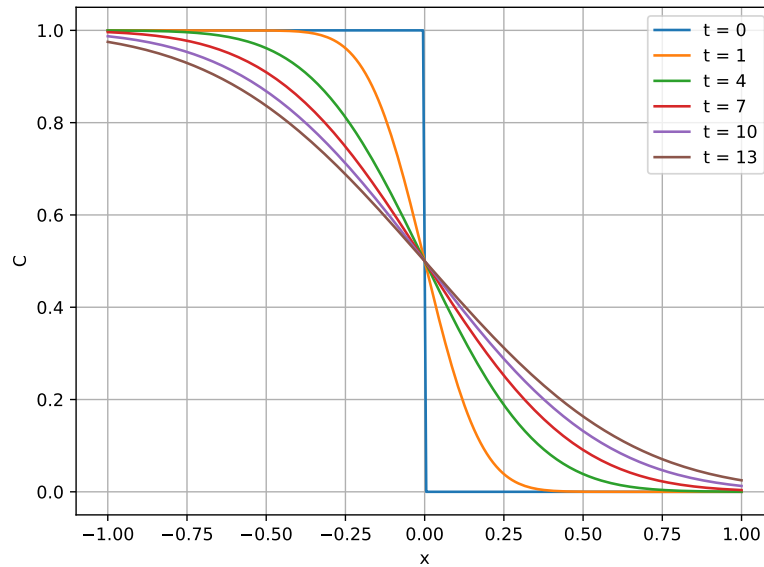


Fig. 8.6 Result of the code listing 8.6.

Analytical solutions

Analytical solutions of the diffusion equation can be obtained for a variety of initial and boundary conditions provided the diffusion coefficient is constant (Crank, 1975). As an example, the solution for a diffusing substance initially confined in a finite region

$$C = C_0, x < 0, C = 0, x > 0, t = 0 \quad (8.15)$$

can be written written in the form (Crank, 1975):

$$C(x, t) = 0.5C_0 \operatorname{erfc} \frac{x}{2\sqrt{Dt}} \quad (8.16)$$

where $\operatorname{erfc}()$ is the complementary error function defined as $1 - \operatorname{erf}()$:

$$\operatorname{erfc}(x) = 1 - \operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_x^{\infty} e^{-t^2} dt \quad (8.17)$$

I will provide you more details about the error function at section 9.2

The code listing 8.6 implements the equation 8.16 in Python.

```

1 import numpy as np
2 from scipy import special
3 import matplotlib.pyplot as plt
4
5 def planeDiff_1D(t, D, x0=0, xmin=-1, xmax=1, Cleft=1, Cright=0,
6   num_points=200):
7     N = num_points
8     x=np.linspace(xmin, xmax, N)
9     deltaC = Cleft - Cright
10
11     C0 = np.piecewise(x, [x < x0, x >= x0], [Cleft, Cright])
12     C = 0.5 * deltaC * (special.erfc((x - x0)/(2 * np.sqrt(D * t
13   )))
14     return x,C,C0
15
16 D = 0.01 # Diffusion coefficient
17
18 fig, ax = plt.subplots()
19
20 for t in range(1, 14, 3):
21
22     x,C,C0 = planeDiff_1D(t=t, D=D)
23     if t==1:
24         leg = "t = " + str(t)
25         plt.plot(x,C0, label="t = 0")
26     leg = "t = " + str(t)
27     ax.plot(x, C, label=leg)
28
29 ax.grid()
30 ax.set_xlabel('x')
31 ax.set_ylabel('C')
32 ax.legend()

```

Listing 8.6 Analytical solution of plane diffusion.

Numerical solution for constant D

The simplest way to discretize the equation 8.2 is by finite difference (Linge & Langtangen, 2017):

$$\frac{C_j^{n+1} - C_j^n}{\Delta t} = D \left[\frac{C_{j+1}^n - 2C_j^n + C_{j-1}^n}{(\Delta x)^2} \right] \quad (8.18)$$

where n and j represent the time and space domain, respectively. This scheme, called FTCS (Forward-Time Central-Space), uses the Euler's method and a central difference scheme to approximate the derivatives in time and in space, respectively. For more details about the theory behind numerical schemes for the solution of partial differential equation please refer to more specialized books (Linge & Langtangen, 2017; Mazumder, 2015; Morton & Mayers, 2005). In Python, Eq. 8.18 can be easily implemented using the code listing 8.7. The finite difference scheme reported in equation 8.18 is stable under the following conditions:

$$\frac{2D\Delta t}{(\Delta x)^2} \leq 1 \quad (8.19)$$

Figure 8.7 compares the results of the analytical and the numerical solutions.

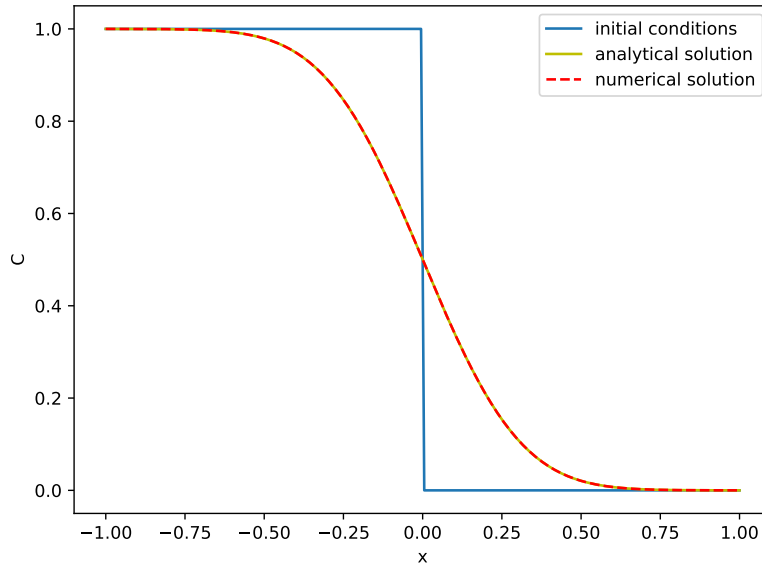


Fig. 8.7 Result of the code listing 8.7.

```

1 def FTCS(u, D, h, dt):
2
3     d2u_dx2 = np.zeros(u.shape,np.float)
4     for i in range(1,len(u)-1):
5         # Central difference scheme in space
6         d2u_dx2[i] = (u[i+1] - 2*u[i] + u[i-1]) / h**2
7
8     # Neuman boundary conditions at i=0 and i=len(u)-1
9     i=0
10    d2u_dx2[i] = (u[i+1] - 2 * u[i] + u[i]) / h**2
11    i=len(u)-1
12    d2u_dx2[i] = (u[i] - 2 * u[i] + u[i-1]) / h**2
13
14    # Euler method for the time domain
15    u1 = u + dt * D * d2u_dx2
16    return u1
17
18 dt = 0.001 #step size of time
19 tf = 3
20
21 def dcomputeDconst(u, D, h, dt, tf):
22
23     nsteps=tf/dt
24     u1 = u
25     for i in range(int(nsteps)):
26         u1 = FTCS(u1, D, h, dt)
27     return u1
28
29 x, C, C0 = planeDiff_1D(t=tf, D=D)
30
31 h=x[1] - x[0] #step size of the 1D space
32 u = C0 # intial conditions
33 C1 = dcomputeDconst(u, D, h, dt, tf)
34
35 fig, ax = plt.subplots()
36 ax.plot(x,C0, label='initial conditions')
37 ax.plot(x,C,'y', label='analytical solution')
38 ax.plot(x,C1,'r--', label='numerical solution')
39 ax.set_xlabel('x')
40 ax.set_ylabel('C')
41 ax.legend()

```

Listing 8.7 Plane Diffusion by finite difference method.

The implementation of the FTCS scheme (i.e., lines 1-16 of the code listing 8.7 could be improved exploiting the potentialities of NumPy (Linge & Langtangen, 2017). In particular, the loop at lines 4-6 could be replaced by a single line of code in the vectorial notation (line 4 of code listing 8.8; Linge and Langtangen, 2017).

```

1 def numpyFTCS(u, D, h, dt):
2
3     d2u_dx2 = np.zeros(u.shape, np.float)
4     d2u_dx2[1:-1] = (u[2:] - 2 * u[1:-1] + u[:-2]) / h ** 2
5
6     # Neuman boundary conditions at i=0 and i=len(u)-1
7     i=0
8     d2u_dx2[i] = (u[i+1] - 2 * u[i] + u[i]) / h ** 2
9     i=len(u)-1
10    d2u_dx2[i] = (u[i] - 2 * u[i] + u[i-1]) / h ** 2
11
12    # Euler method for the time domain
13    u1 = u + dt * D * d2u_dx2
14    return u1

```

Listing 8.8 Using the vectorial notation for the FTCS scheme.

In petrology and volcanology, the process of chemical diffusion is often used to constrain the residence times of crystals in a volcanic plumbing system before an eruption (Costa et al., 2020). As an example, Costa et al., 2003 report a formulation to model diffusion of Mg in plagioclase, also accounting for the influence of anorthite absolute values and gradients on chemical potentials and diffusion coefficients (Costa et al., 2020). The rationale behind the formulation provided by (Costa et al., 2020) bases on the evidence that diffusive fluxes of trace elements may be strongly coupled to major element concentration gradients (i.e., multi-component diffusion).

In the following, I provide a Python implementation for the problem reported in (Costa et al., 2020). It consists of: 1) making the analytical determinations of Mg and An content on zoned plagioclases; 2) constraining the boundary conditions (e.g., equilibrium at the rims); 3) Defining the initial and equilibrium profiles; 4) estimating the diffusion coefficient for Mg; 5) solving the time-dependent form of the diffusion equation by finite differences. The analytical determinations for An are typically performed by Electron Probe Mycro-Analyses (EPMA). Magnesium can be determined either by EPMA or Laser Ablation Inductively Coupled Plasma Mass Spectrometry (LA-ICP-MS).

As an example, Fig. 8.8, reports a rim to rim MgO profile, analyzed by EPMA, on the plagioclase labelled 4202-1 P11 by (Moore et al., 2014).

Following the approach proposed by (Costa et al., 2003), the dependence of Mg trace element partitioning between plagioclase and melt on the anorthite content is approximated by Eq. 8.20 (Costa et al., 2003):

$$R \cdot T \cdot \ln \frac{C_{Mg}^{Pl}}{C_{Mg}^l} = AX_{AN} + B \quad (8.20)$$

where X_{AN} , C_{Mg}^{Pl} , C_{Mg}^l , T , and R are the anorthite molar fraction, the concentration of Mg in plagioclase, the concentration of Mg in the liquid, the temperature,

and the gas constant, respectively (Costa et al., 2003). For the A and B parameters, (Moore et al., 2014) proposed values equal to -21882 and -26352, respectively.

To model the diffusive process, the initial and the equilibrium profiles have been estimated starting from the Eq. 8.20.

In detail, the initial profile is defined by the melt concentration in equilibrium with the crystal core (Moore et al., 2014). Both the initial and equilibrium profiles are calculated using Eq. 8.20, i.e., MgO equal to 7.8 wt % and 8.4 wt %, respectively. As boundary conditions, crystal rims that are in contact with the surrounding melt are open, i.e., Mg values at the rims are those of the equilibrium profile. I used the formulation for the diffusion coefficient reported by (Costa et al., 2003) (Eq. 8.21):

$$D_{Mg} = \left[2.92 \cdot 10^{-4.1 \cdot X_{AN} - 3.1} \exp\left(\frac{-266000}{R \cdot T}\right) \right] \cdot 10^{12} \quad (8.21)$$

The time-dependent form of the diffusion equation for Mg in a plagioclase developed by (Costa et al., 2003) is reported in Eq. 8.22:

$$\begin{aligned} \frac{\partial C_{Mg}}{\partial t} = & \left(D_{Mg} \frac{\partial^2 C_{Mg}}{\partial x^2} + \frac{\partial C_{Mg}}{\partial x} \frac{\partial D_{Mg}}{\partial x} \right) + \\ & - \frac{A}{RT} \left(D_{Mg} \frac{\partial C_{Mg}}{\partial x} \frac{\partial X_{AN}}{\partial x} + C_{Mg} \frac{\partial D_{Mg}}{\partial x} \frac{\partial X_{AN}}{\partial x} + D_{Mg} C_{Mg} \frac{\partial^2 X_{AN}}{\partial x^2} \right) \end{aligned} \quad (8.22)$$

The code listing 8.9 reports the finite difference approximation of the Eq. 8.22. In accordance with (Moore et al., 2014), in the code listing 8.9 I used the FTCS scheme reported in Eq. 8.18. For first order derivatives, the explicit central scheme in space is

$$\frac{\partial C}{\partial x} \approx \frac{C_{j+1}^n - C_{j-1}^n}{2 \cdot \Delta x} \quad (8.23)$$

The notation is the same reported in the Eq. 8.7.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import pandas as pd
4
5 # Model parameters
6 T = 1200.0 # Temperature in Celsius
7 dx = 4.12 # average distance in micron among the analyses
8 dt = 0.9 * 1e4
9 RT = 8.3144 * (T + 273.15)
10 R = dt / dx ** 2
11
12 # Initial Conditions
13 Mydataset = pd.read_excel('Moore_PhD.xlsx')
14 my_Distance = Mydataset.Distance.values
15 mgc = Mydataset.MgO.values
16 An = Mydataset.An_mol_percent.values

```

```

17 AN = An / 100
18 AN_unsmoothed = AN
19 AN_smoothed = np.full(len(AN),0.)
20
21 # Smoothing the AN profile to avoid numerical artifacts
22 D_smoot = np.full(len(AN),0.0005)
23 for i in range(2):
24     AN_smoothed[1:-1] = AN_unsmoothed[1:-1] + R * D_smoot[1:-1]
25     * (AN_unsmoothed[2:] - 2 * AN_unsmoothed[1:-1] +
26     AN_unsmoothed[:-2])
27     AN_smoothed[0] = AN[0]
28     AN_smoothed[len(AN)-1] = AN[len(AN)-1]
29     AN_unsmoothed = AN_smoothed
30
31 D_Mg = 2.92 * 10**(-4.1 * AN_smoothed - 3.1)*np.exp(-266 * 1e3/RT
32     )*1e12 # Eq. 8 in Costa et al., 2003
33
34 fig, ax = plt.subplots(figsize=(7,5))
35
36 # Initial and Equilibrium Profiles
37 A = - 21882
38 B = - 26352
39 K = np.exp((A*AN_smoothed+B)/RT) # Eq. 8 in Moore et al., 2014
40 c_eq = 8.4 * K
41 c_init = 7.8 * K
42 ax.plot(my_Distance, c_eq, linewidth=2, color='#ff464a', label =
43     'Equilibrium Profile')
44 ax.plot(my_Distance, c_init,linewidth=2, color='#342a77', label
45     = 'Initial Profile')
46
47 # The numerical solution start here
48 colors = ['#4881e9', '#e99648', '#e9486e']
49 t_final_weeks = np.array([4,10,21])
50
51 for t_w, color in zip(t_final_weeks,colors):
52
53     C_Mg_new = np.full(len(c_eq),0.)
54     d2AN = np.full(len(c_eq),0.)
55     d2C_Mg = np.full(len(c_eq),0.)
56     dD_Mg = np.full(len(c_eq),0.)
57     dC_Mg = np.full(len(c_eq),0.)
58     dAN = np.full(len(c_eq),0.)
59
60     C_Mg = c_init
61     t_final = int(604800 * t_w/dt)
62     for i in range(t_final):
63         # boundary conditions: Rims are at equilibrium with melt
64         C_Mg_new[0] = c_eq[0]
65         C_Mg_new[len(c_eq)-1] = c_eq[len(c_eq)-1]
66
67         # Finite difference sol. of Eq. 7 in Costa et al., 2003
68         d2AN[1:-1] = (AN_smoothed[2:] - 2 * AN_smoothed[1:-1] +
69         AN_smoothed[:-2])
70         d2C_Mg[1:-1] = C_Mg[2:] - 2 * C_Mg[1:-1] + C_Mg[:-2]

```

```

65     dD_Mg[1:-1] = (D_Mg[2:]-D_Mg[:-2])/2
66     dC_Mg[1:-1] = (C_Mg[2:]-C_Mg[:-2])/2
67     dAN[1:-1] = (AN_smoothed[2:]-AN_smoothed[:-2])/2
68
69     C_Mg_new[1:-1] = C_Mg[1:-1] + R * ( (D_Mg[1:-1] * d2C_Mg
70     [1:-1] + dD_Mg[1:-1] * dC_Mg[1:-1]) - (A/RT) * (D_Mg[1:-1] *
71     dC_Mg[1:-1] * dAN[1:-1] + C_Mg[1:-1] * dD_Mg[1:-1] * dAN
72     [1:-1] + D_Mg[1:-1] * C_Mg[1:-1] * d2AN[1:-1]) )
73     C_Mg = C_Mg_new
74     ax.plot(my_Distance, C_Mg_new, linestyle='--', linewidth=1,
75     label= str(t_w) + ' weeks at 1200 Celsius deg.')
```

```

76 ax.scatter(my_Distance, mgc, marker='o', c='#c7ddf4', edgecolors=
77     'k', s=50, label='Analytical Deteminations', zorder=100,
78     alpha=0.7)
79 ax.set_ylim(0.19,0.27)
80 time_sec = t_final * dt
81 time_weeks = time_sec / 604800
82 ax.legend(title=r'\bf{4202\_1-PI1}$ (Moore et al., 2014)', ncol
83     =2, loc='lower center')
84 ax.set_xlabel(r'Distance [ $\mu$ m$]')
85 ax.set_ylabel('MgO [wt %]')
86 fig.tight_layout()
```

Listing 8.9 Implementation of the Eq. 8.22 in Python. Data are from Moore et al., 2014.

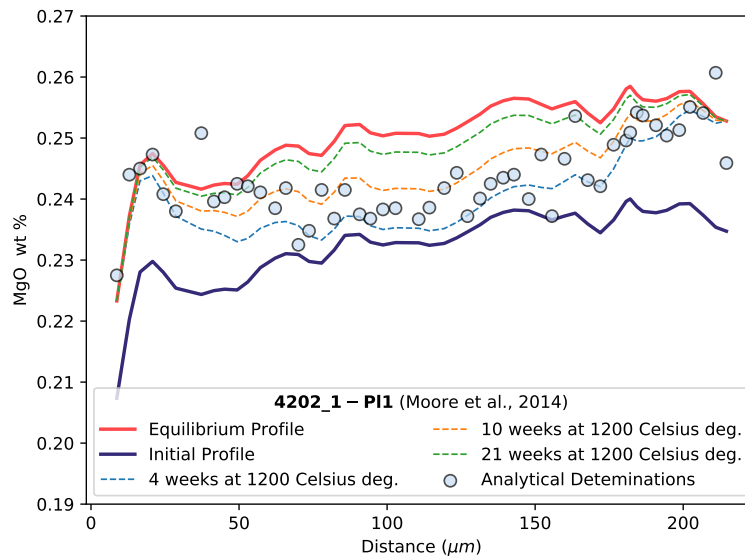


Fig. 8.8 The result of the code listing 8.9.

Part IV
Probability Density Functions and Error
Analysis

Chapter 9

Probability Density Functions and Their Use in Geology

9.1 Probability Distribution and Density Functions (PDF)

Everitt (2006) defines the probability distribution for discrete random variables as the mathematical formula that gives the probability of each value of the variables. For a continuous random variable, it is a function which is graphically described by a curve in the plane (x, y) . For a specific interval $[x_1, x_2]$, the area under the curve (i.e., the definite integral) provides the probability that the investigated variable falls within $[x_1, x_2]$ (Everitt, 2006). The term probability density, also refers to probability distributions (Everitt, 2006).

Definition: a probability density function (PDF) is a function associated with a continuous random variable whose value at any given point in the sample space (i.e., the set of possible values taken by the random variable) provides an estimation of the likelihood of occurrence for that specific the value. All the probability density functions share the following properties and indexes (Hughes & Hase, 2010):

- The PDF is normalized when: $\int_{-\infty}^{\infty} PDF(x)dx = 1$;
- the probability that x lies between the values $x_1 \leq x_2$, is: $P(x) = \int_{x_1}^{x_2} PDF(x)dx$
- mean: $\mu = \int_{-\infty}^{\infty} x \cdot PDF(x)dx$;
- median: $\int_{-\infty}^{Me} PDF(x)dx = \frac{1}{2}$;
- variance: $\sigma^2 = \int_{-\infty}^{\infty} (x - \mu)^2 \cdot PDF(x)dx$;
- skewness: $\mu_3 = \int_{-\infty}^{\infty} (x - \mu)^3 \cdot PDF(x)dx$;

The second point tells us that solving a definite integral in the interval $[x_1, x_2]$ of a variable x describing a geological process of known PDF, we define the probability for the occurrence of that process between x_1 and x_2 . As an example, knowing the PDF of age estimates for a sequence of eruptive events by Zircon dating, we can define the probability for the occurrence of an eruption between two specific ages. We will see a specific example later in the chapter. However, the PDF is rarely known *a priori*. Under specific conditions, our measurements could follow a known PDF. As an example, the different formulations of central limit theorems provide us the

circumstances under which the estimations of a sample mean converge to a normal distribution (Section 9.6)

9.2 The Normal Distribution

Normal probability density function

The normal distribution is a bell-shaped PDF that occurs naturally in many situations. As an example, it finds application when calibrating an analytical device, in error propagation (section 10), and, generally speaking, during the interpretation of a data set resulting from a field campaign, e.g., as consequence of the central limit theorem (section 9.6). The normal probability density function (PDF_N) is defined as follow:

$$PDF_N(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \quad (9.1)$$

where μ and σ are the mean and the standard deviation, respectively. The main characteristics of the normal distribution are:

- normal distributions are bell-shaped with points of inflection at $\mu \pm \sigma$;
- the mean, mode and median are all equal;
- the curve is symmetric at the center (i.e., around the mean, μ);
- all normal curves are non-negative for all x values;
- exactly half of the values are to the left of center and exactly half the values are to the right;
- the limit of $PDF_N(x, \mu, \sigma)$ as x goes to positive or negative infinity is 0;
- the height of any normal curve is maximized at $x = \mu$;
- the total area under the curve is 1;
- the shape of any normal curve depends on its mean μ and standard deviation σ (code listing 9.1 and Fig. 9.1);
- the standardized normal PDF has standard deviation equal to 1 and mean equal to 0.

The SciPy library provides the PDF for a normal, or gaussian, distribution with the function `scipy.stats.norm.pdf()`, but we can also define it using the `def` statement (i.e., our own function; code listing 9.1 and Fig. 9.1).

To get the probability of occurrence for a normal distributed x entity between x_1 and x_2 , we need to solve the definite integral (Eq 9.2).

$$P(x_1 \leq x \leq x_2) = \int_{x_1}^{x_2} PDF_N(x, \mu, \sigma) dx = \frac{1}{\sqrt{2\pi\sigma^2}} \int_{x_1}^{x_2} e^{-\frac{(x-\mu)^2}{2\sigma^2}} dx \quad (9.2)$$

Eq. 9.2 has no analytical solution, but due to its importance, mathematicians developed a specific function to solve it: the error function.

```

1 from scipy.stats import norm
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5
6 # I'm going to define my normal PDF...
7 def normal_pdf(x, mean, std):
8     return 1/(np.sqrt(2*np.pi*std**2))*np.exp(-0.5*((x - mean)
9         **2)/(std**2))
10
11 x = np.arange(-12, 12, .001)
12
13 pdf1 = normal_pdf(x, mean = 0, std = 2)
14
15 #the built-in norm PDF in scipy.stats
16 pdf2 = norm.pdf(x, loc= 0, scale = 2)
17
18 fig = plt.figure(figsize=(7,9))
19 ax1 = fig.add_subplot(3, 1, 1)
20 ax1.plot(x,pdf1, color = '#84b4e8', linestyle = "-", linewidth
21         =6, label="My normal PDF")
22 ax1.plot(x,pdf2, color = '#ff464a', linestyle = "--", label="
23         norm.pdf() in scipy.stats ")
24 ax1.set_xlabel("x")
25 ax1.set_ylabel("PDF(x)")
26 ax1.legend(title = r"Normal PDF with  $\mu=0$  and  $1\sigma=2$ ")
27
28 ax2 = fig.add_subplot(3, 1, 2)
29 for i in [1,2,3]:
30     y = normal_pdf(x,0,i)
31     ax2.plot(x, y, label=r" $\mu = 0$ ,  $1\sigma = " + str(i))$ ")
32 ax2.set_xlabel("x")
33 ax2.set_ylabel("PDF(x)")
34 ax2.legend()
35
36 ax3 = fig.add_subplot(3, 1, 3)
37 for i in [-3,0,3]:
38     y = normal_pdf(x,i,1)
39     ax3.plot(x, y, label=r" $\mu = " + str(i) + ", 1\sigma =$ 
40         = 1")
41 ax3.set_xlabel("x")
42 ax3.set_ylabel("PDF(x)")
43 ax3.legend()
44 fig.tight_layout()

```

Listing 9.1 The normal PDF.

By definition, the error function $erf(x)$ is equal to:

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt \quad (9.3)$$

As a consequence, the solution of the definite integral reported in Eq. 9.2, i.e., a normal PDF in the interval $[x_1, x_2]$ with $x_1 \leq x_2$, has the form reported in Eq. 9.4.

$$P(x_1 \leq x \leq x_2) = \frac{1}{\sqrt{2\pi\sigma^2}} \left[\operatorname{erf}\left(\frac{x_2 - \mu}{\sqrt{2\pi\sigma^2}}\right) - \operatorname{erf}\left(\frac{x_1 - \mu}{\sqrt{2\pi\sigma^2}}\right) \right] \quad (9.4)$$

Also, Eq. 9.2 can be solved numerically using the techniques we reported in Chapter 7. The code listing 9.2 reports the solution of the equation 9.2 using both the Eq. 9.4 and the numerical methods *trapz()* (Eq. 7.14) and *sims()* (Eq. 7.15) available in *scipy.integrate*.

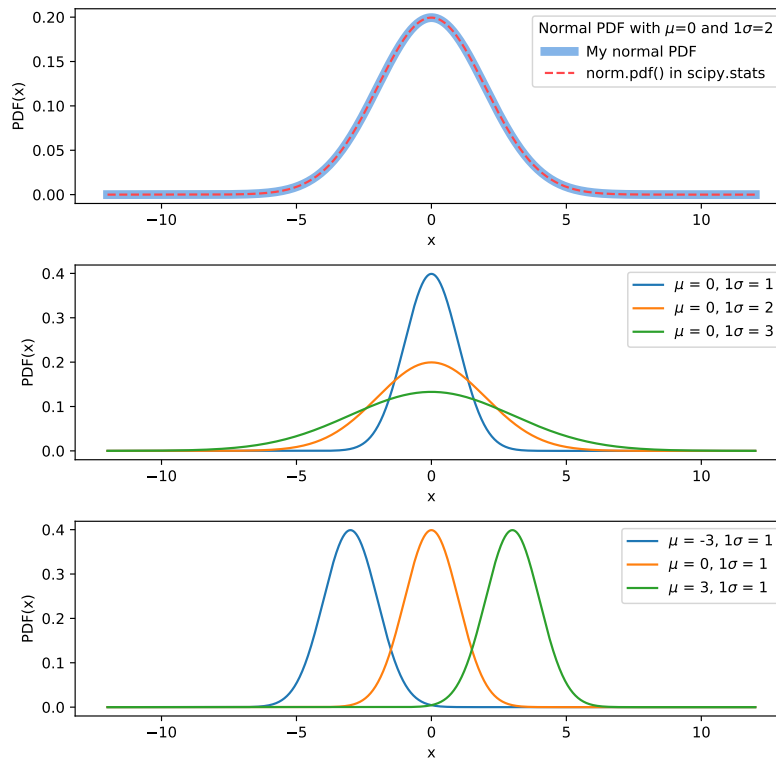


Fig. 9.1 The result of the code reported in the listing 9.1.

```

1 from scipy.stats import norm
2 import numpy as np
3 from scipy import special
4 from scipy import integrate
5
6 def integrate_normal(x1, x2, mu, sigma):
7     S = 0.5*((special.erf((x2-mu)/(sigma*np.sqrt(2))))-(
8         special.erf((x1-mu)/(sigma*np.sqrt(2))))))
9     return S
10
11 my_mu = 0
12 my_sigma = 1
13
14 my_x1 = 0
15 my_x2 = my_sigma
16
17 # The expected value is equal to 0.3413...
18 myS = integrate_normal(x1= my_x1, x2= my_x2, mu = my_mu, sigma
19     = my_sigma)
20
21 x = np.arange(my_x1, my_x2, 0.0001)
22 y = norm.pdf(x, loc = my_mu, scale = my_sigma) # normal_pdf(
23     x, mean = my_mu, std = my_sigma)
24
25 S_trapz = integrate.trapz(y,x)
26 S_simps = integrate.simps(y,x)
27
28 print("Solution Using erf: {:.9f}".format(myS))
29 print("Using the trapezoidal rule, trapz: {:.10f}".format(
30     S_trapz))
31 print("Using the composite Simpson rule, simps: {:.10f}".
32     format(S_simps))
33
34 '''
35 Output:
36 Solution Using erf: 0.341344746
37 Using the trapezoidal rule, trapz: 0.3413205476
38 Using the composite Simpson rule, simps: 0.3413205478
39 '''

```

Listing 9.2 Solving the Eq. 9.2 using the Eq. 9.4 and numerical methods.

Generating a normal sample distribution

The function `numpy.random.normal(loc=0.0, scale=1.0, size=None)` generates random samples from a normal distribution (code listing, 9.3). Generating a random sample with a specific distribution has many applications. In this chapter it is used to familiarize with the properties of the different distributions. However, random

samples can be also used to perform modelling in Earth Sciences. For example, it is at the foundations of error propagation of the Monte Carlo method (Section 10.4).

```

1 import numpy as np
2 from scipy.stats import norm
3 import matplotlib.pyplot as plt
4
5 mu = 0 # mean
6 sigma = 1 # standard deviation
7 normal_sample = np.random.normal(mu, sigma, 15000)
8
9 # plot the histogram of the sample distribution
10 fig, ax = plt.subplots()
11 ax.hist(normal_sample, bins='auto', density=True, color = '#
    c7ddf4', edgecolor= '#000000', label="Random sample with
    normal distribution ")
12
13 # probability density function
14 x = np.arange(-5,5, 0.01)
15 normal_pdf = norm.pdf(x, loc= mu, scale = sigma)
16 ax.plot(x, normal_pdf, color = '#ff464a', linewidth = 1.5,
    linestyle='--', label=r"Normal PDF with  $\mu=0$  and  $1\sigma=1$ "
    =1")
17 ax.legend(title='Normal Distribution')
18 ax.set_xlabel('x')
19 ax.set_ylabel('Probability Density')
20 ax.set_xlim(-5,5)
21 ax.set_ylim(0,0.6)
22
23 # Descriptive statistics
24 arithmetic_mean = normal_sample.mean()
25 standard_deviation = normal_sample.std()
26
27 print('Sample mean equal to {:.4f}'.format(arithmetic_mean))
28 print('Sample standard deviation equal to {:.4f}'.format(
    standard_deviation))
29
30 '''
31 Output: (your results will be slightly different because of the
    pseudo-random nature of the distribution)
32 Sample mean equal to -0.0014
33 Sample standard deviation equal to 1.0014
34 '''

```

Listing 9.3 Generating a random sample with normal distribution ($\mu=0$ and $1\sigma=1$) and a normal PDF having the same μ and 1σ of the random sample.

To note, the use of Monte Carlo simulations are at the basis of many geological studies involving the estimations of uncertainties in the field of mineral exploration mapping (Wang et al., 2020), slope stability (Tobutt, 1982), and groundwater hydrology (Ballio & Guadagnini, 2004).

The code listing 9.3 shows how to generate a random sample of 15000 elements characterized by a $\mu=0$ and $\sigma=1$. Also, the code listing 9.3 develops a normal PDF with the same μ and 1σ of the random sample.

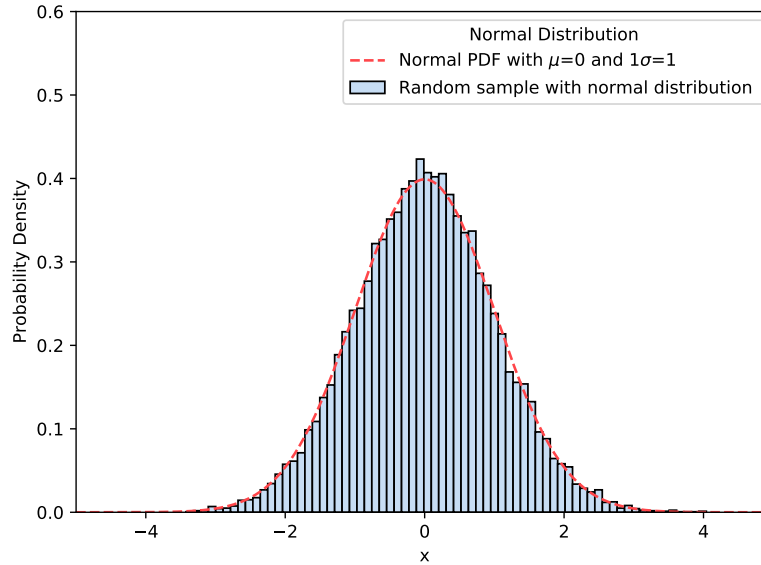


Fig. 9.2 The result of the code listing 9.3.

9.3 The Log-Normal Distribution

The log-normal (or lognormal) distribution is a continuous probability distribution of a random variable whose logarithm is normally distributed. The log-normal distribution has been often invoked as a fundamental rule in Geology (Ahrens, 1953). Currently, it is still widely utilized by geologists, but considering all its potentials and pitfalls (Reimann & Filzmoser, 2000). The PDF for a log-normal distribution is reported in Eq. 9.5.

$$\log PDF_N(x, \mu_n, \sigma_n) = \frac{1}{x} \frac{1}{\sqrt{2\pi\sigma_n^2}} e^{-\frac{(\log(x)-\mu_n)^2}{2\sigma_n^2}} \quad (9.5)$$

where μ_n and σ_n are the mean and the standard deviation of the normal distribution obtained calculating the log (i.e., the natural logarithm) of the random variable. To generate a log-normal distribution, the `scipy.stats.lognorm()` method require the specification of s and scale parameters corresponding to σ_n and e^{μ_n} , respectively.

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 from scipy.stats import norm, lognorm
4
5 colors = ['#342a77', '#ff464a', '#4881e9']
6 Normal_mu = [0,0.5,1]
7 Normal_sigma = [0.5,0.4,0.3]
8 x = np.arange(0.001, 7, .001) # for the log-normal PDF
9 x1 = np.arange(-2.5, 2.5, .001) # for the normal PDF
10
11 fig, (ax1, ax2) = plt.subplots(nrows = 2, ncols = 1, figsize =
    (8,9))
12
13 for mu_n, sigma_n, color in zip(Normal_mu, Normal_sigma, colors):
14     lognorm_pdf = lognorm.pdf(x,s = sigma_n, scale=np.exp(mu_n))
15     r = lognorm.rvs(s = sigma_n, scale = np.exp(mu_n), size =
        15000)
16     ax1.plot(x, lognorm_pdf, color=color, label=r"$\mu_n$ = " +
        str(mu_n) + " - $\sigma_n$ = " + str(sigma_n))
17     ax1.hist(r, bins = 'auto', density = True, color=color,
        edgecolor='#000000', alpha=0.5)
18     logr = np.log(r)
19     normal_pdf = norm.pdf(x1, loc= mu_n, scale = sigma_n)
20     ax2.plot(x1, normal_pdf, color=color, label=r"$\mu_n$ = " +
        str(mu_n) + " - $\sigma_n$ = " + str(sigma_n))
21     ax2.hist(logr, bins = 'auto', density = True, color=color,
        edgecolor='#000000', alpha=0.5)
22     my_mu = logr.mean()
23     ax2.axvline(x=my_mu, color=color, linestyle="--", label=r"
        calculated $\mu_n$ = " + str(round(my_mu,3)))
24     my_sigma = logr.std()
25     print("Expected mean: " + str(mu_n) + " - Calculated mean: "
        + str(round(my_mu,3)))
26     print("Expected std.dev.: " + str(sigma_n) + " - Calculated
        std.dev.: " + str(round(my_sigma,3)))
27
28 ax1.legend(title="log-normal distributions")
29 ax1.set_xlabel('x')
30 ax1.set_ylabel('Probability Density')
31 ax2.legend(title="normal distributions")
32 ax2.set_xlabel('ln(x)')
33 ax2.set_ylabel('Probability Density')
34
35 fig.tight_layout()

```

Listing 9.4 Generating a random samples with log-normal distributions.

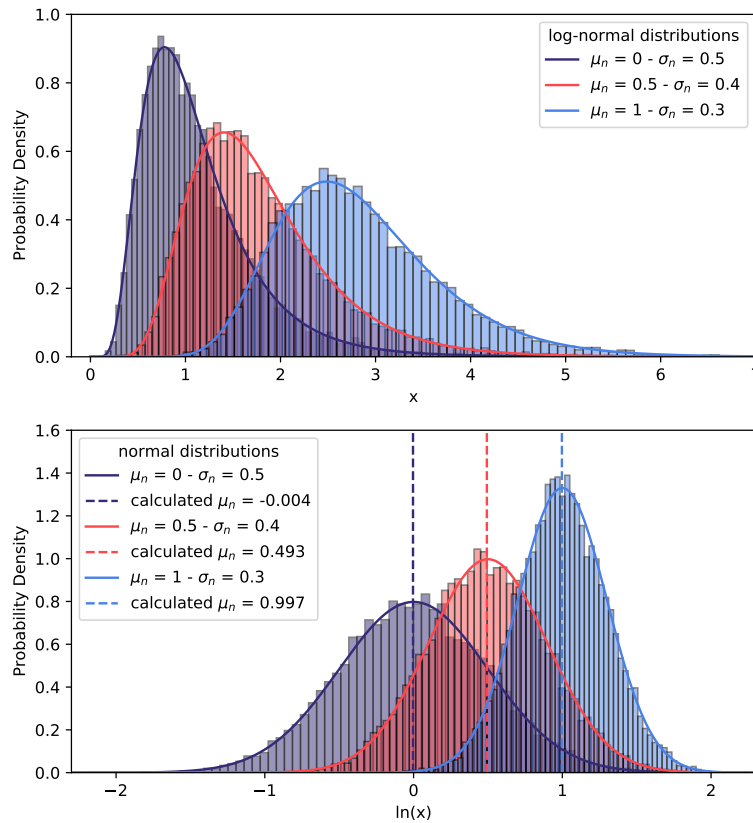


Fig. 9.3 The result of the code listing 9.4.

9.4 Other Useful PDFs for Geological Applications

The `scipy.stats` module allows the management of many probability distributions, useful in geological applications. Examples are the Poisson, Pareto, and Student's *t* distributions that find applications, among other fields, in geochemical determinations (Ulianov et al., 2015), metal exploration (Agterberg, 2018), and geophysical investigations (Troyan & Kiselev, 2010). The table 9.1 reports some probability distributions that are present in the `scipy.stats` module.

9.5 Density Estimation

The process of density estimation consists of reconstructing probability density functions from the observed data (Gramacki, 2018; Silverman, 1998). I will describe

Table 9.1 Selected statistical functions in the `scipy.stats` module.

Function	Distribution	Function	Distribution
<code>alpha()</code>	Alpha cont. random var.	<code>arcsine()</code>	Arcsine cont. random var.
<code>beta()</code>	Beta cont. random var.	<code>bradford()</code>	Bradford cont. random var.
<code>cauchy()</code>	Cauchy cont. random var.	<code>chi()</code>	Chi cont. random var.
<code>chi2()</code>	Chi-squared cont. random var.	<code>cosine()</code>	Cosine cont. random var.
<code>dgamma()</code>	Double gamma cont. random var.	<code>dweibull()</code>	Double Weibul cont. random var.
<code>erlang()</code>	Erlang cont. random var.	<code>expon()</code>	Exponential cont. random var.
<code>halfcauchy()</code>	Half-Cauchy cont. random var.	<code>halfnorm()</code>	Half-normal cont. random var.
<code>laplace()</code>	Laplace cont. random var.	<code>levy()</code>	Levy cont. random var.
<code>logistic()</code>	Logistic cont. random var.	<code>loggamma()</code>	Log gamma cont. random var.
<code>loglaplace()</code>	Log-Laplace cont. random var.	<code>loguniform()</code>	Loguniform cont. random var.
<code>maxwell()</code>	Maxwell cont. random var.	<code>pareto()</code>	Pareto cont. random var.
<code>pearson3()</code>	Pearson type III cont. random var.	<code>powerlaw()</code>	Power-function cont. random var.
<code>rayleigh()</code>	Rayleigh cont. random var.	<code>skewnorm()</code>	Skew-normal cont. random var.
<code>t()</code>	Student's t cont. random var.	<code>uniform()</code>	Uniform cont. random var.
<code>bernoulli()</code>	Bernoulli discr. random var.	<code>binom()</code>	Binomial discr. random var.
<code>boltzmann()</code>	Boltzmann discr. random var.	<code>dlaplace()</code>	Laplacian discr. random var.
<code>geom()</code>	Geometric discr. random var.	<code>poisson()</code>	Poisson discr. random var.

two main approaches to achieve this goal. The first one is parametric. It consists of selecting a known probability density function and fit the observed data with its governing parameters (Gramacki, 2018; Silverman, 1998). As an example, if we would like to fit a bell-shaped distribution with a normal PDF, we start estimating its mean (μ) and standard deviation (σ). Then, the obtained μ and σ values are used to reconstruct a normal PDF and fit the observed distribution. The processes of fitting described in code listing 9.3 and 9.4 are all examples of parametric density estimations.

Although intriguing for its simplicity, the parametric approach is not always the best choice (Gramacki, 2018; Silverman, 1998). As an example, popular PDF are mostly unimodal, but many practical examples in Geology involve multimodal distributions. Also, the choice of a specific known PDF is not always straightforward when working with practical Geological applications. As a consequence, the so-called non-parametric approach is often the best choice (Gramacki, 2018; Silverman, 1998). It relies the attempting of an estimation for the density directly from the data, without making any parametric assumptions about the underlying distribution (Gramacki, 2018; Silverman, 1998).

A density histogram is the simplest form of non-parametric density estimation (Gramacki, 2018; Silverman, 1998). We encountered density histograms earlier in the book at Section 4.2. The development of a density histogram is quite easy. In detail, It consist in dividing the the sample space into intervals called bins (Gramacki, 2018; Silverman, 1998). Then, the density for each bin is estimated using the equation 9.6 (Gramacki, 2018; Silverman, 1998).

$$\hat{f}(x_i - h/2 \leq x < x_i + h/2) = \frac{k_i}{n \cdot h} \quad (9.6)$$

where x_i , k_i , n , and h are the x values at the centre of each bin, i.e., the interval $[x_i - h/2, x_i + h/2]$, the number of observation in the interval $x_i - h/2 \leq x_i < x_i + h/2$, the number of bins, and the bin width, i.e., $h = x_i - x_{i+1} = (x_{max} - x_{min})/n$, respectively. Please note that the symbol \hat{f} refers to the empirical estimation of the PDF.

A more evolved method than density histograms to guess a PDF starting from experimental data is the kernel density estimation (KDE). A KDE is a non-parametric way to estimate the probability density function of a random variable. To understand, let $(x_1, x_2, x_i, \dots, x_n)$ be a univariate independent and identically distributed (i.e., they have the same probability distribution) sample belonging to a distribution with an unknown PDF. We are interested in estimating the shape (\hat{f}) of this PDF. The equation that define a KDE is:

$$\hat{f}(x) = \frac{1}{nh} \sum_{i=1}^n K\left(\frac{x - x(i)}{h}\right) \quad (9.7)$$

where K is the kernel, a non-negative function that integrates to one, i.e., $\int_{-\infty}^{\infty} K(x)dx = 1$, and h (with $h > 0$) is a smoothing parameter called the bandwidth. A range of kernel functions are commonly used: normal, uniform, triangular, biweight, triweight, Epanechnikov, and others (code listing 9.5 and Fig. 9.4).

In Python, there are many different implementations allowing the development of a KDE (Table 9.2).

```

1 from statsmodels.nonparametric.kde import KDEUnivariate
2 import matplotlib.pyplot as plt
3 import numpy as np
4
5 kernels = ['gau', 'epa', 'uni', 'tri', 'biw', 'triw']
6 kernels_names = ['Gaussian', 'Epanechnikov', 'Uniform', '
   Triangular', 'Biweight', 'Triweight']
7 positions = np.arange(1,9,1)
8
9 fig, ax = plt.subplots()
10
11 for kernel, kernel_name, pos in zip(kernels, kernels_names,
   positions):
12     # kernels
13     kde = KDEUnivariate([0])
14     kde.fit(kernel= kernel, bw=1, fft=False, gridsize=2**10)
15     ax.plot(kde.support, kde.density, label = kernel_name,
   linewidth=1.5, alpha=0.8)
16
17
18 ax.set_xlim(-2,2)
19 ax.grid()
20 ax.legend(title='kernel functions')
```

Listing 9.5 Kernel functions available in *KDEUnivariate*().

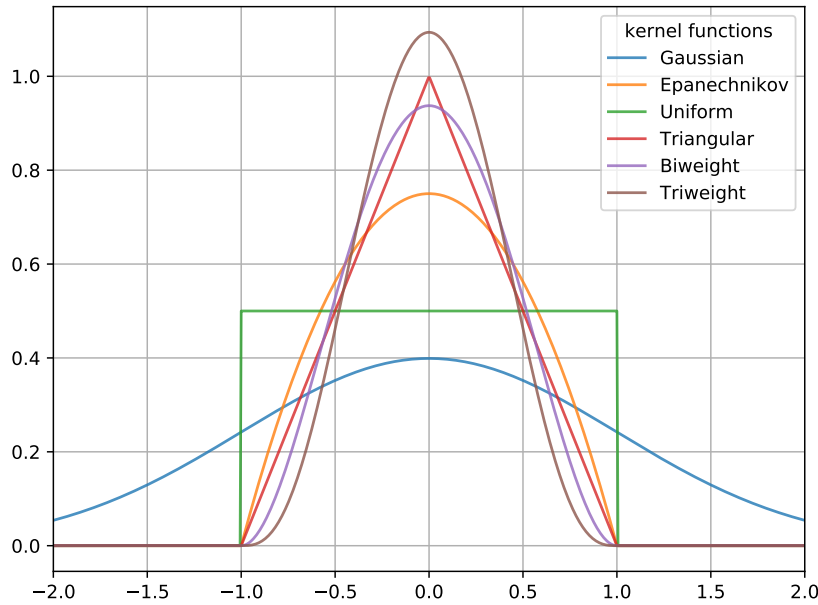


Fig. 9.4 The result of the code listing 9.5.

Table 9.2 Selection of kernel density estimators in Python.

Package	Function	Description
Scipy	<code>gaussian_kde()</code>	kernel-density estimate using Gaussian kernels
Statsmodels	<code>KDEUnivariate()</code>	Univariate kernel density estimator
Statsmodels	<code>KDEMultivariate()</code>	Multivariate kernel density estimator
Scikit-Learn	<code>KernelDensity()</code>	Multivariate kernel density estimator
Seaborn	<code>kdeplot()</code>	Plot univariate or bivariate distributions using kernel density estimation

The code listing 9.6 and Fig. 9.5 shows the application of the `KDEUnivariate()` function to geochemical data. Also, they show the effect of bandwidth selection on the resulting KDE estimate.

As example application of density histograms and KDE for unravelling PDFs in geological applications, the code listing 9.7 and Fig. 9.6 show the reconstruction of $^{238}\text{U}/^{206}\text{Pb}$ Zircon ages for the last 1500 My. Data are from (Puetz, 2018). Due to the recent re-rising linking between magmatism and Mass extinction (Davies et al., 2017; Liu et al., 2017; Tegner et al., 2020), the largest extinction events are also reported.

```

1 from statsmodels.nonparametric.kde import KDEUnivariate
2 import pandas as pd
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
7     sheet_name='Supp_traces')
8
9 x = myDataset.Zr
10 x_eval = np.arange(0,1100,1)
11
12 fig = plt.figure()
13 ax1 = fig.add_subplot(2, 1, 1)
14 # Density Histogram
15 ax1.hist(x, bins= "auto", density = True, label="Density
16     Histogram", color='#c7ddf4', edgecolor='#000000')
17 kde = KDEUnivariate(x)
18 kde.fit()
19 My_kde = kde.evaluate(x_eval)
20 ax1.plot(x_eval, My_kde, linewidth = 1.5, color='#ff464a', label=
21     "gaussian KDE - auto bandwidth selection")
22 ax1.set_xlabel('Zr [ppm]')
23 ax1.set_ylabel('Probability density')
24 ax1.legend()
25
26 ax2 = fig.add_subplot(212)
27 # Density Histogram
28 ax2.hist(x, bins= "auto", density = True, label="Density
29     Histogram", color='#c7ddf4', edgecolor='#000000')
30
31 # KDE
32 # Effect of bandwidth
33 for my_bw in [10,50,100]:
34     kde = KDEUnivariate(x)
35     kde.fit(bw = my_bw)
36
37     My_kde = kde.evaluate(x_eval)
38     ax2.plot(x_eval, My_kde, linewidth = 1.5, label="gaussian KDE
39         - bw: " + str(my_bw))
40
41 ax2.set_xlabel('Zr [ppm]')
42 ax2.set_ylabel('Probability density')
43 ax2.legend()
44
45 fig.tight_layout()

```

Listing 9.6 Example application of KDE in Geology.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import numpy as np
4 from statsmodels.nonparametric.kde import KDEUnivariate
5
6 # import Zircon data from Puetz (2010)
7 mydata = pd.read_excel('1-s2.0-S1674987117302141-mmcl.xlsx',
8     sheet_name='Data')
9 mydata = mydata[(mydata.age206Pb_238U>0)&(mydata.age206Pb_238U
10     <1500)]
11 d = mydata.age206Pb_238U
12
13 # Plot the Density Histogram
14 fig, ax = plt.subplots(figsize=(8,5))
15 bins = np.arange(0,1500,20)
16 ax.hist(d, bins, color='#c7ddf4', edgecolor='k', density=True,
17     label='Density Histogram - bins = 20 My')
18
19 # Compute and plot the KDE
20 age_eval = np.arange(0,1500,10)
21 kde = KDEUnivariate(d)
22 kde.fit(bw=20)
23 pdf = kde.evaluate(age_eval)
24 ax.plot(age_eval, pdf, label = 'Gaussian KDE - bw = 20 Ma',
25     linewidth=2, alpha=0.7, color='#ff464a')
26
27 # Adjust diagram parameters
28 ax.set_ylim(0,0.0018)
29 ax.set_xlabel('Age (My)')
30 ax.set_ylabel('Probability Density')
31 ax.legend()
32 ax.grid(axis='y')
33
34 # Plot mass extinction annotations
35 mass_extinction_age = [444, 359, 252, 66, 0]
36 pdf_mass_extinction_age = kde.evaluate(mass_extinction_age)
37 mass_extinction_name = ["Ordovician-Silurian", "Late Devonian",
38     "Permian-Triassic", "Cretaceous-Paleogene", "Men's
39     Triggered?"]
40 y_offsets = [0.0001, 0.0001, 0.0002, 0.0002, 0.0004]
41 y_texts = [30, 105, 15, 62, 160]
42 x_texts = [30, 30, 30, 30, 30]
43
44 for x, y, name, x_text, y_text, y_offset in zip(
45     mass_extinction_age, pdf_mass_extinction_age,
46     mass_extinction_name, x_texts, y_texts, y_offsets):
47     ax.annotate(name, xy=(x, y + y_offset), xycoords='data',
48         xytext=(x_text, y_text), textcoords='offset points',
49         arrowprops=dict(arrowstyle="->",
50             connectionstyle="angle, angleA=0, angleB=90, rad=10"))
51
52 fig.tight_layout()

```

Listing 9.7 Example application of KDE in Geology.

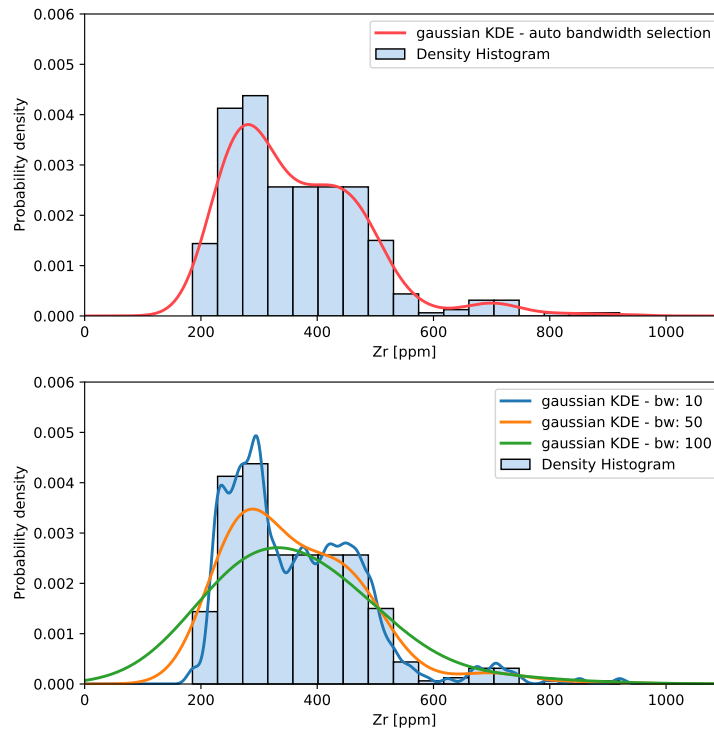


Fig. 9.5 The result of the code listing 9.6.

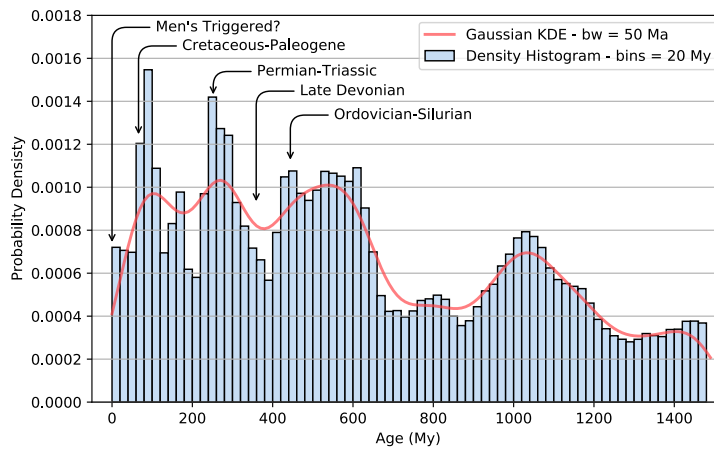


Fig. 9.6 The result of the code listing 9.7.

9.6 The Central Limit Theorem and Normal Distributed Means

There are different ways to report the central limit theorem. The easiest is the one adopted by Hughes and Hase (2010): "the sum of a large number of independent random variables, each with finite mean and variance, will tend to be normally distributed, irrespective of the distribution function of the random variable."

To familiarize with the central limit theorem, the code listing 9.8 and Fig. 9.7, replicate the part of the experiment reported by Hughes and Hase (2010) in Fig 3.7.

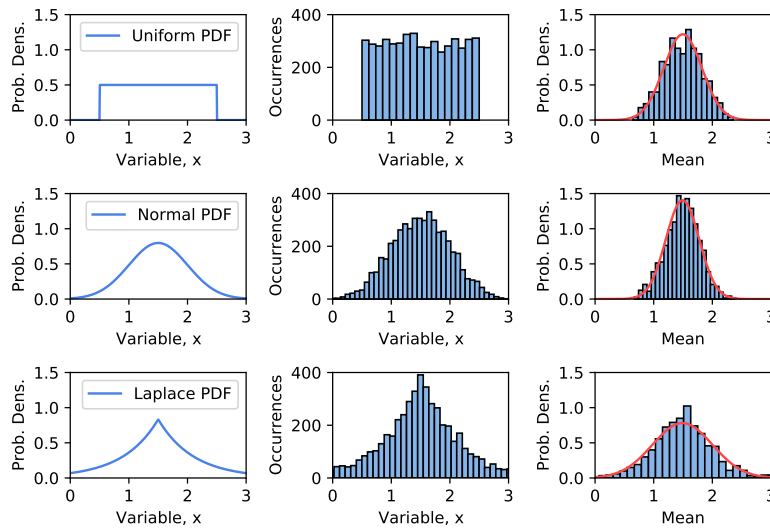


Fig. 9.7 The result of the code listing 9.8.

In detail, the code listing 9.8 starts from three different distribution of the random variable (i.e., uniform, normal and laplace; Tab. 9.1) to create: 1) the relative probability density function (first column of Fig. 9.7), 2) 1000 randomly generated occurrences of the random variable (second column of Fig. 9.7), and 3) the estimation of mean value of the distribution based on 1000 attempts using 3 randomly selected occurrences of random variable (third column of Fig. 9.7).

In accordance with the central limit theorem, the histograms of the estimated means assumes normal distribution (third column of Fig. 9.7) with a mean peaked at 1.5. Also, distribution of the means (third column of Fig. 9.7) is narrower than the width of the original distributions (second column of Fig. 9.7) by a factor \sqrt{N} . Further details and some geological implications of the central limit theorem will be provided and discussed in Chapter 10.

```

1 import numpy as np
2 import scipy.stats as stats
3 import matplotlib.pyplot as plt
4
5 fig = plt.figure(figsize=(8,6))
6
7 dists = [stats.uniform(loc=0.5, scale=2), stats.norm(loc=1.5,
8           scale=0.5), stats.laplace(loc=1.5, scale=0.6)]
9 names = ['Uniform', 'Normal', 'Laplace']
10 x = np.linspace(0,3,1000)
11
12 for i, (dist, name) in enumerate(zip(dists, names)):
13     # Probability Density Function (pdf)
14     pdf = dist.pdf(x)
15     ax1 = fig.add_subplot(3, 3, 3*i+1)
16     ax1.plot(x, pdf, color='#4881e9', label= name + ' PDF')
17     ax1.set_xlim(0,3)
18     ax1.set_ylim(0,1.5)
19     ax1.set_xlabel('Variable, x')
20     ax1.set_ylabel('Prob. Dens.')
21     ax1.legend()
22
23     #Distribution (rnd) of the Random Variable based on the
24     #selected pdf
25     rnd = dist.rvs(size=5000)
26     ax2 = fig.add_subplot(3,3,3*i+2)
27     ax2.hist(rnd, bins='auto', color='#84b4e8', edgecolor='
28             #000000')
29     ax2.set_xlim(0,3)
30     ax2.set_ylim(0,400)
31     ax2.set_xlabel('Variable, x')
32     ax2.set_ylabel('Occurrences')
33
34     ax3 = fig.add_subplot(3,3,3*i+3)
35     mean_dist = []
36     for _ in range(1000):
37         mean_dist.append(dist.rvs(size=3).mean())
38     mean_dist = np.array(mean_dist)
39     ax3.hist(mean_dist, density=True, bins='auto', color='#84
40             b4e8', edgecolor='#000000')
41     normal = stats.norm(loc= mean_dist.mean(), scale=
42             mean_dist.std())
43     ax3.plot(x, normal.pdf(x), color='#ff464a')
44     ax3.set_xlim(0,3)
45     ax3.set_ylim(0,1.5)
46     ax3.set_xlabel('Mean')
47     ax3.set_ylabel('Prob. Dens.')
48
49 fig.tight_layout()

```

Listing 9.8 The central limit theorem (Hughes & Hase, 2010).

Chapter 10

Error Analysis

10.1 Dealing with Errors in Geological Measurements

As reported by Hughes and Hase (2010), the aim of error analysis is to quantify and record the errors associated with the inevitable spread in a set of measurements. This is also true for geological estimations. The following definitions are taken from the book "Measurements and their Uncertainties" by Hughes and Hase (2010). There are two fundamental terms to describe the uncertainties associated to a set of measurements: precision and accuracy. An accurate measurement is one in which the results of the experiments are in agreement with the accepted value. A precise result is one where the spread of measurements is 'small' either relative to the average results or in absolute magnitude. In this chapter, I will also discuss about the meaning of the standard error (i.e., the uncertainty in mean estimations) and how to propagate the uncertainties using two different strategies: the linear methods and the Monte Carlo approach.

Precision and accuracy

To introduce the concepts of precision and accuracy, I am going to use a practical example: the estimation of the figure of merits of an instrumentation used to perform the chemical characterization of geological samples. The definition of the precision and the accuracy of an analytical device is typically performed using a reference material, i.e., a chemical homogeneous sample of known composition (better if certified), analyzed as an unknown. In the following, I report the results obtained during repeated analyses of the USGS BCR2G reference material at the LA-ICP-MS facility of Perugia University over about five years. These results are obtained in very comfortable operating conditions using a large beam diameter (i.e, 80 μm), 10 Hz and laser fluence of 3.5 J/cm². The chemical element reported here is the Lanthanum (La), present at a concentration of 25.6 ± 0.5 ppm (Rocholl, 1998). Data are stored in the USGS_BCR2G.xls file.

```
1 import pandas as pd
2 import scipy.stats as stats
3 import matplotlib.pyplot as plt
4 import numpy as np
5
6 myDataset = pd.read_excel('USGS_BCR2G.xls', sheet_name='Sheet1')
7
8 fig, ax = plt.subplots()
9 ax.hist(myDataset.La, bins='auto', density=True, edgecolor='
  #000000', color = '#c7ddf4', label= "USGS BCR2G")
10 ax.set_xlabel("La [ppm]")
11 ax.set_ylabel("Probability Density")
12
13 x = np.linspace(23,27.5,500)
14 pdf = stats.norm(loc=myDataset.La.mean(), scale=myDataset.La.std
  ()).pdf(x)
15
16 ax.plot(x,pdf, linewidth=2, color='#ff464a', label = 'Normal
  Distribution')
17
18 ax.legend()
```

Listing 10.1 LA-ICP-MS determinations of La in the USGS BCR2G reference material.

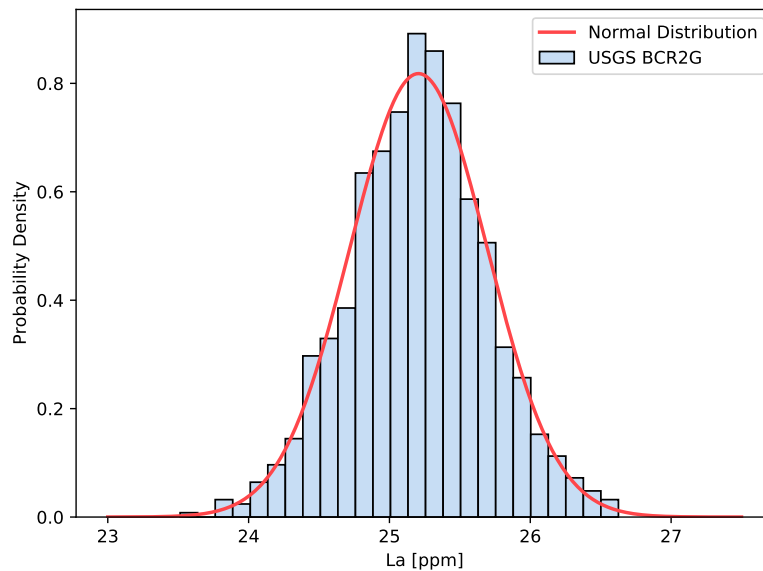


Fig. 10.1 The result of the code listing 10.1.

In detail, the accuracy measure the agreement of our estimates with real values in the unknowns. Typically, the accuracy of an analytical device (LA-ICP-MS in our case) is estimated evaluating the agreement of the estimates to the accepted values of a reference material. The deviation of the mean μ of the measurements from the accepted value R is an estimation of accuracy:

$$Accuracy = 100 \cdot \frac{\mu - R}{R} \quad (10.1)$$

The precision of a set of measurements is the spread of the obtained distribution, and it can be estimated using an index of dispersion (Chapter 5). Typically, the utilized index is the standard deviation, often expressed in percent:

$$Precision = 100 \cdot \frac{\sigma}{R} \quad (10.2)$$

```

1 MyMean = myDataset.La.mean()
2 R = 25.6
3 Accuracy = 100 * (MyMean - R) / R
4 MyStd = myDataset.La.std()
5 Precision = 100 * MyStd / R
6
7 fig, ax = plt.subplots(figsize=(6,5))
8 ax.hist(myDataset.La, bins = 'auto', density = True, edgecolor =
9         '#000000', color = '#c7ddf4', label = 'USGS BCR2G')
10 ax.set_xlabel('La [ppm]')
11 ax.set_ylabel('Probability Density')
12 ax.axvline(x=myDataset.La.mean(), color='#ff464a', linewidth=3,
13           label='Mean of the Measurements:' + str(round(MyMean, 1)) +
14             '[ppm]')
15 ax.axvline(x = R, color='#342a77', linewidth=3, label='Accepted
16           Value')
17 ax.axvline(x = MyMean - MyStd, color = '#4881e9', linewidth = 1)
18 ax.axvline(x = MyMean + MyStd, color = '#4881e9', linewidth = 1)
19 ax.axvspan(MyMean - MyStd, MyMean + MyStd, alpha = 0.2, color =
20           '#342a77', label = r'$1\sigma$')
21 ax.legend(loc='upper center', bbox_to_anchor=(0.5, -0.15),
22         fancybox=False, shadow=False, ncol=2, title = 'Accuracy = ' +
23           str(round(Accuracy, 1)) + '% - Precision = ' + str(round(
24           Precision, 1)) + '%')
25 fig.tight_layout()

```

Listing 10.2 Accuracy an Precision.

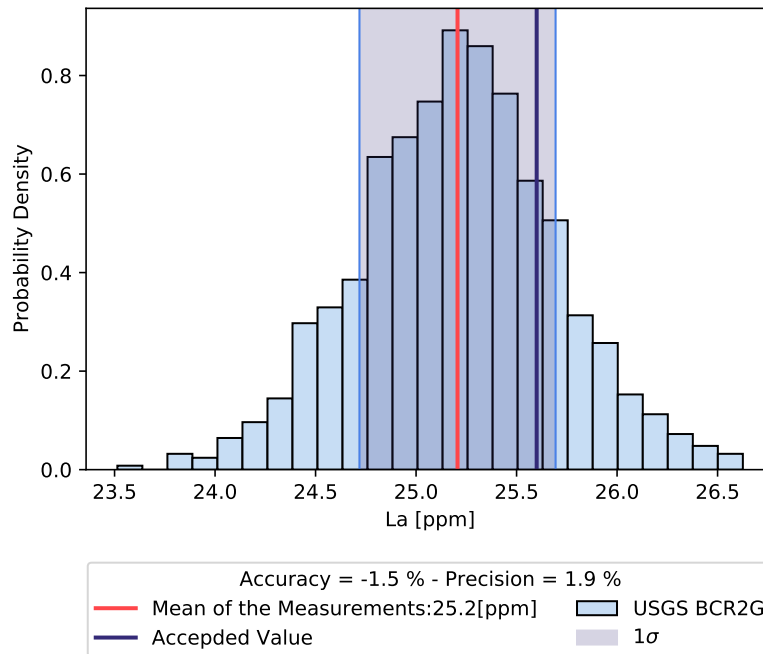


Fig. 10.2 The result of the code listing 10.2.

Confidence intervals

As a consequence of the Central Limit Theorem, a large enough set of measurements on the same target resulting from many random sources of (small) uncertainty will approach to a normal distribution (Fig. 10.1; see Section 9.6 for further details).

The normal distribution enable us to verify the probability of La measurements to lie within one- (68.27%), two- (95.45%), and three-standard deviations (99.27%) interval around the obtained mean value using the Eq. 9.4 (code listing 10.3 and Fig. 10.3). Therefore, to provide a complete picture of our estimations for a quantity x , the results should be provided using the mean value (μ_x), and the confidence intervals (Hughes & Hase, 2010; Taylor, 1997):

$$\mu \pm n\sigma_x \quad (10.3)$$

with $n = 1, 2, 3, \dots$ corresponding to confidence intervals of 68.27%, 95.45%, 99.27%, ... , respectively.


```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def normalPDF(x, mu, sigma):
6     PDF = 1/(sigma * np.sqrt(2 * np.pi)) * np.exp( - (x - mu)
7         **2 / (2 * sigma**2))
8     return PDF
9
10 signal_levels = [1, 2, 3]
11 confidences = [68.27, 95.45, 99.73]
12
13 fig = plt.figure(figsize=(7,8))
14 MyMean = myDataset.La.mean()
15 MyStd = myDataset.La.std()
16
17 x_pdf = np.linspace(MyMean - 4 * MyStd, MyMean + 4 * MyStd,
18                     1000)
19 my_PDF = normalPDF(x_pdf, MyMean, MyStd)
20
21 for signal_level, confidence in zip(signal_levels, confidences):
22     ax = fig.add_subplot(3, 1, signal_level)
23     ax.hist(myDataset.La, bins = 'auto', density = True,
24            edgecolor = '#000000', color = '#c7ddf4', label = 'USGS
25            BCR2G', zorder=0)
26     x_confidence = np.linspace(MyMean - signal_level * MyStd,
27                               MyMean + signal_level * MyStd, 1000)
28     ax.plot(x_pdf, my_PDF, linewidth=2, color='#ff464a', label
29            = 'Normal Distribution', zorder=1)
30     ax.fill_between(x_confidence, normalPDF(x_confidence,
31                               MyMean, MyStd), y2=0, color = '#ff464a', alpha=0.2, label
32            = 'prob. = {}'.format(confidence) + ' %', zorder=1)
33     ax.legend(ncol=3, loc='upper center', title = r'$\mu \sim \text{ppm}
34            \sim $' + str(signal_level) + r'$ \sim \sigma \sim $ = ' + '{:.1f}'.
35            format(MyMean) + r'$ \sim \text{ppm} \sim $' + '{:.1f}'.format(
36            signal_level * MyStd))
37     ax.set_ylim(0, 1.6)
38     ax.set_xlabel('La [ppm]')
39     ax.set_ylabel('prob. dens.')
40
41 fig.tight_layout()

```

Listing 10.3 Confidence intervals.

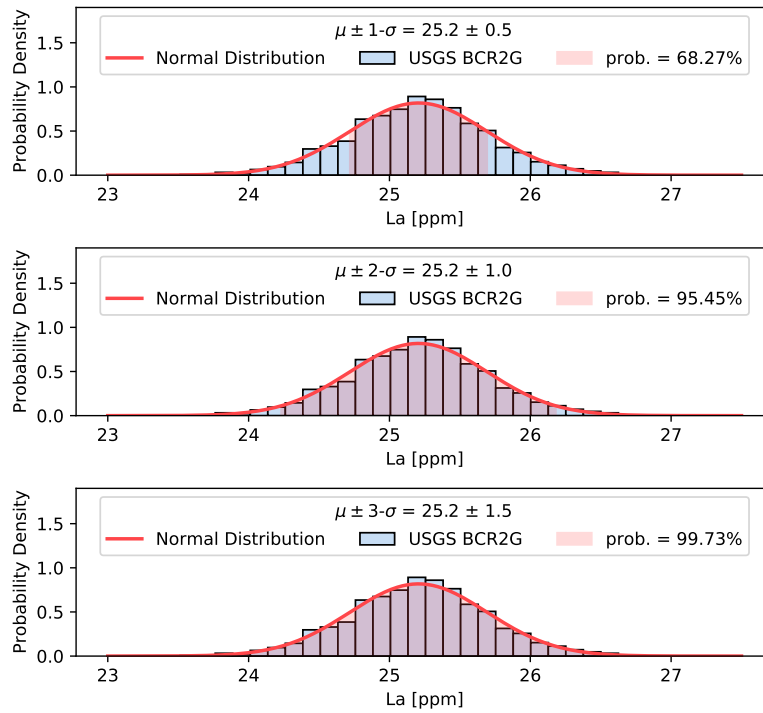


Fig. 10.3 The result of the code listing 10.3.

Uncertainties of mean estimates: the standard error

The standard deviations of the means or standard error (SE) is a measure of the uncertainty in the location of the mean of a set of measurements (Hughes & Hase, 2010):

$$SE = \frac{\sigma_s}{\sqrt{n}} \quad (10.4)$$

As a consequence, mean estimates μ_s should be reported as follow (Hughes & Hase, 2010; Taylor, 1997):

$$\mu_s \pm SE = \mu_s \pm \frac{\sigma_s}{\sqrt{n}} \quad (10.5)$$

The significance of Eq. 10.4 can be evaluated in light of the central limit theorem. Assume you are sampling an homogeneous material (e.g., a geological reference material like the USGS BCR2G) characterized by a perfectly known target value of 1.5 (the unit is not important here) using a well calibrated analytical device (i.e., no accuracy biases).

```

1 import numpy as np
2 import scipy.stats as stats
3 import matplotlib.pyplot as plt
4
5 mean_value = 1.5
6 std_dev = 0.5
7 dist = stats.norm(loc = mean_value, scale = std_dev)
8 x = np.linspace(0,3,1000)
9 fig = plt.figure(figsize=(6,8))
10
11 # Distribution of the Random Variable based on the normal PDF
12 pdf = dist.pdf(x)
13 ax1 = fig.add_subplot(3, 1, 1)
14 ax1.plot(x, pdf, color = '#84b4e8', label = r'$\mu_p$ = 1.5 -
15     1$\sigma_p$ = 0.5')
16 ax1.set_xlim(0,3)
17 ax1.set_ylim(0,1)
18 ax1.set_xlabel('Variable, x')
19 ax1.set_ylabel('Prob. Dens.')
20 ax1.legend(title = 'Parent Distribution')
21
22 # Dependence of the SE on the Central Limit Theorem
23 ax2 = fig.add_subplot(3, 1, 2)
24 std_of_the_mean = []
25 Ns = [2,10,100,500]
26
27 for N in Ns:
28     # Mean Estimation Based on 1000 attempts using N values
29     mean_dist = []
30     for _ in range(1000):
31         mean_dist.append(dist.rvs(size=N).mean())
32     mean_dist = np.array(mean_dist)
33     std_of_the_mean.append(mean_dist.std())
34     normal = stats.norm(loc= mean_dist.mean(), scale =
35         mean_dist.std())
36     ax2.plot(x, normal.pdf(x), label='N = ' + str(N))
37 ax2.set_xlim(0,3)
38 ax2.set_xlabel('Mean')
39 ax2.set_ylabel('Prob. Dens.')
40 ax2.legend(title='Standard Deviation of the Means', ncol=2)
41
42 # SE estimates and the empirically derived std of the Means
43 ax3 = fig.add_subplot(3,1,3)
44 ax3.scatter(Ns,std_of_the_mean, color = '#ff464a', edgecolor =
45     '#000000', label = "Standard Deviation of the Means",
46     zorder = 1)
47 N1 = np.linspace(1,600,600)
48 SE = std_dev / np.sqrt(N1)
49 ax3.plot(N1 , SE, c = '#4881e9', label= 'Standard Error (SE)',
50     zorder = 0)
51 ax3.set_xlabel('N')
52 ax3.set_ylabel('Standard Error, SE')
53 ax3.legend()
54 fig.tight_layout()

```

Listing 10.4 Standard error estimate.

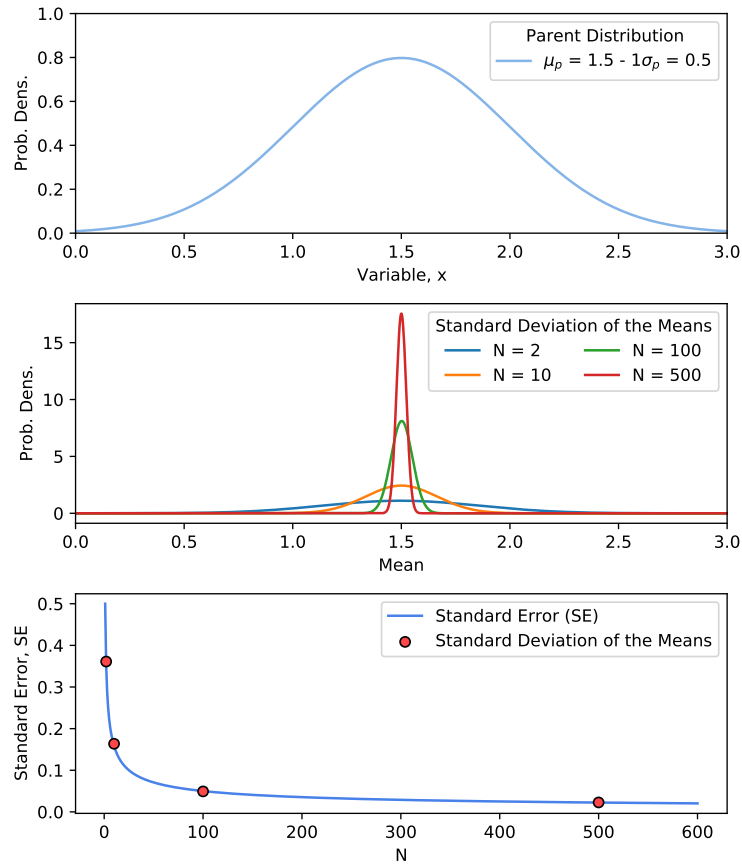


Fig. 10.4 The result of the code listing 10.4.

Because of the many random uncertainties associated to the analytical device, the target population (i.e., the set of all possible measurements) will assume a normal distribution in agreement with the central limit theorem (upper panel of Fig. 10.4). Making the analyses, we will start sampling the target population. What is the uncertainty associated to the mean estimate using n estimates? The standard error is a measure of this uncertainty and can be measured either using the Eq. 10.4, or repeating many times (1000 in the case of code listing 10.4) the mean estimation with N measurements and estimating the standard deviation of the obtained set of means (middle panel of Fig. 10.4). As we are geologists and we trust on the evidence only, in the bottom panel of Fig. 10.4 I report a comparison of the SE obtained using the Eq. 10.4 and the standard deviation values of the means distributions obtained in the above experiment. The code listing 10.4 reports the procedure to unravel the meaning of the SE and create Fig. 10.4).

But what are the information provided by the SE? To answer, please look at code listing 10.4 and Fig. 10.4, where we are sampling (e.g., making the analyses of an unknown geological material, or sampling a geological quantity like a deep or strike) the same normal population of 10.4) characterized by a mean and a standard deviation of 1.5 and 0.5, respectively. Performing few estimates (e.g., 3), we obtain a mean and a standard deviation estimates of 1.56 and 0.51, respectively. In this case, the SE is equal to 0.23. As a consequence we should write the $\mu_s = 1.56 \pm 0.23$ and $\sigma_s = 0.51$. To note, three parameters are needed to define our measurements. Increasing n , the SE decreases progressively, with μ_s becoming a more robust estimate of the mean value of the parent distribution (Fig. 10.4). Always remember that the standard deviation is a measure of the spread of the sampled distribution. It highlights how accurately the mean represents the sampled distribution. On the contrary, the standard error measures how far the sample mean (μ_s) of the measurements is likely to be from the true population mean (μ_p). Finally note that the SE is always smaller than the σ_s .

10.2 Reporting Uncertainties in Binary Diagrams

Errors are always present in empirical estimation (e.g., geological samplings and analytical determinations). As a consequence, uncertainties should be always accounted during data visualization and modelling. Under the assumption of a normal distribution of our estimates (cfc. the central limit theorem described in Section. 9.6), we can set confidence levels at 68, 95, and 99.7 % using 1σ , 2σ , and 3σ , respectively.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 myDataset1 = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
5                             sheet_name='Supp_traces')
6
7 x = myDataset1.Zr
8 y = myDataset1.Th
9 dx = myDataset1.Zr * 0.1
10 dy = myDataset1.Th * 0.1
11
12 fig, ax = plt.subplots()
13 ax.errorbar(x, y, xerr=dx, yerr=dy, marker='o', markersize=4,
14             linestyle = '', color='#c7ddf4', markeredgcolor='k', ecolor=
15             '0.7', label='Recent CFC activity')
16 ax.set_xlabel('Zr [ppm]')
17 ax.set_ylabel('Th [ppm]')
18 ax.legend(loc='upper left')
```

Listing 10.5 Reporting errors in binary diagrams.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.array([250,300,360,480,570,770,870,950])
5 y = np.array([20,25,30,40,50,70,80,100])
6
7
8 fig = plt.figure(figsize=(6,8))
9
10 # xerr and yerr reported as single value
11 dx = 50
12 dy = 10
13 ax1 = fig.add_subplot(3,1,1)
14 ax1.errorbar(x, y, xerr=dx, yerr=dy, marker='o', markersize=6,
15             linestyle = '', color='#c7ddf4', markeredgcolor='k', ecolor=
16             '0.7', label='single value for xerr and yerr')
17 ax1.legend(loc='upper left')
18
19 # xerr and yerr reported as 1D array
20 dx = np.array([25,35,40,120,150,30,30,25])
21 dy = np.array([8,8,6,7,7,35,40,40])
22
23 ax2 = fig.add_subplot(3,1,2)
24 ax2.errorbar(x, y, xerr=dx, yerr=dy, marker='o', markersize=6,
25             linestyle = '', color='#c7ddf4', markeredgcolor='k', ecolor=
26             '0.7', label='xerr and yerr as 1D array')
27 ax2.set_ylabel('Th [ppm]')
28 ax2.legend(loc='upper left')
29
30 # xerr and yerr reported as 2D array
31 dx = np.array
32     ([[80,60,70,100,150,150,20,100],[20,25,30,30,30,30,90,30]])
33 dy = np.array([[10,4,10,15,15,20,5,5],[2,8,4,4,6,7,10,20]])
34
35 ax3 = fig.add_subplot(3,1,3)
36 ax3.errorbar(x, y, xerr=dx, yerr=dy, marker='o', markersize=6,
37             linestyle = '', color='#c7ddf4', markeredgcolor='k', ecolor=
38             '0.7', label='xerr and yerr as 2D array')
39 ax3.set_xlabel('Zr [ppm]')
40 ax3.legend(loc='upper left')
41
42 fig.tight_layout()

```

Listing 10.6 Reporting errors in binary diagrams.

In binary diagrams, errors can be easily reported using the *errorbar()* function of the *matplotlib.pyplot* sub-package (code listing 10.5 and Fig. 10.5).

The *errorbar()* function accepts all the arguments available for *plot()*, plus *xerr*, *yerr*, and the related arguments. In detail, *xerr* and *yerr* refer to the error on the x and y axis, respectively. They can be a float, i.e., a number defining the same error for all the measurements.

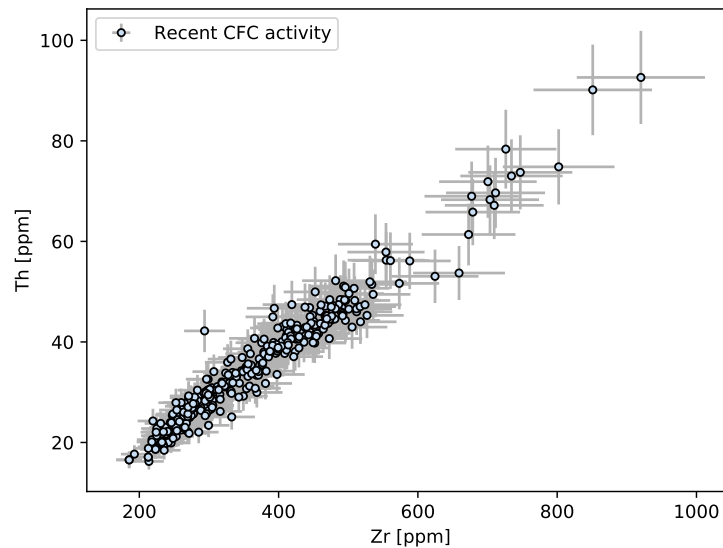


Fig. 10.5 The result of the code listing 10.5.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.array([200,300,360,480,570,770,870,950])
5 y = np.array([10,15,30,40,50,70,80,100])
6 dx = 40
7 dy = 10
8
9 fig = plt.figure()
10 ax1 = fig.add_subplot(2,1,1)
11 ax1.errorbar(x, y, xerr=dx, yerr=dy, marker='o', markersize=4,
12             linestyle = '', color='k', ecolor='0.7', elinewidth=3,
13             capsize=0, label='Recent activity of the CFC')
14 ax1.legend(loc='upper left')
15 ax1.set_xlabel('Zr [ppm]')
16 ax1.set_ylabel('Th [ppm]')
17
18 ax2 = fig.add_subplot(2,1,2)
19 ax2.errorbar(x, y, xerr=dx, yerr=dy, marker='o', markersize=6,
20             linestyle = '', color='#c7ddf4', markeredgcolor='k',
21             ecolor='k', elinewidth = 0.8, capthick=0.8, capsize=3,
22             label='Recent activity of the CFC')
23 ax2.legend(loc='upper left')
24 ax2.set_xlabel('Zr [ppm]')
25 ax2.set_ylabel('Th [ppm]')

```

Listing 10.7 Reporting errors in binary diagrams.

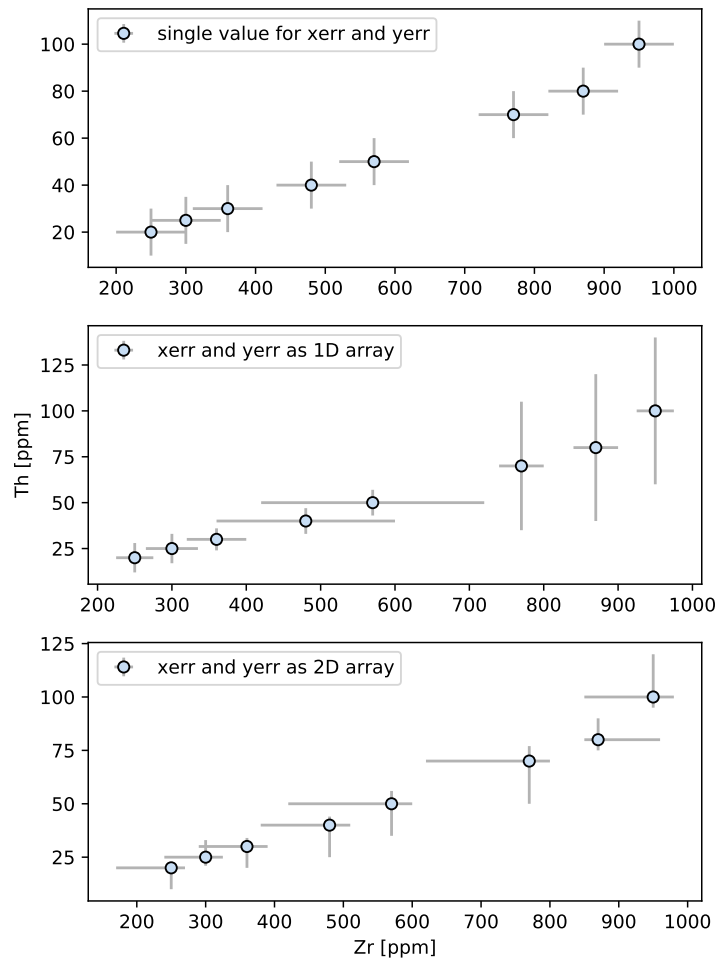


Fig. 10.6 The result of the code listing 10.6.

Also, they can be a 1D or 2D arrays. Using 1D arrays (e.g. Fig. 10.5), a symmetrical error (i.e., $x \pm xerr$) is defined for each single point. Finally, reporting $xerr$ and $yerr$ as 2D array, we report non-symmetrical errors (Fig. 10.6)


```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4
5 def plot_errorbar(x,y, dx, dy, xoffset, yoffset, text, ax):
6     ax.errorbar(x,y, xerr=dx, yerr=dy, marker='', linestyle =
7         '', elinewidth = .5, capthick=0.5, ecolor='k', capsizes=3)
8     ax.text(x + xoffset, y + yoffset, text)
9
10 myDataset1 = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
11     sheet_name='Supp_traces')
12
13 x = myDataset1.Zr
14 y = myDataset1.Th
15
16 dx = 60
17 dy = 7
18
19 errorbar_x = x.max() - x.min() * 0.1
20 errorbar_y = y.min() + y.max() * 0.1
21
22 fig, ax1 = plt.subplots()
23 ax1.scatter(x, y, marker='o', color='#4881e9', edgecolor='k',
24     alpha=0.8, label='Recent activity of the CFC')
25 plot_errorbar(errorbar_x, errorbar_y, dx, dy, dx/4, dy/4, r' $\sigma$ ', ax1)
26
27 ax1.legend(loc='upper left')
28 ax1.set_xlabel('Zr [ppm]')
29 ax1.set_ylabel('Th [ppm]')

```

Listing 10.8 Reporting errors in binary diagrams.

10.3 The Linearized Approach in Error Propagation

Using a linearized approximation (i.e., to a first-order Taylor series expansion) and assuming uncorrelated and statistically independent variables (i.e., the independent variables are not correlated with either the magnitude or error of any other parameter), the general formula for the error propagation assumes the form reported in Eq. 10.6 (Hughes & Hase, 2010; Taylor, 1997):

$$\sigma_z = \sqrt{\left(\frac{\partial z}{\partial a}\right)^2 (\sigma_a)^2 + \left(\frac{\partial z}{\partial b}\right)^2 (\sigma_b)^2 + \left(\frac{\partial z}{\partial c}\right)^2 (\sigma_c)^2 + \dots} \quad (10.6)$$

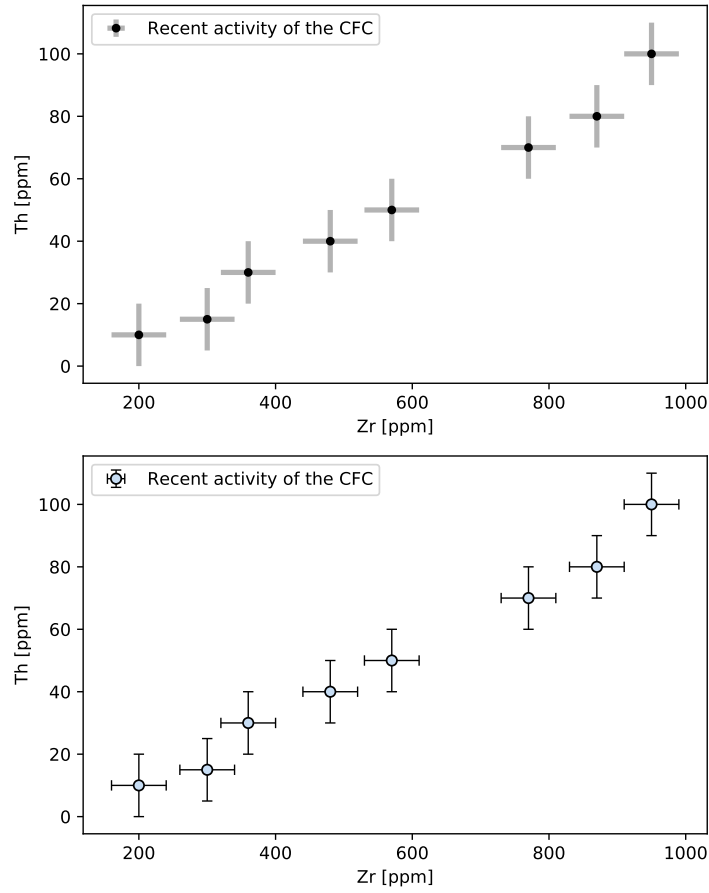


Fig. 10.7 The result of the code listing 10.7.

where z is a multi-variable function $z = f(a, b, c, \dots)$ depending on the measured variables $a \pm \sigma_a, b \pm \sigma_b, c \pm \sigma_c$, etc. Table 10.1 report the application of the Eq. 10.6 to some simple equations of common use that are often useful to solve geological problems. If correlations among the involved variables cannot be neglected (i.e., they are not independent), additional terms should be added. As an example, considering the function $z = f(x, y)$ that depends on measured quantities $x \pm \sigma_x$ and $y \pm \sigma_y$, with covariance σ_{xy} between x and y , the uncertainty in z is given by:

$$\sigma_z = \sqrt{\left(\frac{\partial z}{\partial x}\right)^2 (\sigma_x)^2 + \left(\frac{\partial z}{\partial y}\right)^2 (\sigma_y)^2 + 2 \frac{\partial z}{\partial x} \frac{\partial z}{\partial y} \sigma_{xy}} \quad (10.7)$$

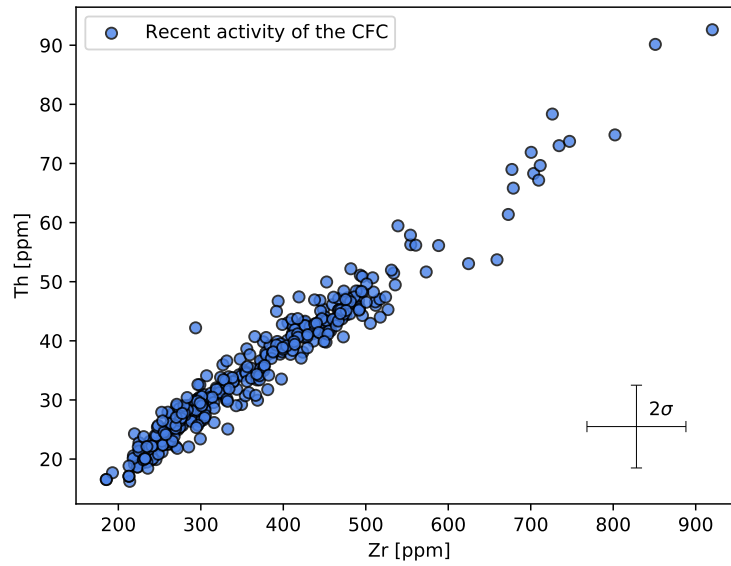


Fig. 10.8 The result of the code listing 10.8.

Table 10.1 Error propagation of some equations of common use that are often useful to solve geological problems, modified from Hughes and Hase, 2010.

Function, Z	Error	Function, Z	Error
$z = 1/a$	$\sigma_z = z^2 \sigma_a$	$z = \exp(a)$	$\sigma_z = z \cdot \sigma_a$
$z = \ln(a)$	$\sigma_z = \sigma_a/a$	$z = 10^a$	$\sigma_z = \sigma_a / [\ln(10) \cdot a]$
$z = a^n$	$\sigma_z = n \cdot a^{n-1} \cdot \sigma_a$	$z = \log_{10}(a)$	$\sigma_z = 10^a \cdot \ln(10) \cdot \sigma_a$
$z = \sin(a)$	$\sigma_z = \cos(a) \cdot \sigma_a$	$z = \cos(a)$	$\sigma_z = \sin(a) \cdot \sigma_a$
$z = a + b$	$\sigma_z = \sqrt{(\sigma_a)^2 + (\sigma_b)^2}$	$z = a - b$	$\sigma_z = \sqrt{(\sigma_a)^2 + (\sigma_b)^2}$
$z = a \cdot b$	$\sigma_z = z \cdot \sqrt{(\frac{\sigma_a}{a})^2 + (\frac{\sigma_b}{b})^2}$	$z = a/b$	$\sigma_z = z \cdot \sqrt{(\frac{\sigma_a}{a})^2 + (\frac{\sigma_b}{b})^2}$

Take in mind that the reported linearized approach, based on the first order truncation of Taylor series expansion, assumes that the magnitude of the error is small (Hughes & Hase, 2010; Taylor, 1997). As a consequence, it is only valid when the involved uncertainties are small enough (e.g., less than 10%, to provide a rough estimation, Hughes and Hase, 2010; Taylor, 1997).

In the simplest cases, you could develop and run python functions to propagate errors. The code listing 10.9 shows two practical examples (i.e., sum and division) based on the rules reported in Table 10.1.

```

1 import numpy as np
2
3 def sum_ab(a, b, sigma_a, sigma_b):
4     z = a + b
5     sigma_z = np.sqrt(sigma_a**2 + sigma_b**2)
6     return z, sigma_z
7
8 def division_ab(a, b, sigma_a, sigma_b):
9     z = a / b
10    sigma_z = z * np.sqrt((sigma_a/a)**2 + (sigma_b/b)**2)
11    return z, sigma_z

```

Listing 10.9 Example application of the rules reported in 10.1 for the sum and division.

Also, you could use the symbolic approach to solve the Eq. 10.6 or the Eq. 10.7. As an example, the code listing 10.10 uses SymPy to propagate errors solving the Eq. 10.6.

```

1 import sympy as sym
2
3 a, b, sigma_a, sigma_b = sym.symbols("a b sigma_a sigma_b")
4
5 def symbolic_error_prop(func, val_a, val_sigma_a, val_b=0,
6   val_sigma_b=0):
7
8     z = sym.lambdify([a,b],func, 'numpy')
9     sigma_z = sym.lambdify([a,b,sigma_a, sigma_b], sym.sqrt((sym.
10    diff(func,a)**2*sigma_a**2)+(sym.diff(func,b)**2*sigma_b**2))
11    , 'numpy')
12
13    val_z = z(a = val_a, b = val_b)
14    val_sigma_z = sigma_z(a = val_a, b = val_b, sigma_a =
15    val_sigma_a, sigma_b = val_sigma_b)
16
17    return val_z, val_sigma_z

```

Listing 10.10 Example application of the symbolic approach to solve the Eq. 10.6.

```

1 my_a = np.array([2,3,5,7,10])
2 my_sigma_a = np.array([0.2,0.3,0.4,0.7,0.9])
3 my_b = np.array([2,3,6,4,8])
4 my_sigma_b = np.array([0.3,0.3,0.5,0.5,0.5])
5
6 # errors propagated using custom functions
7 my_sum_ab_l, my_sigma_sum_ab_l = sum_ab(a = my_a, b = my_b,
    sigma_a = my_sigma_a, sigma_b = my_sigma_b)
8 my_division_ab_l, my_sigma_division_ab_l = division_ab(a=my_a,
    b = my_b, sigma_a = my_sigma_a, sigma_b = my_sigma_b)
9
10 # errors propagated using the symbolic approach
11 my_sum_ab_s, my_sigma_sum_ab_s = symbolic_error_prop(func=a+b,
    val_a= my_a, val_sigma_a = my_sigma_a, val_b= my_b,
    val_sigma_b = my_sigma_b)
12 my_division_ab_s, my_sigma_division_ab_s = symbolic_error_prop
    (func=a/b, val_a= my_a, val_sigma_a = my_sigma_a, val_b=
    my_b, val_sigma_b = my_sigma_b)
13
14 fig = plt.figure(figsize=(8,8))
15 ax1 = fig.add_subplot(2,2,1)
16 ax1.errorbar(x = my_a, y= my_sum_ab_l, xerr=my_sigma_a, yerr=
    my_sigma_sum_ab_l, linestyle='', marker = 'o', ecolor='k',
    elinewidth=0.5, capsize=1, label='Errors by the custom
    functions')
17 ax1.set_xlabel('a')
18 ax1.set_ylabel('a + b')
19 ax1.legend()
20 ax2 = fig.add_subplot(2,2,2)
21 ax2.errorbar(x = my_a, y= my_sum_ab_s, xerr=my_sigma_a, yerr=
    my_sigma_sum_ab_s, linestyle='', marker = 'o', ecolor='k',
    elinewidth=0.5, capsize=1, label='Errors by the symbolic
    approach')
22 ax2.set_xlabel('a')
23 ax2.set_ylabel('a + b')
24 ax2.legend()
25 ax3 = fig.add_subplot(2,2,3)
26 ax3.errorbar(x = my_a, y= my_division_ab_l, xerr=my_sigma_a,
    yerr=my_sigma_division_ab_l, linestyle='', marker = 'o',
    ecolor='k', elinewidth=0.5, capsize=1, label='Errors by
    custom function')
27 ax3.set_xlabel('a')
28 ax3.set_ylabel('a / b')
29 ax3.legend()
30 ax4 = fig.add_subplot(2,2,4)
31 ax4.errorbar(x = my_a, y= my_division_ab_s, xerr=my_sigma_a,
    yerr=my_sigma_division_ab_s, linestyle='', marker = 'o',
    ecolor='k', elinewidth=0.5, capsize=1, label='Errors by
    the symbolic approach')
32 ax4.set_xlabel('a')
33 ax4.set_ylabel('a / b')
34 ax4.legend()
35 fig.tight_layout()

```

Listing 10.11 Error propagation by custom functions reported in the code listing 10.9 and by solving the Eq. 10.6 by the symbolic approach (code listing 10.10).

Finally, the the code listing 10.11 and the Fig. 10.9 compare the results obtained by propagating errors by custom functions based on the rules reported in Tab. 10.1 and by the symbolic approach. As expected, the obtained results reported in Fig. 10.9 are identical.

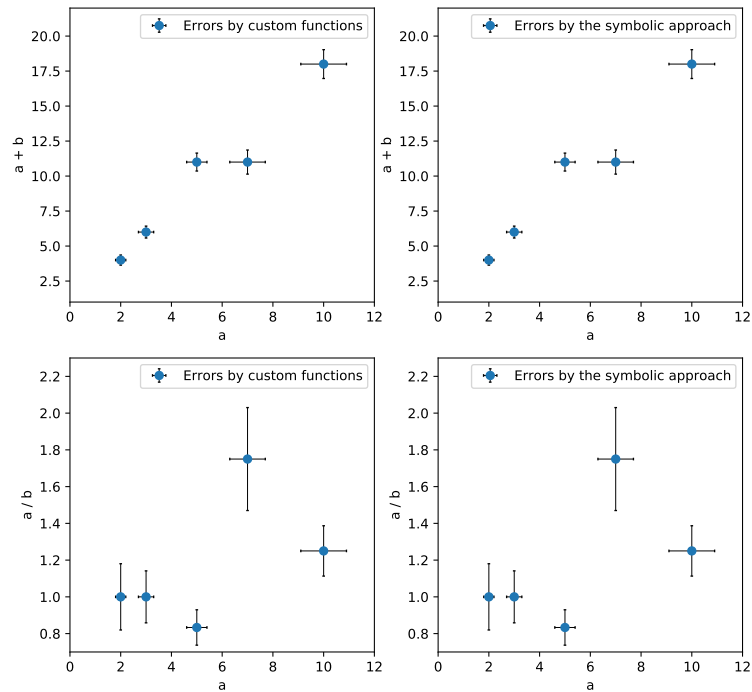


Fig. 10.9 The result of the code listing 10.11.

To provide a geological example, please consider plotting a Rb/Th ratio Vs. L_a for tephtras belonging to the recent volcanic activity of the Campi Flegrei Caldera using the linearized approach for error propagation (code listing 10.12 and Fig. 10.10).

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import sympy as sym
4
5 a, b, sigma_a, sigma_b = sym.symbols("a b sigma_a sigma_b")
6
7 def symbolic_error_prop(func, val_a, val_sigma_a, val_b=0,
8   val_sigma_b=0):
9     z = sym.lambdify([a,b],func, 'numpy')
10    sigma_z = sym.lambdify([a,b,sigma_a, sigma_b], sym.sqrt((
11      sym.diff(func,a)**2*sigma_a**2)+(sym.diff(func,b)**2*
12      sigma_b**2)), 'numpy')
13    val_z = z(a = val_a, b = val_b)
14    val_sigma_z = sigma_z(a = val_a, b = val_b, sigma_a =
15      val_sigma_a, sigma_b = val_sigma_b)
16
17    return val_z, val_sigma_z
18
19 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
20   sheet_name='Supp_traces')
21
22 ratio_y, sigma_ratio_y = symbolic_error_prop(a/b, val_a =
23   myDataset.Rb, val_sigma_a=myDataset.Rb * 0.1, val_b=
24   myDataset.Th, val_sigma_b=myDataset.Th*0.1)
25
26 myDataset['Rb_Th'] = ratio_y
27 myDataset['Rb_Th_1s'] = sigma_ratio_y
28
29 epochs = ['one', 'two', 'three', 'three-b']
30 colors = ['#afbbb5', '#f10e4a', '#27449c', '#f9a20e']
31
32 fig, ax = plt.subplots()
33 for epoch, color in zip(epochs, colors):
34     myData = myDataset[(myDataset.Epoch == epoch)]
35     ax.errorbar(x = myData.La, y= myData.Rb_Th, xerr=myData.La
36       * 0.1, yerr= myData.Rb_Th_1s, linestyle='',
37       markerfacecolor= color, markersize=6, marker='o',
38       markeredgecolor='k', ecolor=color, elinewidth=0.5, capsize
39       =0, label="Epoch " + epoch)
40
41 ax.legend(title='CFC Recent Activity')
42 ax.set_ylabel('Rb/Th')
43 ax.set_xlabel('La [ppm]')

```

Listing 10.12 Rb/Th ratio Vs. La for tephra belonging to the recent volcanic activity of the Campi Flegrei Caldera. Error propagated using the linearized approach.

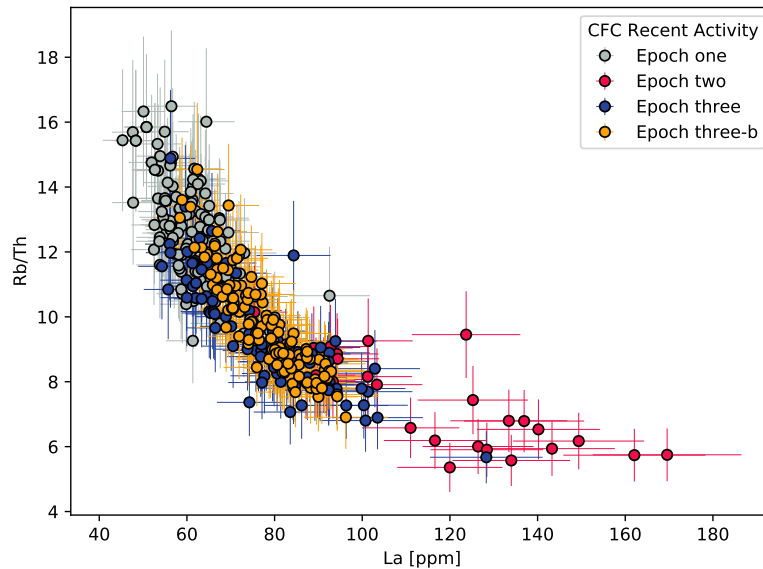


Fig. 10.10 The result of the code listing 10.12.

10.4 The Monte Carlo Approach in Error Propagation

Monte Carlo (MC) numerical modelling, named after the casino tradition in the Principality of Monaco, simulates complex probabilistic events using simple random events (Barbu & Zhu, 2020). In detail, MC methods rely on true-random (TRNG) or pseudo-random number generators (PRNG) to produce sample distributions simulating a target probability density function (Barbu & Zhu, 2020; Johnston, 2018).

What's the difference between TRNGs and PRNGs? TRNGs refer to devices, generally hardware based, producing real (i.e., non-deterministic) random numbers (Johnston, 2018). On the contrary, PRNGs are deterministic algorithms that generate a "random looking" sequence of numbers (Johnston, 2018). However, given the same starting conditions (i.e., same seeding), a PRNG will always return the same sequence of numbers (Johnston, 2018).

In NumPy 1.19, the default PRNG that provide the random sampling of a wide range of distributions (e.g., uniform, normal, etc..) is the PCG64¹. It is a 128-bit implementation of O'Neill's permutation congruential generator (O'Neill, 2014).

¹ <https://www.pcg-random.org>


```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def normalPDF(x, mu, sigma):
5     PDF = 1/(sigma * np.sqrt(2 * np.pi)) * np.exp( - (x - mu)
6         **2 / (2 * sigma**2))
7     return PDF
8
9 def unifromPDF(x, a, b):
10    PDF = np.piecewise(x, [(x>=a) & (x<=b), (x<a) & (x>b)],
11        [1/(b-a), 0])
12    return PDF
13
14 # Random sampling of a normal distribution
15 my_mu, my_sigma = 0, 0.1 # mean and standard deviation
16 sn = np.random.default_rng().normal(loc = my_mu, scale =
17     my_sigma, size = 10000)
18 fig = plt.figure()
19 ax1 = fig.add_subplot(2,1,1)
20 ax1.hist(sn, density = True, bins='auto', edgecolor = 'k',
21     color = '#c7ddf4', label = 'Random Sampling of the Normal
22     Distribution')
23 my_xn = np.linspace(my_mu - 4 * my_sigma, my_mu + 4 * my_sigma
24     , 1000)
25 my_yn = normalPDF(x= my_xn, mu = my_mu, sigma = my_sigma)
26 ax1.plot(my_xn,my_yn,linewidth=2,linestyle='--',color='#ff464a
27     ',label='Target Normal Probability Density Function')
28 ax1.set_ylim(0.0, 7.0)
29 ax1.set_xlabel('x')
30 ax1.set_ylabel('Prob. Density')
31 ax1.legend()
32
33 # Random sampling of a uniform distribution
34 my_a, my_b = -1, 1 # lower and upper bound of the uniform
35     distribution
36 su = np.random.default_rng().uniform(low = my_a, high = my_b,
37     size = 10000)
38 ax2 = fig.add_subplot(2,1,2)
39 ax2.hist(su, density = True, bins='auto', edgecolor = 'k',
40     color = '#c7ddf4', label = 'Random Sampling of the Uniform
41     Distribution')
42 my_xu = np.linspace(-2, 2, 1000)
43 my_yu = unifromPDF(x = my_xu, a = my_a, b = my_b)
44 ax2.plot(my_xu, my_yu, linewidth=2, linestyle='--', color='#
45     ff464a', label = 'Target Uniform Probability Density
46     Function')
47 ax2.set_ylim(0, 1)
48 ax2.set_xlabel('x')
49 ax2.set_ylabel('Prob. Density')
50 ax2.legend()
51
52 fig.tight_layout()

```

Listing 10.13 Random Sampling of a Normal and a Uniform distribution.

The PCG-64 has a period of 2^{128} and supports advancing an arbitrary number of steps as well as 2^{127} streams².

To provide an example on how to perform a random sampling of specific PDFs, the code listing 10.13, shows how to generate a random sequence of numbers (i.e., a random sample distribution) simulating a normal and uniform PDFs, respectively. In detail, the code listing 10.13 uses the `np.random.default_rng()` statement (i.e., line 17), based on the PCG64 PRNG (O’Neill, 2014).

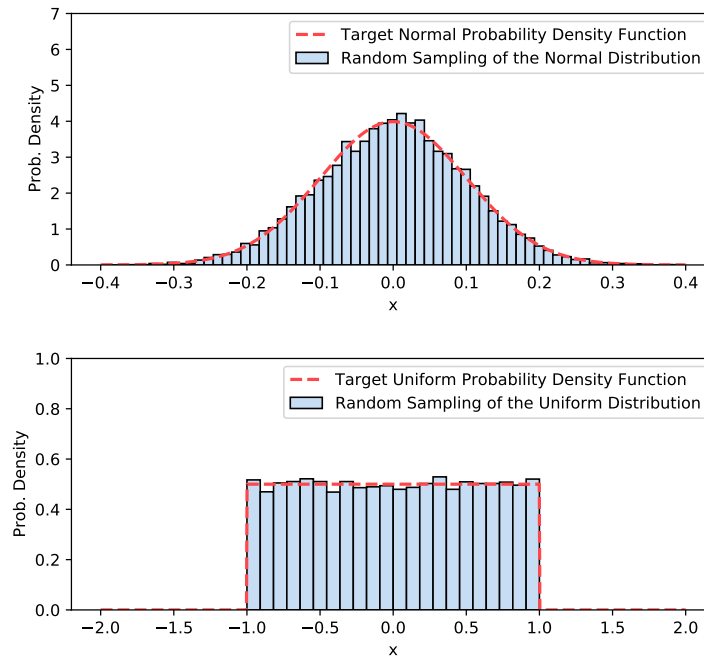


Fig. 10.11 The result of the code listing 10.13.

The other PRNGs currently available in Numpy are reported in Table 10.2.

The code listing 10.14 show how to use a different PRNG than the PCG64 to define the same normal distribution of Fig. 10.11 characterized by a mean (μ) and a standard deviation (σ) of 0 and 0.1, respectively. The results of code listing 10.14 is reported in Fig. 10.12.

² https://numpy.org/doc/stable/reference/random/bit_generators/

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4
5 def normalPDF(x, mu, sigma):
6     PDF = 1/(sigma * np.sqrt(2 * np.pi)) * np.exp( - (x - mu)
7         **2 / (2 * sigma**2))
8     return PDF
9
10 fig = plt.figure(figsize=(6,9))
11
12 # Random sampling of a normal distribution
13 my_mu, my_sigma = 0, 0.1 # mean and standard deviation
14
15 BitGenerators = [np.random.MT19937(), np.random.Philox(), np.
16     random.SFC64()]
17 Names = ['Mersenne Twister PRNG (MT19937)', 'Philox (4x64)
18     PRNG (Philox)', 'Chris Doty-Humphrey\'s SFC PRNG (SFC64)']
19 Indexes = [1,2,3]
20
21 for bit_generator, name, index in zip(BitGenerators,Names,
22     Indexes):
23     sn = np.random.Generator(bit_generator).normal(loc = my_mu
24         , scale = my_sigma, size = 10000)
25     ax = fig.add_subplot(3, 1, index)
26     ax.hist(sn, density = True, bins='auto', edgecolor = 'k',
27         color = '#c7ddf4', label = name)
28     my_xn = np.linspace(my_mu - 4 * my_sigma, my_mu + 4 *
29         my_sigma, 1000)
30     my_yn = normalPDF(x= my_xn, mu = my_mu, sigma = my_sigma)
31     ax.plot(my_xn, my_yn, linewidth=2, linestyle='--', color='
32         #ff464a', label = 'Target Normal PDF')
33     ax.set_ylim(0.0, 7.0)
34     ax.set_xlim(my_mu - 6 * my_sigma, my_mu + 6 * my_sigma)
35     ax.set_xlabel('x')
36     ax.set_ylabel('Probability Density')
37     ax.legend()
38
39 fig.tight_layout()

```

Listing 10.14 Random Sampling of a Normal distribution using different PRNGs.

Please note that for most of the basic everyday tasks in geological modelling (e.g., basic error propagation), all the PRNGs reported in Table 10.2 work satisfactorily, so I suggest to use the default one for simplicity of use.

In error propagation, the MC approach is a proficient technique to be considered when the Eq. 10.6 or its corrected forms (e.g., Eq. 10.7) are inconvenient (Schwartz, 1975).

Table 10.2 Pseudo Random Number Generators (PRNG) currently (ver. 1.19) available in Numpy.

PRNG	Reference	Description
PCG64	(O’Neill, 2014)	128-bit implementation of O’Neill’s permutation congruential generator
MT19937	(Haramoto et al., 2008)	Mersenne Twister pseudo-random number generator
Philox	(Salmon et al., 2011)	A 64-bit counter-based PRNG using weaker (and faster) versions of cryptographic functions
SFC64	http://pracrand.sourceforge.net	Implementation of Chris Doty-Humphrey’s Small Fast Chaotic PRNG

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def gaussian(x, mean, std):
5     return 1/(np.sqrt(2*np.pi*std**2))*np.exp(-0.5*(((x - mean)
6         )**2)/(std**2))
7
8 my_a, my_sigma_a = 40, 8
9 my_b, my_sigma_b = 20, 2
10
11 N = 10000
12 a_normal = np.random.default_rng().normal(my_a, my_sigma_a, N)
13 b_normal = np.random.default_rng().normal(my_b, my_sigma_b, N)
14
15 # Linearized Method
16 my_sum_ab_l, my_sigma_sum_ab_l = sum_ab(a = my_a, b = my_b,
17     sigma_a = my_sigma_a, sigma_b = my_sigma_b)
18 my_x = np.linspace(20,100,1000)
19 my_sum_ab_PDF = gaussian(x = my_x, mean= my_sum_ab_l, std =
20     my_sigma_sum_ab_l)
21
22 # Monte Carlo estimation
23 my_sum_ab_mc = a_normal + b_normal
24 my_sum_ab_mc_mean = my_sum_ab_mc.mean()
25 my_sigma_sum_ab_mc_std = my_sum_ab_mc.std()
26
27 fig, ax = plt.subplots()
28 ax.hist(my_sum_ab_mc, bins='auto', color='#c7ddf4', edgecolor=
29     'k', density=True, label= r'a+b sample distribution by MC
30     ($\mu_{a+b} = $' + "{:.0f}".format(my_sum_ab_mc_mean) + r'
31     - 1$\sigma_{a+b}$' + "{:.0f}".format(
32     my_sigma_sum_ab_mc_std) + ')')
33 ax.plot(my_x, my_sum_ab_PDF, color='#ff464a', linestyle='--',
34     label=r'a+b PDF by linearized error propagation')
35 ax.set_xlabel('a + b')
36 ax.set_ylabel('Probability Density')
37 ax.legend(title='Error Propagation')
38 ax.set_ylim(0,0.07)

```

Listing 10.15 Error propagation by MC.

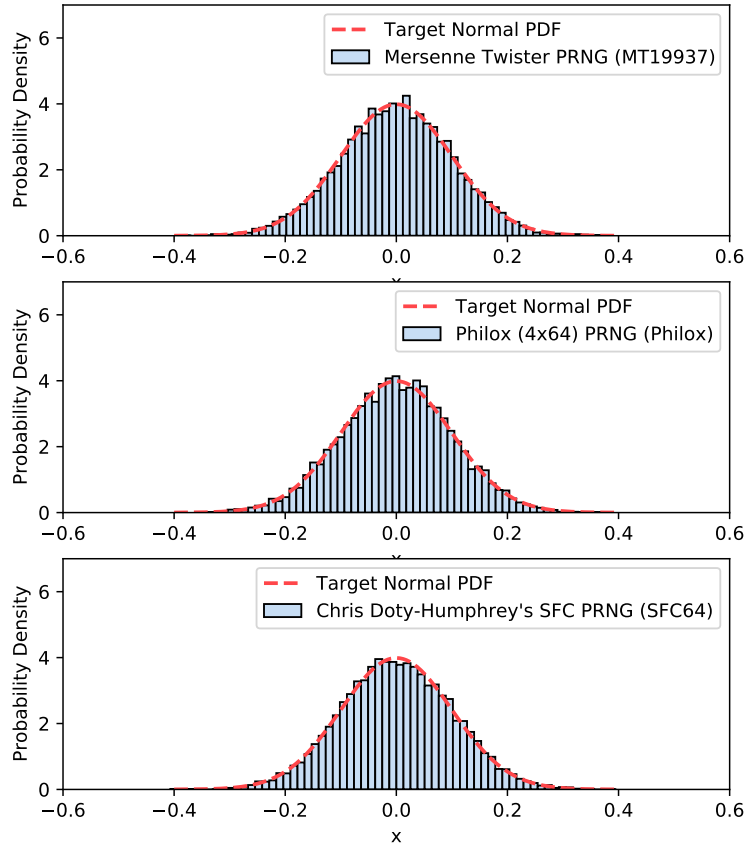


Fig. 10.12 The result of the code listing 10.14.

Please remember that the application of the Eq. 10.6 bases on these strong assumptions (Schwartz, 1975): (a) the involved errors are statistically uncorrelated; (b) the involved variables are independent; and (c), the errors must be sufficiently small relative to the corresponding mean values. A more difficult problem arises when the derivative elements in Eq. 10.6 or Eq. 10.7 can be solved only with a great effort or perhaps not at all (Schwartz, 1975). This problem, however, could be attacked by numerical methods, e.g., by Monte Carlo error propagation (Schwartz, 1975). To provide a detailed description of the MC method is far beyond the scopes of the present introductory book. Here I limit the discussion to a very simple case, i.e., the sum of two variables affected by errors with a normal distribution (code

listing 10.15 and Fig. 10.13). The reported example highlights the power and the simplicity of the MC approach in error propagation. In detail the code listing 10.15 show that, after defining a sample distribution for each parameter (i.e., lines 14 and 15), the error propagation by MC can be easily performed in one line of code (i.e., line 25) without applying any additional equation than the one of interest, the sum in our case.

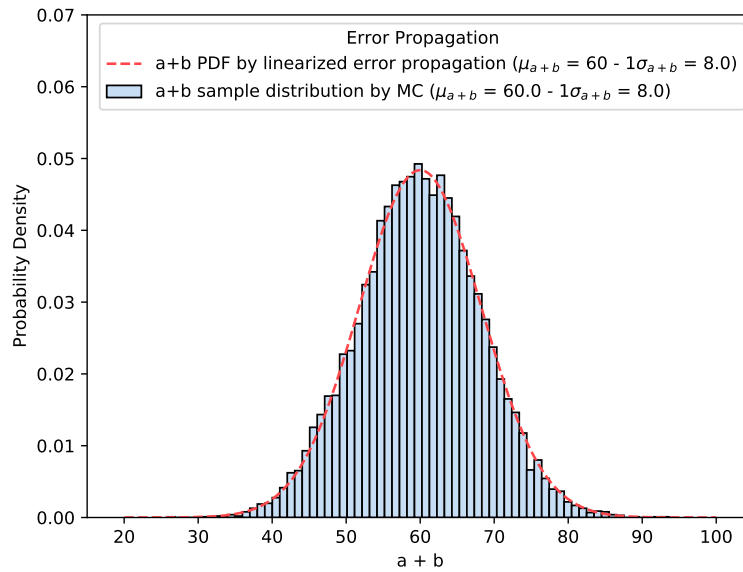


Fig. 10.13 The result of the code listing 10.15.

Part V
Robust Statistics and Machine Learning

Chapter 11

Introduction to Robust Statistics

11.1 Classical and Robust Approaches to Statistics

All statistical methods and techniques bases explicitly or implicitly on some assumptions (Huber & Ronchetti, 2009; Maronna et al., 2006). Among these, the assumption that the observed (i.e., sampled) data follow a normal (Gaussian) distribution is widely adopted (Huber & Ronchetti, 2009; Maronna et al., 2006). This assumption is the basis for all the classical methods in regression, analysis of variance, and multivariate analysis. However, it often happens, and this is true for many geological cases, that a sample of data mostly follows a normal distribution, but some observations are not normally distributed, defining a different pattern. Such atypical data are called outliers. Please note that a single outlier can strongly distort statistical method based non the assumption of normality (e.g. the king-kong effect in linear regression). Also, if the data are assumed to be normally distributed but their actual distribution starts diverging from a Gaussian shape, then classical tests may return unreliable results (Huber & Ronchetti, 2009; Maronna et al., 2006). Definition: "The robust approach to statistical modeling and data analysis aims at deriving methods that produce reliable parameter estimates and associated tests and confidence intervals, not only when the data follow a given distribution exactly, but also when this happens only approximately in the sense just described" (Maronna et al., 2006). Please note that a 'robust' model should converge to the results of classical methods in the case of the assumption behind them (e.g., normal distribution) are satisfied. A complete treatment of robust statistics is far beyond the scope of the present chapter, and I suggest more specific sources for the interested reader. In the following, I will focus on 1) how to check if a sample is normally distributed (i.e., normality tests; 2) robust descriptive statistics and 3) robust linear regression; 4) application of robust statistics in geochemistry.

11.2 Normality Tests

There is not a standard procedure to affirm that a sample follows (or not follows) a normal distribution. However, a reasonable procedure consists of: 1) a preliminary qualitative inspection of the histogram plot then fitted by normal PDF (see section 9.5); 2) a subsequent inspection of a Quantile-Quantile plot, 3) the application of selected statistical tests of normality (Thode, 2002).

Please note that a reasonably large number of observations is needed to detect deviations from normality (Huber & Ronchetti, 2009; Maronna et al., 2006).

Histogram plots and parametric fitting

As reported in section 9.5, reporting the probability density histogram is an easy and efficient way to qualitatively inspect the shape of a sample distribution. In the case of a normal distribution, we expect a symmetric, bell-shaped aspect of the resulting histogram plot. Then, performing a parametric fitting by a normal PDF, we can better evaluate the similarities and the discrepancies between the studied sample, and a normal distribution characterized by the same mean and standard deviation.

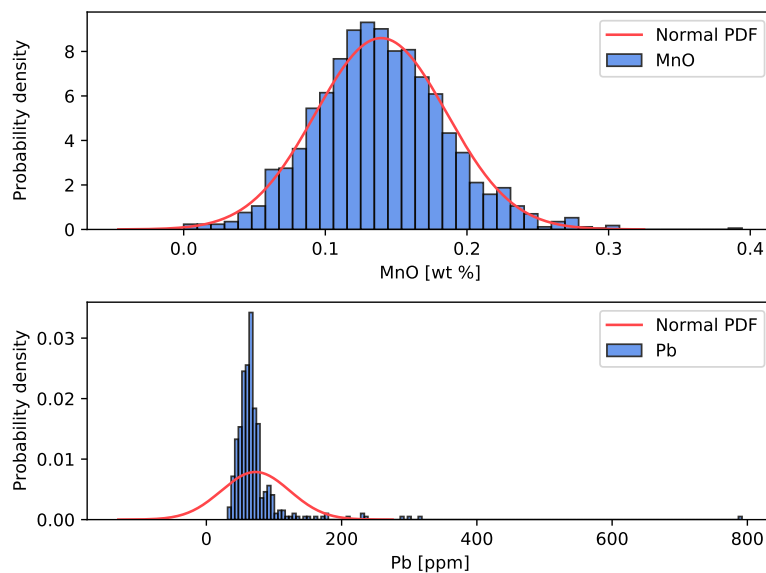


Fig. 11.1 The result of the code listing 11.1.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 from scipy.stats import norm
4 import numpy as np
5
6 myDataset_majors = pd.read_excel('Smith_glass_post_NYT_data.
    xlsx', sheet_name='Supp_majors', engine='openpyxl')
7 myDataset_traces = pd.read_excel('Smith_glass_post_NYT_data.
    xlsx', sheet_name='Supp_traces', engine='openpyxl')
8
9 fig = plt.figure()
10
11 # MnO
12 MnO = myDataset_majors.MNO
13
14 ax1 = fig.add_subplot(2, 1, 1)
15 ax1.hist(MnO, bins= 'auto', density = True, color='#4881e9',
    edgcolor='k', label='MnO', alpha=0.8)
16 a_mean = MnO.mean()
17 std_dev = MnO.std()
18 x = np.linspace(a_mean-4*std_dev, a_mean+4*std_dev,1000)
19 pdf = norm.pdf(x, loc=a_mean, scale=std_dev)
20 ax1.plot(x, pdf, linewidth=1.5, color='#ff464a',label='Normal
    PDF')
21 ax1.set_xlabel('MnO [wt %]')
22 ax1.set_ylabel('Probability density')
23 ax1.legend()
24
25 #Pb
26 Pb = myDataset_traces.Pb
27 Pb = Pb.dropna(how='any')
28 ax2 = fig.add_subplot(2, 1, 2)
29 ax2.hist(Pb, bins= 'auto', density = True, color='#4881e9',
    edgcolor='k', label='Pb', alpha=0.8)
30 a_mean = Pb.mean()
31 std_dev = Pb.std()
32 x = np.linspace(a_mean-4*std_dev, a_mean+4*std_dev,1000)
33 pdf = norm.pdf(x, loc=a_mean, scale=std_dev)
34 ax2.plot(x, pdf, linewidth=1.5, color='#ff464a', label='Normal
    PDF')
35 ax2.set_xlabel('Pb [ppm]')
36 ax2.set_ylabel('Probability density')
37 ax2.legend()
38
39 fig.align_ylabels()
40 fig.tight_layout()

```

Listing 11.1 Plotting the histogram distribution and making a parametric fitting to start assessing the normality of a sample distribution.

As an example, looking at the MnO and Pb distributions of the data set reported in (Smith et al., 2011), we can easily observe a strong departure from a normal

distribution for Pb that is characterized by a right tail (i.e., positive skewness) and a strong outlier at about 790 ppm (code listing 11.1 and figure 11.1). On the contrary, the MnO probability density histogram is almost symmetric with no evidence for the presence of outliers with the exception of a single observation a about 0.39 wt %.

The parametric fitting of the two distributions with a gaussian PDF (code listing 11.1 and figure 11.1), confirms a strong departure from the normality for Pb, and quite good fit for MnO.

Being a qualitative inspection, the plotting of a density histogram and the parametric fitting by a normal distribution allow the recognition of strong departures from the normality. As a consequence, we can certain exclude the normality for the Pb sample. On the contrary, we cannot affirm that MnO is normally distributed, yet (Thode, 2002).

Quantile-quantile plots

A successive step in the investigation for the normality of a sample distribution could consist in the fulfilment of a Quantile-quantile plot (Palettas, 1992). The Quantile-quantile (Q-Q) plot is a graphical representation used to decide if two data sets come from populations characterized by the same distribution. When used to test for the normality of a sample distribution, one of the two data sets is the investigated sample, the second data set is derived by a normal PDF. In detail, we develop a binary diagram where the quantiles of the investigated data set are plotted against the quantiles of a normal distribution.

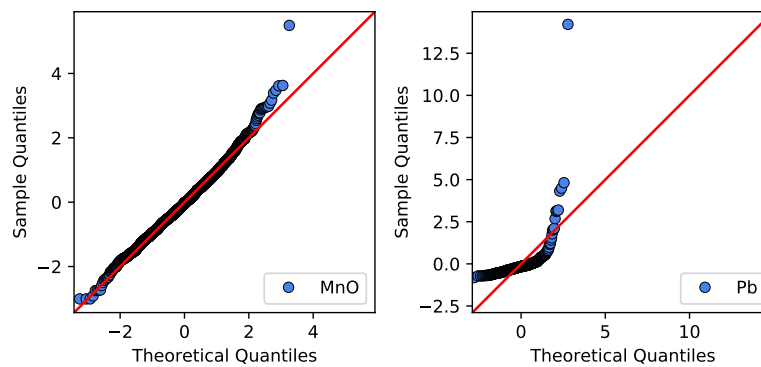


Fig. 11.2 The result of the code listing 11.2.

```
1 import statsmodels.api as sm
2
3 fig = plt.figure()
4
5 ax1 = fig.add_subplot(1, 2, 1)
6 sm.qqplot(data= MnO, fit = True, line="45", ax=ax1,
7           markerfacecolor='#4881e9', markeredgewidth='0.5',
8           markeredgewidth='0.5', markeredgewidth='0.5',
9           markeredgewidth='0.5', label='MnO')
10 ax1.set_aspect('equal', 'box')
11 ax1.legend(loc='lower right')
12
13 ax2 = fig.add_subplot(1, 2, 2)
14 sm.qqplot(data= Pb, fit = True, line="45", ax=ax2,
15           markerfacecolor='#4881e9', markeredgewidth='0.5',
16           markeredgewidth='0.5', markeredgewidth='0.5',
17           markeredgewidth='0.5', label='Pb')
18 ax2.set_aspect('equal', 'box')
19 ax2.legend(loc='lower right')
20
21 fig.tight_layout()
```

Listing 11.2 Q-Q diagrams for MnO and Pb.

If the investigated data set come from a population with a normal distribution, the standardized quantiles (i.e., derived after subtracting mean and dividing by the standard deviation) should fall approximately along a 1:1 reference line.

The larger is the departure from this reference line, the greater is the evidence that the investigated data set does not come from a normal population. As an example, Fig. 11.2 (code listing 11.2) reports two Q-Q plots, evaluating the MnO and Pb samples, respectively. As expected, the Q-Q plot for Pb departs strongly from the accordance with the reference line, demonstrating further the non-normality of the sample. Inspecting the Q-Q plot for MnO, it emerges that the sample quantiles are mostly in agreement with the theoretical ones. However, at least one observation in the Q-Q plot departs from linearity. It correspond to the extreme right size of the distribution (i.e., the outliers above 0.29 wt % observed in Fig. 11.1). Can we assume that MnO follows a normal distribution? To answer, we are going to perform further statistical tests.

Statistical tests

Typically, a statistical test for normality initially assumes that the sample derives from a normal (Gaussian) population (Thode, 2002). This initial assumption is the so-called null hypothesis (H_0). Then, the test elaborates data and returns one or more statistical parameters and one or more threshold values to evaluate the if it is possible to accept or not H_0 (Thode, 2002).

The Shapiro-Wilk (S-W) test is a statistical procedure for testing sample data set for normality (Shapiro & Wilk, 1965). In detail, the S-W test relies on the

determination of the W parameter that is obtained by dividing the square of an appropriate linear combination of the sample order statistics by the usual symmetric estimate of variance (Shapiro & Wilk, 1965). The maximum value of W is 1, corresponding to a normal distribution. Hence, the closer W is to one, the more your sample approach to a normal distribution. Small values for W indicate that your sample is not normally distributed. In the practice, you can reject the null hypothesis that your population is normally distributed if the W -value is under a certain threshold.

The D'Agostino and Pearson's (DA-P) test evaluates two descriptive statistics, i.e., the skewness and kurtosis, to produce a test of normality (R. D'Agostino & Pearson, 1973; R. B. D'Agostino, 1971). In detail, it estimates a statistical parameter, i.e., the p -value, combining the two metrics to quantify the divergence from a Gaussian distribution (R. D'Agostino & Pearson, 1973; R. B. D'Agostino, 1971). As in the case of the S-W test, you can reject the null hypothesis that your population is normally distributed if the p -value is under a certain threshold.

The Anderson-Darling (A-D) test is a modification of the Kolmogorov-Smirnov (K-S) test (Stephens, 1974). It returns a statistics, i.e., a series of computed values, and a list of critical values rather than a single p -value as in the case of the DA-P test. If the returned statistic is larger than reference critical values, then for the corresponding significance level, the null hypothesis that the data come from the chosen distribution, i.e, normal in our case, can be rejected (Stephens, 1974).

The code listing 11.3 highlights how to perform S-W, DA-P, and A-D tests in Python for a geological data set.

```

1 def returns_NormalTests(my_data):
2
3     from scipy.stats import shapiro, anderson, normaltest
4
5     print('-----')
6     print('')
7     stat, p = shapiro(my_data)
8     alpha = 0.05
9     if p > alpha:
10        print('Shapiro test fails to reject H0: looks normal :)')
11    else:
12        print('Shapiro test rejects H0: not normal :(')
13    print('')
14    stat, p = normaltest(my_data)
15    alpha = 0.05
16    if p > alpha:
17        print("D'Agostino and Pearson's test fails to reject H0:
18        looks normal :)")
19    else:
20        print("D'Agostino and Pearson's test rejects H0: not normal
21        :(")
22    print('')
23    result = anderson(my_data)
24    print('Anderson-Darling test:')

```

```

23     for sl, cv in zip(result.significance_level, result.
24         critical_values):
25         if result.statistic < cv:
26             print('%.3f: fails to reject H0: Sample looks normal :)'
27                 % (sl))
28         else:
29             print('%.3f: rejects H0: Sample does not look normal :(
30                 % (sl))
31             print('-----')
32             print('')
33 # Original MnO sample
34 print('Original MnO sample')
35 returns_NormalTests(MnO)
36 # Removing the outliers above 0.27 wt %
37 print('MnO sample without observations above 0.27 wt %')
38 MnO_no_outliers = MnO[MnO < 0.27]
39 returns_NormalTests(MnO_no_outliers)
40 ''' Results:
41 Original MnO sample
42 -----
43 Shapiro test rejects H0: not normal :(
44 D'Agostino and Pearson's test rejects H0: not normal :(
45 Anderson-Darling test:
46 15.000: rejects H0: Sample does not look normal :(
47 10.000: rejects H0: Sample does not look normal :(
48 5.000: rejects H0: Sample does not look normal :(
49 2.500: rejects H0: Sample does not look normal :(
50 1.000: rejects H0: Sample does not look normal :(
51 -----
52 MnO sample without observations above 0.27 wt %
53 -----
54 Shapiro test fails to reject H0: looks normal :)
55 D'Agostino and Pearson's test fails to reject H0: looks normal :)
56 Anderson-Darling test:
57 15.000: fails to reject H0: Sample looks normal :)
58 10.000: fails to reject H0: Sample looks normal :)
59 5.000: fails to reject H0: Sample looks normal :)
60 2.500: fails to reject H0: Sample looks normal :)
61 1.000: fails to reject H0: Sample looks normal :)
62 -----
63 '''
64
65
66
67
68
69
70

```

Listing 11.3 Performing Statistical Tests of Normality for the MnO sample.

11.3 Robust Estimators for Location and Scale

In chapter 5, we reviewed the classical estimators of location and scale (or spread) for a sample distribution. They are the building blocks of descriptive statistics. Examples are the sample mean and standard deviation as estimators for the location and scale, respectively. However, they may fail in the presence of outliers. In these cases, robust estimators are a better choice (Huber & Ronchetti, 2009; Maronna et al., 2006). In the following I provide a light introduction to robust estimators for the location and the scale of univariate sample distributions with their implementation in Python. I suggest the reading of more specialized book for the readers interested in a deeper treatment of the topic (Huber & Ronchetti, 2009; Maronna et al., 2006).

Robust and weak estimators for the location

Among the classical estimators for the location, the arithmetic mean is the most used and widely recognized (cf. Chapter 5). However, the arithmetic mean is strongly affected by the presence of outliers (Huber & Ronchetti, 2009; Maronna et al., 2006). As an example, looking at the Pb distribution in the data set reported in (Smith et al., 2011), we see a positive tail and a strong outlier at 790 ppm (Fig. 11.3). The arithmetic means for Pb is 81 ppm, and it falls at an higher value than most observations, ranging between 50 and 80 ppm (Fig. 11.3). This evidence is the result of the strong influence of positive outliers on the arithmetic mean. As a consequence, we can say that the arithmetic mean is a weak estimator for the location in the presence of outliers. On the contrary, the median value is located at 67 ppm (Fig. 11.3), centered on the interval containing most observations (i.e. 50-80 ppm) and corresponding to the modal bin in Fig. 11.3. This is because, the median is less affected by outliers than the arithmetic mean, resulting a robust estimator for the location in the presence of outliers.

Another approach to provide a robust estimation for the location of a sample distribution is by the Trimmed mean. It consists in defining a criterion to discard a proportion of the largest and smallest values as follow (Huber & Ronchetti, 2009; Maronna et al., 2006): let $\alpha \in [0, 1/2]$ and $m = [n\alpha]$ where $[.]$ stands for the integer part and n for the total number of observations. We define the α -trimmed mean as (Huber & Ronchetti, 2009; Maronna et al., 2006):

$$\mu_\alpha = \bar{z}_\alpha = \frac{1}{n - 2m} \sum_{i=m+1}^{n-m} z_{(i)} \quad (11.1)$$

where $z_{(i)}$ denotes the ordered observations. The limit cases $\alpha = 0$ and $\alpha \rightarrow 0.5$ correspond to the sample mean and median, respectively.

The α -Winsorized mean is similar to the α -trimmed mean but, instead of deleting extreme values as in the trimmed mean, it shifts them towards the bulk of the data (Eq. 11.2).


```

1 import pandas as pd
2 import numpy as np
3 from scipy.stats.mstats import winsorize
4 from scipy.stats import trim_mean
5 import matplotlib.pyplot as plt
6
7 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
8     sheet_name=1, engine='openpyxl')
9
10 el = 'Pb'
11 mySubDataset = myDataset[myDataset.Epoch == 'three-b']
12 mySubDataset = mySubDataset.dropna(subset=[el])
13
14 fig, ax = plt.subplots()
15 a_mean = mySubDataset[el].mean()
16 median = mySubDataset[el].median()
17 trimmed_mean = trim_mean(mySubDataset[el], proportiontocut
18     =0.1)
19 winsorized_mean = np.mean(winsorize(mySubDataset[el], limits
20     =0.1))
21
22 delta = 100 * (a_mean - median) / median
23
24 bins = np.arange(50,240,5)
25 ax.hist(mySubDataset[el], density = True, edgecolor='k',
26     color='#4881e9', bins=bins, label = 'Lead (Pb), Epoch
27     Three')
28 ax.axvline(a_mean, color = '#ff464a', linewidth = 2, label = '
29     Arithmetic Mean: {:.0f} [ppm]'.format(a_mean))
30 ax.axvline(median, color = '#ebb60d', linewidth = 2, label = '
31     Median: {:.0f} [ppm]'.format(median))
32 ax.axvline(trimmed_mean, color = '#8f10b3', linewidth = 2,
33     label = r'Trimmed Mean ( $\alpha = 0.1$ ):' + '{:.0f} [ppm]'
34     .format(trimmed_mean))
35 ax.axvline(winsorized_mean, color = '#07851e', linewidth = 2,
36     label = r'Winsored Mean ( $\alpha = 0.1$ ):' + '{:.0f} [ppm]'
37     .format(winsorized_mean))
38
39 ax.set_xlabel(el + " [ppm]")
40 ax.set_ylabel('probability density')
41 ax.legend()
42 ax.annotate('Large outlier at about 800 ppm', (240, 0.02),
43     (220, 0.02), ha="right", va="center", size=9, arrowprops=
44     dict(arrowstyle='fancy'))
45 ax.annotate('Deviation of the arithmetic\mean from the median
46     : {:.1f} %'.format(delta), (a_mean + 3, 0.03), (a_mean +
47     25, 0.03), ha="left", va="center", size=9, arrowprops=dict
48     (arrowstyle='fancy'))

```

Listing 11.4 Weak and robust estimates for the location.

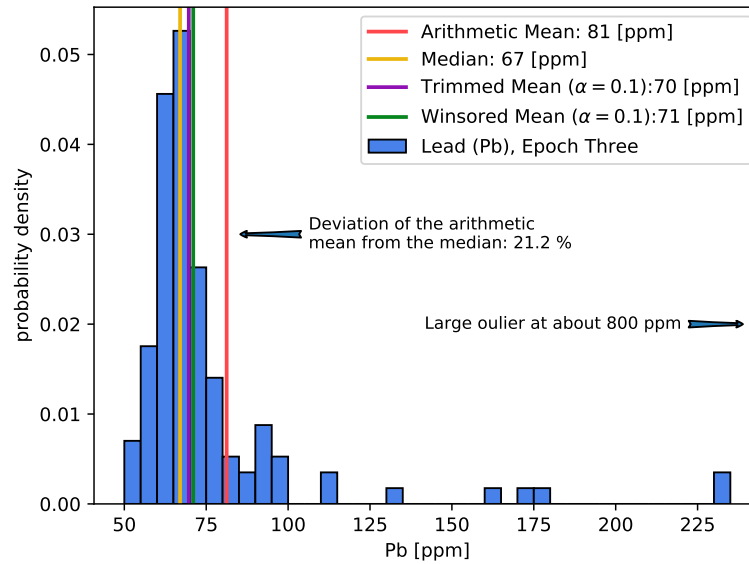


Fig. 11.3 The result of the code listing 11.4.

$$\mu_{wins} = \frac{1}{n} \left(mz_{(m)} + mz_{(n-m+1)} + \sum_{i=m+1}^{n-m} z_{(i)} \right) \quad (11.2)$$

where m and $z_{(i)}$ are defined as for the trimmed mean (Eq. 11.1). In Python the trimmed and Winsorized can be estimated easily using the `trim_mean()` and `winsorize()` methods in `scipy.stats` and `scipy.stats.mstats`, respectively (code listing 11.4 and Fig. 11.3).

Robust and weak estimators for the scale

As for the location, in the Chapter 5 we reviewed the main estimators for the scale of a distribution. Among them, the weaker is the range (Fig. 11.4). Also, the standard deviation is strongly affected by the presence of outliers (Fig. 11.4). Among the estimators for the scale reported in the Chapter 5 is the Inter Quartile Range (IQR; Fig. 11.4). Here, I am going to introduce an additional robust estimator for the scale of a sample distribution named the median absolute deviation about the median (MAD), and defined as (Eq. 11.3):

$$MAD(\mathbf{z}) = MAD(z_1, z_2, \dots, z_n) = Me \{ |\mathbf{z} - Me(\mathbf{z})| \}. \quad (11.3)$$

```

1 import pandas as pd
2 import numpy as np
3 from scipy import stats
4 import matplotlib.pyplot as plt
5
6 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
7     sheet_name=1, engine='openpyxl')
8 el = 'Pb'
9 mySubDataset = myDataset[myDataset.Epoch == 'three-b']
10 mySubDataset = mySubDataset.dropna(subset=[el])
11
12 a_mean = mySubDataset[el].mean()
13 median = mySubDataset[el].median()
14 range_values = [mySubDataset[el].min(), mySubDataset[el].max()]
15 std_dev_values = [a_mean - mySubDataset[el].std(), a_mean +
16     mySubDataset[el].std()]
17 IQR_values = [np.percentile(mySubDataset[el], 25, interpolation =
18     'midpoint'), np.percentile(mySubDataset[el], 75,
19     interpolation = 'midpoint')]
20 MADn_values = [median - stats.median_abs_deviation(mySubDataset[
21     el], scale='normal'), median + stats.median_abs_deviation(
22     mySubDataset[el], scale='normal')]
23
24 scales_values = [range_values, std_dev_values, IQR_values,
25     MADn_values]
26 scale_labels = ['Range', 'Standard Deviation', 'Inter Quartile
27     Range', 'Median Absolute Deviation']
28 locations = [a_mean, a_mean, median, median]
29 location_labels = ['Arithmetic Mean', 'Arithmetic Mean', 'Median',
30     'Median']
31 binnings = ['auto', np.arange(0,300,5), np.arange(50,150,5), np.
32     arange(50,150,5)]
33 indexes = [1,2,3,4]
34
35 fig = plt.figure(figsize=(8,6))
36 for scale_values, location, scale_label, location_label, bins,
37     index in zip(scales_values, locations, scale_labels,
38     location_labels, binnings, indexes):
39     ax = fig.add_subplot(2, 2, index)
40     ax.hist(mySubDataset[el], density = True, edgecolor='k',
41     color='#4881e9', bins=bins)
42     ax.axvline(location, color = '#ff464a', linewidth = 1, label
43     = location_label)
44     ax.axvline(scale_values[0], color='#ebb60d')
45     ax.axvline(scale_values[1], color='#ebb60d')
46     ax.axvspan(scale_values[0], scale_values[1], alpha=0.1, color
47     ='orange', label=scale_label)
48     ax.set_xlabel(el + " [ppm]")
49     ax.set_ylabel('probability density')
50     ax.set_ylim(0, 0.1)
51     ax.legend(loc = 'upper right')
52 fig.tight_layout()

```

Listing 11.5 Weak and robust estimates for the scale.

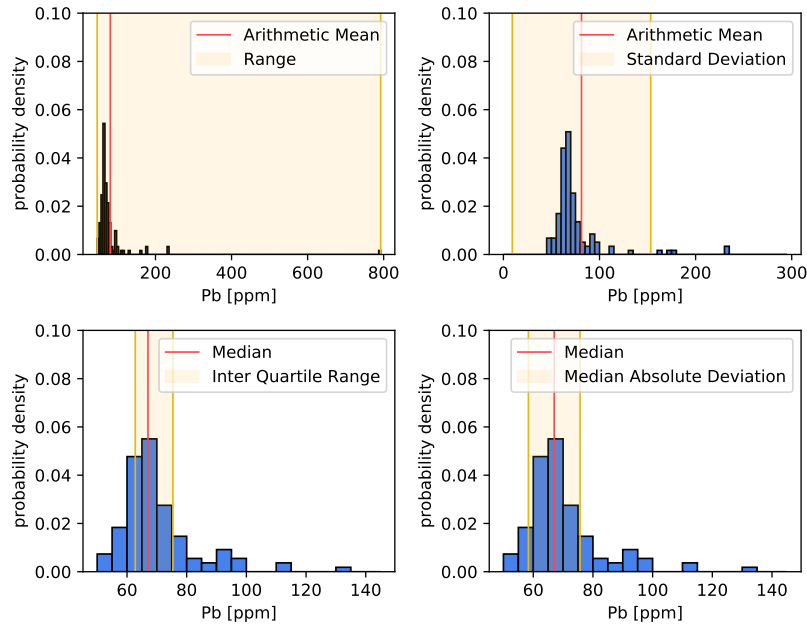


Fig. 11.4 The result of the code listing 11.5.

The MAD uses the sample median twice, first to get an estimate of the location of the data set (i.e., $Me(\mathbf{z})$), and then to compute the sample median of the absolute residuals from the estimated location, i.e., $\{|\mathbf{z} - Me(\mathbf{z})|\}$. To make the MAD comparable to the σ , the normalized MAD (MAD_n) is defined as:

$$MAD_n(\mathbf{z}) = \frac{MAD(\mathbf{z})}{0.6745} \quad (11.4)$$

The rationale behind this choice is that 0.6745 is the MAD of a standard normal random variable, and hence a $N(\mu, \sigma)$ variable has $MAD_n = \sigma$. In Python, the MAD can be computed easily by using the `scipy.stats.median_abs_deviation()` function. To calculate the MAD_n as defined by the Eq. 11.4, we need to set the `'scale'` parameter to `'normal'` explicitly when calling the `median_abs_deviation()` function.

M-estimators of location and scale

The jointly robust estimation of location and scale proposed by Huber (1966), i.e., "Huber's proposal 2", consist in the solution of a location–dispersion model with two unknown parameters (i.e., $\hat{\mu}$ and $\hat{\sigma}$; Eq. 11.5).

```

1 import pandas as pd
2 import numpy as np
3 import statsmodels as st
4 import matplotlib.pyplot as plt
5
6 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
7     sheet_name=1, engine='openpyxl')
8 e1 = 'Pb'
9
10 mySubDataset = myDataset[myDataset.Epoch == 'three-b']
11 mySubDataset = mySubDataset.dropna(subset=[e1])
12
13 norms = [st.robust.norms.HuberT(t=1.345), st.robust.norms.
14     Hampel(a=2.0, b=4.0, c=8.0)]
15 loc_labels = [r"Huber's T function", r"Hampel function"]
16 indexes = [1,2]
17
18 fig = plt.figure(figsize=(6,6))
19
20 for norm, loc_label, index in zip(norms, loc_labels, indexes):
21
22     huber_proposal_2 = st.robust.Huber(c= 1.5, norm = norm)
23     H_loc, H_scale = huber_proposal_2(mySubDataset[e1])
24     ax = fig.add_subplot(2, 1, index)
25     bins = np.arange(50,250,5)
26     ax.hist(mySubDataset[e1], density = True, edgecolor='k',
27         color='#4881e9', bins=bins)
28     ax.axvline(H_loc, color = '#ff464a', linewidth = 2, label=
29         loc_label + " as  $\psi$ : location at {:.1f} [ppm]".format
30         (H_loc))
31     ax.axvline(H_loc + H_scale, color = '#ebb60d')
32     ax.axvline(H_loc - H_scale, color = '#ebb60d')
33     ax.axvspan(H_loc + H_scale, H_loc - H_scale, alpha=0.1,
34         color='orange', label="Huber's estimation for the scale:
35         {:.1f} [ppm]".format(H_scale))
36     ax.set_xlabel(e1 + " [ppm]")
37     ax.set_ylabel('probability density')
38     ax.set_ylim(0, 0.1)
39     ax.legend(loc = 'upper right')
40     ax.annotate('Large outlier at about 800 ppm', (253, 0.04),
41         (230,0.04), ha="right", va="center", size=9, arrowprops=
42         dict(arrowstyle='fancy'))
43 fig.tight_layout()

```

Listing 11.6 M-estimators for the location and the scale: "Huber's proposal 2".

$$\begin{aligned}
 \sum_{i=1}^n \psi \left(\frac{x_i - \hat{\mu}}{\hat{\sigma}} \right) &= 0 \\
 \sum_{i=1}^n \psi^2 \left(\frac{x_i - \hat{\mu}}{\hat{\sigma}} \right) &= (n-1)\beta
 \end{aligned}
 \tag{11.5}$$

where $\hat{\mu}$ and $\hat{\sigma}$ are the maximum likelihood estimators of μ and σ , respectively. In Python, the "Huber's proposal 2" is implemented by the `statsmodels.robust.scale.Huber()` function. By default, it uses the Huber's T as ψ , but other ψ can be selected (e.g., Hampel 17A, Ramsay's Ea, etc).

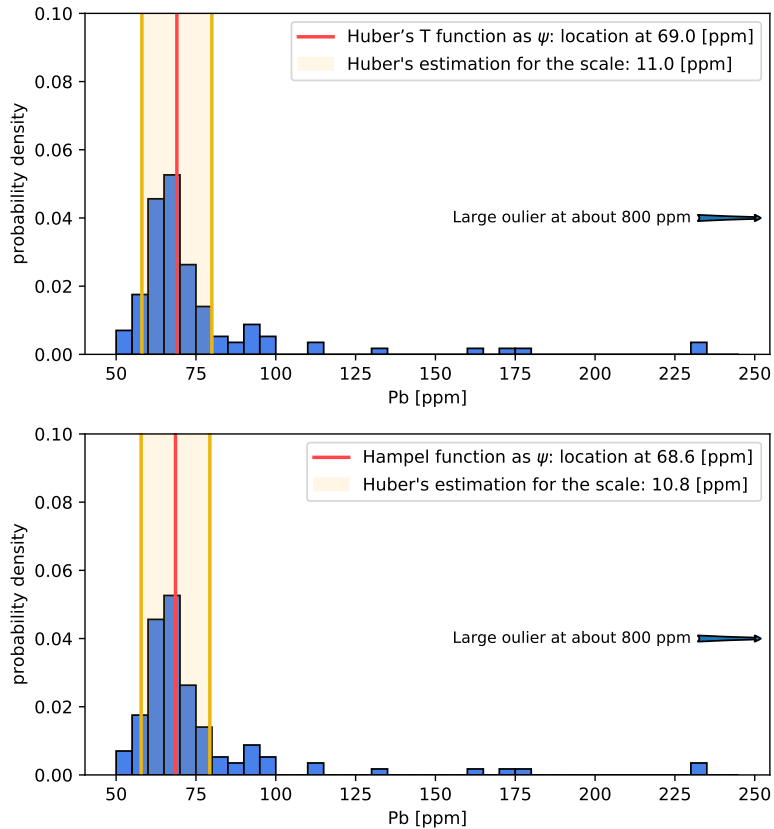


Fig. 11.5 The result of the code listing 11.6.

11.4 Robust Statistics in Geochemistry

In the present section, I review the main conclusions reported by (Reimann & Filzmoser, 2000) on the use of robust statistics in geochemistry. As an example, Reimann and Filzmoser (2000) issued that most of the variables belonging to large data sets from regional geochemical and environmental surveys show neither a normal or lognormal data distribution.

Table 11.1 Application of robust statistics in geochemistry. Developed on the basis of Tab. 3 in Reimann and Filzmoser, 2000.

Location	Recommendation	Here
Arithmetic mean	Should only be used in special cases	Yes
Geometric mean	Can be used, but may be problematic in some cases	Yes
Median	Should be the first choice as location estimator	Yes
Hampel or Huber means	Can be used	Yes
Dispersion	Recommendation	Here
Standard deviation	Should not be used if data outliers exist	Yes
Mad (medmed)	Can be used	Yes
Hinge spread	Can be used	No
Robust spread	Can be used	Yes
Tests for means and variances	Recommendation	Here
t-test	Should not be used	No
F-test	Should not be used	No
Notches in boxplot	Can be used, very easy and fast	Yes
Non-parametric tests	Can be used	Yes
Robust tests	Can be used	No
Multivariate methods	Recommendation	Here
Correlation analysis	Should not be used with the original (untransformed) data	Yes
Regression analysis	Should not be used with the original (untransformed) data	Yes
Robust regression analysis	Can be used, preferably on log-transformed data	No
Non-parametric regression	Can be used, preferably on log-transformed data	Yes
PCA	Very sensible to outlying observations, Should not be used	No
Robust PCA	Can be used, preferably with log-transformed data	No

Even after a transformation devoted to reporting the data set to a normal shape, many of these data sets do not approach a Gaussian distribution (Reimann & Filzmoser, 2000). Typically, the distributions investigated by Reimann and Filzmoser (2000) are skewed, and they contain outliers. Reimann and Filzmoser (2000) concluded that when dealing with regional geochemical or environmental data, normal or lognormal distributions are an exception and not the rule. The conclusions reported by Reimann and Filzmoser (2000) have significant consequences for the further statistical treatment of geochemical and environmental data, mostly requiring a robust approach.

Why geochemical and environmental data are not normally distributed? Reimann and Filzmoser (2000) started arguing that geochemical and environmental data register spatial dependence and spatially dependent data are not, usually, normally distributed. Also, trace element data approaching the detection limit are often truncated, i.e., a significant number of observations are not characterized by a true measured value (Reimann & Filzmoser, 2000). Also, the precision of the analytical determinations degrades with the reduction of element concentration, i.e., values are less precise when approaching detection limits (Reimann & Filzmoser, 2000). Finally, these data sets often reveal outliers, possibly due to analytical issues or derived by another population than the main body of data (Reimann & Filzmoser, 2000).

Table 11.1, is a modification of Tab. 3 in Reimann and Filzmoser, 2000 and it reports frequently used statistical parameters, tests and multivariate methods and their suitability for regional geochemical and environmental data which neither show a normal or lognormal distribution.

Chapter 12

Machine Learning

12.1 Introduction to Machine Learning in Geology

Machine learning (ML) is a sub-field of Artificial Intelligence (AI) concerning the use of algorithms and methods to detect patterns from large data sets and to use the uncovered patterns to predict future trends, classify, or perform other kinds of strategic decisions (Murphy, 2012).

The field of ML has grown significantly over the past two decades, evolving from a “niche approach” to a robust technology with broad scientific and commercial use (Jordan & Mitchell, 2015). For example, ML is now successfully employed in several fields like speech recognition, computer vision, robot control and natural language processing (Jordan & Mitchell, 2015). In principle, any complex problem described by a large enough number of input samples and features well fit ML applications (Jordan & Mitchell, 2015). Notably, in the last decade, many researchers have started investigating the application of ML methods in the Earth Sciences (Abedi et al., 2012; Cannata et al., 2011; Goldstein & Coco, 2014; Huang et al., 2002; Masotti et al., 2006; Petrelli et al., 2017; Petrelli et al., 2020; Petrelli & Perugini, 2016; Petrelli et al., 2003; Zuo & Carranza, 2011). In the following, I am going to introduce the basics of ML in Python highlighting a case study in the field of Earth Sciences.

A common characteristic of ML applications is that they are not developed to process an a priori defined conceptual model but they attempt to disclose the complexities in large data sets through a so-called learning process (Bishop, 2006; Shai & Shai, 2013). It consists in the effort of converting the experience into “expertise” or “knowledge” (Shai & Shai, 2013). To understand, please consider that humans use past experiences to implement their learning processes.

As an example, kids begin learning the alphabet by observing the world around them where they find sounds, written letters, words or phrases. Then, at school, they learn the significance of the alphabet and how to combine the different letters. Similarly, the experience for a ML algorithm is the training data and the output is the learned expertise, e.g., a model that can perform a specific task (Shai & Shai, 2013).

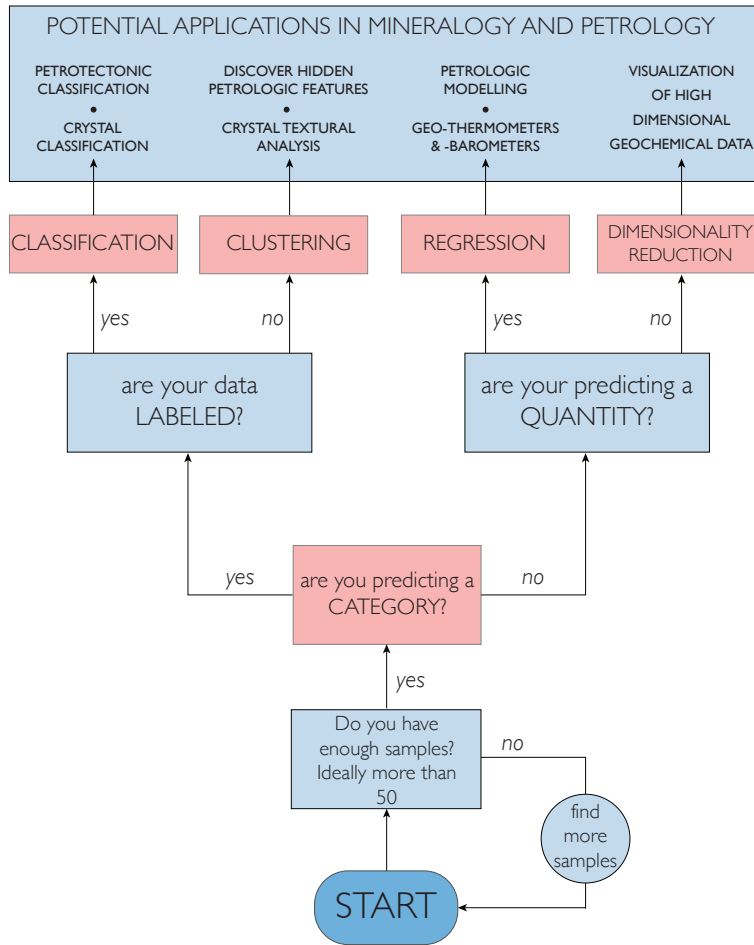


Fig. 12.1 Workflow for the application of ML techniques in petrology and mineralogy (Petrelli & Perugini, 2016).

Broadly, the learning process in ML can be divided into two main fields: (a) unsupervised and (b) supervised learning. In the unsupervised learning, the training data set consists of several input vectors or arrays, without any corresponding target values. On the contrary, in supervised applications, the training data set is labelled, meaning that the algorithm learns through examples (Bishop, 2006).

Figure 12.1 shows a flowchart, modified from Petrelli and Perugini (2016) depicting the main areas of ML (classification, clustering, regression and dimensionality reduction) and their possible use to solve typical mineralogical and petrological problems. As reported in Fig. 12.1, a requirement for the use of an ML technique is the

availability of a proper number of data (indicatively more than 50¹). The main aim of Fig 12.1 is to determine the ML field (i.e., classification, clustering, regression or dimensionality reduction) to approach the problem. The procedure entails a range of choices about the nature of the investigated issue. If the problem involves categories, the first step is to select between labelled and unlabeled data. If the learning data set is labelled, the training process is supervised and it will involve a “classification” problem (Kotsiantis, 2007). An example of a classification problem in petrology is the petro-tectonic identification using geochemical data (Petrelli & Perugini, 2016). If the training data set is unlabeled, the problem is about “clustering” (Jain et al., 1999). The field of clustering as been investigated in petrology since the '80 (e.g., Le Maitre, 1982). As an example, Le Maitre (1982) discussed the basics of clustering in petrology. If the problem does not include a categorization, the subsequent step is to establish whether a quantity must be predicted. If the answer is yes, we are in the field of “regression” (Smola & Schölkopf, 2004). An example application in petrology of ML regression has been provided by Petrelli et al. (2020). Finally, if we are not predicting a quantity, we are in the field of “dimensionality reduction” (Lee & Verleysen, 2009). Dimensionality reduction is particularly useful, for example, in the context of visualization of high-dimensional geological data.

12.2 Machine Learning in Python

To introduce the reader in the use of machine learning techniques to Earth Sciences I will use Scikit-learn². Scikit-learn is a Python library integrating a wide range of state-of-the-art machine learning algorithms (Pedregosa et al., 2011). This package focuses on bringing machine learning to non-specialists using a general-purpose high-level language like Python (Pedregosa et al., 2011).

Scikit-learn represents a robust framework for the solution of Earth Sciences problems in fields of clustering, regression, dimensionality reduction, and classification (Fig. 12.1). Additional examples of Python libraries allowing the development of ML applications are TensorFlow³, Keras⁴, and PyTorch⁵.

¹ https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

² <https://scikit-learn.org>

³ <https://www.tensorflow.org>

⁴ <https://keras.io>

⁵ <https://pytorch.org>

12.3 A Study Case of Machine Learning in Geology

Pyroxene thermo-barometry

Disclosing pre-eruptive temperatures and storage depths in volcanic plumbing systems is a fundamental issue in petrology and volcanology (e.g., Devine et al., 1998; Putirka, 2008; Putirka et al., 2003). To date, the development of geo-thermometers and barometers bases on the thermodynamic characterization of the magmatic system and this approach provides a robust framework, widely applied in the estimation of pre-eruptive magma temperature and storage depths (Masotta et al., 2013; Neave et al., 2019; Nimis, 1995; Nimis & Ulmer, 1998; Putirka, 2008; Putirka et al., 2003). As reported by Petrelli et al. (2020), the conventional calibration procedure for cpx thermometers and barometers consists of five main steps: a) recognize chemical equilibria associated with large variations of entropy and volume, respectively (Putirka, 2008); b) retrieving a statistically robust experimental data set with known T and P (e.g., the LEPR data set; Hirschmann et al., 2008); c) determine the cpx components from chemical analyses; d) define a regression procedure; e) validate the model (Putirka, 2008).

In 2020, Petrelli et al. (2020) proposed a new ML method to retrieve magma temperature and storage depths on the basis of melt-clinopyroxenes and clinopyroxenes only chemistry. In detail, the ML approach proposed by Petrelli et al. (2020) starts from the same basis of the classical approach but it is not based on an 'a priori' defined model, allowing the algorithm to retrieve the elements that are involved in variations of entropy and volume. But what's the main difference between classical approaches and ML ones? In few word, classical approaches base on a simplified thermodynamic framework providing equations to be fitted using experimental data (typically using linear regression). On the contrary ML methods base on the statistical relationships linking variations in the chemistry of CPXs (or CPX-melt couples) and the target variables (i.e., P and T), without necessary providing a thermodynamic framework. In agreement with the workflow reported in Fig. 12.1, the investigations reported in Petrelli et al. (2020) fall in the ML field of regression.

The experimental data set for the training

The experimental data set utilized by Petrelli et al. (2020) to train the model consisted of 1403 experimentally produced clinopyroxenes in equilibrium with a wide range of silicate melt compositions at pressures and temperatures in the range 0.001-40 kbar and 952-1883 K, respectively. As input parameters, Petrelli et al. (2020) used the major element compositions of melt (SiO_2 , TiO_2 , Al_2O_3 , FeO_t , MnO , MgO , CaO , Na_2O , K_2O , Cr_2O_3 , P_2O_5 , H_2O) and clinopyroxene (SiO_2 , TiO_2 , Al_2O_3 , FeO_t , MnO , MgO , CaO , Na_2O , K_2O , Cr_2O_3) phases. Now, we start importing and visualizing the data set shared by Petrelli et al. (2020) using the code listing 11.6 and Figs. 12.2 and 12.3.

```

1 import numpy as np
2 import pandas as pd
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5 from sklearn.preprocessing import StandardScaler
6 from sklearn.ensemble import ExtraTreesRegressor
7 from sklearn.metrics import mean_squared_error
8 from sklearn.metrics import r2_score
9
10 # Import The Training Data Set
11 my_training_dataset = pd.read_excel('
    GlobalDataset_Final_rev9_TrainValidation.xlsx', usecols = "A:
    M,O:X,Z:AA", skiprows=1, engine='openpyxl')
12 my_training_dataset.columns = [c.replace('.1', 'cpx') for c in
    my_training_dataset.columns]
13 my_training_dataset = my_training_dataset.fillna(0)
14
15 Train_Labels = np.array([my_training_dataset.Sample_ID]).T
16 X0_train = my_training_dataset.iloc[:, 1:23]
17 Y_train = np.array([my_training_dataset.T_K]).T
18
19 fig = plt.figure(figsize=(8,8))
20 x_labels_melt = [r'SiO2', r'TiO2', r'Al2O3', r'
    FeOt', r'MnO', r'MgO', r'CaO', r'Na2O', r'K2O', r'
    Cr2O3', r'P2O5', r'H2O']
21 for i in range(0,12):
22     ax1 = fig.add_subplot(4, 3, i+1)
23     sns.kdeplot(X0_train.iloc[:, i], fill=True, color='k',
24                 facecolor='#c7ddf4', ax = ax1)
25     ax1.set_xlabel(x_labels_melt[i] + ' [wt. %] the melt')
26 fig.align_ylabels()
27 fig.tight_layout()
28
29 fig1 = plt.figure(figsize=(6,8))
30 x_labels_cpx = [r'SiO2', r'TiO2', r'Al2O3', r'FeOt
    ', r'MnO', r'MgO', r'CaO', r'Na2O', r'K2O', r'
    Cr2O3']
31 for i in range(0,10):
32     ax2 = fig1.add_subplot(5, 2, i+1)
33     sns.kdeplot(X0_train.iloc[:, i+12], fill=True, color='k',
34                 facecolor='#c7ddf4', ax = ax2)
35     ax2.set_xlabel(x_labels_cpx[i] + ' [wt. %] in cpx')
36 fig1.align_ylabels()
37 fig1.tight_layout()

```

Listing 12.1 Importing and visualizing the training data set in Petrelli et al., 2020.

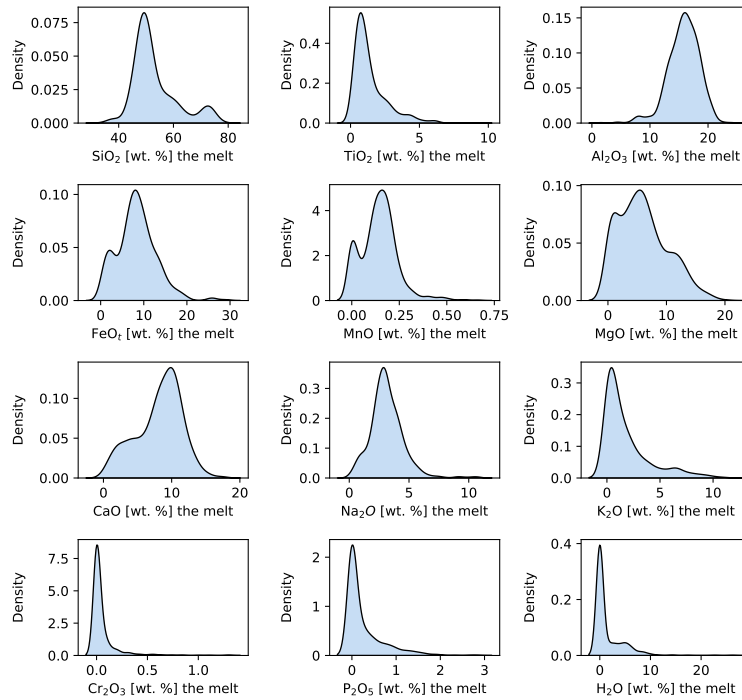


Fig. 12.2 Chemical composition of the melt phase in the training data set by Petrelli et al. , 2020.

Standardization

Standardization of the data set is a common requirement for many machine-learning estimators.

For instance, many ML algorithms assume that all features are centered around zero and have variance in the same order. If a feature has a variance that is orders of magnitude larger than others, it might play a dominant role and make the algorithm unable to learn from other features correctly as expected.

The easiest way to perform the normalization of a data set is removing the mean and scaling to unit variance (Eq. 12.1):

$$\tilde{x}_e^i = \frac{x_e^i - \mu^e}{\sigma_p^e} \quad (12.1)$$

where \tilde{x}_e^i , x_e^i are the transformed and the original value of each component belonging to the sample distribution of chemical analyses of the element e (i.e., SiO₂, TiO₂, etc...), characterized by an average μ^e and a standard deviation σ_p^e .

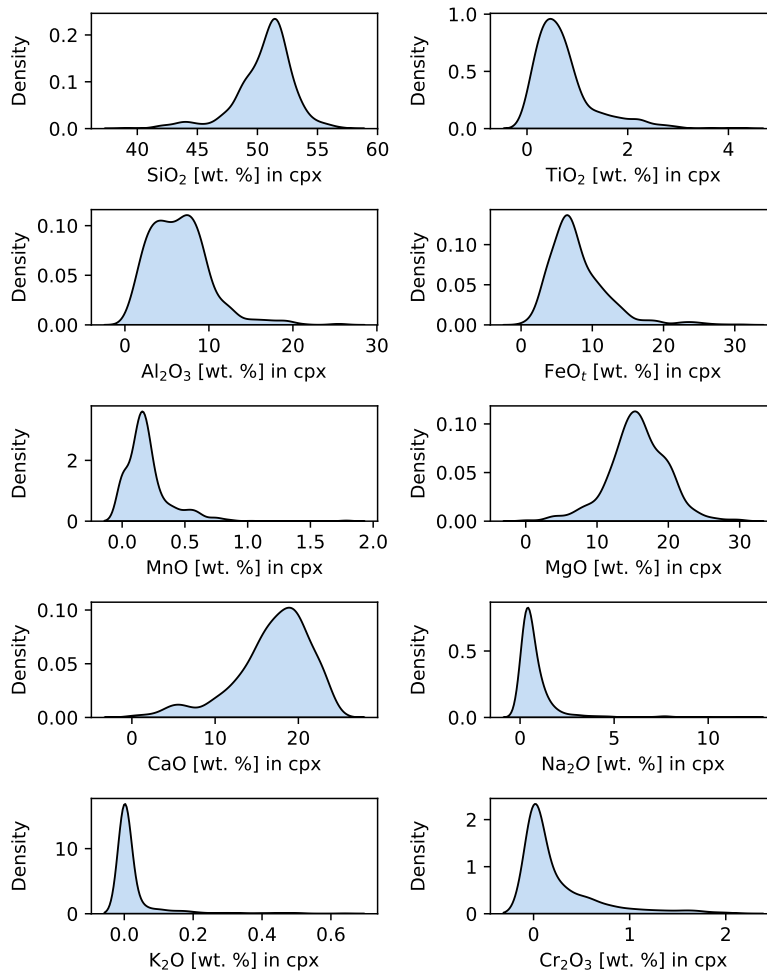


Fig. 12.3 Chemical composition of the clinopyroxene phase in the training data set by Petrelli et al., 2020.

Scikit-learn implements the Eq. 12.1 in the *sklearn.preprocessing.StandardScaler()* class, a set of methods (i.e., functions) allowing the scaling of both the training data set and unknown samples.

Also, scikit-learn implements additional scalers and transformers. In scikit-learn, scaler and transformers perform linear and non-linear transformations, respectively.

As an example, *MinMaxScaler()* scales all feature belonging to the data set between 0 and 1. Table 12.1 summarizes the main scalers and the transformers available in scikit-learn.

The *QuantileTransformer()* provides non-linear transformations in which distances between marginal outliers and inliers are shrunk. Finally, the *PowerTransformer()* provides non-linear transformations in which data is mapped to a normal distribution to stabilize variance and minimize skewness.

Table 12.1 Scalers and Trasformers in Scikit-learn. Descriptions are taken from the official documentation of Scikit-learn.

Scaler	Description
<code>sklearn.preprocessing.StandardScaler()</code>	Standardize features by removing the mean and scaling to unit variance (Eq. 12.1)
<code>sklearn.preprocessing.MinMaxScaler()</code>	Transform features by scaling each feature to a given range. The default rannges is [0,1]
<code>sklearn.preprocessing.RobustScaler()</code>	Scale features using statistics that are robust to outliers. This Scaler removes the median and scales the data according to the quantile range. The default quantile range is the IQR (Inter-Quartile Range).
Tranformer	Description
<code>sklearn.preprocessing.PowerTransformer()</code>	Apply a power transform featurewise to make data more Gaussian-like. Power transforms are a family of parametric, monotonic transformations that are applied to make data more Gaussian-like. During the writing of the rpresent book, <i>PowerTransformer</i> supported the Box-Cox transform and the Yeo-Johnson transform.
<code>sklearn.preprocessing.QuantileTransformer()</code>	Transform features using quantiles information. This method transforms the features to follow a uniform or a normal distribution. Therefore, for a given feature, this transformation tends to spread out the most frequent values. It also reduces the impact of (marginal) outliers: this is therefore a robust preprocessing scheme.

In Petrelli et al. (2020), the data set has been processed by the *StandardScaler()*. To better understands, the lines 1 and 2 of the code listing 12.2 show how to apply the *StandardScaler()* to the data displayed in Fig. 12.2 and Fig. 12.3.

Also, Fig. 12.4 and Fig. 12.5 report the result of the *StandardScaler()* operated in the code listing 12.2 for the melt and clinopyroxene data, respectively. Now, all the features (i.e., each chemical element oxide) are characterized by zero mean and unit variance. To note, the tree based methods described in the following section and utilized here as application proxy in Geology of ML do not strictly require standardization. However, performing the standardization helps in data visualization, and it is useful when you are applying different methods to the same problem to

compare performances with scale sensitive algorithms like Support Vector Machines (SVM; Hearst et al., 1998). Generally speaking, the algorithms which depends on distance measures among the predictors are the ones requiring standardization.

```

1 scaler = StandardScaler().fit(X0_train)
2 X_train = scaler.transform(X0_train)
3
4 fig2 = plt.figure(figsize=(8,8))
5 for i in range(0,12):
6     ax3 = fig2.add_subplot(4, 3, i+1)
7     sns.kdeplot(X_train[:, i],fill=True, color='k', facecolor='#
8         ffdfab', ax = ax3)
9     ax3.set_xlabel('scaled ' + x_labels_melt[i] + ' the melt')
10 fig2.align_ylabels()
11 fig2.tight_layout()
12
13 fig3 = plt.figure(figsize=(6,8))
14 for i in range(0,10):
15     ax4 = fig3.add_subplot(5, 2, i+1)
16     sns.kdeplot(X_train[:, i+12],fill=True, color='k', facecolor=
17         '#ffdfab', ax = ax4)
18     ax4.set_xlabel('scaled ' + x_labels_cpx[i] + ' in cpx')
19 fig3.align_ylabels()
20 fig3.tight_layout()

```

Listing 12.2 Application of the StandardScaler() to the data reported in Fig. 12.2 and Fig. 12.3.

Training and testing the model

As humans learn from the experience, ML algorithms learn from data. The role of the scaled training data set is to provide the learning experience for a ML algorithm.

In Petrelli et al. (2020), many ML methods have been evaluated to find the best regressor for the investigated problem. They are: Single Decision Trees (Breiman et al., 2017), Random Forests (Breiman, 2001), Stochastic Gradient Boosting (Friedman, 2002), Extremely Randomized Trees (ERT, Geurts et al., 2006), and k-nearest neighbors (Bentley, 1975). How does they work?

Single Decision Trees: a single decision tree model (Breiman et al., 2017) partitions the features space into a set of regions. Then it fits a simple model for each region (Zhang & Haghani, 2015). To understand how does the decision tree model work for a regression problem, I am going to provide you the example reported by Zhang and Haghani (2015). Consider a continuous response variable Y and two independent variables X1 and X2. To make a regression, the first step consists of splitting the space defined by X1 and X2 into two regions and model the response Y (mean of Y) individually in each region. Then, the process continues with the splitting of each region into two more regions and proceed until some stopping rule is met. During each partition process, the best fit is achieved through the selection of

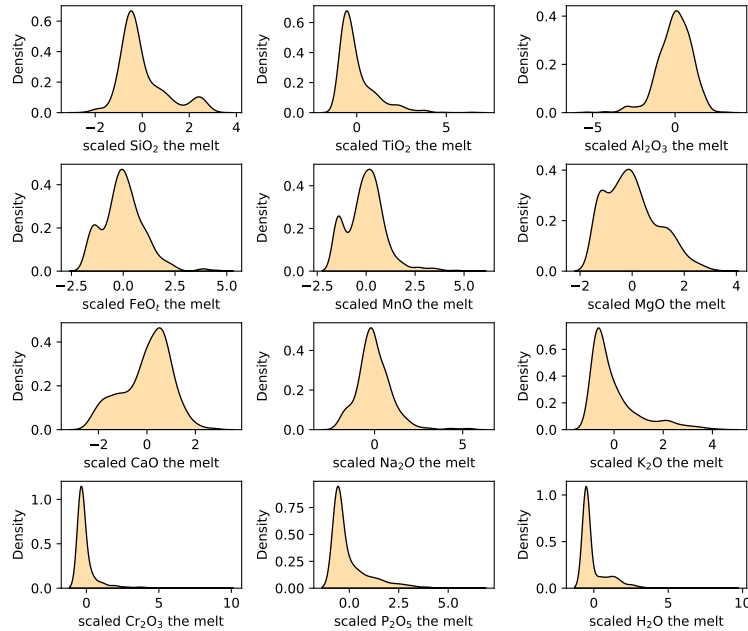


Fig. 12.4 Result of the application of the `StandardScaler()` to the data reported in Fig. 12.2.

variables and a split-point (Zhang & Haghani, 2015). The single tree algorithm is at the base of random forest, gradient boosting regression, and extremely randomized tree methods. More details about the single decision tree model can be found in Breiman et al. (2017).

The Random Forest: the Random Forest algorithm (Breiman, 2001) combines two established machine-learning principles (Zhang & Haghani, 2015): Breiman’s “bagging” predictors (Breiman, 1996) and the random features selection (Ho, 1998). The bagging is a method for producing multiple versions of a predictor and using these to get an aggregated predictor (Breiman, 1996). The multiple versions are created by making bootstrap replicates of the learning set and using these as new learning sets (Breiman, 1996). In the Random Forest algorithm, the Bagging predictors have been used to generate a diverse subset of data for training base models (Zhang & Haghani, 2015). For a given training data set with sample size n , bagging produces k new training set, each with sample size n , by sampling from the original training data set uniformly and with replacement (Zhang & Haghani, 2015). Through sampling with replacement (i.e., bootstrap), some observations appear more than once in the produced sample, while other observations will be ‘left out’ of the sample (Zhang & Haghani, 2015). Then, k base models are trained using the newly created k training set and coupled through averaging for the regression or majority voting for classification (Zhang & Haghani, 2015). A detailed description of the Random

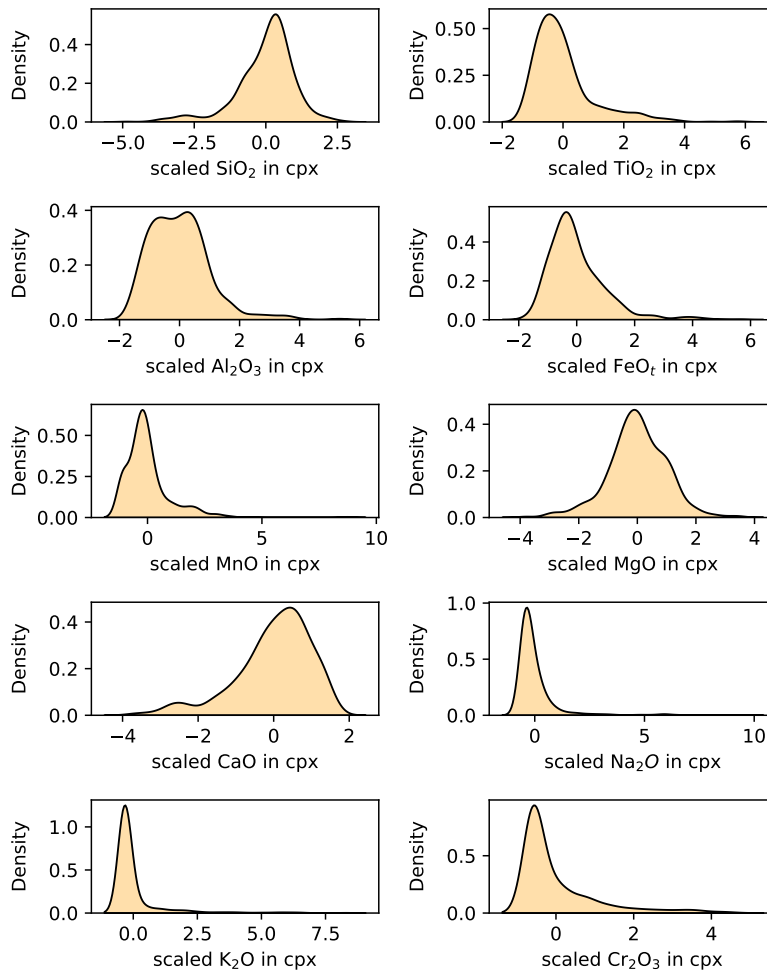


Fig. 12.5 Result of the application of the StandardScaler() to the data reported in Fig. 12.3.

Forest algorithm can be found in Breiman (2001), Natekin and Knoll (2013), and Zhang and Haghani (2015).

Gradient Boosting: distinct from bagging predictors, the boosting method creates base models sequentially (Friedman, 2002; Zhang & Haghani, 2015). In the Gradient Boosting algorithm, the prediction capability is progressively improved through developing multiple models in sequence and focusing on these training cases that are difficult to estimate (Zhang & Haghani, 2015). As a key feature in the boosting process, examples that are hard to estimate using the previous base models appear more often in the training data than the ones that are correctly estimated (Friedman, 2002; Zhang & Haghani, 2015). In detail, each successive base model is intended

to correct the errors made by its previous base models (Zhang & Haghani, 2015). A detailed description of the Gradient Boosting algorithm can be found in Friedman (2002) and Zhang and Haghani (2015).

The Extremely Randomized Trees: the Extremely Randomized Trees algorithm builds an ensemble of regression trees according to the top-down procedure proposed by Geurts et al. (2006). Its two main differences with other tree-based ensemble methods are: 1) it splits nodes by choosing cut-points fully at random and 2) it uses the whole learning sample (rather than a bootstrap replica) to grow the trees (Geurts et al., 2006). It has two main parameters: the number of attributes randomly selected at each node and the minimum sample size for splitting a node (Geurts et al., 2006). It works several times with the (full) original learning sample to generate an ensemble model (Geurts et al., 2006). The predictions of the trees are aggregated to yield the final prediction, by majority vote in classification problems and arithmetic average in regression problems (Geurts et al., 2006). A complete description of the Extremely Randomized Trees algorithm is reported in Geurts et al. (2006).

k-nearest neighbors: K-nearest neighbors is a simple algorithm that collects all available cases and predicts the numerical target based on an estimation of similarity (e.g., distance functions; Bentley, 1975). In detail, it typically uses an inverse distance weighted average of the K nearest neighbors (Bentley, 1975). The weight of each training instance can be uniform or computed using a kernel function, which could depend on the distance (as opposed to similarity) between itself and the test instance. To note, the prediction using a single neighbor is just the target value of the nearest neighbor (Bentley, 1975). A common distance metric used to measure the distance between two instances is the Euclidean distance metric. A detailed description of the K-nearest neighbors algorithm can be found in Bentley (1975).

The scikit-learn implementation of Single Decision Trees (Breiman et al., 2017), Random Forests (Breiman, 2001), Stochastic Gradient Boosting (Friedman, 2002), Extremely Randomized Trees (ERT, Geurts et al., 2006), and k-nearest neighbors (Bentley, 1975) are reported in Table 12.2.

Table 12.2 ML regressors reported in Petrelli et al. (2020).

Algorithm	scikit-learn
Single Decision Trees	class sklearn.tree. DecisionTreeRegressor()
The Random Forest	class sklearn.ensemble. RandomForestRegressor()
Gradient Boosting	class sklearn.ensemble. GradientBoostingRegressor()
The Extremely Randomized Trees	class sklearn.ensemble. ExtraTreesRegressor()
k-nearest neighbors	class sklearn.neighbors. KNeighborsRegressor()

The training and test processes can be easily performed in scikit-learn as described in the code listing 12.3.

They consist in the following steps (code listing 12.3): 1) define and train the algorithm (in our case the Extremely Randomized Trees method; lines 2 and 5); 2) import the test data set and scale it in accordance with the rules used for the train data set (lines 8-17); 3) make a prediction on the test data set (line 20); 4) select one or more metrics to evaluate the results (lines 23 and 23); 5) make the evaluation for the results (lines 28-35 producing the Fig 12.6).

```

1 # Define the regressor, in our case the Extra Tree Regressor
2 regr = ExtraTreesRegressor(n_estimators=550, criterion='mse',
   max_features=22, random_state=280) # random_state fixed for
   reproducibility
3
4 # Train the model
5 regr.fit(X_train, Y_train.ravel())
6
7 # Import the test data set
8 my_test_dataset = pd.read_excel('GlobalDataset_Final_rev9_Test.
   xlsx', usecols = "A:M,O:X,Z:AA", skiprows=1, engine='openpyxl
   ')
9 my_test_dataset.columns = [c.replace('.1', 'cpx') for c in
   my_test_dataset.columns]
10 my_test_dataset = my_test_dataset.fillna(0)
11
12 X0_test = my_test_dataset.iloc[:, 1:23]
13 Y_test= np.array([my_test_dataset.T_K]).T
14 Labels_test = np.array([my_test_dataset.Sample_ID]).T
15
16 # Scale the test dataset
17 X_test_scaled = scaler.transform(X0_test)
18
19 # Make the prediction on the test data set
20 predicted = regr.predict(X_test_scaled)
21
22 # Evaluate the results using the R2 and RMSE
23 r2 = r2_score(Y_test, predicted)
24 RMSE = np.sqrt(mean_squared_error(predicted, Y_test))
25
26 # Plot data
27 fig, ax = plt.subplots(figsize=(6,6))
28 ax.plot([1050,1850],[1050,1850], c='#000000', linestyle='--')
29 ax.scatter(Y_test,predicted, color='#ad1010', edgecolor='#000000',
   label=r"ExtraTreesRegressor - R$^2$=" + "{:.2f}".format(r2)
   + " - RMSE=" + "{:.0f}".format(RMSE) + " K")
30 ax.legend()
31 ax.axis('scaled')
32 ax.set_xlabel('Expected Temperature values [K]')
33 ax.set_ylabel('Predicted Temperature values [K]')

```

Listing 12.3 Training and testing the ExtraTreesRegressor() algorithm for temperature predictions.

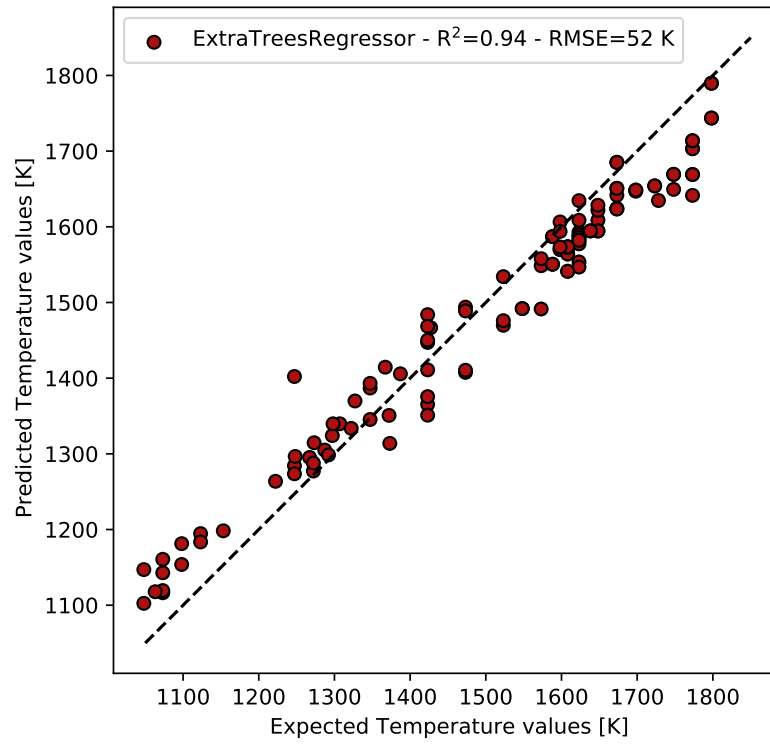


Fig. 12.6 Result of the code listing 12.3.

Part VI
Appendices

Appendix A

Python Packages Specifically Developed for Geologists

pyrolite

pyrolite¹ is a set of tools to handle and visualize geochemical data. The python package includes functions to work with compositional data, to transform geochemical variables (e.g. elements to oxides), functions for common plotting tasks (e.g. spiderplots, ternary diagrams, bivariate and ternary density diagrams), and numerous auxiliary utilities.

ObsPy

ObsPy² is an open-source project dedicated to provide a Python framework for processing seismological data. It provides parsers for common file formats, clients to access data centers and seismological signal processing routines which allow the manipulation of seismological time series.

APSG

APSG³ defines several new python classes to easily manage, analyze and visualize orientational structural geology data.

¹ <https://pyrolite.readthedocs.io/en/master/>

² <https://github.com/obspy/obspy/wiki>

³ <https://apsg.readthedocs.io/en/stable/index.html>

GemPy

GemPy⁴ is a tool for generating 3D structural geological models in Python. As such, it enables you to create complex combinations of stratigraphical and structural features such as folds, faults, and unconformities. It was furthermore designed to enable probabilistic modeling to address parameter and model uncertainties.

Segyio

Segyio⁵ is a small LGPL licensed C library for easy interaction with SEG-Y and Seismic Unix formatted seismic data, with language bindings for Python and Matlab. Segyio is an attempt to create an easy-to-use, embeddable, community-oriented library for seismic applications. Features are added as they are needed; suggestions and contributions of all kinds are very welcome.

Pyrocko

Pyrocko⁶ is an open source seismology toolbox and library. Most of Pyrocko is written in the Python programming language, a few parts are written in C.

gprMax

gprMax⁷ is open source software that simulates electromagnetic wave propagation. It solves Maxwell's equations in 3D using the Finite-Difference Time-Domain (FDTD) method. gprMax was designed for modelling Ground Penetrating Radar (GPR) but can also be used to model electromagnetic wave propagation for many other applications.

Lasio, welly and PetroPy

Lasio⁸ is a Python package to read and write Log ASCII Standard (LAS) files, used for borehole data such as geophysical, geological, or petrophysical logs. It's compatible with versions 1.2 and 2.0 of the LAS file specification, published by the Canadian Well Logging Society. Support for LAS 3 is being worked on. In principle it is designed to read as many types of LAS files as possible, including

⁴ <https://www.gempy.org>

⁵ <https://github.com/equinor/segyio>

⁶ <https://git.pyrocko.org/pyrocko/pyrocko>

⁷ <https://www.gprmax.com>

⁸ <https://github.com/kinverarity1/lasio/>

ones containing common errors or non-compliant formatting. Sometimes we want a higher-level object, for example to contain methods that have nothing to do with LAS files. We may want to handle other well data, such as deviation surveys, tops (aka picks), engineering data, striplogs, synthetics, and so on. This is where welly⁹ comes in. Welly uses lasio for data I/O, but hides much of it from the user. We recommend you look at both projects before deciding if you need the 'well-level' functionality that welly provides. Welly is a family of classes to facilitate the loading, processing, and analysis of subsurface wells and well data, such as striplogs, formation tops, well log curves, and synthetic seismograms. PetroPy¹⁰ is a python petrophysics package allowing scientific python computing of conventional and unconventional formation evaluation. Reads las files using lasio. Includes a petrophysical workflow and a log viewer based on XML templates.

SimPEG

Simulation and Parameter Estimation in Geophysics (SimPEG)¹¹ is a python package for simulation and gradient based parameter estimation in the context of geophysical applications.

Devito

Devito¹² is a Python package to implement optimized stencil computation (e.g., finite differences, image processing, machine learning) from high-level symbolic problem definitions. Devito builds on SymPy and employs automated code generation and just-in-time compilation to execute optimized computational kernels on several computer platforms, including CPUs, GPUs, and clusters thereof.

pyGIMLi

pyGIMLi¹³ is an open-source library for modelling and inversion and in geophysics. The object-oriented library provides management for structured and unstructured meshes in 2D and 3D, finite-element and finite-volume solvers, various geophysical forward operators, as well as Gauss-Newton based frameworks for constrained, joint and fully-coupled inversions with flexible regularization.

⁹ <https://github.com/agile-geoscience/welly>

¹⁰ <https://github.com/toddheitmann/PetroPy>

¹¹ <https://github.com/simpeg/simpeg>

¹² <http://www.devitoproject.org>

¹³ <https://www.pygimli.org>

HyVR

The Hydrogeological Virtual Reality simulation package (HyVR)¹⁴ is a Python module that helps researchers and practitioners generate subsurface models with multiple scales of heterogeneity that are based on geological concepts. The simulation outputs can then be used to explore groundwater flow and solute transport behaviour. This is facilitated by HyVR outputs in common flow simulation packages' input formats. As each site is unique, HyVR has been designed that users can take the code and extend it to suit their particular simulation needs.

Landlab

Landlab¹⁵ is an open-source Python-language package for numerical modeling of Earth surface dynamics. It contains: 1) a gridding engine which represents the model domain. Regular and irregular grids are supported; 2) library of process components, each of which represents a physical process (e.g., generation of rain, erosion by flowing water); 3) utilities that support general numerical methods, file input/output, and visualization. In addition Landlab contains a set of Jupyter notebook tutorials providing an introduction to core concepts and examples of use.

pyGeoPressure

pyGeoPressure is an open source python package designed for pore pressure prediction with both well log data and seismic velocity data. Though lightweighted, pyGeoPressure is able to perform whole workflow from data management to pressure prediction. The main features of pyGeoPressure are: 1) Overburden (or Lithostatic) Pressure Calculation; 2) Eaton's method and Parameter Optimization; 3) Bowers' method and Parameter Optimization; 4) Multivariate method and Parameter Optimization.

¹⁴ <https://github.com/driftingtides/hyvr>

¹⁵ <https://github.com/landlab/landlab>

Appendix B

Introduction to Object Oriented Programming

B.1 Object-oriented programming

Definition: "Object-oriented programming (OOP) is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods)."¹.

In details, the main building blocks of OOP are classes and objects.

A class is an abstraction that is used to create objects. Typically, classes represent broad categories, like the items of an online shop or a physical objects that share attributes. All the objects created using a specific class share the same attributes (e.g., colors, sizes, etc..). In the practice, a class is the blueprint, whereas an object (i.e., instance) contains real data and it is built from a class. The creation of a new object from a class is defined as instantiating an object.

As an example we could define a class for the items of an online shops containing the following attributes: color, size, description, and price. Then, we could start defining each single object, characterized by a specific color, size, description, and price. When we define a `Dataframe` or a `Figure` in python we are creating objects using the class `pandas.DataFrame()` and `matplotlib.figure.Figure()`, respectively.

Also, classes contain functions, called methods. These functions are defined within the class and perform actions or computations for the objects belonging to that specific class. As an example, `.var()` and `.mean()` are methods available for the objects belonging to the class `pandas.DataFrame()`.

B.2 Defining classes, attributes and methods in Python

The class statement followed by the name of the class and a colon define a new class. Note that Python class names are written in capitalized words notation by

¹ https://en.wikipedia.org/wiki/Object-oriented_programming

convention. As an example the code listing B.1 define a class named Circle after importing the NumPy library that will be required in the successive developments of the class.

```
1 import numpy as np
2
3 class Circle:
4     # Attributes an methods here
```

Listing B.1 Defining a new class in Python

The attributes of the class are defined using a method called `__init__()`. The method `__init__()` can contain many parameters, but the first one is always a variable called `self`.

As an example, in the code listing B.2 we define the attribute `radius` to the class Circle.

```
1 import numpy as np
2
3 class Circle:
4
5     def __init__(self, radius):
6         self.radius = radius
```

Listing B.2 Adding attributes to a class

Finally, methods are functions that are defined inside a class and can only be called from an object belonging to that specific class. The code listing B.3 report the implementation of the methods `description()`, `area()`, `circumference()`, and `diameter()`, respectively.

```
1 import numpy as np
2
3 class Circle:
4
5     def __init__(self, radius):
6         self.radius = radius
7
8     # my first Instance method
9     def description(self):
10        return "circle with radius equal to {:.2f}".format(self.
11            radius)
12
13    # my secong instance method
14    def area(self):
15        return np.pi * self.radius ** 2
16
17    # my secong instance method
```

```
17     def circumference(self):
18         return 2 * np.pi * self.radius
19
20     # my tird instance method
21     def diameter(self):
22         return 2 * np.pi
```

Listing B.3 Adding methods to a class

Finally, the code listing B.4 explain how to create (i.e., instantiate) a new Circle object named my my_Circle, print its description and calculate its area.

```
1 import numpy as np
2
3 class Circle:
4
5     def __init__(self, radius):
6         self.radius = radius
7
8     # my first Instance method
9     def description(self):
10        return "circle with radius equal to {:.2f}".format(self.
11            radius)
12
13    # my secong instance method
14    def area(self):
15        return np.pi * self.radius ** 2
16
17    # my secong instance method
18    def circumference(self):
19        return 2 * np.pi * self.radius
20
21    # my tird instance method
22    def diameter(self):
23        return 2 * np.pi
24
25 my_Circle = Circle(radius=2)
26
27 # Description
28 print(my_Circle.description())
29
30 # Calculate and report the area
31 my_Area = my_Circle.area()
32
33 # Reporting the area of my_Circle
34 print("The area of a {} is equal to {:.2f}".format(my_Circle.
35     description(), my_Area))
```

Listing B.4 Instatiating an object from the Circle class and using methods (i.e., functions)

Appendix C

The Matplotlib Object Oriented API

C.1 Matplotlib Application Programming Interfaces

As reported in the Section 3.1, there are two main Application Programming Interfaces (APIs) to use Matplotlib:

OO-style: Using the OO-style, you explicitly define the objects governing the content and the aesthetics of a diagram (i.e., figures and axes) and call methods on them to create your diagram.

pyplot style: It is the simplest way of plotting in matplotlib. Using the pyplot style, you rely on pyplot that automatically creates and manages the objects governing your diagram. Then, you use the pyplot functions for plotting.

Regarding the use of a specific style, the official documentation of matplotlib reports (Feb, 2021): "Matplotlib's documentation and examples use both the OO and the pyplot approaches (which are equally powerful), and you should feel free to use either (however, it is preferable pick one of them and stick to it, instead of mixing them). In general, we suggest to restrict pyplot to interactive plotting (e.g., in a Jupyter notebook), and to prefer the OO-style for non-interactive plotting (in functions and scripts that are intended to be reused as part of a larger project)."¹

C.2 Matplotlib Object Oriented API

As reported in the Section 1.2 and Appendix B, when using the Object Oriented programming paradigm, everything is an object, instantiated from a class. The following descriptions are taken and adapted from the matplotlib official documentation².

¹ <https://matplotlib.org/stable/tutorials/introductory/usage.html>

² <https://matplotlib.org>

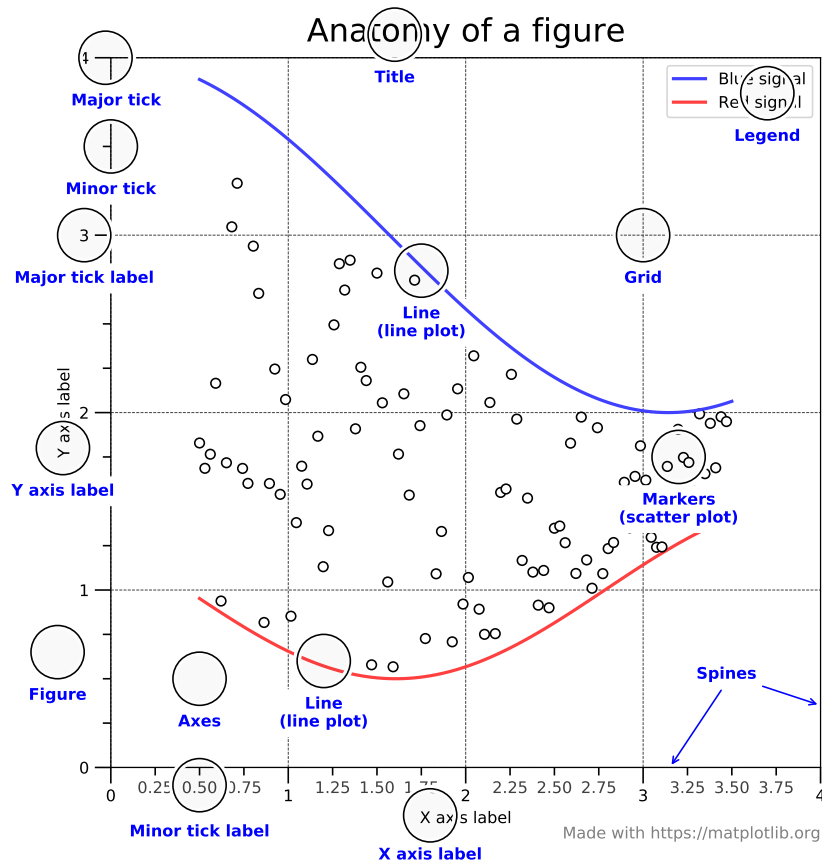


Fig. C.1 Anatomy of a matplotlib Figure

The main classes governing a diagram in matplotlib are:

Figure: the figure object embeds the whole diagram and it keeps track of all the child Axes, the artists (e.g., titles, figure legends, etc), and the canvas. A figure can contain any number of Axes, but will typically have at least one.

Axes: Axes are what you typically think when using the word 'plot'. It is the region of the Figure where you plot your data. A given figure can host many Axes, but a specific Axes object can only be in one Figure.

Axis: The Axis take care of setting the graph limits and generating the ticks (i.e., the marks on the axis) and ticklabels (i.e., strings labeling the ticks). The location of the ticks is determined by an object called Locator, and the ticklabel strings are formatted by a Formatter. Tuning the Locator and Formatter allows you to get a very fine control over tick locations and labels. Data limits can be also controlled via the

axes.Axes.set_xlim() and axes.Axes.set_ylim() methods). Each Axes has a title (set via set_title()), an x-label (set via set_xlabel()), and a y-label set via set_ylabel()). Note that Axes and Axis are two different type of objects in matplotlib.

Artists: In few word, an artist is any object that you can see within a Figure. Artists includes Text objects, Line2D objects, collections objects and many other objects. When the figure is rendered, all of the artists are drawn to the canvas.

The Fig. C.1 (generated using a script available in the official documentation³) reports the main anatomy of a matplotlib Figure. In detail, Fig. C.1 highlights the main objects that you can manage to personalize further a matplotlib diagram.

C.3 Fine Tuning of Geological Diagrams using the OO-style

Using the OO-style, we can access to any class in matplotlib. These classes provide us many attributes an methods to perform the fine tuning a geological diagram.

To provide an example, the code listing C.1 highlights, with embedded referencing to the official documentation, how to perform additional personalizations to a geological diagram we developed in the present book (Fig. 4.2). Making the fine tuning, we will improve further the quality of our artworks. Fig C.2 reports the result of code listing C.1.

```

1 import matplotlib.pyplot as plt
2 import matplotlib as mpl
3 from matplotlib.ticker import MultipleLocator, FormatStrFormatter
  , AutoMinorLocator
4 import pandas as pd
5 import numpy as np
6
7 myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
  sheet_name='Supp_traces')
8
9 fig, ax = plt.subplots()
10 # Figure managment
11 # https://matplotlib.org/stable/api/_as_gen/matplotlib.figure.
  Figure.html
12
13 # Axes managment
14 # https://matplotlib.org/stable/api/axes_api.html
15
16 # select your style
17 # https://matplotlib.org/stable/gallery/style_sheets/
  style_sheets_reference.html
18 mpl.style.use('ggplot')
19
20 # Make the plot

```

³ <https://matplotlib.org/stable/gallery/showcase/anatomy.html>

```

21 ax.hist(myDataset.Zr, density=True, bins='auto', color='Tab:blue'
    , edgcolor='k', alpha=0.8, label = 'CFC recent activity')
22
23 # Commonnly used personalizations
24 ax.set_xlabel('Zr [ppm]')
25 ax.set_ylabel('Probability density')
26 ax.set_title('Zr sample distribution')
27 ax.set_xlim(-100, 1100)
28 ax.set_ylim(0,0.0055)
29 ax.set_xlabel(r'Zr [ $\mu \cdot g^{-1}$ ]')
30 ax.set_ylabel('Probability density')
31 ax.set_xticks(np.arange(0, 1100 + 1, 250)) # adjust the x tick
    frequency
32 ax.set_yticks(np.arange(0, 0.0051, .001)) # adjust the y tick
    frequency
33
34
35 # Major and minor ticks
36 # https://matplotlib.org/stable/gallery/ticks_and_spines/
    major_minor_demo.html
37
38 ax.xaxis.set_minor_locator(AutoMinorLocator())
39
40 ax.tick_params(which='both', width=1)
41 ax.tick_params(which='major', length=7)
42 ax.tick_params(which='minor', length=4)
43
44 ax.yaxis.set_minor_locator(MultipleLocator(0.0005))
45
46 ax.tick_params(which='both', width=1)
47 ax.tick_params(which='major', length=7)
48 ax.tick_params(which='minor', length=4)
49
50
51 # Spine management
52 # https://matplotlib.org/stable/api/spines_api.html
53
54 ax.spines["top"].set_color("#363636")
55 ax.spines["right"].set_color("#363636")
56 ax.spines["left"].set_color("#363636")
57 ax.spines["bottom"].set_color("#363636")
58
59 # Spine placement
60 # https://matplotlib.org/stable/gallery/ticks_and_spines/
    spine_placement_demo.html
61
62 # Advanced Annotations
63 # https://matplotlib.org/stable/tutorials/text/annotations.html#
    plotting-guide-annotation
64 ax.annotate("Mean Value",
65             xy=(myDataset.Zr.mean(), 0.0026), xycoords='data',
66             xytext=(myDataset.Zr.mean() + 250, 0.0035),
        textcoords='data',
67             arrowprops=dict(arrowstyle="fancy",

```

```

68         color="0.5",
69         shrinkB=5,
70         connectionstyle="arc3,rad=0.3",
71     ),
72 )
73
74 ax.annotate("Modal \n value ",
75            xy=(294, 0.0045), xycoords='data',
76            xytext=(0, 0.005), textcoords='data',
77            arrowprops=dict(arrowstyle="fancy",
78                            color="0.5",
79                            shrinkB=5,
80                            connectionstyle="arc3,rad=-0.3",
81                        ),
82 )
83
84 # Legend
85 # https://matplotlib.org/stable/api/legend_api.html
86 ax.legend(title = 'My Legend')
87
88 fig.tight_layout()

```

Listing C.1 Advanced personalization of matplotlib diagrams

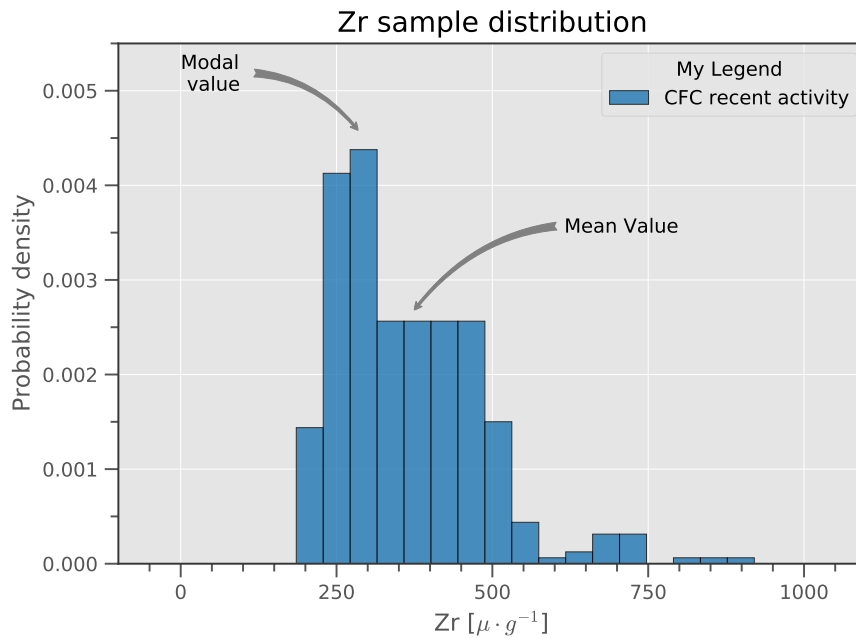


Fig. C.2 Result of the personalization of code listing C.1

Appendix D

Working with pandas

D.1 How to perform common operations in pandas

Importing an Excel file:

```
1 In [1]: import pandas as pd
2
3 In [1]: myDataset = pd.read_excel('Smith_glass_post_NYT_data.xlsx',
  sheet_name='Supp_traces')
```

Importing an .csv file:

```
1 In [1]: import pandas as pd
2
3 In [1]: myDataset = MyData = pd.read_csv('DEM.csv')
```

Get the column labels

```
1 In [3]: myDataset.columns
2 Out[3]: Index(['Analysis no.', 'Strat. Pos.', 'Eruption', 'controlcode', 'Sample', 'Epoch', 'Crater size', 'Date of analysis', 'Si/bulk cps', 'SiO2* (EMP)', 'Sc', 'Rb', 'Sr', 'Y', 'Zr', 'Nb', 'Cs', 'Ba', 'La', 'Ce', 'Pr', 'Nd', 'Sm', 'Eu', 'Gd', 'Tb', 'Dy', 'Ho', 'Er', 'Tm', 'Yb', 'Lu', 'Hf', 'Ta', 'Pb', 'Th', 'U'], dtype='object')
```

Get the shape (i.e. height and width) of a DataFrame

```
1 In [4]: myDataset.shape
2 Out[4]: (370, 37)
```

Selecting a single column

```
1 In[5]: myDataset['Rb']
2 Out[5]:
3 0      355.617073
4 1      367.233701
5 2      293.320592
6 3      344.871192
7 4      352.352196
8      ...
9 365    358.479709
10 366    405.655463
11 367    328.080366
12 368    333.859656
13 369    351.240272
14 Name: Rb, Length: 370, dtype: float64
```

or

```
1 In[6]: myDataset.Rb
2 Out[6]:
3 0      355.617073
4 1      367.233701
5 2      293.320592
6 3      344.871192
7 4      352.352196
8      ...
9 365    358.479709
10 366    405.655463
11 367    328.080366
12 368    333.859656
13 369    351.240272
14 Name: Rb, Length: 370, dtype: float64
```

Selecting the first two rows of the whole DataFrame

```
1 In[7]: myDataset[0:2]
2 Out[7]:
3 Analysis no.  ...      Pb      Th      U
4 0      ...      60.930984  35.016435  9.203411
5 1      ...      59.892427  34.462577  10.459280
6 [2 rows x 37 columns]
```


Selecting the first four rows of a single columns

```

1 In[8]: myDataset['Rb'][0:4]
2 Out[8]:
3 0    355.617073
4 1    367.233701
5 2    293.320592
6 3    344.871192
7 Name: Rb, dtype: float64

```

Converting the first four rows of a single columns to a NumPy array

```

1 Out[9]: myDataset.Rb[0:4].to_numpy()
2 Out[9]: array([355.61707274, 367.23370121, 293.32059158,
344.87119168])

```

Selecting a single cell

```

1 In[10]: myDataset['Rb'][4]
2 Out[10]: 352.3521959503882

```

or, using the indexes of the columns and rows (note that rows and columns are reversed than the previous example)

```

1 In[11]: myDataset.iloc[4,11]
2 Out[11]: 352.3521959503882

```

Sorting

```

1 In[12]: myDataset.sort_values(by='SiO2* (EMP)', ascending=False)
2 Out[12]:
3 Analysis no. ... SiO2* (EMP) ... Th U
4 228 ... 62.410000 ... 56.114101 15.548608
5 236 ... 62.410000 ... 47.402098 12.345041
6 ... ... ... ... ...
7 304 ... 54.425402 ... 16.539421 5.256582
8 318 ... 54.425402 ... 16.539421 5.256582
9 [370 rows x 37 columns]

```

Filtering:

1) define a sub DataFrame containing all the samples with Zirconium above 400

```
1 In[13]: myDataset1 = myDataset[myDataset.Zr > 400]
```

2) define a sub DataFrame containing all the samples with Zirconium between 400 and 450

```
1 In[14]: myDataset2 = myDataset[((myDataset.Zr > 400)&(myDataset.Zr < 500))]
```

Managing Missing Data:

1) drop any rows that have missing data

```
1 In[15]: myDataset3 = myDataset.dropna(how='any')
2 In[16]: myDataset.shape
3 Out[16]: (370, 37) <- the original data set
4 In[17]: myDataset3.shape
5 Out[17]: (366, 37) <- 4 samples contained missing data
```

2) replace missing data with a fixed value (e.g., 5)

```
1 In[18]: myDataset4 = myDataset.fillna(value=5)
```

Further Readings

Part I: Python for Geologists, a kick-off

- Beazley, D. M., & Jones, B. K. (K. (2013). *Python cookbook*.
- Bisong, E. (2019). Matplotlib and Seaborn. *Building machine learning and deep learning models on google cloud platform* (pp. 151–165). Apress. https://doi.org/10.1007/978-1-4842-4470-8{_}12
- Bressert, E. (2012). *SciPy and NumPy: An Overview for Developers*. O'Reilly Media, Inc.
- Chen, D. Y. (2017). *Pandas for everyone : Python data analysis*. Addison-Wesley Professional.
- Dowek, G., & Lévy, J.-J. (2011). *Introduction to the Theory of Programming Languages*. Springer London. <https://doi.org/10.1007/978-0-85729-076-2>
- Downey, A. (2016). *Think Python*.
- Gabbrielli, M., & Martini, S. (2010). *Programming Languages: Principles and Paradigms*. Springer London. <https://doi.org/10.1007/978-1-84882-914-5>
- Hunt, J. (2019). *A Beginners Guide to Python 3 Programming*. Springer International Publishing. <https://doi.org/10.1007/978-3-030-20290-3>
- Lubanovic, B. (2019). *Introducing Python : modern computing in simple packages*.
- Matthes, E. (2019). *Python crash course : a hands-on, project-based introduction to programming*.
- Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., . . . Scopatz, A. (2017). SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3, e103. <https://doi.org/10.7717/peerj-cs.103>
- Paper, D. (2020). *Hands-on Scikit-Learn for Machine Learning Applications*. Apress. <https://doi.org/10.1007/978-1-4842-5373-1>
- Rollinson, H. (1993). *Using Geochemical Data: Evaluation, Presentation, Interpretation*. Routledge.
- Rossant, C. (2018). *IPython Cookbook, Second Edition*. Pack.
- Smith, V., Isaia, R., & Pearce, N. (2011). Tephrostratigraphy and glass compositions of post-15 kyr Campi Flegrei eruptions: implications for eruption history and chronostratigraphic markers. *Quaternary Science Reviews*, 30(25-26), 3638–3660. <https://doi.org/10.1016/J.QUASCIREV.2011.07.012>
- Sweigart, A. (2019). *Automate the boring stuff with Python : practical programming for total beginners*.
- Turbak, F. A., & Gifford, D. K. (2008). *Design Concepts in Programming Languages*. MIT Press.
- Van Roy, P., & Haridi, S. (2004). *Concepts, Techniques, and Models of Computer Programming*. MIT Press.

Part II: Describing Geological Data

- Blundy, J., & Wood, B. (1994). Prediction of crystal-melt partition coefficients from elastic moduli. *Nature*, 372(6505), 452–454. <https://doi.org/10.1038/372452a0>
- Branch, M. A., Coleman, T. F., & Li, Y. (1999). A Subspace, Interior, and Conjugate Gradient Method for Large-Scale Bound-Constrained Minimization Problems. *SIAM Journal on Scientific Computing*, 21(1), 1–23. <https://doi.org/10.1137/S1064827595289108>
- Chatterjee, S., & Hadi, A. S. (2013). *Regression Analysis by Example* (fifth edit).
- Healy, K. (2019). *Data Visualization: A Practical Introduction*.
- Kopka, H., & Daly, P. W. (2003). *Guide to LaTeX*. Addison-Wesley Professional.
- Lamport, L. (1994). *LaTeX: A document preparation system, User's guide and reference manual*. Addison Wesley.
- Meltzer, A., & Kessel, R. (2020). Modelling garnet-fluid partitioning in H₂O-bearing systems: a preliminary statistical attempt to extend the crystal lattice-strain theory to hydrous systems. *Contributions to Mineralogy and Petrology*, 175(8), 80. <https://doi.org/10.1007/s00410-020-01719-8>
- Mittelbach, F., Goossens, M., Braams, J., Carlisle, D., & Rowley, C. (2004). *No Title The LaTeX Companion, 2nd edition*. Addison-Wesley Professional.
- Montgomery, D. C., Peck, E. A., & G. Geoffrey, V. (2012). *Introduction to Linear Regression Analysis*. Wiley.
- Moré, J. J. (1978). The Levenberg-Marquardt algorithm: Implementation and theory. In G. Watson (Ed.), *Numerical analysis. lecture notes in mathematics* (pp. 105–116). Springer, Berlin, Heidelberg. <https://doi.org/10.1007/BFb0067700>
- Motulsky, H., & Christopoulos, A. (2004). *Fitting models to biological data using linear and nonlinear regression : a practical guide to curve fitting*. Oxford University Press.
- Ross, S. M. (2017). *Introductory statistics* (4th Edition). Academic Press.
- Seber, G. A. F. (A. F., & Wild, C. J. (J. (1989). *Nonlinear regression*. Wiley.
- Tufte, E. (2001). *The Visual Display of Quantitative Information* (2nd editio). Graphics Press.
- Voglis, C., & Lagaris, I. E. (2004). A Rectangular Trust Region Dogleg Approach for Unconstrained and Bound Constrained Nonlinear Optimization. In T. Simos & G. Maroulis (Eds.), *Wseas international conference on applied mathematics, corfu, greece* (pp. 562–565). Taylor; Francis Inc. <https://doi.org/10.1201/9780429081385-138>

Part III: Integrals and Differential Equations in Geology

- Agarwal, R. P., & O'Regan, D. (2008). *An Introduction to Ordinary Differential Equations*. Springer. <https://doi.org/9780387712758>

- Anderson, D. L. (1989). *Theory of the Earth*. Blackwell Scientific Publications.
- Atkinson, K. E., Han, W., & Stewart, D. (2009). *Numerical Solution of Ordinary Differential Equations*. John Wiley & Sons, Inc. <https://doi.org/10.1002/9781118164495>
- Burd, A. (2019). *Mathematical Methods in the Earth and Environmental Sciences*. Cambridge University Press. <https://doi.org/10.1017/9781316338636>
- Costa, F., Chakraborty, S., & Dohmen, R. (2003). Diffusion coupling between trace and major elements and a model for calculation of magma residence times using plagioclase. *Geochimica et Cosmochimica Acta*, 67(12), 2189–2200. [https://doi.org/10.1016/S0016-7037\(02\)01345-5](https://doi.org/10.1016/S0016-7037(02)01345-5)
- Costa, F., Shea, T., & Ubide, T. (2020). Diffusion chronometry and the timescales of magmatic processes. *Nature Reviews Earth & Environment*, 1(4), 201–214. <https://doi.org/10.1038/s43017-020-0038-x>
- Crank, J. (1975). *The mathematics of diffusion* (2nd ed.). Clarendon Press Oxford [England].
- Dziewonski, A. M., & Anderson, D. L. (1981). Preliminary reference Earth model. *Physics of the Earth and Planetary Interiors*, 25(4), 297–356. [https://doi.org/10.1016/0031-9201\(81\)90046-7](https://doi.org/10.1016/0031-9201(81)90046-7)
- Fick, A. (1855). Ueber Diffusion. *Annalen der Physik und Chemie*, 170(1), 59–86. <https://doi.org/10.1002/andp.18551700105>
- Griffiths, D. F., & Higham, D. J. (2010). *Numerical Methods for Ordinary Differential Equations*. Springer London. <https://doi.org/10.1007/978-0-85729-148-6>
- King, D., Billingham, J., & Otto, S. R. (2003). *Differential equations. Linear, non-linear, ordinary, partial*. Cambridge University Press.
- Li, Z., Qiao, Z., & Tang, T. (2017). *Numerical Solution of Differential Equations*. Cambridge University Press. <https://doi.org/10.1017/9781316678725>
- Linge, S., & Langtangen, H. P. (2017). *Finite Difference Computing with PDEs* (1st ed.). Springer International Publishing. <https://doi.org/10.1007/978-3-319-55456-3>
- Mazumder, S. (2015). *Numerical methods for partial differential equations: Finite difference and finite volume methods*. Academic press.
- Meurer, A., Smith, C. P., Paprocki, M., Čertík, O., Kirpichev, S. B., Rocklin, M., Kumar, A., Ivanov, S., Moore, J. K., Singh, S., Rathnayake, T., Vig, S., Granger, B. E., Muller, R. P., Bonazzi, F., Gupta, H., Vats, S., Johansson, F., Pedregosa, F., . . . Scopatz, A. (2017). SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3, e103. <https://doi.org/10.7717/peerj-cs.103>
- Moore, A., Coogan, L., Costa, F., & Perfit, M. (2014). Primitive melt replenishment and crystal-mush disaggregation in the weeks preceding the 2005–2006 eruption 9° 50' N, EPR. *Earth and Planetary Science Letters*, 403, 15–26. <https://doi.org/10.1016/J.EPSL.2014.06.015>
- Morton, K. W., & Mayers, D. F. (2005). *Numerical Solution of Partial Differential Equations*. Cambridge University Press. <https://doi.org/10.1017/CBO9780511812248>
- Priestley, H. (1997). *Introduction to Integration*. Oxford University Press.

- Slavinić, P., & Cvetković Marko. (2016). Volume calculation of subsurface structures and traps in hydrocarbon exploration — a comparison between numerical integration and cell based models. *Open Geosciences*, 8(1). <https://doi.org/10.1515/geo-2016-0003>
- Strang, G., Herman, E., OpenStax College, & Open Textbook Library. (2016). *Calculus. Volume 1*. OpenStax - Rice University.
- Zill, D. (2012). *A First Course in Differential Equations with Modeling Applications*. Cengage Learning, Inc.

Part IV: Probability Density Functions and Error Analysis

- Agterberg, F. (2018). Statistical Modeling of Regional and Worldwide Size-Frequency Distributions of Metal Deposits. *Handbook of mathematical geosciences* (pp. 505–523). Springer International Publishing. https://doi.org/10.1007/978-3-319-78999-6{_}26
- Ahrens, L. H. (1953). A fundamental law of geochemistry. <https://doi.org/10.1038/1721148a0>
- Ballio, F., & Guadagnini, A. (2004). Convergence assessment of numerical Monte Carlo simulations in groundwater hydrology. *Water Resources Research*, 40(4), 4603. <https://doi.org/10.1029/2003WR002876>
- Barbu, A., & Zhu, S.-C. (2020). *Monte Carlo Methods* (1st Editio). Springer Nature.
- Davies, J., Marzoli, A., Bertrand, H., Youbi, N., Ernesto, M., & Schaltegger, U. (2017). End-Triassic mass extinction started by intrusive CAMP activity. *Nature Communications*, 8(1), 15596. <https://doi.org/10.1038/ncomms15596>
- Everitt, B. (2006). *The Cambridge Dictionary of Statistics* (3rd ed.). Cambridge University Press.
- Gramacki, A. (2018). *Nonparametric Kernel Density Estimation and Its Computational Aspects* (Vol. 37). Springer International Publishing. https://doi.org/10.1007/978-3-319-71688-6{_}4
- Haramoto, H., Matsumoto, M., & L'Ecuyer, P. (2008). A Fast Jump Ahead Algorithm for Linear Recurrences in a Polynomial Space. *Sequences and their applications - seta 2008* (pp. 290–298). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-85912-3{_}26
- Hughes, I., & Hase, T. (2010). *Measurements and their Uncertainties: A practical guide to modern error analysis*. Oxford University Press, US.
- Johnston, D. (2018). *Random number generators - principles and practices: a guide for engineers and programmers*. Walter de Gruyter GmbH.
- Liu, S. A., Wu, H., Shen, S. Z., Jiang, G., Zhang, S., Lv, Y., Zhang, H., & Li, S. (2017). Zinc isotope evidence for intensive magmatism immediately before the end-Permian mass extinction. *Geology*, 45(4), 343–346. <https://doi.org/10.1130/G38644.1>

- O'Neill, M. E. (2014). PCG : A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation.
- Puetz, S. J. (2018). A relational database of global U–Pb ages. *Geoscience Frontiers*, 9(3), 877–891. <https://doi.org/10.1016/J.GSF.2017.12.004>
- Reimann, C., & Filzmoser, P. (2000). Normal and lognormal data distribution in geochemistry: Death of a myth. Consequences for the statistical treatment of geochemical and environmental data. *Environmental Geology*, 39(9), 1001–1014. <https://doi.org/10.1007/s002549900081>
- Rocholl, A. (1998). Major and Trace Element Composition and Homogeneity of Microbeam Reference Material: Basalt Glass USGS BCR-2G. *Geostandards and Geoanalytical Research*, 22(1), 33–45. <https://doi.org/10.1111/j.1751-908X.1998.tb00543.x>
- Salmon, J. K., Moraes, M. A., Dror, R. O., & Shaw, D. E. (2011). Parallel random numbers: As easy as 1, 2, 3. *SC '11: Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–12. <https://doi.org/10.1145/2063384.2063405>
- Schwartz, L. M. (1975). Random Error Propagation by Monte Carlo Simulation. *Analytical Chemistry*, 47(6), 963–964. <https://doi.org/10.1021/ac60356a027>
- Silverman, B. W. (1998). *Density Estimation for Statistics and Data Analysis*. Chapman; Hall/CRC.
- Taylor, J. R. (1997). *An Introduction to Error Analysis: The Study of Uncertainties in Physical Measurements* (2nd edition). University Science Books.
- Tegner, C., Marzoli, A., McDonald, I., Youbi, N., & Lindström, S. (2020). Platinum-group elements link the end-Triassic mass extinction and the Central Atlantic Magmatic Province. *Scientific Reports*, 10(1), 1–8. <https://doi.org/10.1038/s41598-020-60483-8>
- Tobutt, D. C. (1982). Monte Carlo Simulation methods for slope stability. *Computers and Geosciences*, 8(2), 199–208. [https://doi.org/10.1016/0098-3004\(82\)90021-8](https://doi.org/10.1016/0098-3004(82)90021-8)
- Troyan, V., & Kiselev, Y. (2010). *Statistical Methods of Geophysical Data Processing*. World Scientific Publishing Company.
- Ulianov, A., Müntener, O., & Schaltegger, U. (2015). The ICPMS signal as a Poisson process: a review of basic concepts. *Journal of Analytical Atomic Spectrometry*, 30(6), 1297–1321. <https://doi.org/10.1039/C4JA00319E>
- Wang, Z., Yin, Z., Caers, J., & Zuo, R. (2020). A Monte Carlo-based framework for risk-return analysis in mineral prospectivity mapping. *Geoscience Frontiers*, 11(6), 2297–2308. <https://doi.org/10.1016/j.gsf.2020.02.010>

Part V: Robust Statistics and Machine Learning

- Abedi, M., Norouzi, G.-H., & Bahroudi, A. (2012). Support vector machine for multi-classification of mineral prospectivity areas. *Computers and Geosciences*, 46, 272–283. <https://doi.org/10.1016/j.cageo.2011.12.014>

- Bentley, J. L. (1975). Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9), 509–517. <https://doi.org/10.1145/361002.361007>
- Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Springer.
- Breiman, L. (1996). Bagging Predictors. *Machine Learning*, 24(2), 123–140. <https://doi.org/10.1023/A:1018054314350>
- Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5–32. <https://doi.org/10.1023/A:1010933404324>
- Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (2017). *Classification and regression trees*. CRC Press. <https://doi.org/10.1201/9781315139470>
- Cannata, A., Montalto, P., Aliotta, M., Cassisi, C., Pulvirenti, A., Privitera, E., & Patanè, D. (2011). Clustering and classification of infrasonic events at Mount Etna using pattern recognition techniques. *Geophysical Journal International*, 185(1), 253–264. <https://doi.org/10.1111/j.1365-246X.2011.04951.x>
- D'Agostino, R., & Pearson, E. S. (1973). Tests for departure from normality. *Biometrika*, 60, 613–622.
- D'Agostino, R. B. (1971). An omnibus test of normality for moderate and large sample size. *Biometrika*, 58, 341–348.
- Devine, J. D., Murphy, M. D., Rutherford, M. J., Barclay, J., Sparks, R. S. J., Carroll, M. R., Young, S. R., & Gardner, I. E. (1998). Petrologic evidence for pre-eruptive pressure-temperature conditions, and recent reheating, of andesitic magma erupting at the Soufriere Hills Volcano, Montserrat, W.I. *Geophysical Research Letters*, 25(19), 3669–3672. <https://doi.org/10.1029/98GL01330>
- Friedman, J. H. (2002). Stochastic gradient boosting. *Computational Statistics and Data Analysis*, 38(4), 367–378. [https://doi.org/10.1016/S0167-9473\(01\)00065-2](https://doi.org/10.1016/S0167-9473(01)00065-2)
- Geurts, P., Ernst, D., & Wehenkel, L. (2006). Extremely randomized trees. *Machine Learning*, 63(1), 3–42. <https://doi.org/10.1007/s10994-006-6226-1>
- Goldstein, E., & Coco, G. (2014). A machine learning approach for the prediction of settling velocity. *Water Resources Research*, 50(4), 3595–3601. <https://doi.org/10.1002/2013WR015116>
- Hearst, M. A., Dumais, S. T., Osuna, E., Platt, J., & Scholkopf, B. (1998). Support vector machines. *IEEE Intelligent Systems and their Applications*, 13(4), 18–28. <https://doi.org/10.1109/5254.708428>
- Hirschmann, M. M., Ghiorso, M. S., Davis, F. A., Gordon, S. M., Mukherjee, S., Grove, T. L., Krawczynski, M., Medard, E., Till, C. B., . Medard, E., & Till, C. B. (2008). Library of Experimental Phase Relations (LEPR): A database and Web portal for experimental magmatic phase equilibria data. *Geochemistry, Geophysics, Geosystems*, 9(3), n/a–n/a. <https://doi.org/10.1029/2007GC001894>
- Ho, T. (1998). The random subspace method for constructing decision forests. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(8), 832–844. <https://doi.org/10.1109/34.709601>

- Huang, C., Davis, L., & Townshend, J. (2002). An assessment of support vector machines for land cover classification. *International Journal of Remote Sensing*, 23(4), 725–749. <https://doi.org/10.1080/01431160110040323>
- Huber, P. J., & Ronchetti, E. M. (2009). *Robust Statistics, 2nd Edition*. Wiley.
- Jain, A., Murty, M., & Flynn, P. (1999). Data clustering: A review. *ACM Computing Surveys*, 31(3), 264–323. <https://doi.org/10.1145/331499.331504>
- Jordan, M., & Mitchell, T. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245), 255–260. <https://doi.org/10.1126/science.aaa8415>
- Kotsiantis, S. (2007). Supervised machine learning: A review of classification techniques. *Informatica (Ljubljana)*, 31(3), 249–268.
- Le Maitre, R. (1982). *Numerical Petrology*. Elsevier.
- Lee, J., & Verleysen, M. (2009). Quality assessment of dimensionality reduction: Rank-based criteria. *Neurocomputing*, 72(7-9), 1431–1443. <https://doi.org/10.1016/j.neucom.2008.12.017>
- Maronna, R. A., Martin, R. D., & Yohai, V. J. (2006). *Robust Statistics: Theory and Methods* (2nd editio). Wiley. <https://doi.org/10.1002/0470010940>
- Masotta, M., Mollo, S., Freda, C., Gaeta, M., & Moore, G. (2013). Clinopyroxene–liquid thermometers and barometers specific to alkaline differentiated magmas. *Contributions to Mineralogy and Petrology*, 166(6), 1545–1561. <https://doi.org/10.1007/s00410-013-0927-9>
- Masotti, M., Falsaperla, S., Langer, H., Spampinato, S., & Campanini, R. (2006). Application of Support Vector Machine to the classification of volcanic tremor at Etna, Italy. *Geophysical Research Letters*, 33(20). <https://doi.org/10.1029/2006GL027441>
- Murphy, K. (2012). *Machine Learning*. MIT Press.
- Natekin, A., & Knoll, A. (2013). Gradient boosting machines, a tutorial. *Frontiers in Neuroinformatics*, 7(DEC), 21. <https://doi.org/10.3389/fnbot.2013.00021>
- Neave, D. A., Bali, E., Guðfinnsson, G. H., Halldórsson, A., Kahl, M., Schmidt, A.-s., & Holtz, F. (2019). Clinopyroxene–liquid equilibria and geothermobarometry in natural and experimental tholeiites: the 2014–2015 Holuhraun eruption, Iceland David. *Journal of Petrology*, *accepted*.
- Nimis, P. (1995). A Clinopyroxene Geobarometer for Basaltic Systems Based on Crystal-Structure Modeling. *Contribution to mineralogy and petrology*, 121(2), 115–125.
- Nimis, P., & Ulmer, P. (1998). Clinopyroxene geobarometry of magmatic rocks Part 1: An expanded structural geobarometer for anhydrous and hydrous, basic and ultrabasic systems. *Contribution to mineralogy and petrology*, 133(1-2), 122–135.
- Palettas, P. (1992). Probability Plots and Modern Statistical Software. In C. Page & R. LePage (Eds.), *Computing science and statistics*. Springer. https://doi.org/10.1007/978-1-4612-2856-1{_}84
- Pedregosa, F., Varoquaux, G. G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É.

- (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(85), 2825–2830.
- Petrelli, M., Bizzarri, R., Morgavi, D., Baldanza, A., & Perugini, D. (2017). Combining machine learning techniques, microanalyses and large geochemical datasets for tephrochronological studies in complex volcanic areas: New age constraints for the Pleistocene magmatism of central Italy. *Quaternary Geochronology*, 40, 33–44. <https://doi.org/10.1016/j.quageo.2016.12.003>
- Petrelli, M., Caricchi, L., & Perugini, D. (2020). Machine Learning Thermobarometry: Application to Clinopyroxene-Bearing Magmas. *Journal of Geophysical Research: Solid Earth*, 125(9). <https://doi.org/10.1029/2020JB020130>
- Petrelli, M., & Perugini, D. (2016). Solving petrological problems through machine learning: the study case of tectonic discrimination using geochemical and isotopic data. *Contributions to Mineralogy and Petrology*, 171(10). <https://doi.org/10.1007/s00410-016-1292-2>
- Petrelli, M., Perugini, D., Moroni, B., & Poli, G. (2003). Determination of travertine provenance from ancient buildings using self-organizing maps and fuzzy logic. *Applied Artificial Intelligence*, 17(8-9), 885–900. <https://doi.org/10.1080/713827251>
- Putirka, K. (2008). *Introduction to minerals, inclusions and volcanic processes* (Vol. 69). <https://doi.org/10.2138/rmg.2008.69.1>
- Putirka, K., Mikaelian, H., Ryerson, F., & Shaw, H. (2003). New clinopyroxene-liquid thermobarometers for mafic, evolved, and volatile-bearing lava compositions, with applications to lavas from Tibet and the Snake River Plain, Idaho. *American Mineralogist*, 88(10), 1542–1554. <https://doi.org/10.2138/am-2003-1017>
- Reimann, C., & Filzmoser, P. (2000). Normal and lognormal data distribution in geochemistry: Death of a myth. Consequences for the statistical treatment of geochemical and environmental data. *Environmental Geology*, 39(9), 1001–1014. <https://doi.org/10.1007/s002549900081>
- Shai, S.-S., & Shai, B.-D. (2013). *Understanding machine learning: From theory to algorithms* (Vol. 9781107057). Cambridge University Press. <https://doi.org/10.1017/CBO9781107298019>
- Shapiro, S. S., & Wilk, M. B. (1965). An Analysis of Variance Test for Normality (Complete Samples). *Biometrika*, 52(3/4), 591. <https://doi.org/10.2307/2333709>
- Smith, V., Isaia, R., & Pearce, N. (2011). Tephrostratigraphy and glass compositions of post-15 kyr Campi Flegrei eruptions: implications for eruption history and chronostratigraphic markers. *Quaternary Science Reviews*, 30(25-26), 3638–3660. <https://doi.org/10.1016/J.QUASCIREV.2011.07.012>
- Smola, A., & Schölkopf, B. (2004). A tutorial on support vector regression. *Statistics and Computing*, 14(3), 199–222. <https://doi.org/10.1023/B:STCO.0000035301.49549.88>
- Stephens, M. A. (1974). EDF Statistics for Goodness of Fit and Some Comparisons. *Journal of the American Statistical Association*, 69, 730–737.

- Thode, H. C. (2002). *Testing for Normality* (1st). Marcel Dekker.
- Zhang, Y., & Haghani, A. (2015). A gradient boosting method to improve travel time prediction. *Transportation Research Part C: Emerging Technologies*, 58, 308–324. <https://doi.org/10.1016/j.trc.2015.02.019>
- Zuo, R., & Carranza, E. (2011). Support vector machine: A tool for mapping mineral prospectivity. *Computers and Geosciences*, 37(12), 1967–1975. <https://doi.org/10.1016/j.cageo.2010.09.014>

