JOHANNES GUTENBERG UNIVERSITÄT MAINZ

JG|U

NUMERICAL ALGORITHMS

Luca Doria

Institut für Kernphysik
J.J. Becher-Weg 45 55128 Mainz (Germany)
doria@uni-mainz.de

Computer Science
Wintersemester 2022-2023

Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität Mainz

September 2022

Contents

1 Introduction **1**

2 Errors and Number Representation **5**

2.1 Approximation Errors 5
2.2 Rounding Errors 5
2.3 Absolute and Relative Errors 6
2.4 Error Propagation 7
2.5 Number Representation 9
2.6 Machine-specific Precision 10
2.7 Subtraction and Machine Precision 12
2.8 More on Number Representations 13

3 Linear Systems **15**

3.1 Introduction 15
3.2 The Condition Number 17
3.3 Tha Gauss Elimination Algorithm 18
3.4 Pivoting 21
3.5 Tri-diagonal and Diagonally Dominant Matrices 23
3.6 LU Decomposition 24
3.7 LU Decomposition: General Case 27
3.8 Choleski Decomposition 30
3.9 Steepest Descent and Conjugate Gradient Methods 31
3.10 Eigenvalues and Eigenvectors: the Power Method 39
3.11 Eigenvalues and Eigenvectors: the Jacobi Method 41

4 Approximation and Interpolation **49**
4.1 Linear Interpolation 49
4.2 Parabolic Interpolation 51
4.3 Cubic Splines Interpolation 52
4.4 Cubic Splines with Smooth Second Derivatives 56
4.5 Lagrange Interpolation 59
4.6 Least Squares Method 61
4.7 Linear Interpolation with the Least Squares Method 62
4.8 Error on the Estimated Linear Parameters 63

5 Root Finding **67**
5.1 Incremental Method 67
5.2 Bisection Method 68
5.3 Newton's Method 69
5.4 Multidimensional Newton's Method 71
5.5 Secant Method 73
5.6 Brent's Method 74

6 Numerical Integration **77**
6.1 Introduction 77
6.2 Newton-Cotes Methods 78
6.3 The Trapezoidal Rule 78
6.4 Simpson's Rule 80
6.5 Romberg Integration 81
6.6 Gaussian Quadrature: Introduction 86
6.7 Gaussian Quadrature: a more general case 87
6.8 Orthogonal Polynomials 88
6.9 The Gaussian Quadrature Algorithm 89
6.10 Multidimensional Integration 90
6.11 Introduction to Stochastic Integration 93
6.12 Monte Carlo Integration 94
6.13 Importance Sampling 95
6.14 Gaussian Random Numbers: Box-Muller Transformation . . 96

7 Numerical Differentiation **97**
7.1 Backward and Forward Differences 97
7.2 Central Difference 98
7.3 Second Derivative 99

7.4 Another derivation . . . 100
7.5 Derivatives with Interpolation . . . 102

8 Numerical Ordinary Differential Equations 103
8.1 Introduction to Initial Value Problems . . . 103
8.2 Euler's Method . . . 104
8.3 Runge-Kutta Method: RK2 . . . 105
8.4 Higher-order Runge Kutta Methods: RK4 . . . 108
8.5 Fouth-Order Runge-Kutta Method in two Dimensions . . . 108
8.6 Fourth-Order Runge-Kutta Method for Second-order Differential Equations . . . 109
8.7 Taylor Expansion Methods . . . 110
8.8 Stability Analysis . . . 110
8.9 Adaptive-Mesh Methods . . . 111
8.10 Application: Predator-Prey Model . . . 112
8.11 Boundary Value Problems . . . 117
8.12 Central Difference Method for Boundary Value Problems . . . 117
8.13 Upwind Difference Method . . . 118
8.14 Leapfrog method for second order differential equations . . . 119
8.15 Leapfrog Method: Application to the Damped Oscillator . . . 121
8.16 The Numerov Method . . . 123
8.17 Application to the Schrödinger Equation: Particle in a Box Potential . . . 125
8.18 Application to the Schrödinger Equation: The Hydrogen Atom . . . 126

9 Elliptic Equations 131
9.1 Introduction . . . 131
9.2 Boundary Value Problems for Elliptic Equations . . . 132
9.3 Dirichlet Problem on a Rectangle . . . 132

10 Parabolic Equations 139
10.1 Definition of the problem . . . 139
10.2 Explicit Method for the Initial-Boundary Problem for the Heat Equation . . . 139
10.3 Explicit Central Difference Method . . . 140
10.4 Implicit Central Difference Method . . . 141
10.5 The Crank-Nicolson Method . . . 142

10.6 Reaction-Diffusion Systems 144
10.7 Bidimensional Systems: Turing Instability 144
10.8 A non-linear Turing Instability Example 145

11 Financial Applications **149**
11.1 Introduction: the Ito Formula 149
11.2 The Black-Scholes Equation 151
11.3 Analytic Solution 153
11.4 Numerical schemes for the Black-Scholes Equation 156
11.5 Monte Carlo Approach 159

12 Hyperbolic Equations **163**
12.1 Introduction 163
12.1.1 The Cauchy Problem 163
12.1.2 The Initial Boundary Problem 165
12.2 Explicit Method for the Initial Boundary Problem 166
12.3 Implicit Method for the Initial Boundary Problem 167
12.4 The Lax-Wendroff Method 168

13 The Navier-Stokes Equations **171**
13.1 Introduction 171
13.2 2-Dimensional Flow 172
13.3 A general Finitie-Difference Scheme 172
13.3.1 Boundary conditions 173
13.3.2 Numerical approximation scheme 174

14 The Fourier Transform **177**
14.1 Introduction 177
14.2 Fast Fourier Transform 178
14.3 Aliasing 179

Chapter 1 | Introduction

Many situations require a numerical solution of a mathematical problem using a computer. Typical examples is the calculation of an integral or the solution of a differential equation. Actually there are more cases where the latter examples are not solvable analytically than the contrary: actually the possibility of an analytical solution should be regarded as a rare case.
While integrals of polynomial functions have analytical solutions, other fundamental integrals like

$$\int_a^b e^{-x^2} dx \quad , \tag{1.1}$$

does'nt, although it plays a central role in probability theory and in other applications.
Concerning differential equations, even simple mechanical systems like the pendulum, described by

$$\partial_{tt} x + \omega^2 \sin x = 0 \quad , \tag{1.2}$$

do not have an analytical solution. Only the "small angle approximation" $\sin x \approx x$ yields the known oscillatory solution $x \sim \sin x$.
Moving to non-linear systems, the difficulty of finding closed solutions increases even more.
The latter simple considerations show the need of reliable numerical methods for solving a very large class of problems relevant in manny applications.
The formulation of a numerical scheme must fulfill some important requirements:

1. **Consistency:** If the numerical scheme is based on a discretization, the equations must become exact if the length of the discrete step tends to zero.

2. **Stability:** Numerical errors should not grow during the computation

3. **Convergence:** The solution must converge to the exact one as the discrete steps tend to zero.

4. **Conservation:** If conservation laws are implied, these should be fulfilled by the numerical method.

5. **Boundedness:** Solutions must stay bounded.

Not all the listed requirements must hold for every method but all of them apply to the solution of equations for example.
In these notes, we will describe in a simple way different numerical methods for solving common mathematical problems in the form of computer algorithms. For most of the methods, a `Python` code will show its implementation. The codes are essential and only describe the core of the algorithms and they can be expanded with more functionalities, checks, object orientation or other programming techniques.
The implementation of the algorithms on a computer presents additional challenges in terms of errors induced by the finite representation of numbers and the need to keep the computation within manageable time and space (memory) limits.
In the next chapters, after the discussion of the errors that might be introduced by the use of a concrete computer, we will describe different mathematical problems like the solution of linear systems, interpolation, solution of algebraic equations, derivation, and integration. In the second part, we will apply the numerical derivation methods to the solution of ordinary differential equations and then to partial differential equations (PDE). This last field is extremely vast and here we will only describe the most simple methods also showing some applications to biology and finance. Among the PDEs, the Navier-Stokes equation governing fluid motion has a special standing given its difficulty. Here we will only mention the most basic problems in devising a numerical scheme for this very important equation which has wide application in many scientific and engineering fields.
We conclude with a brief discussion of the numerical Fourier transform describing one algorithm for accelereting its calculation.

Aknowledgements

I thank the students of the Winter Semester 2022-2023 of the Computer Science department at the Johannes Gutenberg University (Mainz, Germany) for the many suggestions and corrections while preparing these notes.

Luca Doria
(JGU Mainz, February 1st, 2023).

CHAPTER 2 | Errors and Number Representation

The result of a calculation with a digital computer might be affected by errors. In the following, we will refer to **accuracy** as the capability to match the *true* value with a calculated one. The word **precision** refers instead to the number of digits we can use in a computation. A computer works with a definite representation of numbers which implies always a finite precision.

2.1 Approximation Errors

Some numerical calculations can involve an infinite series of steps for reaching the true result. A typical example is the approximation of functions using the Taylor expansion. For example, we might want to calculate the value of π with the Leibniz series

$$\pi = 4\sum_{i=0}^{\infty}\frac{(-1)^i}{2i+1} = 4\cdot(1-\frac{1}{3}+\frac{1}{5}-\frac{1}{7}+...) \quad . \tag{2.1}$$

In this case, since we cannot perform the infinite sum, we have to truncate the series to a certain step $i = N$: the *accuracy* of the result will depend on N. Note that the series terms become smaller and smaller, and at a certain point we will reach the limit of the number representation of the computer: we will be thus limited also in *precision*.

2.2 Rounding Errors

As we said, the representation of numbers on a computer is finite and if expressing a number needs a very large number of digits, necessarily part of them will be lost. A

typical example are computations involving irrational numbers, like $(\sqrt{2})^2 - 2 = 0$. The result seems obvious, but if you try to perform the calculation with `Python`

```
>>> import numpy as np
>>> (np.sqrt(2))**2-2
4.440892098500626e-16
```

the result is different. The calculation of `sqrt(2)` produces a number with infinite digits and only a fine amount of them is used for the next calculation (the squaring). Swapping the square root with the square

```
>>> np.sqrt(2**2)-2
0.0
```

produces instead the correct result: no infinite-digits numbers are involved. Even the subtraction in the above example is affected by a rounding error.

2.3 Absolute and Relative Errors

Denoting with x the *true* value of a quantity and with $\bar{x}$ the result of a computation for calculating x, we define the **absolute error** the difference between them

$$\Delta x = \bar{x} - x \quad . \tag{2.2}$$

The absolute error can be positive or negative, but most of the times we are interested only in its magnitude, therefore we consider $|\Delta x| = |\bar{x} - x|$. Another practically useful definition is the **error bound**

$$|\Delta x| \leq \epsilon \quad \Rightarrow \quad \bar{x} - \epsilon \leq x \leq \bar{x} + \epsilon \quad , \tag{2.3}$$

where ϵ is an upper limit for the absolute error.
Another common expression (especially in the natural sciences [1]) for the error bound is

$$x = \bar{x} \pm \epsilon \quad . \tag{2.4}$$

In many concrete cases, the absolute error might not quantify exactly how accurate our numerical result is. Consider the following example where the "real" number is

$$x_1 = 1000.0$$

[1] In natural sciences this expression for the error usually involves statistical errors. In the present case, we are referring to the absolute, or maximal error.

and the estimated one is

$$\bar{x}_1 = 999.0$$

The absolute error is $\Delta x_1 = -1$. Consider now the case

$$x_2 = 1000000.0$$

and the estimated one is

$$\bar{x}_2 = 999999.0$$

The absolute error $\Delta x_2 = -1$ is the same as in the case of x_1, but in this case we clearly did a better job in calculating a larger number relatively to its precision. These considerations lead to the definition of the **relative error**

$$\delta x = \frac{\Delta x}{x} = \frac{\bar{x} - x}{x} \quad . \tag{2.5}$$

Analogously to the absolute error, we can introduce an error bound for the relative error

$$|\delta x| = \left| \frac{\Delta x}{x} \right| \leq \epsilon \tag{2.6}$$

2.4 Error Propagation

A natural problem is the combination of quantities affected by an error, or more generically, the calculation of a function with arguments affected by errors. The errors *propagate* through the operations needed for calculating the function leading to a result with a combined error. In the following, we will start with simple operations like sums and products and then tackle the most general case.

Addition and Subtraction of Errors

Consider the difference between two quantities $x = x_1 - x_2$ and the same for the calculated ones $\bar{x} = \bar{x_1} - \bar{x_2}$. Applying the definition of absolute error,

$$\Delta x = \Delta(x_1 - x_2) = (x_1 - x_2) - (\bar{x_1} - \bar{x_2}) = \tag{2.7}$$

$$x_1 - x_2 - (x_1 - \Delta x_1 - x_2 + \Delta x_2) = \Delta x_1 - \Delta x_2 \quad . \tag{2.8}$$

Since we are interested in error bounds, we can consider only absolute values of the errors and applying the triangular inequality we obtain

$$|\Delta x| \leq |\Delta x_1| + |\Delta x_2| \quad . \tag{2.9}$$

An analogous calculation for the sum of two quantities leads to the same result, as can be directly verified. The result tells us that the absolute error of the sum or difference of two quantitites is bounded by the sum of the errors of the single variables[2].
A very important case is represented by the situation where x_1 is very close to x_2 and we have to assess the error of their difference. Considering now the relative error

$$|\delta x| = \left|\frac{\Delta x}{x}\right| \leq \frac{\Delta x_1 + \Delta x_2}{|x_1 - x_2|} \approx (|\delta x_1| + |\delta x_2|)\frac{|x_1|}{|x_1 - x_2|} \quad , \tag{2.10}$$

where we substituted the realtive errors $\delta x_{1/2} = \Delta x_{1/2}/x_{1/2}$ and considered $x_1 \sim x_2$. It is evident that if x_1 and x_2 have close values, the relative error becomes very large. This effect is sometimes referred to *catastrophic cancellation*. Numerically subtracting two similar numbers requires always care, especially if their difference is close to the computer's precision.

Multiplication and Division

We consider now the case $x = x_1 \cdot x_2$ where the computed value is $\bar{x} = \bar{x}_1 \cdot \bar{x}_2$. Remembering the definition of relative error, we can write $\bar{x}_{1/2} = x_{1/2}(1 - \delta x_{1/2})$ and use it in the definition of relative error for x

$$\delta x = \frac{\bar{x} - x}{x} = \frac{\bar{x_1}\bar{x_2} - x_1 x_2}{x_1 x_2} = \frac{x_1(1 - \delta x_1)x_2(1 - \delta x_2) - x_1 x_2}{x_1 x_2} = \delta x_1 + \delta x_2 \quad , \tag{2.11}$$

where we neglected the higher-order term $\delta x_1 \delta x_2$. Applying again the triangular inequality

$$|\delta x| \leq |\delta x_1| + |\delta x_2| \quad , \tag{2.12}$$

we observe that the relative error of the product is bounded by the sum of the relative errors of the factors. The $x = x_1/x_2$ case leads exactly to the same result.

[2]In the case of statistical errors, the errors add in quadrature. The numerical errors are not statistical, and the presented treatment corresponds to a "worst case scenario" where the error is maximal. In practice, one error can be large, and the other one very small, but the sum of the two errors will always contain all the possible combination of cases.

The General Case of Error Propagation

In the previous sections, we analyzed the error propagation in the case of the four basic arithmetic operations among variables and now we will tackle the more general case of a sequence of operations involving variables with an error, summarized (in the case of a single variable) by a function y=f(x):

$$\Delta y = \bar{y} - y = f(\bar{x}) - f(x) \quad . \tag{2.13}$$

The presence of an error changes the result of the function and in order to estimate this change, we can expand f with a Taylor series

$$\Delta y = f(x + \Delta x) - f(x) = f(x) + \frac{df}{dx}(\bar{x} - x) + ... - f(x) \approx \frac{df}{dx}(\bar{x} - x) \quad , \tag{2.14}$$

where we disregarded high-order terms assumung $\bar{x} - x$ as a small quantity. The previous formula gives a recipe for calculating the absolute error of a function of one variable. The relative errors is thus

$$\delta y = \frac{\Delta y}{y} \approx \frac{1}{f(x)}\frac{df}{dx}\Delta x = \frac{x}{f(x)}\frac{df}{dx}\delta x \quad . \tag{2.15}$$

In the multi-variable case $y = f(x_0, x_1, ..., x_n - 1)$, the absolute error is given by

$$\Delta y \approx \sum_{i=0}^{n-1} \frac{\partial f}{\partial x_i}\Delta x_i \quad , \tag{2.16}$$

and the relative error by

$$\delta y \approx \sum_{i=0}^{n-1} \frac{x_i}{f(x_0, x_1, ..., x_{n-1})}\frac{\partial f}{\partial x_i}\delta x_i \quad . \tag{2.17}$$

2.5 Number Representation

Internally, a computer represents numbers as bit strings. The `Python` language does not restrict the number of digits for an integer number, and there are no limits to how large an integer can be. Real numbers are represented with 64 bits (this is the `float` type) in the form

$$\pm \quad \text{mantissa} \quad \times \quad 10^{\text{exponent}} \quad . \tag{2.18}$$

The 64 bit limit means that not all the real numbers can be represented but there is a limit in precision, which is given by these two numbers

$$\pm 4.9 \times 10^{-324} \quad \leftrightarrow \quad \pm 1.8 \times 10^{308} \quad . \tag{2.19}$$

In this representation, real numbers can be represented with a precision of 1 part in $2^{52} \approx 2.2 \times 10^{-16}$ (or 15-16 decimal digits) which is sufficient for almost all the applications.
Technically, `Python` uses 64 bits as follows:

- 1 bit for the overall sign (positive or negative)
- 11 bits for the exponent
- 52 bits for the significant digits.

2.6 Machine-specific Precision

The machine precision ϵ_m can be defined as the smallest number larger than 1 which can be expressed by the representation of the real numbers adopted. In the previous section, we saw that in the mantissa/exponent representation and 64 bits, this number is

$$\boxed{\epsilon_m \approx 2.2 \times 10^{-16} \quad .} \tag{2.20}$$

It is interesting to explore the behaviour of the numbers as they come close to the machine precision limit. In order the look directly at what happens, we can try to take a number and apply some operations to reduce it, for example, we can half it many times:

```
1 #start with a number close to the machine precision
    limit
2 x = 0.5**50
3 for step in range(0,3):
4       x = x/2
5       print(step, 1+x, x)
```

The output of the previous code fragment is

```
0 1.0000000000000004 4.440892098500626e-16
1 1.0000000000000002 2.220446049250313e-16
2 1.0 1.1102230246251565e-16
```

The second line displays the smallest increment from 1, while in the last line no increment is present and the variable x stays equal to 1.
We can also try to get close to the machine limit with a direct calculation:

```
>>> 1.0 + 2.3e-16
1.0000000000000002
>>> 1.0 + 1.6e-16
1.0000000000000002
>>> 1.0 + 1.12e-16
1.0000000000000002
>>> 1.0 + 1.1e-16
1.0
```

The last test looks surprising at first sight: $1.6e-16$ for example is smaller than the machine precision, but still we obtain a number which is larger than one. The explanation lies in the *rounding*: $1.6e-16$ is rounded up to the machine precision 2.2e-16, while 1.1e-16 is rounded down to 1.
An interesting side-note is that very small numbers can be stored: for example we can assign 10^{-300} to a variable and our representation system will work without problems. Instead, the number $1.0 + 10^{300}$ will be truncated, since the precision is limited to $O(10^{-16})$.

2.7 Subtraction and Machine Precision

We have already seen, that subtracting numbers can lead to large errors. If we consider only the machine precision error ϵ_m and the relative error of the difference given by Eq. 2.10 we have

$$|\delta x| \leq \frac{|x_1|}{|x_1 - x_2|} \cdot 2\epsilon_m \quad . \tag{2.21}$$

The last result shows that in general, the relative subtraction error can be potentially much larger that the machine precision.
When numbers are close to each other, the error can be significant and might lead to a loss of significant figures. Consider this example where we subtract close numbers:

$$1.84759264759824671064 - 1.84759264759824000000 =$$

$$0.00000000000000671064$$

The calculation is trivial "by hand", but this is what a `Python` code would produce:

```
>>> 1.84759264759824671064 - 1.84759264759824000000
6.661338147750939e-15
```

The figures after the 16th one are affected by the machine precision error. This can be seen also letting `Python` print out the first number, which has more figure than it is allowed by the machine precision:

```
>>> 1.84759264759824671064
1.8475926475982467
```

while the second number is represented correctly. From Eq. 2.21, we have $\epsilon_m \approx 0.06$ or a 6% relative error in the result.

2.8 More on Number Representations

Not all floating point numbers are exactly representable on a machine with finite precision. It might be surprising that for example numbers like 0.1, 0.2, and 0.3 are not exactly representable. This can be seen with the following simple code:

```
>>> x = 0.1 + 0.2
>>> print(x)
0.30000000000000004
```

If you try to print just 0.1 or 0.2, the interpreter will display the correct result, but this happens only because the representation error is so small, that it is truncated in the output. Instead, in the sum $0.1 + 0.2$ the errors in both numbers accumulate and become visible. It turns out, that a number is exactly representable only if it is of the form x_1/x_2 where x_1 and x_2 are integers and x_2 is a power of 2. Clearly, the machine precision error remains and depends on how many bits you use in your representation. In more detail, a floating-point number x is calculated fixing a base **b** (which is 2 usually, or 10), an exponent **e** and a "significant" **s**, or mantissa. For example, $x = 12345.678 = 1.2345678 \times 10^3$ and in general

$$x = \frac{s}{b^{p-1}} \cdot b^e \quad ,$$

where **p** is the precision. From the formula, it is clear that if b=2, the number 0.1=1/10 cannot be exactly represented.
Actually, the number in binary is the infinite sequence 0.00011001100110011..., where the pattern of 0011 repeats, therefore we cannot store such a number[3]. Indeed, the binary of 10 is 1010 and when we divide 1 by 1010, the division process

[3]Number representation once lead to a tragic episode, as reported by Computer Scientist R. Brent: During the first Iraq war (1991) an Iraqi Scud missile was fired towards Dhahran, Saudi Arabia. A US Patriot missile failed to intercept it, leading to the death of 28 soldiers and 97 more injured. It turned out that the Patriot system had an internal clock that incremented every 0.1 seconds, and the time (in seconds) was determined by multiplying the counter value by a 24-bit approximation to 1/10. As we learned, numbers like 1/N are infinite binary sequences, if they are not a power of 2. The Patriot system was multiplying by a number close to 0.0999999 instead of 0.1000000. The Patriot was intended to be a mobile system that would run for only a few hours at one site, and in that case the rounding error would not be serious. However, in Dhahran it ran for

will never terminate and we obtain a periodic number in base 2.
An important consequence of this not exact representation for certain numbers is that it is always dangerous comparing floating-point numbers coming from different calculations. For example, the following comparison will fail

```
>>> x = 0.1 + 0.2
>>> y = 0.3
>>> x == y
False
```

The common and safe solution to this problem is to compare the difference between floating-point numbers and check if it is smaller than a quantitity we can accept.

100 hours, accumulating a rounding error of 0.34 seconds. The Patriot became confused (presumably because it had two different values for the time), could not track the incoming Scud missile, treating it as a false alarm. The press reported that the Patriot "missed" Scud, but actually the Patriot never left the ground!

CHAPTER 3 | Linear Systems

3.1 Introduction

Linear systems are algebric systems of the type

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2 \\ \qquad\qquad \vdots \quad\quad \vdots \quad\quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + ... + a_{nn}x_n = b_n \end{cases} \tag{3.1}$$

where in general $n > 1$ and x_i (i=1..n) are often called the *unknowns*. The system of linear equations can be recast in matrix form **Ax=b**, or explicitly

$$\underbrace{\begin{pmatrix} a_{11} & a_{12} & ... & a_{1n} \\ a_{21} & a_{22} & ... & a_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ a_{n1} & a_{n2} & ... & a_{nn} \end{pmatrix}}_{\mathbf{A}} \cdot \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}}_{\mathbf{x}} = \underbrace{\begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{pmatrix}}_{\mathbf{b}} \tag{3.2}$$

Before attempting a numerical solution of a linear system, we must be sure that a solution exists. Moreover, the solution must be also unique for a meaningful numerical approach. The following theorem provides the result we need:

Theorem 1. *The system 3.2 has one unique solution if and only if* $det(A) \neq 0$.

The following theorem gives a constructive receipt for finding the solutions of a linear system and it has as direct consequence Theorem 1:

Theorem 2. *(Cramer's theorem) The system 3.2 has solutions*

$$x_i = \frac{det(A|b_i)}{det(A)}$$

where the matrix $(A|b_i)$ *is obtained substituting the i-th column with the vector* **b**.

From Cramer's theorem, the condition $det(A) \neq 0$ is clearly required.

Example:

Considering the linear system

$$\begin{cases} x_1 + x_2 - x_3 = 0 \\ x_1 - x_2 + x_3 = 1 \\ -x_1 + x_2 + x3 = -1 \end{cases}$$

After having verified that $det(A) = -4 \neq 0$, the solutions according to Theorem 2 are

$$x_1 = \frac{det\begin{pmatrix} 0 & 1 & -1 \\ 1 & -1 & 1 \\ -1 & 1 & 1 \end{pmatrix}}{det\begin{pmatrix} 1 & 1 & -1 \\ 1 & -1 & 1 \\ -1 & 1 & 1 \end{pmatrix}} = \frac{-2}{-4} = \frac{1}{2}$$

$$x_2 = \frac{det\begin{pmatrix} 1 & 0 & -1 \\ 1 & 1 & 1 \\ -1 & -1 & 1 \end{pmatrix}}{det\begin{pmatrix} 1 & 1 & -1 \\ 1 & -1 & 1 \\ -1 & 1 & 1 \end{pmatrix}} = \frac{2}{-4} = -\frac{1}{2}$$

$$x_3 = \frac{det\begin{pmatrix} 1 & 1 & 0 \\ 1 & -1 & 1 \\ -1 & 1 & -1 \end{pmatrix}}{det\begin{pmatrix} 1 & 1 & -1 \\ 1 & -1 & 1 \\ -1 & 1 & 1 \end{pmatrix}} = \frac{0}{-4} = 0$$

In general, for matrices with few zero entries, the calculation of the determinants becomes quickly intractable, since the direct computations grow as $O(n!)$, where n is the number of equations (and unknowns). This is the reason why we will discuss computationally more efficient algorithms.

3.2 The Condition Number

Formally, the solution of the system $Ax = b$ is $x = A^{-1}b$ and it is thus possible only if A is invertible. Since we would like to solve the problem numerically, are there case in between "invertible" and "not-invertible" which might affect a numerical algorithm from finding a solution? The answer is given by the calculation of the **condition number**, which we will describe in the following.
First, we define a *matrix norm*, which gives a measure of how large overall the matrix elements are. This definition is not unique, but a widely used one is the quadratic sum of all the matrix elements

$$\| A \| = \sqrt{\sum_{i=1}^{N} \sum_{j=1}^{N} a_{ij}^2} \quad . \tag{3.3}$$

The definition is convenient, since the squaring operation disregards signs and takes into account only the magnitude and it resembles the Euclidean norm (actually for matrices sometimes this is called the Frobenius norm). A similar definition could be based on the sum of the absolute values of the matrix elements, for example.
For assessing how easily a linear system is solvable, we might be tempted to look at the value of the determinant, which is connected to the invertibility of the matrix. This reasoning is wrong, since the solution $x = A^{-1}b$ depends also from the vector b. In general, the correct question to ask is: how a function $y = f(x)$ is sensitive to the input, or in other words, how a change δx in the x variable induces a change δy through f ? We can thus define the *condition number* C as the maximum of the ratio between the two changes, or, even better, the ratio between the relative changes:

$$C = \max \left[\left(\frac{\delta x}{x} \right) / \left(\frac{\delta y}{y} \right) \right] \quad . \tag{3.4}$$

For a linear function, the vector plays the role of y so that the condition number C is the maximum ratio of the relative error in $x = A^{-1}b$ to the relative error in b. Assuming an error in b equal to δb and introducing a suitable norm for assessing the "magnitude" of matrices and vectors

$$C = \frac{\| A^{-1}\delta b \|}{\| \delta b \|} / \frac{\| A^{-1} b \|}{\| b \|} = \frac{\| A^{-1}\delta b \|}{\| \delta b \|} \cdot \frac{\| b \|}{\| A^{-1} b \|} \quad . \tag{3.5}$$

We are interested in the maximum value of C:

$$\max_{b,\delta b\neq 0} C = \max_{\delta b\neq 0}\left[\frac{\parallel A^{-1}\delta b \parallel}{\parallel \delta b \parallel}\right]\cdot \max_{b\neq 0}\left[\frac{\parallel b \parallel}{\parallel A^{-1}b \parallel}\right] = \parallel A^{-1} \parallel \parallel A \parallel \qquad (3.6)$$

The condition number C quantifies how sensitive the "output" b is from the "input" x: of C is of order unity, small errors in the input will not be amplified by the function. The condition number is an intrinsic property of the function (A, in the case of linear systems) and it is not depending from numerical errors or specific numerical algoritms. If C is not much larger than one, the matrix is said to be *well-conditioned*. If C is very large, then the matrix is *ill-conditioned*. An ill-conditoned matrix is almost singular, and the solution of a linear system is very sensitive to numerical errors. A matrix that is not invertible has $C = \infty$.
It is worth noting, that C does not depend directly from the determinant and an overall scaling of the matrix elements does not change it.

3.3 Tha Gauss Elimination Algorithm

This method aims at simplifying the matrix **A** as much as possible reducing some of its elements to zero by means of row or column linear transformations. These transformations do not change the solutions of the system. In the following, we will explain the method by means of a concrete example. Consider the linear system with $det(A) = 290 \neq 0$

$$\begin{array}{c} A \\ B \\ C \\ D \end{array}\left\{\begin{array}{rcrcrcrcl} 4x_1 & - & x_2 & + & 2x_3 & - & x_4 & = & 2 \\ x_1 & + & 4x_2 & - & x_3 & + & x_4 & = & 2 \\ x_1 & - & 2x_2 & - & 3x_3 & + & x_4 & = & 4 \\ & & x_2 & & & - & 4x_4 & = & 0 \end{array}\right.$$

The idea is to successively eliminate the unknowns (the procedure is indeed called **elimination**). Starting with x_1, we can divide Eq. A by 4 and then subtract it to B and C, obtaining the equivalent system

$$\begin{array}{c} B' \\ C' \\ D' \end{array}\left\{\begin{array}{rcrcrcl} \frac{17}{4}x_2 & - & \frac{3}{2}x_3 & + & \frac{5}{4}x_4 & = & \frac{3}{2} \\ -\frac{7}{4}x_2 & - & \frac{7}{2}x_3 & + & \frac{5}{4}x_4 & = & \frac{7}{2} \\ x_2 & & & - & 4x_4 & = & 0 \end{array}\right.$$

Now it is the turn of eliminating x_2 multiplying B' by $7/17$ and adding it to C' and D':

$$\begin{array}{ll} C'' & \left\{ \begin{array}{rcrcl} -\frac{70}{17}x_3 & + & \frac{30}{17}x_4 & = & \frac{70}{17} \\ \frac{6}{17}x_3 & - & \frac{73}{17}x_4 & = & -\frac{6}{17} \end{array} \right. \\ D'' & \end{array}$$

The last step will be to eliminate x_3 multiplying C'' by 6/70 arriving to a single equation for x_4 which can be readily solved:

$$D''' \quad -\frac{29}{7}x_4 = 0 \Rightarrow x_4 = 0$$

Considering the obtained equations, the system was transformed to the equivalent one (written in matrix form)

$$\begin{array}{c} A \\ B' \\ C'' \\ D''' \end{array} \begin{pmatrix} 4 & -1 & 2 & -1 \\ 0 & \frac{17}{4} & -\frac{3}{2} & \frac{5}{4} \\ 0 & 0 & -\frac{70}{17} & \frac{30}{17} \\ 0 & 0 & 0 & -\frac{29}{7} \end{pmatrix} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{pmatrix} \tag{3.7}$$

The matrix **A** is now in "upper triangular" form and the last equation has only one unknown, which once found can be substituted in C", giving x_3. After that, x_3 can be substituted in B' to obtain x_2 and so on, in a procedure called **backward substitution**. In order to build a numerical algorithm implementing elimination and backward substitution, we have to formalize in a general way the operations we carried out in the previous example.
First we introduce the notation a_{ij} for denoting the element in the i-th row and in the j-th column of the matrix, and the index $k = 1..n-1$ for labelling the step of the algorithm.
Looking at the previous example, it is not difficult to derive the relations

$$a_{ij}^{k+1} = a_{ij}^{(k)} - \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} a_{kj}^{(k)} \quad , \quad j = k, k+1, .., n \quad , \tag{3.8}$$

$$b_i^{k+1} = b_i^{(k)} - \frac{a_{ik}^{(k)}}{a_{kk}^{(k)}} b_k^{(k)} \quad . \tag{3.9}$$

The latter equations describe how to obtain the new matrix elements (step k+1) starting from the previous ones (step k) multiplying by the correct factor and then subtracting from the previous matrix row.
After elimination, backward substitution is described by

$$x_n = \frac{b_n^{(n)}}{a_{nn}^{(n)}} \tag{3.10}$$

which gives the trivial solution of the last variable, with which we can find all the other ones with

$$x_i = \frac{b_i^{(i)} - \sum_{j=i+1}^{n} a_{ij}^{(i)} x_j}{a_{ii}^{i}} \quad i = n-1, n-2, ...1 \quad . \tag{3.11}$$

The `Python` code 3.1 implements the elimination and backward substitution procedures described by equations 3.8, 3.9, and 3.10, 3.11.

```python
import numpy as np
import sys

def GaussElimination(a,b):

    n = len(b)
    #Elimination
    for k in range(0,n-1):
        for i in range(k+1,n):
            if a[k,k]==0:
                sys.exit("ERROR: div. by zero.")
            else:
                if (a[i][k] !=0):
                    ratio = a[i][k]/a[k][k]
                    #Note this "vectorized" numpy loop
                    a[i,k+1:n] = \
                        a[i,k+1:n]-ratio*a[k,k+1:n]
                    b[i] = b[i] - ratio*b[k]

    #Backward Substitution
    for k in range(n-1,-1,-1):
        b[k] = (b[k]-np.dot(a[k,k+1:n],b[k+1:n]))/a[k,k]

    return b
```

Listing 3.1: Gauss Elimination

Code 3.1 differs slightly from Eq. 3.8: a check on the pivot element is added and if $a_{ik} = 0$ the elimination is skipped for saving computing time. The index j in the code starts with $k + 1$. This means that the original value of the matrix is never replaced with a zero. This is irrelevant, since the backsubstitution phase never accesses the lower triangular part of the matrix.

3.4 Pivoting

Observing equations 3.8, 3.9 of the elimination procedure, it is evident that the division by $a_{kk}^{(k)}$ (called **pivot** elements) can result in numerical problems if they are either zero or very small in comparison to the other elements.
A common strategy for dealing with this problem is called **pivoting**.

Pivoting consists in choosing at each elimination step k the row index j for which

$$|a_{jk}^{(k)}| = \max_{j>k} |a_{ik}^{(k)}| \tag{3.12}$$

and swap rows k and j.
For example, consider the linear system

$$\begin{array}{c} A \\ B \\ C \\ D \end{array} \left\{ \begin{array}{rcrcrcrcl} & & x_2 & & & - & x_4 & = & 0 \\ x_1 & - & 2x_2 & - & 3x_3 & + & x_4 & = & 4 \\ x_1 & + & 4x_2 & - & x_3 & + & x_4 & = & 2 \\ 4x_1 & - & x_2 & + & 2x_3 & - & x_4 & = & 2 \end{array} \right.$$

The pivot element a_{11} is zero and thus we cannot find a suitable number to multiply equation A with. We notice though that $|a_{41}| = 4$ is the largest element of the first column (or $|a_{41}| = \max |a_{i1}|$) and before applying the elimination step, we can swap equation A with equation D. The pivoting procedure can be applied at every step of the elimination process.
For understanding the numerical problem in more details, we can consider the very simple linear problem

$$\left\{ \begin{array}{l} \epsilon x + 2y = 4 \\ x - y = 1 \end{array} \right. ,$$

where x, y are the unknowns and $\epsilon \approx 0$ is a small parameter. Writing the previous system in matrix form, we can try to find the solution applying the Gauss elimination algorithm. We would like to have the lower-left element equal to zero, subtracting to the second row the first one, divided by ϵ:

$$\begin{pmatrix} \epsilon & 2 & 4 \\ 1 & -1 & 1 \end{pmatrix} \rightarrow \begin{pmatrix} \epsilon & 2 & 4 \\ 0 & -1-2/\epsilon & 1-4/\epsilon \end{pmatrix}$$

Having eliminated one unknown in the last row, we can solve for the y variable obtaining the system

$$\left\{ \begin{array}{l} \epsilon x + 2y = 4 \\ y = \frac{4-\epsilon}{2+\epsilon} \approx 2 \end{array} \right. .$$

Approximating $y = 2$ since ϵ is very small, we can insert this solution in the first equation obtaining the puzzling result $\epsilon x = 0$ which cannot be solved.
Now let's apply the pivoting, i.e. swap the rows since the lower row starts with a larger value than the first and apply Gauss elimination

$$\begin{pmatrix} 1 & -1 & 1 \\ \epsilon & 2 & 4 \end{pmatrix} \rightarrow \begin{pmatrix} 1 & -1 & 1 \\ 0 & 2+\epsilon & 4-\epsilon \end{pmatrix}$$

This time, the system is reduced to

$$\begin{cases} x - y = 1 \\ y = \frac{4-\epsilon}{2+\epsilon} \approx 2 \end{cases} .$$

and substituting $y = 2$ in the first equation we obtain $x = 3$ which is the correct solution within the $\epsilon \approx 0$ approximation. Note that after pivoting, ϵ does not appear in denominators anymore, eliminating the numerical problem present before. Pivoting can be applied (if needed) at every step of the ellimination algorithm. More precisely, swapping only rows (or only columns) is called *partial pivoting*, while in general both rows and columns can be swapped. In most cases, swapping only rows (or only columns) is sufficient and prevents the accumulation of numerical errors in the case of very large matrices.
Approximately, Gauss elimination can be applied to systems up to few hundreds of equations: after that, rounding errors may accumulate yielding inaccurate solutions.

3.5 Tri-diagonal and Diagonally Dominant Matrices

In many concrete applications, matrices assume a particular form called **tridiagonal**. A matrix A is tridiagonal if it has the following form

$$A = \begin{pmatrix} a_{11} & a_{12} & 0 & 0 & 0 & 0 & ... & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 & 0 & ... & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 & 0 & ... & 0 \\ 0 & 0 & ... & ... & ... & ... & ... & 0 \\ 0 & 0 & 0 & ... & ... & ... & ... & 0 \\ ... & ... & ... & ... & ... & a_{n-1,n-2} & a_{n-1,n-1} & a_{n-1,n} \\ 0 & 0 & 0 & 0 & 0 & 0 & a_{n,n-1} & a_{n,n} \end{pmatrix} \tag{3.13}$$

In the matrix A, all the elements not belonging to the three major diagonals are zero.
Diagonally dominant matrices are matrices for which the diagonal elements are larger than all the others:

$$|a_{ii}| \geq \sum_{j=1, j\neq i}^{n} |a_{ij}| \quad . \tag{3.14}$$

with strict inequality for at least one value of i. A relevant theorem related to the given definitions is the following:

Theorem 3. *Let the linear system $Ax = b$ have a matrix A which is tridiagonal, diagonally dominant, and satisfy*

$$\begin{array}{ll} a_{ii} < 0 & i = 1, 2, ..., n \\ a_{j,j+1} > 0 & j = 1, 2, ..., n-1 \\ a_{j+1,j} > 0 & j = 1, 2, ..., n-1 \end{array} \tag{3.15}$$

Then, there exists an unique solution of the system.

A more general theorem with less restrictive assumptions is the following

Theorem 4. *If the linear system $Ax = b$ is diagonally dominant with the strict inequality*

$$|a_{ii}| > \sum_{j=1, j\neq i}^{n} |a_{ij}| \quad \forall i \quad , \tag{3.16}$$

then $det(A) \neq 0$. Under the same assumption, this theorem is clearly valid also for tridiagonal systems.

3.6 LU Decomposition

The Gauss elimination algorithm finds solution to the linear system $Ax = b$. In many applications, the matrix A remains the same, while the vector b changes. In such cases, the Gauss algorithm should be applied for every different vector b. An alternative is given by the LU decomposition algorithm, which gives a clever decomposition of the matrix A as the product of two matrices: one is upper-triangular (U) and the other one is lower-triangular (L) such that A=LU. Once this decomposition is done, the solution of the linear system is fast and can be repeated many times for different vectors b.
A first observation is that the upper-triangular matrix U can be constructed with the Gauss elimination algorithm and therefore a system like Ux=b is solvable by backward substitution. It is easy to see that the system Lx=b is instead solved by **forward substitution**. Let's see how this can be done in practice. Let's consider a linear system $Ax = b$, where A has the tridiagonal form of Eq. 3.13 and assume

we can factorize A such that $A = L \cdot U$ where

$$L = \begin{pmatrix} p_1 & 0 & 0 & 0 & 0 & ... & 0 \\ a_{21} & p_2 & 0 & 0 & 0 & ... & 0 \\ 0 & a_{32} & p_3 & 0 & 0 & ... & 0 \\ 0 & 0 & ... & ... & ... & ... & 0 \\ 0 & 0 & 0 & ... & ... & ... & 0 \\ ... & ... & ... & ... & ... & ... & ... \\ 0 & 0 & 0 & 0 & 0 & a_{n,n-1} & p_n \end{pmatrix} \tag{3.17}$$

is **lower triangular** and

$$U = \begin{pmatrix} 1 & q_1 & 0 & 0 & 0 & ... & 0 \\ 0 & 1 & q_2 & 0 & 0 & ... & 0 \\ 0 & 0 & 1 & q_3 & 0 & ... & 0 \\ 0 & 0 & ... & ... & ... & ... & 0 \\ 0 & 0 & 0 & ... & ... & ... & 0 \\ ... & ... & ... & ... & ... & ... & q_{n-1} \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.18}$$

is **upper triangular**. The matrix elements p_i (i=1,..,n) and q_i (i=1,..,n-1) have to be determined. Calculating the product of matrices 3.17 and 3.18 we obtain

$$L \cdot U = \begin{pmatrix} p_1 & p_1 q_1 & 0 & 0 & 0 & ... & 0 \\ a_{21} & a_{21} q_1 + p_2 & p_2 q_2 & 0 & 0 & ... & 0 \\ 0 & a_{32} & a_{32} q_2 + p_3 & q_3 & 0 & ... & 0 \\ 0 & 0 & ... & ... & ... & ... & 0 \\ 0 & 0 & 0 & ... & ... & ... & 0 \\ ... & ... & ... & ... & ... & ... & .... \\ 0 & 0 & 0 & 0 & 0 & a_{n,n-1} & a_{n,n-1} q_{n-1} + p_n \end{pmatrix} \tag{3.19}$$

Comparing matrix 3.19 with the matrix A, we can derive the following relations

$$\begin{array}{ll} & p_1 = a_{11} \quad , \\ q_1 = \frac{a_{12}}{p_1} & p_2 = a_{22} - a_{21} q_1 \quad , \\ q_2 = \frac{a_{23}}{p_2} & p_3 = a_{33} - a_{32} q_2 \quad , \\ ... & ... \\ q_{n-1} = \frac{a_{n-1,n}}{p_{n-1}} & p_n = a_{nn} - a_{n,n-1} q_{n-1} \quad , \end{array} \tag{3.20}$$

which can be condensed in

$$\begin{array}{l} p_1 = a_{11} \\ q_{j-1} = a_{j-1,j}/p_{j-1} \quad j = 2,3,...,n \\ p_j = a_{jj} - a_{j,j-1}q_{j-1} \end{array} \tag{3.21}$$

Now that we know how to LU-decompose the initial matrix A, we have to solve the corresponding linear system $Ax = b \Rightarrow LUx = b$. The strategy is the following: first we set

$$Ux = z \quad , \tag{3.22}$$

and then

$$Lz = b \quad . \tag{3.23}$$

Starting from the last equation, the system looks like

$$\begin{array}{lcl} p_1 z_1 & = & b_1 \\ a_{21}z_1 + p_2 z_2 & = & b_2 \\ a_{32}z_2 + p_3 z_3 & = & b_3 \\ ... & ... & ... \\ a_{n,n-1}z_{n-1} + p_n z_n & = & b_n \end{array} \tag{3.24}$$

By *forward substitution* (first solve for z_1, then for z_2, ...) we have

$$\begin{array}{lcl} z_1 & = & b_1/p_1 \\ z_2 & = & (b_2 - a_{21}z_1)/p_2 \\ z_3 & = & (b_3 - a_{32}z_2)/p_3 \\ ... & ... & ... \\ z_n = (b_n - a_{n,n-1}z_{n-1})/p_n & & \end{array} \tag{3.25}$$

or in a form easy to translate into a computer code (compare it with the Gauss elimination algorithm)

$$\begin{array}{l} z_1 = \frac{b_1}{p_1} \\ z_j = \frac{b_j - a_{j,j-1}z_{j-1}}{p_j} \quad j = 2,3,...,n \end{array} \tag{3.26}$$

Now that we have the vector z, we can solve $Ux = z$ and obtain the vector x:

$$\begin{array}{lll} x_1 + q_1 x_2 & = & z_1 \\ x_2 + q_2 x_3 & = & z_2 \\ ... & ... & ... \\ x_{n-1} + q_{n-1} x_n & = & z_{n-1} \\ x_n & = & z_n \end{array} \tag{3.27}$$

The last system can be easily solved by *backward substitution*:

$$\begin{array}{l} x_n = z_n \\ x_j = z_j - x_{j+1} q_j \quad j = n-1, n-2, ..., 1 \end{array} \tag{3.28}$$

An important observation is that the solution of the tridiagonal system has computational complexity $O(n)$, while the Gauss elimination has $O(n^3)$.

3.7 LU Decomposition: General Case

It is possible to decompose a generic matrix (not necessarily tri-diagonal) in the product of a lower-diagonal matrix L and an upper-diagonal matrix U. The decomposition is not unique and a choice has to be made usually about the diagonal elements. Since L and U have non-zero diagonals, a choice could be to fix the diagonal elements of L to 1. We illustrate the method with a 3x3 matrix A = LU with

$$L = \begin{pmatrix} 1 & 0 & 0 \\ l_{21} & 1 & 0 \\ l_{31} & l_{32} & 1 \end{pmatrix} \quad U = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} \quad . \tag{3.29}$$

Performing the multiplication

$$A = LU = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ u_{11} l_{21} & u_{12} l_{21} + u_{22} & u_{13} l_{21} + u_{23} \\ u_{11} l_{31} & u_{12} l_{31} + u_{22} l_{32} & u_{13} l_{31} + u_{23} l_{32} + u_{33} \end{pmatrix} \quad . \tag{3.30}$$

Let is now try to simplify this matrix with the Gauss elimination algorithm. The first step is

$$\text{Row } 2 \rightarrow \text{Row } 2 - l_{21} \cdot \text{Row } 1$$

$$\text{Row } 3 \rightarrow \text{Row } 3 - l_{31} \cdot \text{Row } 1$$

This operation eliminates the matrix entries a_{21} and a_{31}, respectively, resulting in the equivalent matrix

$$A' = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & u_{22}l_{32} & u_{23}l_{32} + u_{33} \end{pmatrix} \quad . \tag{3.31}$$

Appliying the last step

$$\text{Row } 3 \rightarrow \text{Row } 3 - l_{32} \cdot \text{Row } 2$$

we eliminate a_{32} obtaining

$$A'' = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{pmatrix} \quad . \tag{3.32}$$

The result of the elimination process proved the following results:

1. The matrix U corresponds to the upper triangular matrix obtained with the Gauss elimination method.

2. The off-diagonal elements of the matrix L are the *pivot* elements used in the elimination algorithm. In other words, the elements l_{ij} are used to eliminate the elements a_{ij}.

The latter results show that the LU decomposition algorithm (or the **Doolittle algorithm** in this particular choice of the diagonal elements of L) is identical to the Gauss elimination algorithm: the matrix U is produced by the elimination process, while the matrix L is composed by the *pivot* elements.
An interesting implementation of the algorithm stores the results in one single matrix for saving memory space. The idea is to store the elements of L in the zero elements of U, remembering that the diagonal of L has all the entries equal to 1:

$$(LU) = \begin{pmatrix} u_{11} & u_{12} & u_{13} \\ l_{21} & u_{22} & u_{23} \\ l_{31} & l_{32} & u_{33} \end{pmatrix} \quad . \tag{3.33}$$

The algorithm is implemented in the `python` code 3.2.

```python
import numpy as np

#LU decomposition (Doolittle version)
def LUdecomposition(A):
    n = len(A)

    #Perform Gauss elimination for calculating U
    for k in range(0,n-1):
        for i in range(k+1,n):
            if A[i,k] != 0.0:
                Lambda = A[i,k]/A[k,k]
                A[i,k+1:n] = A[i,k+1:n] - \
                    Lambda * A[k,k+1:n]

                #Record the L matrix elements
                A[i,k] = Lambda
    return A

def LUsolver(A,b):
    n = len(A)
    #Forward substitution
    for k in range(1,n):
        b[k] = b[k] - np.dot(A[k,0:k],b[0:k])
    #Backward substitution
    for k in range(n-1,-1,-1):
        b[k] = (b[k] - np.dot(A[k,k+1:n],b[k+1:n])) \
            /A[k,k]
    return b

#Example
A = np.array([[1,2,23],[3,14,5],[6,7,8]])
b = np.array([15,6,37])

A = LUdecomposition(A)
X = LUsolver(A,b)

#Solutions of the linear system
print(X)
```

Listing 3.2: LU Decomposition

3.8 Choleski Decomposition

Another possible matrix decomposition, which is very useful in many applications is the Choleski decomposition, which can be applied to Hermitian positive-definite matrices. A complex matrix is **Hermitian** if it is equal to its complex-conjugated transpose (denoted with A^*). An Hermitian real matrix A is symmetric, or $A = A^T$. A matrix A is positive-definite if $v^T A v > 0 \ \ \forall v \neq 0$, which is equivalent to the following properties:

- A is equivalent to a diagonal matrix with positive real elements.
- A is Hermitian and all its eigenvalues are real and positive.
- A is Hermitian, and all its leading principal minors are positive.
- There is a matrix U with $det(U) \neq 0$ with conjugate transpose U^* such that $A = U^* U$

The last equivalent property leads to the Choleski decomposition of a hermitian matrix A:

$$A = LL^* \quad , \tag{3.34}$$

where L is a lower triangular matrix (U was instead an upper triangular matrix). The formulas for the decomposition can be inferred from a simple n=3 example with real matrix elements

$$A = LL^T = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \cdot \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{pmatrix} \tag{3.35}$$

Performing the multiplication LL^T and comparing with the corresponding elements of A we obtain

$$L = \begin{pmatrix} \sqrt{a_{11}} & a_{21}/l_{11} & a_{31}/l_{11} \\ a_{21}/l_{11} & \sqrt{a_{22} - l_{21}^2} & (a_{32} - l_{31}l_{21})/l_{22} \\ a_{31}/l_{11} & (a_{32} - l_{31}l_{21})/l_{22} & \sqrt{a_{33} - l_{31}^2 - l_{32}^2} \end{pmatrix} \tag{3.36}$$

The last result allows us to infer the general formulas for the matrix elements of L

$$
\begin{aligned}
l_{jj} &= \pm\sqrt{a_{jj} - \sum_{k=1}^{j-1} l_{jk}^2} \\
l_{ij} &= \frac{1}{l_{jj}}\left(\sum_{k=1}^{j-1} l_{ik} l_{jk}\right) \qquad i > j
\end{aligned}
\tag{3.37}
$$

When the Choleski decomposition can be used, it is about twice as fast with respect to the LU decomposition, although asymptotically the methods are equivalent. The Choleski decomposition method is implementes in code 3.3.

```
import numpy as np

def Choleski(A):
    n = len(A)
    for k in range(n):
        A[k,k] = np.sqrt(A[k,k] - \
                 np.dot(A[k,0:k],A[k,0:k]))
        for i in range(k+1,n):
            A[i,k] = \
            (A[i,k] - np.dot(A[i,0:k],A[k,0:k]))/A[k,k]
    for k in range(1,n): A[0:k,k] = 0.0
    return A

#Example
A = np.array([[4,-2,2],[-2,2,-4],[2,-4,11]])
L = Choleski(A)

print("LLt = ",np.dot(L,np.transpose(L)))
```

Listing 3.3: Choleski Decomposition

3.9 Steepest Descent and Conjugate Gradient Methods

The algorithms examined up to now were "direct" methods. Another class of methods involves "iterative" methods which try to approximate the exact solution with iterative steps starting from an initial guess solution. These algorithms can outperform direct methods when the matrix is sparse and very large, since for sparse

matrices we can record only the non-zero values of the elements and this can not only same memory but also speed up matrix-vector multiplications. Iterative procedures are also safe against rounding errors, since they correct by themselves through the iterations.
We will describe here one of the most popular iterative methods: the conjugated gradient method and the less efficient steepest descent method. We consider first the minimization of the following linear function

$$L(x) = \frac{1}{2}x^T Ax - bx \quad , \tag{3.38}$$

where b,x are vectors and A is a *symmetric and positive definite* matrix. The minimization with respect to x involves the gradient of the function L and setting it to zero

$$\nabla L(x) = Ax - b = 0 \quad . \tag{3.39}$$

The last result shows that the minimization of L solves the linear system $Ax = b$. We would like to find the minimum of L with an iterative procedure, starting from guess solution x_0 and then iterate with successive corrections

$$x_{k+1} = x_k + \alpha_k s_k \quad . \tag{3.40}$$

The number α_k is the *step length* and is chosen for minimizing L in the *direction* given by the vector s_k.
Substituting the last equation in the linear system we have

$$A(x_k + \alpha_k s_k) = b \quad , \tag{3.41}$$

and introducing the errors, or *residuals* $e_k = b - Ax_k$ we can rewrite

$$\alpha As_k = e_k \quad . \tag{3.42}$$

Multiplying Eq. 3.42 by s_k^T to the left and solving for the step length

$$\alpha_k = \frac{s_k^T e_k}{s_k^T As_k} \quad . \tag{3.43}$$

Now we have a procedure for updating the step length: we still have to choose the direction s_k.
A good choice seems to be

$$s_k = e_k = -\nabla L(x) \quad . \tag{3.44}$$

The reason is that with the latter choice, we move in the direction of where the the function is steeper: indeed such algorithm is called **steepest descent method**. This is not the best choice we can do and actually its convergence can be slow.
A more efficient algorithm is based on the following update rule

$$s_{k+1} = e_{k+1} + \beta_k s_k \quad , \tag{3.45}$$

where the number β_k is chosen is such a way that two successive directions are *conjugated*:

$$s_{k+1}^T A s_k = 0 \quad . \tag{3.46}$$

The last choice consists in the **conjugated gradient method**: the two successive directions are orthogonal in the metric induced by the matrix A. Concretely, this means that the new direction does not spoil the previous minimization step and the advantage gained is always conserved. In other words, two successive minimization steps do not interfere with oneanother. To see this, we substitute Eq. 3.45 in Eq. 3.46 obtaining

$$(e_{k+1}^T + \beta_k s_k^T) A s_k = 0 \quad , \tag{3.47}$$

which we can solve for β

$$\beta_k = -\frac{e_{k+1}^T A s_k}{s_k^T A s_k} \quad . \tag{3.48}$$

With the last update rule, the algorithm is complete. As starting parameters, we can choose x_0 (the closer to the real solution, the less steps will be needed), and $e_0 = b - Ax_0$ with the starting direction $s_0 = e_0$.
We remark again that the method works for a symmetric and positive definite matrix: this ensures that if a vector v_1 is conjugated to a vector v_2 through A, *i.e* $v_1 A v_2 = 0$, also v_2 is conjugated to v_1 and $A v_1 v_2 = 0$.
Positive definiteness ensures that α and β do not change sign during the iterations which would otherwise result in a non-convergence of the algorithm.
Code 3.4 implements the conjugated gradient method, which we also summarize in the following:

- Choose an initial vector x_0, if possible close to the solution.
- Calculate the initial error e_0 and proceed in the steppest descent direction $s_0 = e_0$
- For $k = 0, .., N_{max}$, where N_{max} a maximum number of iterations, or stop when $|e_k| < \epsilon$ with ϵ a suitable acceptable error, **DO:**

1. $x_{k+1} = x_k + \alpha_k s_k$ and $e_k = b - Ax_{k+1}$ with:

$$\alpha_k = \frac{s_k^T e_k}{s_k^T A s_k} \quad .$$

2. $\beta_k = -\frac{e_{k+1}^T A s_k}{s_k^T A s_k}$
3. $s_{k+1} = e_{k+1} + \beta_k s_k$

- **END DO.**

```python
import numpy as np
from math import sqrt

def ConjugatedGradient(A,b,x,maxiter,err):

    e = b-np.dot(A,x)
    s = e

    for i in range(0,maxiter):

        u = np.dot(A,s)
        alpha = np.dot(s,e) / np.dot(s,u)

        x = x + alpha*s
        e = b - np.dot(A,x)

        if ( sqrt(np.dot(e,e)) ) < err:
            break
        else:
            beta = - np.dot(e,u) / np.dot(s,u)
            s = e + beta*s

    return x,i

#Linear system to solve
A=[[3,2,1] , [2,3,2] , [1,2,3]]
b =[1,2,3]

#Starting guess solution
x = np.array([0,0,0])

maxiter = 30      #maximum number of iterations
tolerance = 1e-6 #precision wanted

x,iterations = ConjugatedGradient(A,b,x,maxiter,
    tolerance)

print("Iterations = ",iterations)
print("Solution x = ",x)
```

Listing 3.4: Conjugated Gradient Algorithm

The following code employs the previous one for plotting the iterative steps of the algorithm for arriving to the solution of the linear problem.

```
from itertools import product
import numpy as np
import matplotlib.pyplot as plt
from math import sqrt

def L(x):
    Ax = np.dot(A, x)
    xAx = np.dot(x, Ax)
    bx = np.dot(b, x)
    return 0.5 * xAx - bx

#creates a x,y,z=L(x,y) mesh using the function L
def create_mesh(f):
    x = np.arange(-5, 5, 0.025)
    y = np.arange(-5, 5, 0.025)
    X, Y = np.meshgrid(x, y)
    Z = np.zeros(X.shape)
    mesh_size = range(len(X))
    for i, j in product(mesh_size, mesh_size):
        x_coord = X[i][j]
        y_coord = Y[i][j]
        Z[i][j] = L(np.array([x_coord, y_coord]))
    return X, Y, Z

#creates a contour plot from the x,y,z mesh
def plot_contour(ax, X, Y, Z):
    ax.set(title='Iteration Path',xlabel='$x_1$',ylabel=
    '$x_2$')
    CS = ax.contour(X, Y, Z)
    ax.clabel(CS, fontsize='small', fmt='%1.2f')
    ax.axis('square')
    return ax
```

Listing 3.5: Plot of iteration steps part 1/2

```
def ConjugatedGradient(A,b,x,maxiter,err):
    e = b-np.dot(A,x)
    s = e
    path = [x] #for recording the steps
    for i in range(0,maxiter):
        u = np.dot(A,s)
        alpha = np.dot(s,e) / np.dot(s,u)
        x = x + alpha*s
        e = b - np.dot(A,x)
        path.append(x)
        if ( sqrt(np.dot(e,e)) ) < err:
            break
        else:
            beta = - np.dot(e,u) / np.dot(s,u)
            s = e + beta*s
    return x,i,np.array(path)

#Linear problem to solve
A = np.array([[2.23,-0.56],[-0.56,2.23]])
b = np.array([1,2.7])

x0=np.array([-2,2])
x,iterations,path = ConjugatedGradient(A,b,x0,10,1e-6)

fig, ax = plt.subplots(figsize=(6, 6))
X, Y, Z = create_mesh(L)
ax = plot_contour(ax, X, Y, Z)
ax.plot(path[:,0], path[:,1], linestyle='--', marker='o'
    , color='blue')
ax.plot(path[-1,0], path[-1,1], 'ro')
plt.show()
```

Listing 3.6: Plot of iteration steps part 2/2

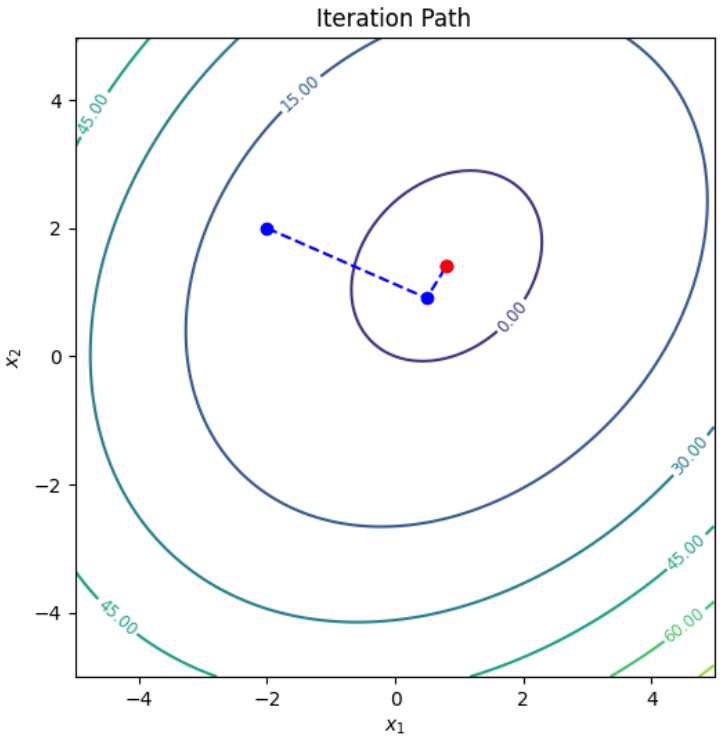

Figure 3.1: Output of the previous code. The steps (bllue line) of the algorithm are plotted over a contour map of the quadratic function $L(x) = \frac{1}{2}x^T Ax - b^T x$.

3.10 Eigenvalues and Eigenvectors: the Power Method

In many problems related to a matrix A, it is required to find a set of constants λ_i and vectors v_i such that

$$Av_i = \lambda_i v_i \quad , \tag{3.49}$$

where λ_i are the *eigenvalues* corresponding to the *eigenvectors* v_i. The last equation can be rewritten as

$$(A - \lambda_i I)v_i = 0 \quad , \tag{3.50}$$

(I is the identity matrix) which has a non-trivial solution if $det(A - \lambda_i I) = 0$, requiring that v_i are non-zero vectors.
In general, finding the eigenvalues and the eigenvectors of large matrices is a complex problem. In some applications, it is sufficient to evaluate the *spectral radius*, which is the largest of the eigenvalues. This can be achieved with a simpler algorithm called the **power method**. We assume that $\lambda_1 > \lambda_2 > ... > \lambda_n$, λ_1 is real,

$$|\lambda_1| > |\lambda_i| \quad , \quad , i = 2, 3, ..n \quad , \tag{3.51}$$

and A has n linearly independent eigenvectors v_i normalized such that the maximum component is equal to 1. If a vector x^0 is an initial guess and the next vectors at step $k + 1$ are obtained with

$$x^k = Ax^{k-1} \quad , \quad k = 1, 2, .., n \quad , \tag{3.52}$$

If x^0 is represented by a linear combination of the eigenvectors (we write it on the eigenvector base)

$$x^0 = c_1 v_1 + c_2 v_2 + ...c_n v_n \quad , \quad c_1 \neq 0 \quad , \tag{3.53}$$

Acting with A on the last vectors, we obtain the sequence:

$$\begin{aligned}
x^1 = \quad & Ax^0 = c_1\lambda_1 v_1 + c_2\lambda_2 v_2 + ... + c_n\lambda_n v_n \\
x^2 = \quad & Ax^1 = c_1\lambda_1^2 v_1 + c_2\lambda_2^2 v_2 + ... + c_n\lambda_n^2 v_n \\
... \quad & ... \\
... \quad & ... \\
x^k = \quad & Ax^{k-1} = c_1\lambda_1^k v_1 + c_2\lambda_2^k v_2 + ... + c_n\lambda_n^k v_n
\end{aligned}$$

Note that the upper indices k indicate the iteration step, while the upper index of the eigenvalues λ is an exponent.
Rewriting the last iteration as

$$x^k = \lambda_1^k \left[c_1 v_1 + c_2 \left(\frac{\lambda_2}{\lambda_1} \right)^k v_2 + ... + c_n \left(\frac{\lambda_n}{\lambda_1} \right)^k \right] \quad , \tag{3.54}$$

and as k becomes large (in the limit $k \to \infty$) we have

$$\begin{aligned} x^k &\longrightarrow \lambda_1^k c_1 v_1 \\ x^{k+1} &\longrightarrow \lambda_1^{k+1} c_1 v_1 \quad , \end{aligned}$$

and taking the ratio of the last two iterations gives an estimate of the largest eigenvalue

$$\frac{x^{k+1}}{x^k} \longrightarrow \lambda_1 \quad . \tag{3.55}$$

The `Python` code 3.7 implements the power method in a single loop, where the vector is continuously renormalized with the previous one's norm.
More in details, the algorithm works as follows

1. Start with a vector x normalized to 1

2. Calculate `x_k1 = Ax`

3. Calculate `|x_k1|`

4. Set `x=x_k1/|x_k1|` and go to step 2 until convergence.

The result will be `|x_k1|`=λ_1 and $x = v_1$. The sign of the eigenvalue is determined looking if $xk1$ changes sign between iterations. If this happes, the sign is negative, otherwise, positive.

```python
import numpy as np
from math import sqrt

#A symmetrix matrix
A = np.array([[1, 2, 3],\
              [2,-3, 4],\
              [3, 4, 3]])

#Initial guess vector with unit norm
x = np.array([1,0,0])

#Iterations
N=100

for i in range(N):
    x_old = x.copy()
    x_k1 = np.dot(A,x)
    x_k1_norm = np.sqrt(np.dot(x_k1,x_k1))

    x = x_k1/x_k1_norm #Normalize

    if (np.dot(x_old,x)<0.0):
        sign = -1.0
        x=-x
    else: sign = 1.0
    if np.dot(x_old-x,x_old-x)<1.0e-6 : break

lambda1 = sign * x_k1_norm
print("Iterations = ",i)
print("Largest Eigenvalue = ",lambda1)
```

Listing 3.7: Power Method

3.11 Eigenvalues and Eigenvectors: the Jacobi Method

The Jacobi method (1846) returns the eigenvectors and the eigenvalues of **symmetric** matrices, which are particularly relevant in many applications. We remind

here that the eigenvalues of a symmetric matrix are always real and if the matrix is positive-definite, they are also positive. Another relevant property of symmetric matrices is that their eigenvectors are orthonormal.
The basic idea of the Jacobi algorithm, is to apply successive rotations in order to diagonalize the matrix and then obtain the eigenvalues as its diagonal elements. First we will discuss similarity transformations and rotations in particular, and then the application to the eigenvalue problem.

Similarity Transformations

Considering the eigenvalue problem $Ax = \lambda x$, we can apply the transformation matrix $x = Px'$ (P is a non-singular matrix). Substituting the transformation into the eigenvalue problem and multiplying by the inverse of P

$$P^{-1}APx' = \lambda P^{-1}Px' \Rightarrow A'x' = \lambda x' \quad , \tag{3.56}$$

where we defined the transformed matrix $A' = P^{-1}AP$. Since λ are scalars, they are not changed by the transformation and we conclude that A and A' have the same eigenvalues: A and A' are thus called *similar* and P encodes a *similarity transformation*.
If we could find P such that A' is diagonal, the eigenvalue problem would be solved, since the diagonal terms of A' are exactly the sought eigenvalues. Moreover, if A' is diagonal, $(A' - \lambda I)x = 0$ implies that the eigenvectors are of the form $x_i = (0, 0, ...1, ...0, 0)$ and the X matrix built with the eigenvectors as columns is the identity matrix and therefore:

$$X = PX' = PI = P \quad , \tag{3.57}$$

and we conclude that the similarity transformation P contains the eigenvectors of A.

Rotations

We introduce now special similarity transformations $x = Rx'$ which have the geometrical interpretation of rotations around an axis. The rotation matrices R

have the form (8-dimensional example)

$$R = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \cos\theta & 0 & 0 & \sin\theta & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & -\sin\theta & 0 & 0 & \cos\theta & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \quad , \tag{3.58}$$

From now on, we simplify the notation defining $c = \cos\theta$ and $s = \sin\theta$. The c and s elements in a generic rotation matrix are found in the positions (k,k), (l,k), (l,k), (l,l), while the other diagonal elements are equal to 1 and the other off-diagonal elements are zero. The relevant property of rotation matrices is $R^{-1} = R^T$: the transpose is equal to the inverse. Applying an orthogonal matrix as similarity transformation to a matrix A

$$A' = R^{-1}AR = R^T AR \quad , \tag{3.59}$$

therefore A' will have the same eigenvalues of A and moreover (thanks to the orthogonality of R) A' will also be symmetric, like A. The transformation in Eq. 3.59 can be written algebrically as

$$\begin{aligned} A'_{kk} &= c^2 A_{kk} + s^2 A_{ll} - 2csA_{kl} \\ A'_{ll} &= c^2 A_{ll} + s^2 A_{kk} + 2csA_{kl} \\ A'_{kl} = A'_{lk} &= (c^2 - s^2)A_{kl} + cs(A_{kk} - A_{ll}) \\ A'_{ki} = A'_{ik} &= cA_{ki} - sA_{li} \quad \text{for} \quad i \neq k \quad i \neq l \\ A'_{li} = A'_{il} &= cA_{li} + sA_{ki} \quad \text{for} \quad i \neq k \quad i \neq l \end{aligned} \tag{3.60}$$

Jacobi Diagonalization

The basic idea is to apply successive rotations to the matrix A for reducing to zero all the off-diagonal elements. An obivious problem is that a rotation could spoil the effect of the previous one. The nice feature of the rotations is though that zeroed off-diagonal elements might re-appear, but with a smaller magnitude. This means that we can write our similarity transformation as

$$P = R_1 \cdot R_2 \cdot R_3 \cdot ... \tag{3.61}$$

where R_i are orthogonal matrices (rotations). Let's consider the off-diagonal elements A'_{kl} in the list 3.60 and force them to be equal to zero, obtaining

$$(c^2 - s^2)A_{kl} + cs(A_{kk} - A_{ll}) = 0 \Rightarrow \tan 2\theta = -\frac{2A_{kl}}{A_{kk} - A_{ll}} \quad , \tag{3.62}$$

where we used the trigonometric identities $c^2 - s^2 = \cos 2\theta$ and $cs = (\sin 2\theta)/2$. Eq. 3.62 can be solved for θ and the orthogonal transformation is fully determined. Numerically, there is a better algorithm for the calculation of c and s. Considering instead of the tangent its reciprocal or cotangent

$$\phi = \cot 2\theta = -\frac{2A_{kl}}{A_{kk} - A_{ll}} \quad , \tag{3.63}$$

and the trigonometrical identity

$$\tan 2\theta = \frac{2t}{1 - t^2} \quad \text{with} \quad t = \tan\theta \quad , \tag{3.64}$$

we have

$$\tan 2\theta = \frac{1}{\phi} = \frac{2t}{1 - t^2} \quad \Rightarrow \quad t^2 + 2\phi t - 1 = 0 \quad . \tag{3.65}$$

Solving the last quadratic equation, the roots are $t = -\phi \pm \sqrt{\phi^2 + 1}$. The root $|t| < 1$ corresponding to $|\theta| \leq 45^\circ$ is the most numerically stable transformation, therefore we choose the plus-sign root if $\phi > 0$ and the minus-sign if $\phi \leq 0$:

$$t = \mathrm{sgn}(\phi)\left(-|\phi| + \sqrt{\phi^2 + 1}\right) \quad . \tag{3.66}$$

Multiplying the last equation by $|\phi| + \sqrt{\phi^2 + 1}$ and solving for t

$$t = \frac{\mathrm{sgn}(\phi)}{|\phi| + \sqrt{\phi^2 + 1}} \approx \frac{1}{2\phi} \quad . \tag{3.67}$$

The second-last manipulation can be used for minimizing round-off errors in the case where ϕ is large and the approximation $(1/2\phi)$ is useful for preventing overflows in the calculation of ϕ^2.
Now that we have a stable calculation of t, using known trigonometric relations we have

$$c = \frac{1}{\sqrt{1 + t^2}} \tag{3.68}$$

$$s = tc \tag{3.69}$$

Going back to the condition for zeroing the off-diagonal elements (Eq. 3.62), we can derive

$$A_{ll} = A_{kk} + A_{kl}\frac{c^2 - s^2}{cs} \quad , \tag{3.70}$$

and substituting the last expression in all the instances of A_{ll} in the equations 3.60 we obtain the new relations

$$\begin{aligned} A'_{kk} &= A_{kk} - tA_{kl} \\ A'_{ll} &= A_{ll} + tA_{kl} \\ A'_{kl} &= A'_{lk} = 0 \\ A'_{ki} = A'_{ik} &= A_{ki} - s(A_{li} + \omega A_{ki}) \quad \text{for} \quad i \neq k \quad i \neq l \\ A'_{li} = A'_{il} &= A_{li} + s(A_{ki} - \omega A_{li}) \quad \text{for} \quad i \neq k \quad i \neq l \end{aligned} \tag{3.71}$$

where $\omega = s/(1+c)$ is introduced for obtaining all the equation in a form where the new value is equal to the old value plus/minus a factor.

The Algorithm

At the beginning of the algorithm, the similarity transformation P is initialized with the identity matrix I. The rotation is

$$P' = PR \tag{3.72}$$

but we have to decide which off-diagonal elements to eliminate first (equivalently, around which axis make the rotation). Jacobi's original procedure required to zero the largest element, but this implies scanning the matrix before every rotation and this can slow down the algorithm for very large dimensions.
An alternative procedure is to apply the rotation to every element if its magnitude is larger than a certain threshold. After a first rotation of all the elements above threshold, the threshold can be lowered and the procedure repeated. The overall algorithm can be summarized in the following steps

1. Initialize P=I and find the largest element A_{kl} in the upper halp of A (it is a symmetric matrix).

2. Compute ϕ and then derive c, s, and ω.

3. Apply the rotation 3.71.

4. Update the transformation matrix P with Eq. 3.72, where only the off-diagonal elements are affected:

$$\begin{aligned} P'_{ik} &= P_{ik} - s(P_{il} + \omega P_{ik}) \\ P'_{il} &= P_{il} + s(P_{ik} - \omega P_{il}) \end{aligned} \tag{3.73}$$

5. Repeat until $|A_{kt}| < \epsilon$ with ϵ the chosen error tolerance.

At the end of the iterations, the matrix A will be diagonal and the diagonal elements will be the estimated eigenvalues, while the columns of matrix P will eventually contain the eigenvectors. The code implementing Jacobi's algorithm is showed (divided in two halves) in Codes 3.8 and 3.9. The function `Jacobi` accepts a matrix A as input as well as a tolerance (how close to zero a matrix element should be). `Jacobi` implements the version of the algorithm where the rotations done by the sub-function `Rotation` are always applied to the largest matrix element found by the sub-function `maxElement`.

```python
from math import sqrt
import numpy as np

def Jacobi(A,tol=1.0e-9):
    #Largest off-diagonal element (upper half)
    def maxElement(A):
        n = len(A) ; themax = 0.0
        for i in range(0,n-1):
            for j in range(i+1,n):
                if abs(A[i,j]) >= themax:
                    themax = abs(A[i,j])
                    k = i
                    l = j
        return themax,k,l

    #Apply a rotation to male A[k,l]=0
    def Rotation(A,P,k,l):
        n = len(A)
        diff = A[l,l] - A[k,k]
        if abs(A[k,l])<abs(diff)*1.0e-36:
            t = A[k,l]/diff
        else:
            phi = diff/(2.0*A[k,l])
            t = 1.0 / (abs(phi) + np.sqrt(phi**2 + 1.0))
            if phi < 0.0: t = - t
        c = 1.0 / np.sqrt(t**2 + 1.0)
        s = t*c
        omega = s/(1.0 + c)
        tmp = A[k,l]
        A[k,l] = 0.0
        A[k,k] = A[k,k] - t*tmp
        A[l,l] = A[l,l] + t*tmp
```

Listing 3.8: Jacobi's Algorithm (1/2)

```
        for i in range(k): #if i<k
            tmp = A[i,k]
            A[i,k] = tmp - s*(A[i,l] + omega*tmp)
            A[i,l] = A[i,l] + s*(tmp - omega*A[i,l])
        for i in range(k+1,l): #if k<i<l
            tmp = A[k,i]
            A[k,i] = tmp - s*(A[i,l] + omega*A[k,i])
            A[i,l] = A[i,l] + s*(tmp - omega*A[i,l])
        for i in range(l+1,n): #if i>l
            tmp = A[k,i]
            A[k,i] = tmp - s*(A[l,i] + omega*tmp)
            A[l,i] = A[l,i] + s*(tmp - omega*A[l,i])
        for i in range(k): #Update P
            tmp = P[i,k]
            P[i,k] = tmp - s*(P[i,l] + omega*P[i,k])
            P[i,l] = P[i,l] + s*(tmp - omega*P[i,l])

    n = len(A)
    maxRotations = 5 * (n**2) #Rule of thumb
    P = np.identity(n)*1.0
    for i in range(maxRotations):
        themax, k, l = maxElement(A)
        if themax < tol: return np.diagonal(A),P
        Rotation(A,P,k,l)
    print("Jacobi Method not converged...")
```

Listing 3.9: Jacobi's Algorithm (2/2)

CHAPTER 4 | Approximation and Interpolation

A typical problem arising in concrete applications is the approximation of discrete data. If a discrete dataset is available, a way to study it is to find a (continuous) function which reproduces the data but also "fills the gaps" between them. Once such function is obtained, one can study the function instead of the data, obtaining effectively a compression of the initial information. Another problem is checking if a specific model given in the form of a function describes a discrete dataset or not. It is useful to introduce the definition of **discrete function**, which is a function defined only on a finite number of points:

$$y_i = f(x_i) \quad , \quad x_i \in \mathbb{R}_{i+1} \quad , \quad i = 0, 1, 2, ...n \quad . \tag{4.1}$$

In this chapter, numerical algorithms for approximating and interpolating discrete data will be described.

4.1 Linear Interpolation

This is the simplest interpolation algorithm, where the idea is to approximate the dataset with pieacewise linear functions, or connect the points of the dataset with lines.
If the dataset is defined on a grid of $n+1$ points $a = x_0 < x_1 < ... < x_n = b$ and the spacing between the points is $h = (b-a)/n$, then a linear interpolation function L is given by

$$L(x) = y_i + \frac{y_{i+1} - y_i}{h}(x - x_i) \quad , x_i \leq x \leq x_{i+1} \quad . \tag{4.2}$$

which is implemented in code 4.1

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

#Linear interpolation function
def LinearInterpolation(x,X,Y):
    L=0
    n = np.size(X)
    h = (X[n-1] - X[0])/(n-1)
    for i in range(0,n-1):
        if (X[i]<=x and x<=X[i+1]):
            L = Y[i] + (Y[i+1]-Y[i])/h * (x-X[i])
    return L

#Example data
x = np.array([1,2,3,4,5,6,7])
y = np.array([1,2,5,7,4,4,2])

#Interpolate (return y given a single x)
N=100
xf = np.linspace(1,7,N)
yf = np.zeros(N)
k=0
for i in xf:
    yf[k] = LinearInterpolation(i,x,y)
    k = k+1

#Plot
plt.plot(x,y,'ro')
plt.plot(xf,yf)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Listing 4.1: Linear Interpolation

In the code, the linear interpolation function is written for producing a single point as output, given a single input value. This implies passing to the function every time the dataset. The output is showed in Fig. 4.1.

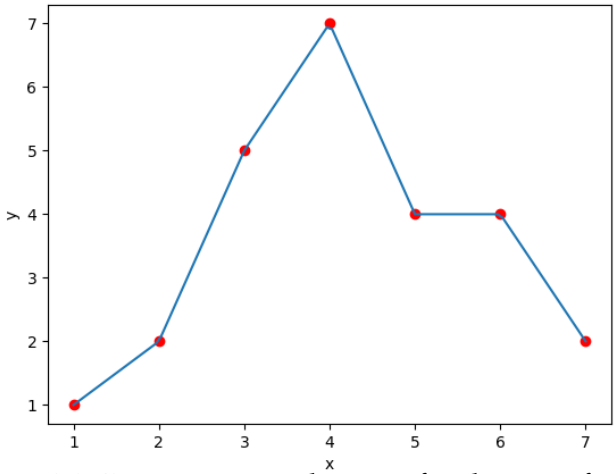

Figure 4.1: Linear interpolation of a discrete function.

4.2 Parabolic Interpolation

If linear interpolation were not sufficient as an approximation, a parabolic interpolation between points can be used instead.
Considering three non-collinear points $(x_0, y_0), (x_1, y_1), (x_2, y_2)$, we can choose $x_1 = 0$ without loss of generality (a translation is always possible). The choice for x_1 implies $x_0 = -h$ and $x_2 = h$. If the three points must lie on a parabola $y = a + bx + cx^2$ then we have

$$\begin{aligned} y_0 &= a - bh + ch^2 \\ y_1 &= a \\ y_2 &= a + bh + ch^2 \end{aligned} \tag{4.3}$$

and solving for the coefficients a, b, c:

$$\begin{aligned} a &= y_1 \\ b &= \tfrac{y_2 - y_0}{2h} \\ c &= \tfrac{y_2 - 2y_1 + y_0}{2h^2} \quad . \end{aligned} \tag{4.4}$$

For assuring that the solution we found describes really a parabola, we have to require that $c \neq 0$. Since $h \neq 0$, we must have

$$\begin{gathered} y_2 - 2y_1 + y_0 = (y_2 - y_1) - (y_1 - y_0) \neq 0 \quad \Rightarrow \\ \Rightarrow (y_2 - y_1) \neq (y_1 - y_0) \end{gathered}$$

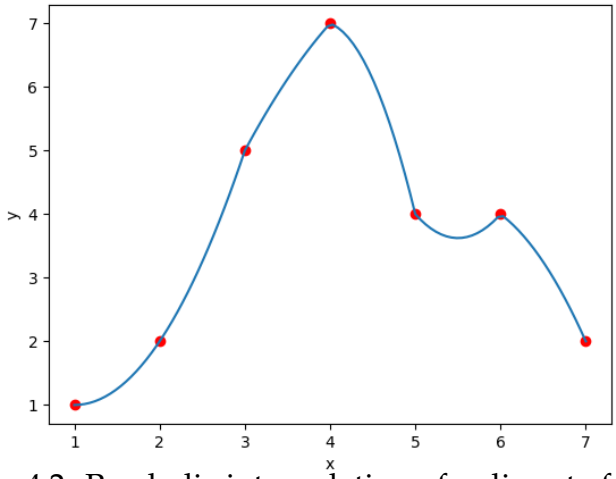

Figure 4.2: Parabolic interpolation of a discrete function.

but the last condition is assured by the non-collinearity of the points. If two points are collinear, the formula falls back in the linear interpolation case. Taking into account a shift of the variable x and the results of Eq. 10.1 we can derive the parabolic interpolation equation

$$P(x) = y_i + \frac{y_{i+1}-y_{i-1}}{2h}(x - x_i) + \frac{y_{i+1}-2y_i+y_{i-1}}{2h^2}(x - x_i)^2$$
$$x_{i-1} \leq x \leq x_{i+1} \quad , \quad i = 1, 3, 5, ..., n-1 \quad . \tag{4.5}$$

Considering the same dataset used in the linear interpolation example, an analog code was written for producing Fig. 4.2.

4.3 Cubic Splines Interpolation

A major drawback of linear and parabolic interpolation is the discontinuity of the first derivative of the obtained functions. The first derivative is discontinuous exactly at the points of the discrete dataset and this might not be realistic in certain applications. A typical example is when the discrete points correspond to positions

in space and the derivative of the interpolating function is thus the velocity. The idea is to use a higher order polynomial function for "glueing" not only the pieces of the function at the discrete points, but also its first derivatives. This is the basis of the **cubic splines** method.
Let's consider again three points where we fix $x_0 = 0$ and therefore $x_1 = h$ and $x_2 = 2h$. We require that the points belong to a cubic function of the form $y = a + bx + cx^2 + dx^3$:

$$\begin{aligned} y_0 &= a \\ y_1 &= a + bh + ch^2 + dh^3 \\ y_2 &= a + 2bh + 4ch^2 + 8dh^3 \end{aligned} \tag{4.6}$$

The derivative of the cubic function is $y' = b + 2cx + 3dx^2$, which implies $y'_0 = b$. With the last condition, we can solve for the cubic function coefficients:

$$\begin{aligned} a &= y_0 \\ b &= y'_0 \\ c &= \frac{8y_1 - 7y_0 - y_2 - 6hy'_0}{4h^2} \\ d &= \frac{3y_0 - 4y_1 + y_2 + 2hy'_0}{4h^3} \quad , \end{aligned} \tag{4.7}$$

which lead to the cubic function

$$y(x) = y_0 + y'_0 x + \frac{8y_1 - 7y_0 - y_2 - 6hy'_0}{4h^2} x^2 + \frac{3y_0 - 4y_1 + y_2 + 2hy'_0}{4h^3} x^3 \quad . \tag{4.8}$$

We assume that we know the value of the derivative at the first point y'_0 but we still need a relation for evaluating the derivative at the other points. Taking the first derivative of Eq. 4.8 and comparing it with the derivative of the cubic equation in $x_2 = 2h$ we have

$$y'(x_2) = b + 4ch + 12dh^2 \quad , \tag{4.9}$$

and substituting the coefficients given in Eqs. 4.7 after some algebra we find

$$y'_2 = y'_0 + 2\frac{y_2 - 2y_1 + y_0}{h} \quad . \tag{4.10}$$

The obtained equations 4.8 and 4.10 can be readily generalized to the **cubic spline formulas**

$$
\begin{aligned}
S(x) = \quad & y_{i-1} + y'_{i-1}(x - x_{i-1}) + \frac{8y_i - 7y_{i-1} - y_{i+1} - 6hy'_{i-1}}{4h^2}(x - x_{i-1})^2 + \\
& + \frac{3y_{i-1} - 4y_i + y_{i+1} + 2hy'_{i-1}}{4h^3}(x - x_{i-1})^3 \\
& y'_{i+1} = y'_{i-1} + 2\frac{y_{i+1} - 2y_i + y_{i-1}}{h} \quad , \quad i = 1, 3, 5, ..n - 1 \quad .
\end{aligned}
\tag{4.11}
$$

In the following code 4.2, the spline interpolation formulas are implemented, and the output of is showed in Fig. 4.3.

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

#Example data
x = np.array([1,2,3,4,5,6,7,8,9,10])
y = np.array([1,2,5,7,4,3,2,4,3,1])

def SplineInterpolation(X,Y,Y10,npoints):

    n = np.size(X)
    h = (X[n-1] - X[0])/(n-1)
    S   = np.zeros(npoints)
    Y1  = np.zeros(np.size(X))

    Y1[0] = Y10
    k=0
    grid = np.array([1,3,5,7])
    for i in grid:

        for x in np.linspace(X[i-1],X[i+1],100):

            A = 8.0*Y[i]-7*Y[i-1]-Y[i+1]-6.0*h*Y1[i-1]
            B = 3.0*Y[i-1]-4.0*Y[i]+Y[i+1]+2.0*h*Y1[i-1]

            S[k] = Y[i-1] + Y1[i-1]*(x-X[i-1]) + \
                   A/4.0/h/h*(x-X[i-1])**2 + \
                   B/4.0/h/h/h*(x-X[i-1])**3
            k=k+1
            Y1[i+1] = \
            Y1[i-1] + 2.0/h*(Y[i+1]-2*Y[i]+Y[i-1])

    return S,Y1

S,Y1 = SplineInterpolation(x,y,1,400)
xf = np.linspace(1,9,np.size(S))

#Plot
plt.plot(x,y,'ro')
plt.plot(xf,S)
plt.xlabel('x')
plt.ylabel('y')
plt.show()
```

Listing 4.2: Spline Interpolation

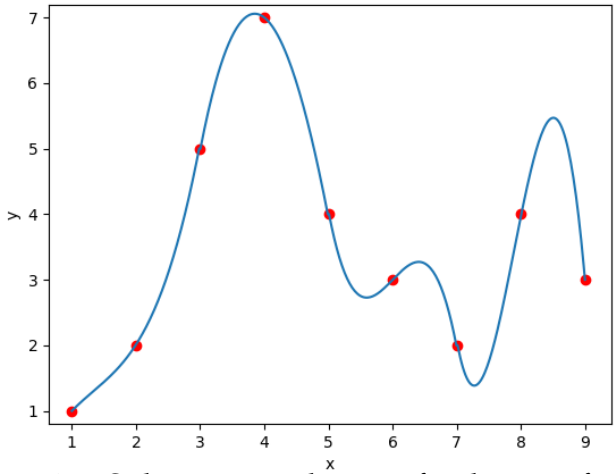

Figure 4.3: Spline interpolation of a discrete function.

Looking at the figure and comparing it with the outputs of linear and parabolic interpolation, it is clear that the continuity of the first derivative generates smoother curves without kinks at the discrete dataset points. Many different interpolation schemes can be constructed along the lines of the cubic splines and we will show one more in the next section.

4.4 Cubic Splines with Smooth Second Derivatives

In some applications, it could be important the guarantee the continuity of the first *and* second derivative. In the previous method, we forced a cubic function to pass through 3 consecutive points. Since a cubic function has 4 parametes, the knowledge 3 points was not sufficient and we provided one initial value of the first derivative. In the method we will describe here, we will use only 2 consecutive points and a cubic function: the additional missing information will be provided by the first *and* second derivative.

Assuming again $x_0 = 0$ and $x_1 = h$ with $y = a + bx + cx^2 + dx^3$ we have

$$\begin{aligned} y_0 &= a \\ y'_0 &= b \\ y''_0 &= 2c \\ y_1 &= a + bh + ch^2 + dh^3 \quad , \end{aligned} \tag{4.12}$$

so in the interval $[0, x_1]$ the cubic function is

$$y(x) = y_0 + y'_0 x + \frac{1}{2} y''_0 x^2 + \frac{y_1 - y_0 - h y'_0 - (1/2)h^2 y''_0}{h^3} x^3 \quad . \tag{4.13}$$

Considering now all the intervals $[x_i, x_{i+1}]$, we can write the general algorithm. First, in analogy to the derivation of Eq. 4.13, we calculate the first and second derivatives:

$$y'_i = y'_{i-1} + h y''_{i-1} + 3\frac{y_i - y_{i-1} - h y'_{i-1} - (1/2)h^2 y''_{i-1}}{h} \quad , \tag{4.14}$$

$$y''_i = y''_{i-1} + 6\frac{y_i - y_{i-1} - h y'_{i-1} - (1/2)h^2 y''_{i-1}}{h^2} \quad , \tag{4.15}$$

and then the final piecewise function

$$\begin{aligned} y(x) = \quad & y_i + y'_i(x - x_i) + \tfrac{y''_i}{2}(x - x_i)^2 + \\ & \tfrac{y_{i+1} - y_i - h y'_i - (1/2)h^2 y''_i}{h^3}(x - x_i)^3 \quad . \end{aligned} \tag{4.16}$$

Besides the usual iterative solution, we can show that the cubic splines with continuous second derivatives can be obtained from the solution of a *tridiagonal* linear

system of equations.
If S(x) is the interpolating function we would like to find, it has to have the property $S(x_i) = y_i$ for $i = 0, 1, ..., n$. If we call S_i the $i - th$ piece of the function, the continuity of the derivatives implies:

$$\begin{array}{l} S_i'(x_i) = S_{i+1}'(x_i) \quad i = 1, 2, ..., n-1 \\ S_i''(x_i) = S_{i+1}''(x_i) = z_i \quad , \end{array} \tag{4.17}$$

where we called z_i the values of the second derivatives. The idea now is to build a function expressing the fact that the second derivative is continuous:

$$S_i''(x) = z_{i-1}\frac{x_i - x}{h} + z_i\frac{x - x_{i-1}}{h} \quad , \quad x_{i-1} \leq x \leq x_i \quad . \tag{4.18}$$

It is easy to verify that $S_i''(x_i) = z_i$ and $S_i''(x_{i-1}) - z_{i-1}$.
By direct integration of Eq. 4.18 we have

$$S_i'(x) = -z_{i-1}\frac{(x_i - x)^2}{2h} + z_i\frac{(x - x_{i-1})^2}{2h} + c_1 \quad , \tag{4.19}$$

$$S_i(x) = z_{i-1}\frac{(x_i - x)^3}{6h} + z_i\frac{(x - x_{i-1})^3}{6h} + c_1 x + c_2 \quad . \tag{4.20}$$

From the last equation, since $y_i = S(x_i)$:

$$\begin{array}{l} y_i = \frac{1}{6}z_i h^2 + c_1 x_i + c_2 \quad , \\ y_{i-1} = \frac{1}{6}z_{i-1} h^2 + c_1 x_{i-1} + c_2 \quad , \end{array} \tag{4.21}$$

we can solve for the constants c_1 and c_2 obtaining

$$\begin{array}{l} c_1 = \frac{(y_i - y_{i-1}) - h^2(z_i - z_{i-1})/6}{h} \quad , \\ c_2 = \frac{(x_i y_{i-1} - x_{i-1} y_i) - h^2(x_i z_{i-1} - x_{i-1} z_i)/6}{h} \quad . \end{array} \tag{4.22}$$

Substituting the coefficients in Eq. 4.20 we obtain the piecewise interpolating cubic function we can solve for the constants c_1 and c_2 obtaining

$$\begin{array}{l} S(x) = \\ z_{i-1}\frac{(x_i - x)[(x_i - x)^2 - h^2]}{6h} + \\ z_i\frac{(x - x_{i-1})[(x - x_{i-1})^2 - h^2]}{6h} + \\ \frac{y_{i-1}(x_i - x) + y_i(x - x_{i-1})}{h} \quad . \end{array} \tag{4.23}$$

We still need to find the values of the z parameters (the second derivatives). We can take the first derivative of Eq. 4.23

$$\begin{array}{l} S'(x) = \\ z_{i-1}\frac{h^2-3(x_i-x)^2}{6h}+ \\ z_i\frac{3(x-x_{i-1})^2-h^2}{6h}+ \\ \frac{y_i-y_{i-1}}{h} \quad . \end{array} \tag{4.24}$$

and impose its continuity $S_i(x_i) - S_{i+1}(x_i)$:

$$\frac{z_{i-1}h}{6} + \frac{z_i h}{3} + \frac{y_i - y_{i-1}}{h} = -\frac{z_i h}{3} - \frac{z_{i+1}h}{6} + \frac{y_{i+1} - y_i}{h} \quad . \tag{4.25}$$

Bringing all the unkowns on the left side we have

$$\frac{1}{6}z_{i-1} + \frac{2}{3}z_i + \frac{1}{6}z_{i+1} = \frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} \quad . \tag{4.26}$$

The last equation describes exactly a tridiagonal system which we learned to solve in the previous chapter.

4.5 Lagrange Interpolation

The methods outlined in the previous sections were quite specialized: one chooses a polynomial and a number of given informations (couple of points, triples of points, derivatives, and so on) and then constructs the piecewise interpolating function. A method due to Laplace is able generalize the procedure for any given number of points.
For explaining the method, we start from the parabolic interpolation of three points $(x_0, y_0), (x_1, y_1), (x_2, y_2)$. Lagrange reasoned that the interpolating function must depend from $y_{0,1,2}$ and therefore assumed a solution of the form

$$y(x) = y_0 A(x) + y_1 B(x) + y_2 C(x) \quad . \tag{4.27}$$

A,B, and C must be second degree polynomials. Since at the nodes the function must work correctly, we must have $y(x_0) = y_0$, $y(x_1) = y_1$, and $y(x_2) = y_2$.

This happens if

$$\begin{array}{lll} A(x_0)=1 & A(x_1)=0 & A(x_2)=0 \\ B(x_0)=0 & B(x_1)=1 & B(x_2)=0 \\ C(x_0)=0 & C(x_1)=0 & C(x_2)=1 \end{array} \tag{4.28}$$

and it is not difficult to see that second-order polynomials satisfying these conditions are

$$A = \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} \quad , \tag{4.29}$$

$$B = \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} \quad , \tag{4.30}$$

$$C = \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \quad . \tag{4.31}$$

Multiplying the parentheses and with some algebra, it can be proved that the obtained solution is identical to the one found in Sec. 4.2: just the form of it is different, and much more useful for a generalization.
The structure of A,B, and C is evident. For example, since A is associated with the point x_0, we see that it is not present in the numerator, while in the denomitator it gets subtracted by the other remaining points. The same observation applies to B and C. Suppose now that we would like to interpolate a cubic through 4 points instead of 3 (we add the point (x_3, y_3)). Following the previous considerations, we should now have

$$A = \frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)} \quad , \tag{4.32}$$

and so on for B, C, and a new coefficient D. The generalization to k points is given by the **Lagrange interpolation formula**. Defining the **cardinal functions**

$$Q_j = \frac{(x-x_0)(x-x_1)...(x-x_{j-1})(x-x_{j+1})...(x-x_{k-1})(x-x_k)}{(x_j-x_0)(x_j-x_1)...(x_j-x_{j-1})(x_j-x_{j+1})...(x_j-x_{k-1})(x_j-x_k)} \quad , \tag{4.33}$$

we can define the Lagrange formula

$$P_k(x) = \sum_{j=0}^{k} y_j Q_j(x) \quad . \tag{4.34}$$

Besides degenerate cases, P_k is a polynomial of degree k and represents a piecewise interpolation formula for k nodes.

4.6 Least Squares Method

Up to now, the nodes contained in the discrete dataset to interpolate, belonged to the interpolating functions. We are now looking for a different method which generates a function which does not necessarily go through the nodes, but it will do it as much as possible. Such a method answers also the need to fit a specific model to a dataset. The method of the least squares is designed for fitting a generic function (not only polynomials) to a dataset.
The idea of the method is quite simple: minimize the deviation of the function from the dataset points. This can be formalized defining the "distance" of the function from a point i as the "error" $e_i = f(x_i) - y_i$. We are not interested in the "direction" of the error, so we might consider $|e_i|$. This choice is not convenient, since at the end we would like to do a minimization, which involves derivatives and the modulus has no continuous derivative everywhere. A more convenient choice is e_i^2 which removes the sign while having a continuous derivative.
What we have to do now is to minimize the overall error E on all nodes:

$$E = \sum_{i=0}^{n} e_i^2 \quad . \tag{4.35}$$

The minimization is done taking derivatives of the error function with respect to the parameters a_i of the fitting function:

$$\frac{\partial E}{\partial a_i} = 0 \quad . \tag{4.36}$$

The latter system of equations have as solution the optimal values of the parameters a_i.

4.7 Linear Interpolation with the Least Squares Method

Let's assume we would like to approximate a dataset (x_i, y_i) $(i = 1, .., N)$ with a linear function of the form $f(x) = a + bx$. The error function will be

$$E(a, b) = \sum_i [y_i - (a + bx_i)]^2 \quad . \tag{4.37}$$

Differentiating with respect to the parameters for minimizing the error

$$\frac{\partial E}{\partial a} = -2 \sum_i^N [y_i - (a + bx_i)] = 0 \quad , \tag{4.38}$$

$$\frac{\partial E}{\partial b} = -2 \sum_i^N [y_i - (a + bx_i)] x_i = 0 \quad , \tag{4.39}$$

which leads to the following result

$$Na + b \sum_i x_i = \sum_i y_i \quad , \tag{4.40}$$

$$a \sum_i x_i + b \sum_i x_i^2 = \sum_i x_i y_i \quad . \tag{4.41}$$

Solving for the parameters a and b, we have

$$a = \frac{\sum_i y_i \sum_i x_i^2 - \sum_i x_i \sum_i x_i y_i}{N \sum_i x_i^2 - \left(\sum_i x_i\right)^2} \quad , \tag{4.42}$$

$$b = \frac{N \sum_i x_i y_i - \sum_i x_i \sum_i y_i}{N \sum_i x_i^2 - \left(\sum_i x_i\right)^2} \quad . \tag{4.43}$$

A more suggestive form of the equations can be obtained introducing the definitions

$$N\sigma_{xx}^2 = \sum_i (x_i - \bar{x})^2 = \left(\sum_i x_i^2\right) - N\bar{x}^2 \quad , \tag{4.44}$$

$$N\sigma_{yy}^2 = \sum_i (y_i - \bar{y})^2 = \left(\sum_i y_i^2\right) - N\bar{y}^2 \quad , \tag{4.45}$$

$$N\sigma_{xy} = \sum_i (x_i - \bar{x})(y_i - \bar{y}) = \left(\sum_i x_i y_i\right) - N\bar{x}\bar{y} \quad . \tag{4.46}$$

where we introduced the *mean values* $N\bar{x} = \sum_i x_i$ and $N\bar{y} = \sum_i y_i$. The number σ_{xy} is often called *covariance* of the two variables x and y and measures how well the points (x,y) are aligned along a line. Rewriting the least squares coefficients with the covariance and the sum of the squared deviation σ_{xx} (the *variance*) we have

$$a = \bar{y} - b\bar{x} \quad , \tag{4.47}$$

$$b = \frac{\sigma_{xy}}{\sigma_{xx}} \quad . \tag{4.48}$$

The quality of the obtained fit can be assessed calculating the *correlation coefficient*

$$\rho = \frac{\sigma_{xy}}{\sqrt{\sigma_{xx}\sigma_{yy}}} \quad , \tag{4.49}$$

which by construction has a maximum (minimum) value of 1 (-1) in the case of a perfect (anti-)correlation.
An example of a linear fit is showed in Fig. 4.7.

4.8 Error on the Estimated Linear Parameters

The parameters a and b of the linear function $f(x) = a + bx$ fitted to a data set (x_i, y_i) (i=0,1,..,N) have also an uncertainty which stems from the uncertainty of the y_i values. The uncertainty σ_y on the y_i values can be estimated with Eq. 4.50:

$$\sigma_y = \frac{1}{N}\sum_i [y_i - (a + bx_i)]^2 \quad . \tag{4.50}$$

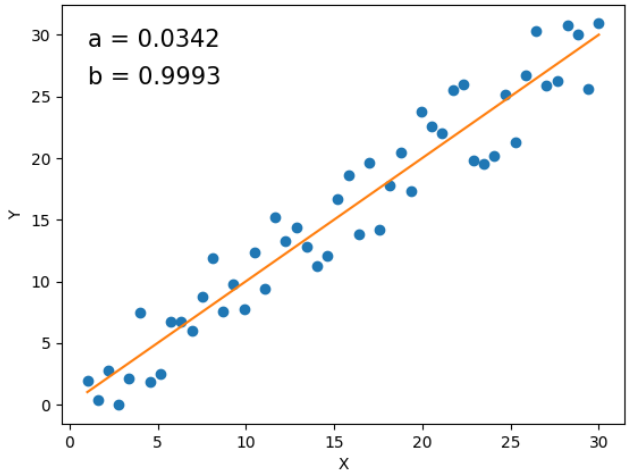

Figure 4.4: Linear fit of a set of 30 data points generated with function $f(x) = a + bx + RDM$ and parameters $a = 0$, $b = 1$. RDM is a random number in the range [-2,2].

Here and in the following, for simplifying the notation we assume that all the sums are made over the entire available dataset: $\sum_{i=0}^{N} \rightarrow \sum_i$.
The previous equation gives a measure of the "dispersion" of the data around the fitted line [1]. Having a definition for σ_y, we can now apply error propagation to the coefficients a and b:

$$\sigma_a^2 = \sum_i \left(\frac{\partial a}{\partial y_i}\right)^2 \sigma_y^2 \quad ,$$

$$\sigma_b^2 = \sum_i \left(\frac{\partial b}{\partial y_i}\right)^2 \sigma_y^2 \quad ,$$

where a and b are expressed by Eqs. 4.42 and 4.43.
In the following , we consider the angular coefficient b and start with calculating

[1] Statistically, the correct formula has an N-2 in the denominator instead of N. This comes for compensating for the previous knowledge of the two parameters a and b. The formula in general can be justified with the maximum likelihood principle applied to gaussian distributed data: in this discussion we neglect these statistical details for concentrating more on the algorithms themselves.

the derivative

$$\frac{\partial b}{\partial y_i} = \frac{1}{\Delta}\frac{\partial b}{\partial y_i}\left(N\sum_i x_i y_i - \sum_i x_i \sum_i y_i\right) = \frac{1}{\Delta}\left(Nx_i - \sum_i x_i\right) \quad , \tag{4.51}$$

where we define[2] $\Delta = N\sum_i x_i^2 - \left(\sum_i x_i\right)^2$ which can be factorized outside the derivative since it depends only on x_i.
Now we can substitute the derivative in the error propagation equation

$$\sigma_b^2 = \frac{1}{\Delta^2}\sum_i \left(Nx_i - \sum_j x_j\right)^2 \sigma_y^2 = \frac{\sigma_y^2}{\Delta^2}\left[N^2\sum_j x_j^2 + N\left(\sum_j x_j\right)^2 - 2N\sum_i x_i \sum_j x_j\right] \quad . \tag{4.52}$$

The last two terms in the parenthesis can be subtracted and the numerator can be identifyed as $N \cdot \Delta$:

$$\sigma_b^2 = \frac{\sigma_y^2}{\Delta^2}\left[N^2\sum_j x_j^2 - N\left(\sum_j x_j\right)^2\right] = \frac{N\sigma_y^2}{\Delta} \quad . \tag{4.53}$$

Repeating the same procedure for σ_a^2, the formulas for the calculation of the uncertainties on the linear regression parameters are

$$\sigma_a^2 = \frac{\sigma_y^2 \sum_i x_i^2}{\Delta} \quad , \tag{4.54}$$

$$\sigma_b^2 = \frac{N\sigma_y^2}{\Delta} \quad . \tag{4.55}$$

[2] the name Δ for this quantity indicates that it is actually the determinant of the linear system to be solved for obtaining a and b.

CHAPTER 5 | Root Finding

A common problem is the solution of a non-linear or linear equation or a system of equations $f(x) = 0$. Geometrically, the same problem can be viewed (for example, in one dimension) as finding the x-axis crossing point of a function $y = f(x)$. The solution of linear systems of equations analyzed in Chap. 3 can be considered part of the root finding algorithms.
In general, root finding works best when we know already that a solution is contained in a known interval.

5.1 Incremental Method

This method is likely the simplest one which can be applied for the search of an equation's roots. The idea is to make small steps dx until a change of sign in the function is detected. If the function is positive before the step and negative after it (or the other way around), it means that we crossed the x-axis and we found a root. A simple implementation of this concept is given by the following code 5.1

```python
#Incremental Method algorithm
def IncrementalMethod(f,a,b,dx):
    x1 = a
    x2 = a+dx

    f1 = f(a)
    f2 = f(x2)

    while f1*f2 > 0.0:
        if x1 >= b: return None,None
        x1 = x2
        f1 = f2
        x2 = x1 + dx
        f2 = f(x2)
    else: return x1, x2
```

Listing 5.1: Incremental Method

In the code, the condition `f1*f2 > 0` checks if the function changes sign after a small step dx.
This simple algorithm has some potential problems. This first one is computational: if the steps are small and the interval large (in the case we do not have any guesses about the location of the root), the convergence might be slow (especially if the algorithm has to be executed several times). Another obvious issue consists in the inability to detect roots which are closer than dx.
Another problem is that singularities can be identified as roots. The classic case is $f(x) = \tan(x)$, where at $x = n\pi$ $n = 1,3,5,..$ the function changes sign without crossing the axis.
Double roots are also not detected by this algorithm: try for example to find the roots of $(x-1)^2 = 0$ and then the roots of the equivalent equation $x^2 - 2x + 1 = 0$.

5.2 Bisection Method

A slight improvement could be to constrain the location of the root from both sides, instead of moving in small steps from one side of the interval to the other in one direction. This idea is contained in the bisection method, where the interval is iteratively halved until the root is reached.

The presence of a root between two points x_1 and x_2 at its sides leads to the condition we have seen in the previous section $f(x_1) \cdot f(x_2) < 0$.
The idea of bisection is to calculate a point $x_3 = (x_1 + x_2)/2$ between x_1 and x_2 and then check the sign of $f(x_1) \cdot f(x_3) < 0$ and $f(x_2) \cdot f(x_3) < 0$.
The algorithm proceeds iteratively halving one of the two intervals where the product is negative.

```
from math import ceil,log
def BisectionMethod(f,x1,x2,tol):
    f1=f(x1)
    if f1==0: return a
    f2=f(x2)
    if f2==0: return b
    if f1*f2 > 0.0:
        print("Root not in the interval")
        return 0
    n = ceil(log(abs(x2-x1)/tol)/log(2.0))
    for i in range(n):
        x3 = 0.5 * (x1+x2)
        f3 = f(x3)
        if f3 == 0.0: return x3
        if f2*f3 < 0.0:
            x1 = x3
            f1 = f3
        else:
            x2 = x3
            f2 = f3
    return (x1+x2)/2.0
```

Listing 5.2: Bisection Method

Also the previous code 5.2 has problems with the identification of double roots.

5.3 Newton's Method

Newton's method (sometimes called Newton-Raphson method) exploits the knowledge of the derivative of a function for locating the roots. The requirement to know the derivative might also be seen as a drawback of this algorithm.

Newton's method starts with a guess solution x_0. If $f(x_0) = 0$ we found a root and if it is not the case, we generate a next guess solution in the following way. We consider the tangent line to f at x_0: this line must cross the x-axis in a location between x_0 and the root. In general, this choice generates a larger step than the one chosen in an incremental method. This leads to an average faster convergence. The equation of the tangent is

$$y - f(x_0) = f'(x_0)(x - x_0) \tag{5.1}$$

and the intersection $x_1$with the x axis is given by the condition $y = 0$

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)} \quad . \tag{5.2}$$

which leads to the iterative solution

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)} \quad , \quad f'(x_1) \neq 0 \quad . \tag{5.3}$$

When $x_{i+1} = x_i$ (or $|x_{i+1} - x_i| < \epsilon$), we obtain the value of the root. This algorithm can converge to the root faster than the previous ones and can identify also double roots. The number of steps is dependent from the required precision and the derivative must be calculated in advance. The method can be applied to both linear and non-linear problems.
A further improvement of the algorithm is the **Generalized Newton Method** where we consider the interative equation

$$x_{i+1} = x_i - \omega\frac{f(x_i)}{f'(x_i)} \quad , \quad f'(x_1) \neq 0 \quad , \tag{5.4}$$

where ω is a suitable weight which might speed up the convergence. Usually, $0 < \omega < 2$, with $\omega = 1$ the special case of Newton's algorithm. It can be proved, that the error at each step of Newton's algorithm is given by

$$E_{i+1} = -\frac{f''(x_i)}{f'(x_i)}E_i^2 \quad , \tag{5.5}$$

thus the method converges quadratically.
A last observation is about the calculation of the derivatives. We said that a drawback of Newton's algorithm might be the required knowledge of the first derivatives. This problem could be circumvented letting the computer, through a numerical method, calculate them.

5.4 Multidimensional Newton's Method

Another way to derive Newtons's method is to Taylor-expand the function f

$$f(x_{i+1}) \approx f(x_i) + f'(x_i)(x_{i+1} - x_i) + ... \quad . \tag{5.6}$$

If x_i and x_{i+1} are close, we can drop higher orders and keep only the first and if x_{i+1} is a root (i.e. $f(x_{i+1} = 0)$ we obtain Newton's formula.
The method can readily generalized to more dimensions, where the Taylor expansion around a vector $\bar{x}$ is

$$f(\bar{x} + \Delta\bar{x}) \approx f(\bar{x}) + J(\bar{x})\Delta\bar{x} + ... \quad . \tag{5.7}$$

The *Jacobian* J is the matrix of first derivatives

$$J_{ij} = \frac{\partial f_i}{\partial x_j} \quad , \tag{5.8}$$

and $\Delta\bar{x}$ is a vector step. If $f(\bar{x} + \Delta\bar{x}) = 0$, we have to solve the matrix equation $J(\bar{x})\Delta\bar{x} = -f(\bar{x})$ obtaining the new step

$$\Delta\bar{x} = -J^{-1}(\bar{x})f(\bar{x}) \quad . \tag{5.9}$$

We can now summarize the multidimensional Newton's algorithm with the following list of steps

1. Choose a starting vector $\bar{x_i}$ and evaluate $f(\bar{x_i})$ with $i = 0$.
2. Calculate the Jacobian $J(\bar{x})$.
3. Solve Eq. 5.9.
4. Calculate the new vector $x_{i+1} = x_i + \Delta\bar{x_i}$.
5. repeat steps 2, 3, 4, 5, until $|\Delta\bar{x}| < \epsilon$.

Note that Eq. 5.9 is a matrix equation and thus it could be solved for example with the *Gauss Algorithm* presented in Chap. 3.
In this multidimensional version of the algorithm, the calculation of the derivatives might be even more impractical as the number of equations grows. A numerical calculation of the derivatives becomes even more important than in the one-dimensional case. Without relying to sophisticate algorithms for the calculation of the derivative, for this application we could use directly the definition

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(\bar{x} + he_j) - f_i(\bar{x})}{h} \quad ,$$

where e_j is a unit vector representing a direction and, for consistency with the equations introduced before, $\Delta\bar{x} = he_j$.
The multi-dimensional Newton's method is implemented in the code 5.3 which also includes checks for arresting the algorithm. The first check, stops the iteration when

$$\frac{||f(\vec{x})||}{D} < \text{tolerance} \quad , \tag{5.10}$$

where D is the number of dimensions and $||\cdot||$ is the Euclidean norm. The second check controls if the steps become too small with respect to the tolerance:

$$\frac{||\Delta\vec{x}||}{\max|\vec{x}|} < \text{tolerance} < 1 \quad . \tag{5.11}$$

The algorithm (see code 5.3) requires many evaluations of the function f, which might be expensive, especially for the $\mathcal{O}(n)$ time complexity of the Jacobian calculation. A simplification might be keeping the Jacobian constant when close to the solution (after Δx becomes smaller than a number decided beforehand, for example).

```python
import numpy as np
from math import sqrt

def NewtonN(f, x, h=1.0e-4, maxint=20, tol=1.0e-6):

    def Jacobian(f,x):
        n = len(x)
        J = np.zeros((n,n))
        f0 = f(x)
        for i in range(n):
            tmp = x[i]
            x[i] = tmp + h
            f1 = f(x)
            x[i] = tmp
            J[:,i] = (f1-f0) / h
        return J, f0

    for i in range(maxint):
        J, f0 = Jacobian(f,x)
        if sqrt(np.dot(f0,f0)/len(x)<tol): return x
        dx = GaussElimination(J,-f0)
        x = x + dx
        if sqrt(np.dot(dx,dx)) < \
           tol * max(abs(np.amax(x)),1.0):
            return x

    print("Maximum number of iterarations reached.")
```

Listing 5.3: N-dim Newton Method

5.5 Secant Method

This method is similar to Newton's method but it does not require the calculation of the derivative. The idea at the basis of this algorithm is considering a line between two starting points x_1 and x_2

$$y = \frac{f(x_1) - f(x_2)}{x_1 - x_2}(x - x_1) + f(x_1) \quad , \tag{5.12}$$

and calculate the point x_3 where it intersects the x-axis

$$y = 0 \Rightarrow x_3 = x_2 - f(x_2)\frac{x_2 - x_1}{f(x_2) - f(x_1)} \quad . \tag{5.13}$$

The points x_1 and x_3 can be now taken for a second calculation of the "secant" line. The iteration of the process should converge to the point where the function f crosses the x-axis and therefore to the root of the equation

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})} \quad . \tag{5.14}$$

One advantage of the secant method over the bisection one is that no fixed interval should be specified. On the other hand, without a fixed interval, the algorithm could diverge.

5.6 Brent's Method

The "hybrid" method of Brent [1] evolves the bisection algorithm including parabolic instead of linear interpolation. It is generally faster than bisection, depending on the smoothness of the function. Like bisection, the advantege of Brent's method lies in the fact that it does not need explicitly the calculation of derivatives.
In the bisection method, starting from two points x_1 and x_2 supposedly containing the root in the interval $[x_1, x_2]$, a third point $x_3 = (x_1 + x_2)/2$ is constructed. Bisection continues looking in which direction the function changes sign, while in Brent's algorithm, an *inverse parabolic interpolation* is used leveraging on the three obtained points. Since we would like to obtain a "new" x-coordinate, we use parabolic interpolation where the roles of f and x are interchanged in the *cardinal functions*

$$x(f) = \frac{(f - f_2)(f - f_3)}{(f_1 - f_2)(f_1 - f_3)}x_1 + \frac{(f - f_1)(f - f_3)}{(f_2 - f_1)(f_2 - f_3)}x_2 + \frac{(f - f_1)(f - f_2)}{(f_3 - f_1)(f_3 - f_2)}x_3 \quad , \tag{5.15}$$

[1] Richard P. Brent is an Australian mathematician and Computer scientist and devised this algorithm in 1973. Hi was a PhD student of George E. Forsythe (1917-1972), founder of the Stanford Computer science department and the person who coined the term "Computer Science". His wife, mathemticin Alexandra W. Illmer Forsythe, was one of the first female Computer scientists and the first ever person writing a textbook on the subject: "Computer Science: A First Course" (1969).

where $f_i = f(x_i)$ and $f = f(x)$.
Since we look for a crossing of the interpolation with the x-axis, we choose f=0, and after some algebra

$$x(0) = -\frac{f_2 f_3 x_1(f_2 - f_3) + f_3 f_1 x_2(f_3 - f_1) + f_1 f_2 x_3(f_1 - f_2)}{(f_1 - f_2)(f_2 - f_3)(f_3 - f_1)} \quad . \tag{5.16}$$

The updated candidate root is $x_{new} = x_3 + \Delta x$ so that the "change" in abscissa is $\Delta x = x(0) - x_3$ which can be rewritten as

$$\Delta x = f_3 \frac{x_3(f_1 - f_2)(f_2 - f_3 + f_1) + f_2 x_1(f_2 - f_3) + f_1 x_2(f_3 - f_1)}{(f_2 - f_1)(f_3 - f_1)(f_2 - f_3)} \quad . \tag{5.17}$$

A code implementing the algorithm is shown in code 5.4. The code is rather simplified and a number of checks can be added. For example, a check should be done in order to make sure that the root does not lie at the borders of the interval and that the solution is really inside the interval. The bisection method is still used if quadratic interpolation results in a point outside the search interval. The convergence of the method is usually very fast for smooth functions. In general, Brent's method needs less steps with respect to bisection, although a single step is computationally more expensive. This is the most widely used root finding method in numerical libraries, which also implement more numerical checks at different stages of the computation.

```
def BrentMethod(f,a,b,maxiter=20,tol=1e-6):
    x1 = a
    x2 = b
    f1 = f(x1)
    f2 = f(x2)
    x3 = (a+b)/2
    for i in range(maxiter):
        f3 = f(x3)

        # Reduce interval
        if f1*f3<0.0: b = x3
        else: a = x3

        # Output root if interval small enough
        if (b-a) < tol*max(abs(b),1.0): return 0.5*(a+b)

        # Parabolic interpolation
        A = x3*(f1-f2)*(f2-f3+f1) + \
                f2*x1*(f2-f3) + \
                f1*x2*(f3-f1)
        B = (f2-f1)*(f3-f1)*(f2-f3)

        # Check division by zero and shift point
        try: dx = f3 * A/B
        except ZeroDivisionError: dx = b-a
        x = x3 + dx

        # If interpolation out of interval, do bisection
        if (b-x)*(x-a) < 0.0:
            dx = 0.5*(b-a)
            x = a + dx

        # Set new reduced interval
        if x<x3:
            x2 = x3
            f2 = f3
        else:
            x1 = x3
            f1 = f3
        x3 = x
        print(x)
    print("Maximum iterations reached")
```

Listing 5.4: Brent's Method

CHAPTER 6 | Numerical Integration

6.1 Introduction

Numerical integration is also called *quadrature* and many algorithms exist depending on the wanted precision, form of the integrating function and dimensions. Numerical integration is an extremely important technique, given the fact that actually very few functions can be integrated analytically. Actually, picking a random function from the space of all functions, the probability that such function can be analytically integrated is zero.
Broadly speaking, there are two classes of algorithms: Newton-Cotes (NC) and Gaussian integration (GI). A separate category encompasses stochastic techniques like Monte Carlo methods.
NC techniques are characterized by an equal spacing in the integration variable, while GI can also use variable-step mashes of points. NC are based on the interpolation formulas studied in the previous chapter and are generally very effective for smooth functions. An advantage of GI methods is their ability to integrate also functions containing for example singularities.
In general, the quadrature problem can be defined as

$$I = \int_a^b f(x) \approx \sum_{i=0}^{n} a_i f(x_i) \quad , \tag{6.1}$$

where the integral I of a function $f(x)$ is approximated by a finite sum over a mesh of points x_i (the "nodes" or *nodal abscissas*). The coefficients a_i are sometimes called the *weights*.

6.2 Newton-Cotes Methods

The NC methods are based on the integration of a function interpolating $f(x)$. This can be achieved in a general way considering Lagrange's interpolation formula in Eq. 4.5: in this way, we can rewrite f(x) as a polynomial of degree n

$$P_n(x) = \sum_{i=0}^{n} f(x_i) Q_i(x) \quad , \tag{6.2}$$

where $Q_i(x)$ are the cardinal functions defined in Eq. 4.5. Substituting in Eq. 6.1

$$I = \int_a^b P_n(x)dx = \sum_{i=0}^{n} \left(f(x_i) \int_a^b Q_i(x)dx \right) = \sum_{i=0}^{n} a_i f(x_i) \quad , \tag{6.3}$$

where the weights a_i are

$$a_i = \int_a^b Q_i(x)dx \quad , \quad i = 0, 1, 2, ..., n \quad . \tag{6.4}$$

It looks like we have reduced the integration problem to another one (the calculation of the weights), but the Lagrange interpolation formula involves polynomials, for which the integrals can be carried out.

6.3 The Trapezoidal Rule

The trapezoidal rule is the simplest numerical algorithm of the NC type and implies interpolating linearly the function to integrate. The linear interpolation corresponds to $n = 1$ and the cardinal functions are:

$$Q_0(x) = \frac{x - x_1}{x_0 - x_1} \quad , \quad Q_1(x) = \frac{x - x_0}{x_1 - x_0} \quad , \tag{6.5}$$

which we can integrate to obtain the weights

$$a_0 = \frac{1}{h} \int_a^b (x - b)dx = \frac{1}{2h}(b - a)^2 = \frac{h}{2} \quad ,$$

$$a_1 = \frac{1}{h} \int_a^b (x - a)dx = \frac{1}{2h}(b - a)^2 = \frac{h}{2} \quad .$$

Substitution in Eq. 6.3 gives

$$I = \frac{h}{2}(f(a) + f(b)) \quad . \tag{6.6}$$

The last formula corresponds to the area of the trapezoid one obtains approximating the function f with a line in the interval $[a, b]$. Indeed a simpler derivation would have been to consider the line from the consecutive points (with spacing h) x_0 to x_1

$$y = y_0 + \frac{y_1 - y_0}{h}(x - x_0) \quad , \tag{6.7}$$

and integrating

$$\int_{x_0}^{x_1} y(x)dx = \left[y_0 x + \frac{y_1 - y_0}{h} \frac{(x - x_0)^2}{2} \right]_{x_0}^{x_1} = \frac{h}{2}(y_0 + y_1) \quad . \tag{6.8}$$

In practice, the x axis is divided in a regular mesh of points and the formula is applied piecewise

$$I \approx \frac{h}{2} \sum_{i=0}^{n-1} [f(x_i) + f(x_{i+1})] \quad . \tag{6.9}$$

The latter formula can be rearranged for obtaining a small computational advantage

$$I \approx \frac{h}{2} \left[(y_0 + y_1) + (y_1 + y_2) + ...(y_{n-1} - y_n) \right] \tag{6.10}$$

where $y_i = f(x_i)$. Summing the repeated terms, we arrive to the final formula for the **trapezoidal rule**

$$I = \int_a^b f(x)dx \approx \frac{h}{2} \left[y_0 + 2 \left(\sum_{i=1}^{n-1} y_i \right) + y_n \right] \quad . \tag{6.11}$$

It is interesting to estimate the error involved in the trapeziodal approximation of the integral I and an answer is contained in the following theorem:

Theorem 5. *If $f(x)$ is C^2 in the interval $[a, b]$ and the error E_i in $[x_i, x_{i+1}]$ is*

$$E_i = \int_{x_i}^{x_{i+1}} f(x)dx - \frac{h}{2} \left[f(x_i) + f(x_{i+1}) \right] \quad ,$$

then the error has the upper limit

$$|E_i| \leq \frac{h^3}{12} \max_{[x_i, x_{i+1}]} |f''(x)| \quad .$$

The convergence of the method is assured by

Theorem 6. *Under the assumptions of previous theorem, if the total error is* $E = \sum_i E_i$ *, then*

$$\lim_{h \to 0} E = 0$$

A final note about the error: a more complete analysis yields

$$E = a_1 h^2 + a_2 h^4 + a_3 h^6 + ... \quad . \tag{6.12}$$

The last formula proves that the error does not follow exactly a h^3 behaviour as Theorem 5 might seem to imply. This is in fact not the case, since the evaluation of the second derivative f'' is not completely independent from h.

6.4 Simpson's Rule

Simpson's rule (sometimes called "1/3 rule") has a similar time computational complexity as the trapezoidal rule, but a higher accuracy. It can be derived using $n = 2$ in the NC formula (the trapezoidal rule had $n = 1$). Instead of applying the NC formula, we will derive Simpson's algorithm starting from a parabola interpolating three points $(x_0, y_0), (x_1, y_1), (x_2, y_2)$ choosing $x_0 = -h$, $x_1 = 0$, $x_2 = h$ (see Eq. 4.5)

$$P(x) = y_1 + \frac{y_2 - y_0}{2h} x + \frac{y_2 - 2y_1 + y_0}{2h^2} x^2 \quad . \tag{6.13}$$

By direct integration

$$\int_{x_0}^{x_2} P(x) dx = \frac{h}{3}(y_0 + 4y_1 + y_2) \quad . \tag{6.14}$$

Since the result does not depend on $x_{i=0,1,2}$, it can be generalized to

$$\int_{x_1}^{x_{i+2}} P(x) dx = \frac{h}{3}(y_i + 4y_{i+1} + y_{i+2}) \quad . \tag{6.15}$$

For the integration of a function $f(x)$ in an interval $[a,b]$ divided in n equal sub-intervals ($h = (b-a)/n$), interpolating every three points we can approximate its integral I with

$$\begin{aligned} I &= \frac{h}{3}\sum_{i=0}^{n/2-1}(y_{2i} + 4y_{2i+1} + y_{2i+2}) \\ &= \int_a^b f(x)dx = \frac{h}{3}\left[(y_0 + 4y_1 + y_2) + (y_2 + 4y_3 + y_4) + ... \right. \\ &\left. + (y_{n-2} + 4y_{n-1} + y_n)\right] \quad . \end{aligned}$$

Regrouping the terms in the second line of the previous equation, we can rewrite the Simpson's rule in an alternative form for computational purposes, minimizing the function calls

$$I = \int_a^b f(x)dx \approx \frac{h}{3}\left[y_0 + 4y_1 + 2y_2 + 4y_3 + ... + 2y_{n-2} + 4y_{n-1} + y_n\right] \quad . \tag{6.16}$$

The error estimate for Simpson's rule is given by the following

Theorem 7. *If f(x) is C^4 in the interval $[a,b]$, its total error is*

$$|E| \leq \frac{h^5}{90}\sum_{i=0}^{n/2-1}\left(\max_{[x_{2i},x_{2i+2}]}\left|\frac{d^4f(x)}{dx^4}\right|\right) \quad .$$

The latter result shows also how much this method is precise with respect to the trapezoidal rule, where the error scales with the smaller power h^3.

6.5 Romberg Integration

Before introducing this high-order integration technique, we have first to discuss an extrapolation algorithm known as **Richardson Extrapolation**. Suppose we have to calculate numerically a quantity I and our numerical estimation $i(h)$ depends from a small parameter h. If the error we make in the estimation is E(h) and the error scales with some power p of h we have

$$I = i(h) + ch^p \quad , \tag{6.17}$$

where c is some constant independent from h.
We can calculate numerically I on two different grids of points such that h_1 are the steps on the first grid and h_2 are different steps (coarser or finer) on a second grid:

$$I = i(h_1) + ch_1^p$$
$$I = i(h_2) + ch_2^p$$

We can now eliminate the constant c in the previous equations and solve for I obtaining

$$I = \frac{\left(\frac{h_1}{h_2}\right)^p i(h_2) - i(h_1)}{\left(\frac{h_1}{h_2}\right)^p - 1} \quad . \tag{6.18}$$

For simplicity, it is common to choose one grid twice as smaller than the other, thus $h_2 = h_1/2$ and the extrapolation formula becomes

$$I = \frac{2^p i(h_1/2) - i(h_1)}{2^p - 1} \quad . \tag{6.19}$$

Why the previous formula is an extrapolation formula will become clear in discussing the following integration algorithm, which will merge the trapezoidal rule with Richardson extrapolation.
Given a function f(x) to integrate, we introduce a sequence of finer and finer grids with width

$$h = \frac{b-a}{2^{i-1}} \quad , \tag{6.20}$$

such that each grid is half as smaller than the previous one (remembering the choice we made in Eq. 6.19). Now we introduce the notation

$$I_i = R_{i,1} \quad , \tag{6.21}$$

where I_i is the integral of f evaluated with the trapezoidal rule on 2^{i-1} sub-intervals (or *frames*) of $[a, b]$.
The algorithm starts with the calculation of $I_1 = R_{1,1}$: in this case $h = (b-a)$ so we have only one frame. Then, we calculate $I_2 = R_{2,1}$ with $p = 2$ obtaining an integration over two frames. Now we can use Richardson extrapolation to eliminate

the $O(h^2)$ error and going into the second step of the algorithm:

$$R_{2,2} = \frac{2^2 R_{2,1} - R_{1,1}}{2^2 - 1} = \frac{4}{3} R_{2,1} - \frac{1}{3} R_{1,1} \quad . \tag{6.22}$$

The next step is the calculation with the trapezoidal rule of $I_3 = R_{3,1}$ and obtain the next extrapolation using the previous finer grid of two frames

$$R_{3,2} = \frac{4}{3} R_{3,1} - \frac{1}{3} R_{2,1} \quad . \tag{6.23}$$

After this calculation, we can take p=4

$$R_{3,3} = \frac{2^4 R_{3,2} - R_{2,2}}{2^4 - 1} = \frac{16}{15} R_{3,2} - \frac{1}{15} R_{2,2} \quad . \tag{6.24}$$

obtaining a $O(h^6)$ error.
How the algorithm works is shown in Fig. 6.5 in a graphical way. The figure suggests that the Romberg algorithm can be stored in a triangular matrix where the diagonal represent the results. When two successive diagonal elements are smaller than a chosen tolerance, the algorithm can stop. Actually the matrix representation is only useful for understanding the algorithm, since computationally only arrays are needed.
The generalization of the previous formulas is

$$R_{i,j} = \frac{4^{j-1} R_{i,j-1} - R_{i-1,j-1}}{4^{j-1} - 1} \quad , \quad i > 1 \quad , j = 2, 3, .., i \quad . \tag{6.25}$$

The `Python` code 6.1 implements the Romberg algorithm. The code employes an iterative version of the trapezoidal rule adapted for 2^{k-1} frames, which works as follows. Defining the integration interval length as L=(b-a), for one frame (or $k = 1$) the trapezoidal integral is

$$I_1 = \frac{L}{2}(f(a) + f(b)) \quad .$$

For $k = 2$ (two frames)

$$I_2 = \frac{L}{4}(f(a) + 2f(a + \frac{L}{2}) + f(b)) = \frac{I_1}{2} + \frac{L}{2} f(a + \frac{L}{2}) \quad ,$$

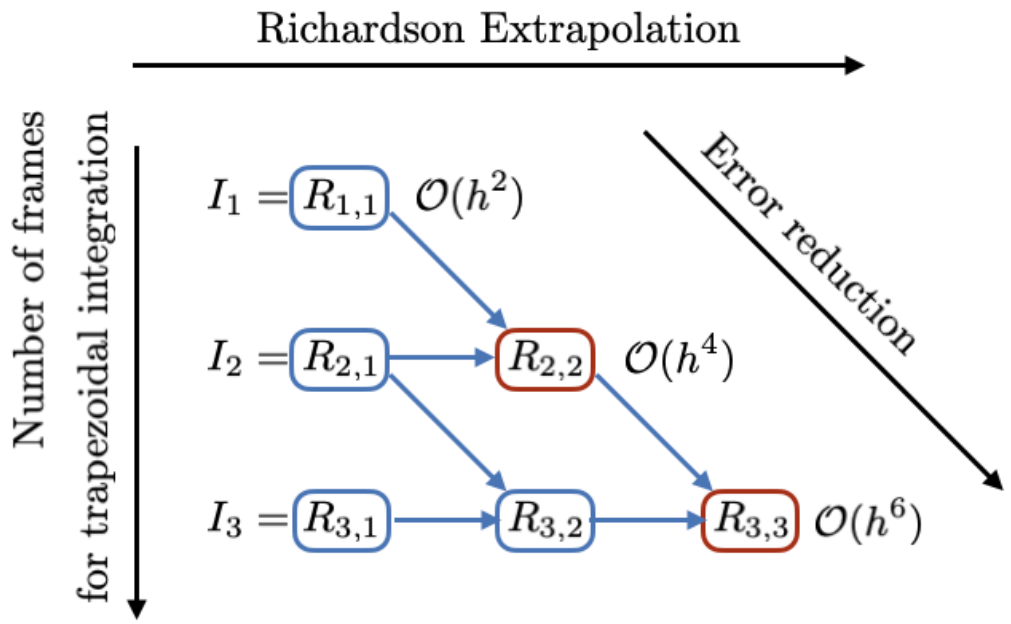

Figure 6.1: Graphical representation of the Romberg integration algorithm steps where two trapezoidal integrations on different grids are merged through Richardson extrapolation.

and for $k = 3$ (4 frames)

$$I_3 = \frac{L}{8}(f(a) + 2f(a+\frac{L}{4}) + 2f(a+\frac{H}{2}) + 2f(a+\frac{3L}{4}) + f(b)) = \frac{I_2}{2} + \frac{L}{4}\left[f(a+\frac{L}{4}) + f(a+\frac{3L}{4})\right] \quad .$$

```python
import numpy as np

#Iterative Trapezoid Routine
def Trapezoid(f,a,b,Iold,k):
    if k==1: Inew = (f(a)+f(b))*(b-a)/2.0
    else:
        n = 2**(k-2)
        h = (b-a)/n
        x = a + h/2.0
        sum = 0.0
        for i in range(n):
            sum = sum + f(x)
            x = x + h
        Inew = (Iold + h*sum)/2.0
    return Inew

#Romberg Integration
def Romberg(f,a,b,tol):

    #Richardson Extrapolation
    def Richardson(r,k):
        for j in range(k-1,0,-1):
            c = 4.0**(k-j)
            r[j] = (c*r[j+1] - r[j]) / (c-1.0)
        return r

    r = np.zeros(21)
    r[1] = Trapezoid(f,a,b,0.0,1)
    r_old = r[1]

    for k in range(2,21):
        r[k] = Trapezoid(f,a,b,r[k-1],k)
        r = Richardson(r,k)
        if (r[1]-r_old) < tol*max(abs(r[1]),1.0):
            return r[1],2**(k-1)
        r_old = r[1]
    print("Romberg Integration not converged")

def f(x):creturn np.sin(x)
a=0
b=np.pi
print(Romberg(f,a,b,1e-6))
```

Listing 6.1: Romberg Integration

In general, for $k > 1$ we can derive

$$I_k = \frac{I_{k-1}}{2} + \frac{L}{2^{k-1}} \sum_{i=1}^{2^{i-1}} f\left[a + \frac{(2i-1)L}{2^{k-1}}\right] \quad , k = 2, 3, .. \quad . \tag{6.26}$$

6.6 Gaussian Quadrature: Introduction

Gaussian quadrature is a method for integrating a function using the approximation

$$\int_a^b f(x)dx \approx \sum_i w(x_i) f(x_i) \quad , \tag{6.27}$$

where the problem is to find the *weights* w_i and the nodes x_i, in analogy to the Newton-Cotes method.
We are going to introduce the idea of Gaussian quadrature restricting for now to the integration interval $[a = -1, b = 1]$.
The idea of the method is to approximate the function f(x) with a polynomial

$$f(x) \approx a_0 + a_1 x + a_2 x^2 + a_3 x^3 ... \quad . \tag{6.28}$$

Let us try to consider a linear approximation first. According to Eq. 6.27 we have

$$\int_{-1}^{1} f(x)dx = w(x_1) f(x_1) \quad , \tag{6.29}$$

and we have to find the weight and the node x_1. For finding two unknowns we need two equations. A first equation is found supposing f equal to a constant (zero-order) and we can choose 1 as its value. The second equation assumes a linear function

$$\int_{-1}^{1} 1 \cdot dx = w_1 \cdot 1 = 2 \quad ,$$
$$\int_{-1}^{1} x \cdot dx = w_1 \cdot x_1 = 0 \quad .$$

The solutions are $x_1 = 0$ and $w_1 = 2$ and we discover the not surprising result

$$\int_{-1}^{1} f(x)dx = 2f(0) \quad . \tag{6.30}$$

The latter formula is the lowest order approximation of an integral and it is exact in the case of a linear function.
Consider now the case of a *cubic* approximation $f = a_0 + a_1 x + a_2 x^2 + a_3 x^3$

$$\int_{-1}^{1} f(x)dx = w(x_1)f(x_1) + w(x_2)f(x_2) \quad . \tag{6.31}$$

A cubic has four coefficients and the latter equation has also four unknowns ($x_{1,2}$, $w_{1,2}$). With an analog procedure as the one used in the linear approximation

$$\begin{aligned}
\int_{-1}^{1} 1 \cdot dx &= w_1 + w_2 = 2 \quad , \\
\int_{-1}^{1} x \cdot dx &= w_1 x_1 + w_2 x_2 = 0 \quad , \\
\int_{-1}^{1} x^2 \cdot dx &= w_1 x_1^2 + w_2 x_2^2 = 2/3 \quad , \\
\int_{-1}^{1} x^3 \cdot dx &= w_1 x_1^3 + w_2 x_2^3 = 0 \quad ,
\end{aligned}$$

noting also that all the odd powers result in a zero integral, since we integrate an odd function on a symmetric interval.
Solving the latter system of equations we have $w_1 = w_2 = 1$ and $x_{1,2} = \mp 1/\sqrt{3}$. Thus we obtain the following approximation formula

$$\int_{-1}^{1} f(x)dx = f\left(-\frac{1}{\sqrt{3}}\right) + f\left(\frac{1}{\sqrt{3}}\right) \quad , \tag{6.32}$$

which is *exact* for cubic functions.
Continuing the same procedure to high (odd) orders, we can obtain in principle an increasingly better approximation of the function $f(x)$.

6.7 Gaussian Quadrature: a more general case

More in general, the Gaussian Quadrature procedure allows the estimation of integrals of the type

$$I = \int_a^b w(x)f(x)dx \quad , \tag{6.33}$$

where $w(x)$ is called *weighting function*. As we did in the previous section, we will try to approximate f(x) with a polynomial $P_m(x)$ which we choose of degree $m = 2n + 1$ (an odd order, as before)

$$\int_a^b w(x)P_m(x)dx = \sum_{i=0}^{n} a_i P_m(x_i) \quad , m \leq 2n+1 \quad . \tag{6.34}$$

Note that now w indicates the weighting function and the weights defined in the previous section are now called a_i.
Concretely, taking for example the interval $[0, \infty]$, $w(x) = e^{-x}$ and adopting a cubic interpolation

$$\begin{aligned}
\int_0^\infty e^{-x}dx = a_0 + a_1 = 1 \quad , \\
\int_0^\infty e^{-x}xdx = a_0x_0 + a_1x_1 = 1 \quad , \\
\int_0^\infty e^{-x}x^2dx = a_0x_0^2 + a_1x_1^2 = 2 \quad , \\
\int_0^\infty e^{-x}x^3dx = a_0x_0^3 + a_1x_1^3 = 6 \quad .
\end{aligned}$$

Solving the system we obtain

$$\begin{aligned}
x_0 = 2 - \sqrt{2} \quad , x_1 = 2 + \sqrt{2} \quad , \\
a_0 = \frac{\sqrt{2}+1}{2\sqrt{2}} \quad , \frac{\sqrt{2}-1}{2\sqrt{2}} \quad ,
\end{aligned}$$

and the integration formula finally is

$$\int_0^\infty e^{-x}f(x)dx \approx \frac{1}{2\sqrt{2}}\left[(\sqrt{2}+1)f(2-\sqrt{2}) + (\sqrt{2}-1)f(2-\sqrt{2})\right] \quad , \tag{6.35}$$

which is exact if f(x) is a cubic polynomial.

6.8 Orthogonal Polynomials

Before presenting the general Gaussian Quadrature algorithm, we have to introduce the **orthogonal polynomials**. A set of polynomials $\phi_n(x)$ with degrees

Polynomial	Interval $[a,b]$	w(x)	Normalization
Legendre $p_n(x)$	$[-1,1]$	1	2/(2n+1)
Chebyshev $T_n(x)$	$[-1,1]$	$1/\sqrt{1-x^2}$	$\pi/2$ n>0
Laguerre $L_n(x)$	$[0,\infty]$	e^{-x}	1
Hermite $H_n(x)$	$[-\infty,\infty]$	e^{-x^2}	$\sqrt{\pi}2^n n!$

$n = 0, 1, 2, ..$ is said to be *orthogonal* with respect to a weighting function $w(x)$ over an interval $[a,b]$ if

$$\int_a^b w(x)\phi_m(x)\phi_n(x)dx = 0 \quad , m \neq n \quad . \tag{6.36}$$

The polynomials listed in Tab. 6.8 are among the most used and were introduced originally as solutions of certain differential equations. The normalization N is defined as $N = \int_a^b w(x)\phi_n^2 dx$.
Considering the Legendre polynomials, the first are

$$\begin{aligned} P_1(x) &= x \quad , \\ P_2(x) &= \frac{1}{2}(3x^2-1) \quad , \\ P_3(x) &= \frac{1}{2}(5x^3-3x) \quad , \\ P_4(x) &= ... \quad . \end{aligned}$$

If one tries to calculate the roots of P_1 and P_2, we discover that they are exactly the nodes we obtained in Sec. 6.6. Indeed, the Legendre polynomials are orthogonal in $[-1,1]$, which is the interval we considered for the integration. This observation points towards a connection between integration formulas and orthogonal polynomials.

6.9 The Gaussian Quadrature Algorithm

Here we will give the general Gauss Quadrature formulas, which will be proven separately in the next section.
The solution of the integral $I = \int_a^b w(x)f(x)dx$ can be expressed as an integral where the function $f(x)$ is approximated by an orthogonal polynomial and

$$\int_a^b w(x)P_{2n+1}(x)dx = \sum_{i=0}^{n} a_i P_{2n+1}(x_1) \quad , \tag{6.37}$$

where the nodes x_i are the zeros of the polynomial $P_{2n+1}(x)$ and the weights a_i can be obtained from

$$a_i = \int_a^b w(x) C_i(x) dx \quad , i = 0, 1, 2, .., n \quad , \tag{6.38}$$

where C_i are the Lagrange's cardinal functions already introduced in Chapt. 4.
For **Legendre Polynomials**

$$a_i = \frac{2}{(1 - x_i)[P'_{n+1}(x_i)]^2} \quad . \tag{6.39}$$

For **Laguerre Polynomials**

$$a_i = \frac{1}{x_i[L'_{n+1}(x_i)]^2} \quad . \tag{6.40}$$

For **Hermite Polynomials**

$$a_i = \frac{2^{n+2}(n+1)!\sqrt{\pi}}{[H'_{n+1}(x_i)]^2} \quad . \tag{6.41}$$

Note that Gauss quadrature in the $[-1, 1]$ interval can be applied to any finite interval $[a, b]$ by a proper change of variables $dx/dy = (b - a)/2$.

6.10 Multidimensional Integration

Integration in more than one dimension is a rather broad topic and here we will present only one simple solution which leverages on the one-dimensional integration routines already analyzed. The most simple idea is to apply the one-dimensional algorithm to each coordinate, while keeping the other coordinates fixed. This can be realized in an iterative way over an n-dimensional grid of points. A clever recursive algorithm is suggested by a close analysis of multidimensional integration. Taking as example 3-dimensional integration, in order to perform the integral I, it can always be written in this form

$$I = \int dxdydz f(x, y, z) = \int_a^b \int_{y1(x)}^{y2(x)} \int_{z1(x,y)}^{z2(x,y)} f(x, y, z) dxdydz \quad . \tag{6.42}$$

The most internal integral can be seen as a function of two variables

$$F_2(x,y) = \int_{z1(x,y)}^{z2(x,y)} f(x,y,z)dz \quad , \tag{6.43}$$

while the second integral is a function of one variable

$$F_1(x) = \int_{y1(x)}^{y2(x)} F_2(x,y)dy \quad . \tag{6.44}$$

With the latter definitions, the integral is thus

$$I = \int_a^b F_1(x)dx \quad . \tag{6.45}$$

As anticipated, the derived formulas can be translated in a recursive algorithm where we integrate Eq. 6.45 with a 1-dimensional integration algorithm. The algorithm (adapted from `Numerical Recipes in C`) is showed in code 6.2 exploiting a simple 1-dimensional trapezoidal integration function.
The obvious general problem of multidimensional integration is the large amount of computation required as the dimension grows. Algorithms along the lines of the one described here can be quite precise and practical up to dimensions 3-4, but after that usually different techniques are required, as we will see in the next section.

```python
#1D Integration (Trapezoidal rule in this case)
def Trapezoid(f,a,b,n=10):
    h = (b-a)/float(n)
    x = a
    I=0
    for i in range(int(n)):
        I = I + (f(x) + f(x+h))*h/2.0
        x = x + h
    return I

#Function to integrate
def F(x,y,z): return x + y + z

#Integration limits on y
def y1(x): return 0
def y2(x): return 2

#Integration limits on z
def z1(x,y): return 0
def z2(x,y): return 2

xsav=0
ysav=0

def Integrate3D(a,b):
    return Trapezoid(F1,a,b)

def F1(x):
    global xsav
    xsav = x
    return Trapezoid(F2,y1(x),y2(x))

def F2(y):
    global ysav
    ysav = y
    return Trapezoid(F3,z1(xsav,y),z2(xsav,y))

def F3(z): return F(xsav,ysav,z)

print("Integral = ",Integrate3D(0,2))
```

Listing 6.2: 3D Integration

6.11 Introduction to Stochastic Integration

Stochastic integration is a method for estimating integrals based on the generation of random numbers. In order to understand the method, we start with a simple example. We would like to calculate the area of the quarter of a circle with unit radius. The exact area is $C = \pi/4$ and the quarter circle is completely contained in a square of unit side.
For estimating the circle area C, we can generate N random points (x, y) with $x < 1$ and $y < 1$ and check how many points p land inside the circle, checking if $\sqrt{x^2 + y^2} < 1$. The number of generated points will be proportional to the square's area Q=1, while the number of points landing inside the circle will be proportional to the circle's area, therefore

$$\frac{Q}{C} = \frac{N}{p} \Rightarrow C = \frac{p}{N} \quad . \tag{6.46}$$

Generating random numbers we have a method for estimating C and since $\pi = 4 \cdot C$, we can even estimate the number π with the same procedure, as showed in the example code 6.3

```
from math import *
import random

def EstimatePI(N):
    p=0
    for i in range(N):
        x = random.random()
        y = random.random()
        if (sqrt(x**2+y**2)<1): p=p+1
    return p/N * 4.0

print("pi = ",EstimatePI(100000))
```

Listing 6.3: π estimation

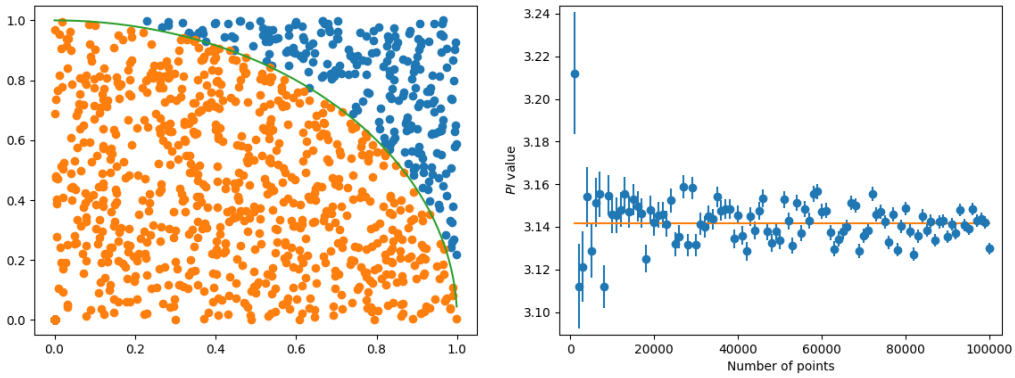

Figure 6.2: **(Left)** Generated 1000 random points in the unit square. The ratio between the square and circle area is equal to the ratio of the number of points landing in the circle (orange) and the total number of generated points. **Right:** Value of π estimated with Monte Carlo integration on a circle as a function of the number of generated points N. The error decreases as $1/\sqrt{N}$.

6.12 Monte Carlo Integration

If we have to calculate the multidimensional integral

$$I = \int_A f(\bar{x}) d\bar{x} \quad , \tag{6.47}$$

over a set A, we can estimate it considering only a fine set of points x_{rnd} in A chosen randomly. If the total integral (the "volume") of the set A is $V = \int_A d\bar{x}$, we can use the approximation

$$\tilde{I} \approx V \cdot \frac{1}{N} \sum_i f(x_i) = V \cdot \langle f \rangle \quad . \tag{6.48}$$

where $\langle f \rangle$ is the average of the function over the N random points. The last formula is the simplest form of the **Monte Carlo** algorithm and its convergence to the true value of I is guaranteed by the *law of large numbers*.
This method is very efficient for high-dimensional integrals, since we probe only a subset of points in A, while grid methods will have to calculate the function over

the whole set A. Given the stochastic nature of the method, a statistical error in the estimate is involved. This can be calculated considering the variance defined as

$$Var(f) = \sigma^2 = \frac{1}{N-1}\sum_{i=1}^{N}\left(f(x_{i,gen}) - \langle f\rangle\right)^2 \quad , \tag{6.49}$$

we can now calculate the variance of the estimator $I \approx V \cdot \langle f\rangle$

$$Var(\tilde{I}) = \frac{V^2}{N^2}\sum_{i=1}^{N} Var(f) = \frac{V^2\sigma^2}{N} \quad . \tag{6.50}$$

The square of the variance gives the root mean square error

$$Err(\tilde{I}) = \frac{V\sigma}{\sqrt{N}} \quad . \tag{6.51}$$

which shows that the error decreases as the square root of the number of generated random points (see Fig. 6.11).

6.13 Importance Sampling

The simple Monte Carlo algorithm described in the latter section can be further improved. The random points were generated according to an uniform random distribution within the integration volume. This in general is not very efficient: we could sample more points where the function is larger and contributes more to the integral for example. This observation leads to the idea of **importance sampling**.
In general, the expectation value E of a variable or a function is defined as

$$E[f] = \int_A \rho(\bar{x}) f(\bar{x}) d\bar{x} \quad , \tag{6.52}$$

with respect to a probability distribution ρ. If we generate N random numbers according to ρ, the last integral can be approximated by $(1/N)\sum_i f(x_i)$. This observation suggests the importance sampling Monte Carlo method. If we have to integrate a function f over a set A, we can perform the following manipulation

$$\int_A f(x)dx = \int_A f(x)\frac{\rho(x)}{\rho(x)}dx = \int_A \frac{f(x)}{\rho(x)}\rho(x)dx \approx \frac{1}{N}\sum_{i=1}^{N}\frac{f(x_i)}{\rho(x_i)} \quad , \tag{6.53}$$

where the random numbers x_i are sampled from the distribution ρ.
The advantage of the method is in the reduction of the variance if ρ is very close to f.

6.14 Gaussian Random Numbers: Box-Muller Transformation

In many applications, it is relevant to generate random numbers with a Gaussian distribution. The Box-Muller algorithm (BM) is one of the available methods for achieving this. BM considers two Gaussian distributions in the two variables X and Y

$$P(X) = \frac{1}{\sqrt{2\pi}} e^{-\frac{X^2}{2}} \quad ,$$

$$P(Y) = \frac{1}{\sqrt{2\pi}} e^{-\frac{Y^2}{2}} \quad .$$

The product of the two latter distributions in spherical coordinates $X = R\cos\theta$ and $Y = R\sin\theta$ is

$$P(X,Y) = P(X)\cdot P(Y) = \frac{1}{2\pi} e^{\frac{-(X^2+Y^2)}{2}} \longrightarrow P(R,\theta) = P(R) = \frac{1}{2\pi} e^{\frac{-R^2}{2}} \quad . \tag{6.54}$$

If we now generate two *uniformly distributed* random numbers x and y, we can write

$$x = \frac{\theta}{2\pi} \Rightarrow \theta = 2\pi x \quad , \tag{6.55}$$

because the product distribution in the new coordinates does not depend on the "angle" θ and

$$y = e^{\frac{-R^2}{2}} \Rightarrow R = \sqrt{-2\ln y} \quad . \tag{6.56}$$

Having R(x,y) and θ(x,y), we can restore the original coordinates obtaining the **Box-Muller** formulas

$$cccX = R\cos\theta = \sqrt{-\ln y}\cdot\cos(2\pi x) \quad , \tag{6.57}$$

$$Y = R\cos\theta = \sqrt{-\ln y}\cdot\sin(2\pi x) \quad . \tag{6.58}$$

Summarizing, the BM method generates two random numbers with Gaussian distribution starting from two uniformly distributed random numbers.

Chapter 7 | Numerical Differentiation

The approximation of derivatives is a central problem in numerical analysis, since it plays a key role in differential equations and therefore in many engineering and science applications. Differentiation is a limiting process and on a computer it will be always affected by round-off errors and therefore the derivative of a function will never be as precise as the computation of the function itself.
In this chapter, we will derive the most important and commonly used approximation schemes for the derivatives.

7.1 Backward and Forward Differences

Starting from the definition of derivative, a numerical approximation can be directly derived

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x+\Delta x) - f(x)}{\Delta x} \approx \frac{f(x+h) - f(x)}{h} \quad , \tag{7.1}$$

where h has to be "small".
In the definition, Δx can be positive or negative. In the positive case, the approximation is called *forward difference*, in the negative case, *backward difference* approximation.
The *grid size* h instead has to be a positive number.
Considering only *interior points* $(x_1, x_2, ..., x_{n-1})$ of a grid $x_0, x_1, ..., x_n$, a numerical approximation of the derivative must depend from two points such that we can write (choosing consecutive points)

$$f'(x_i) = af(x_i) + bf(x_{i+1}) \quad . \tag{7.2}$$

We can now Taylor-expand the second term, remembering that $x_{i+1} - x_i = h$

$$f(x_{i+1}) \approx f(x_i) + hf'(x) + O(h^2). \tag{7.3}$$

Inserting Eq. 7.3 in Eq. 7.2

$$f'(x_i) = (a+b)f(x_i) + bhf'(x_i) + O(h^2) \quad . \tag{7.4}$$

If we choose $a + b = 0$ and $bh = 1$, from which $b = 1/h$ and $a = -1/h$, we satisfy the equation up to first order. Inserting these results in Eq. 7.2 we obtain

$$f'(x_i) \approx \frac{f(x_{i+i}) - f(x_i)}{h} \quad , \tag{7.5}$$

which is a **two-point forward difference approximation** with error $E = (h/2)f''(x)$. With an analogous procedure, we could find

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{h} \quad , \tag{7.6}$$

the **two-point backward difference approximation**.

7.2 Central Difference

In the last section, we used two points for approximating a derivative: here we will consider three points

$$f'(x_i) = af(x_i) + bf(x_{i+1}) + cf(x_{i-1}) \quad . \tag{7.7}$$

Substituting the Taylor expansions for the last two terms up to third order

$$f(x_{i+1}) \approx f(x_i) + hf'(x_i) + \frac{h^2}{2}f''(x) + O(h^3)$$

$$f(x_{i-1}) \approx f(x_i) - hf'(x_i) + \frac{h^2}{2}f''(x) - O(h^3) \quad ,$$

we obtain

$$f'(x_i) = (a+b+c)f(x_i) + h(b-c)f'(x_i) + \frac{h^2}{2}(b+c)f''(x_i) + O(h^3) \quad .$$

For satisfying the last equation we choose

$$\begin{cases} a+b+c=0 \\ h(b-c)=1 \\ b+c=0 \end{cases} \Rightarrow \begin{cases} a=0 \\ b=1/(2h) \\ c=-1/(2h) \end{cases} \quad .$$

Inserting these results back in Eq.7.7 we obtain the three-point approximation

$$f'(x_i) \approx \frac{f(x_{i+1}) - f(x_{i-1})}{2h} \quad . \tag{7.8}$$

The last formula is in fact a two-point approximation, since the third point disappeared from the calculation. This is why this result is better known as **central difference approximation**.
The error is now improved with respect to forward and backward differences, since we expanded the functions up to third order

$$E = \frac{h^2}{12}\left[f^{(3)}(k_1) + f^{(3)}(k_2)\right] \quad , \tag{7.9}$$

where $x_i < k_1 < x_{i+1}$ and $x_{i-1} < k_2 < x_i$.

7.3 Second Derivative

For approximating the second derivative, we have to consider at least three points

$$f''(x_i) = af(x_i) + bf(x_{i+1}) + cf(x_{i-1}) \quad . \tag{7.10}$$

In this case, for keeping a non-zero second derivative, we have to expand $f(x_{i+1})$ and $f(x_{i-1})$ to the fourth order, obtaining

$$f''(x_i) = (a+b+c)f(x_i) + h(b-c)f'(x_i) + \frac{h^2}{2}(b+c)f''(x_i) + \frac{f^{(3)}(x_i)}{6} + \frac{h^4}{24}\left[bf^{(4)}(k_1) + cf^{(4)}(k_2)\right] \quad .$$

With the choice

$$\begin{cases} a+b+c=0 \\ b-c=1 \\ h^2(b+c)=2 \end{cases} \Rightarrow \begin{cases} a=-2/h^2 \\ b=1/h^2 \\ c=1/h^2 \end{cases} .$$

the **central difference approximation for the second derivative** becomes

$$f''(x_i) \approx \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{h^2} \quad , \tag{7.11}$$

with error

$$E = \frac{h^2}{24}\left[f^{(4)}(k_1) + f^{(4)}(k_2)\right] \quad , \tag{7.12}$$

where $x_i < k_1 < x_{i+1}$ and $x_{i-1} < k_2 < x_i$.
An alternative (perhaps simpler) derivation of the central difference formula for the second derivative consists in calculating the Taylor expansion of $f(x+h) + f(x-h)$.

7.4 Another derivation

In this section we present another procedure for deriving different numerical derivative formulas, which is based on the Taylor expansions

$$(a) \quad f(x+h) = f(x) + hf'(x) + \frac{h^2}{2}f''(x) + O(h^3)$$
$$(b) \quad f(x-h) = f(x) - hf'(x) + \frac{h^2}{2}f''(x) + O(h^3)$$
$$(c) \quad f(x+2h) = f(x) + 2hf'(x) + \frac{(2h)2}{2}f''(x) + O(h^3)$$
$$(d) \quad f(x-2h) = f(x) - 2hf'(x) + \frac{(2h)2}{2}f''(x) + O(h^3)$$

	$f(x-2h)$	$f(x-h)$	$f(x)$	$f(x+h)$	$f(x+2h)$
$2hf'(x)$	0	-1	0	1	0
$h^2f''(x)$	0	1	-2	1	0
$2h^3f'''(x)$	-1	2	0	-2	1
$h^4f^{(4)}(x)$	1	-4	6	-4	1

Table 7.1: Coefficients of the numerical derivatives up to fourth order.

and their combinations

$$(e) \quad f(x+h)+f(x-h) = 2f(x)+h^2f''(x)+\frac{h^4}{12}+O(h^6)$$

$$(f) \quad f(x+h)-f(x-h) = 2hf'(x)+\frac{h^3}{3}f'''(x)+O(h^5)$$

$$(g) \quad f(x+2h)+f(x-2h) = 2f(x)+4h^2f''(x)+\frac{4h^4}{3}f^{(4)}(x)+O(h^6)$$

$$(h) \quad f(x+2h)-f(x-2h) = 4hf'(x)+\frac{8h^3}{3}f'''(x)+O(h^5)$$

Solving equation (f) for f'(x) yields directly the central difference approximation

$$f'(x) = \frac{f(x+h)-f(x-h)}{2h} - \frac{h^2}{6}f'''(x)+O(h^5) \approx \frac{f(x+h)-f(x-h)}{2h}+O(h^2)$$

Solving equation (e) for $f''(x)$ yields the approximation of the second derivative (see previous section).
Higher order derivatives can be obtained with other combinations. For example, considering Eq. (f) and (g), and eliminating $f'(x)$, and solving for $f'''(x)$:

$$f'''(x) = \frac{f(x+2h)-4f(x+h)+6f(x)-4f(x-h)+f(x-2h)}{h^4} \quad . \tag{7.13}$$

Note how higher order derivatives require the computation of the function at more points. The central difference approximation results up to the fourth order can be summarized in the Tab. 7.1

7.5 Derivatives with Interpolation

The idea behind this class of algorithms is simple: interpolate a number of consecutive points of the discrete grid and then evaluate the analytic derivative of the interpolating function. A typical choice can be the parabolic or cubic spline interpolation. The interpolating polynomial is usually of low order, since the derivative can amplify the oscillations of the polynomial.
These methods are computationally more expensive than the ones described before but can be quite useful if the grid of points is not regular, with different spaces between points.

CHAPTER 8 | Numerical Ordinary Differential Equations

8.1 Introduction to Initial Value Problems

In science and egineering, the solution of differential equations is a very relevant problem. In particular, many equations describing for example mechanical systems, are of the form

$$y'' = f(x, y, y') \quad , \tag{8.1}$$

where in concrete applications the variable x represents the time. The general problem consists in finding a continuous function which solves Eq. 8.1 for a certain $x \geq a$ and *initial conditions* $y(a) = y_0$ and $y'(a) = y'_0$. Without loss of generality, it is often assumed $a = 0$, which is the "time-zero" point and y_0 and y'_0 can be viewed in many physics applications as the initial position and initial velocity.
This **initial value problem** involving a second order differential equation can be transformed in a first order differential equation with the transformation $y' = v$ such that

$$\begin{cases} y'' = f(x, y, y') \\ f(0) = y_0 \\ f'(0) = y'_0 \end{cases} \Rightarrow \begin{cases} v' = f(x, y, v) \\ y(0) = y_0 \\ v(0) = v_0 \end{cases} \quad .$$

From a numerical point of view, no single version of the problem gives significant advantages, although in certain cases it could be useful to avoid the calculation of second derivatives.

8.2 Euler's Method

This very simple method solves the first-order initial value problem

$$\begin{cases} y' = f(x,y) \\ y(0) = y_0 \end{cases}$$

directly using the forward difference approximation for the derivative

$$\frac{y_{i+1} - y_i}{h} = f(x_i, y_i) \quad \Rightarrow \quad y_{i+1} = y_i + hf(x_i, y_i) \quad . \tag{8.2}$$

The obtained iterative form can be directly implemented in a loop starting from a value y_0 and evolving the solution until the desired "time".
The convergence of Euler's method is described by the following

Theorem 8. *If Y(x) is an exact solution of the initial value problem 8.2, Y is continuous in $[0, L]$, and $\partial f / \partial y$ exists and is bounded and continuous in $[0, L]$, then there are constants $M > 0$, $N > 0$ independent from the grid size h such that*

$$|e_i| \leq \frac{(e^{LM} - 1)Nh}{2M} \quad i = 1, 2, .., n \quad , \tag{8.3}$$

and

$$\lim_{h \to 0} |e_i| = 0 \quad , \tag{8.4}$$

where $e_i = Y(x_i) - y_i$ is the error of the numerical approximation at each grid point.

The theorem is useful in proving the convergence and its speed, but it does not account for rounding errors which might accumulate along the solution.
We can assume that there is a number R equal to the maximum absolute rounding error. For example, if we have to round y_i to the fifth digit, $R = 5 \cdot 10^{-6}$. In the presence of a rounding error bounded by R, we have the following relevant result

$$|e_i| \leq e^{LM} \left[|r_0| + \frac{1}{M} \left(\frac{Nh}{2} + \frac{R}{h} \right) \right] \quad , \tag{8.5}$$

where r_0 is the rounding error at the beginning of the algorithm on the initial condition $y_0 \to y_0 + r_0$.

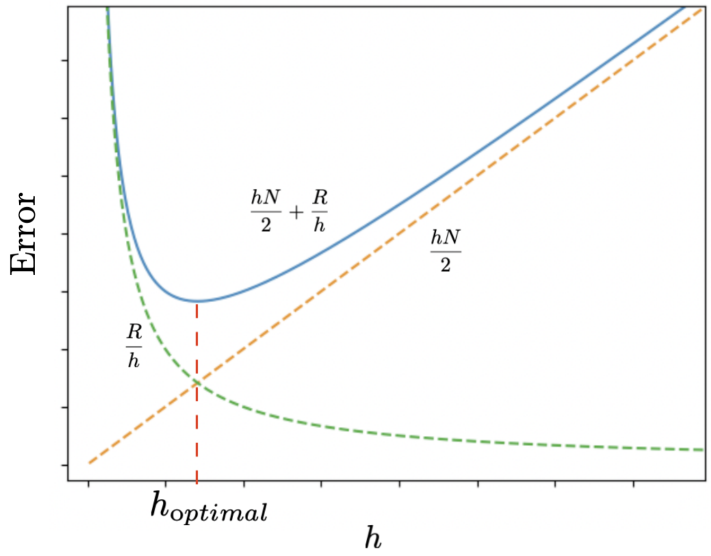

Figure 8.1: Dependence of the error in the presence of round-off as a function of the grid spacing h. The value $h_{optimal}$ is the best trade-off between discretization and round-off errors.

8.3 Runge-Kutta Method: RK2

This method includes Euler's method as special case, but in general it is designed for enhancing the precision of the solutions. The basic idea is to exploit the knowledge of the function f at any point x_i of the grid. To do this, consider the *ansatz*

$$\frac{y_{i+1} - y_i}{h} = ef(x_i, y_i) + bf(x_i + ch, y_i + dh) \quad , \tag{8.6}$$

where b, c, d, e are to be determined (e=1,b=0 corresponds to Euler's method). Considering the Taylor expansion for a function of two variables to second order, the last term in the last equation becomes

$$f(x_i + ch, y_i + dh) = f(x_i, y_i) + \left[chf_x + dhf_y\right] + \frac{1}{2}\left[c^2h^2f_{xx} + 2cdh^2f_{xy} + d^2h^2f_{yy}\right] \quad , \tag{8.7}$$

with the notation $f_x = \partial f/\partial x$, $f_{xy} = \partial^2 f/(\partial x \partial y)$ and so on. Substituting Eq. 8.7 in Eq. 8.6 and rearranging the terms yields

$$y_{i+1} = y_i + h(e+b)f(x_i, y_i) + \frac{h^2}{2}\left[2bcf_x(x_i, y_y) + 2bdf_y(x_i, y_i)\right] + \frac{h^3}{6}\left[3bc^2 f_{xx}(x_i, y_i) + 6bcdf_{xy}(x_i, y_i) + 3bd^2 f_{yy}(x_i, y_i)\right] + \mathcal{O}(h^4) \quad . \tag{8.8}$$

The reason why in the last equation we regrouped the coefficients (1/2) and (1/6) is that now we have the form

$$y_{i+1} = y_i + hF_1 + \frac{h^2}{2}F_2 + \frac{h^3}{6}F_3 + ... \quad , \tag{8.9}$$

which looks like a Taylor expansion of a function of one variable.
Suppose now that Y(x) is the *true* solution of the initial value problem and we expand it again in a Taylor series on our grid of spacing h

$$Y_{i+1} = Y_i + hY_i' + \frac{1}{2}h^2 Y_i'' + \frac{h^3}{6}Y^{(3)} + ... \quad . \tag{8.10}$$

Y satisfies the original differential equation $Y' = f(x, Y)$ and differentiating again using the chain rule

$$Y'' = \frac{\partial f}{\partial x} + \frac{\partial f}{\partial y}\frac{\partial y}{\partial x} = f_x + f_y f$$

$$Y^{(3)} = \frac{d}{dx}\left[f_x + f_y f\right] = f_{xx} + 2ff_{xy} + f^2 f_{yy} + f_x f_y + f f_y^2 \quad ,$$

and substituting in Eq. 8.10 we obtain

$$Y_{i+1} = Y_i + hf(x_i, Y_i) + \frac{h^2}{2}\left[f_x(x_i, Y_y) + f_y(x_i, Y_i)f(x_i, Y_i)\right] + \frac{h^3}{6}\left[f_{xx}(x_i, Y_i) + 2f(x_i, Y_i)f_{xy}(x_i, Y_i) + f_{yy}(x_i, y_i)f^2(x, Y_i) + f_x(x_i, Y_i)f_y(x_i, Y_i) + f(x_i, Y_i)f_y^2(x_i, Y_i)\right] + \mathcal{O}(h^4) \quad . \tag{8.11}$$

After these calculations we ended up with

- Eq. 8.8: An expansion of the *numerical* solution
- Eq. 8.11: An expansion of the *exact* solution

We have now to choose the coefficients b, c, d, e such that the two expansions agree up to order h^2 (agreement to order h would yield Euler's formula):

$$\begin{array}{l}(A)\\(B)\\(C)\\(D)\end{array}\left\{\begin{array}{l} b+e=1 \\ 2bcf_x + 2bdf_y = f_x + ff_y \\ 2bc = 1 \\ 2bd = 1 \end{array}\right. \Rightarrow \left\{\begin{array}{l} c = \frac{1}{2} \\ d = f(x_i, y_i) \end{array}\right. \quad . \qquad .$$

In the previous system (left) of requirements, equations (A) and (B) are needed for matching the Taylor series, but they are not sufficient to determine four parameters. Without loss of generality, we can make the additional choices (C) and (D).
Using these values in the initial *ansatz* Eq. 8.6 we obtain the **Runge-Kutta formula**

$$y_{i+1} = y_i + \frac{h}{2} f(x_i, y_i) + \frac{h}{2} f(x_{i+1}, y_i + hf(x_i, y_i)) \quad . \qquad (8.12)$$

Noting that the term $K_0 = hf(x_i, y_i)$ appears twice in the formula, defining $K_1 = hf(x_{i+1}, y_i + K_0)$ we can describe a 3-step algorithm for the Runge-Kutta approximation

1. $K_0 = hf(x_i, y_i)$,
2. $K_1 = hf(x_{i+1}, y_i + K_0)$,
3. $y_{i+1} = y_i + \frac{1}{2}(K_0 + K_1)$.

8.4 Higher-order Runge Kutta Methods: RK4

The method described in the previous section can be generalized to an *ansatz*

$$y_{i+1} = y_i + h\left[a_1 f(p_1, g_1) + a_2 f(p_2, g_2) + ...a_k f(p_k, g_k)\right] \quad , \tag{8.13}$$

where $x_i \leq p_i \leq x_{i+1}$ and g are the functions calculated at certain shifted points on the grid.
The most widely used approximation scheme for the initial value problem because if its computational simplicity and accuracy, is the **fourth order Runge-Kutta method**, derived by W. Kutta in 1901.

1. $K_0 = hf(x_i, y_i)$,
2. $K_1 = hf(x_i + \frac{h}{2}, y_i + \frac{K_0}{2})$,
3. $K_2 = hf(x_i + \frac{h}{2}, y_i + \frac{K_1}{2})$,
4. $K_3 = hf(x_{i+1}, y_i + K_2)$
5. $y_{i+1} = y_i + \frac{1}{6}(K_0 + 2K_1 + 2K_2 + K_3)$.

This scheme agrees with the Taylor expansion up to $\mathcal{O}(h^4)$ terms and it is one of the possible approximations given the choice of the free parameters available, as done in the previous section.

8.5 Fouth-Order Runge-Kutta Method in two Dimensions

If we are dealing with the two-dimensional initial value problem

$$\begin{cases} y' = F(x, y, v) \\ v' = G(x, y, v) \end{cases} \quad , \quad \begin{cases} y(0) = a \\ v(0) = b \end{cases} \quad ,$$

also in this case a fourth-order Runge-Kutta approximation scheme can be derived. The calculations are involved although along the same lines of Section 8.3 and

here we present only the result, which can be understood by analogy to the one-dimensional case:

$$
\begin{array}{ll}
K_0 = hF(x_i, y_i, v_i) & M_0 = hG(x_i, y_i, v_i) \\
K_1 = hF(x_i + \frac{h}{2}, y_i + \frac{K_0}{2}, v_i + \frac{M_0}{2}) & M_1 = hG(x_i + \frac{h}{2}, y_i + \frac{K_0}{2}, v_i + \frac{M_0}{2}) \\
K_2 = hF(x_i + \frac{h}{2}, y_i + \frac{K_1}{2}, v_i + \frac{M_1}{2}) & M_2 = hG(x_i + \frac{h}{2}, y_i + \frac{K_1}{2}, v_i + \frac{M_1}{2}) \\
K_3 = hF(x_{x+1}, y_i + K_2, v_i + M_2) & M_3 = hG(x_{x+1}, y_i + K_2, v_i + M_2)
\end{array}
$$

$$
\begin{aligned}
y_{i+1} &= y_i + \tfrac{1}{6})(K_0 + 2K_1 + 2K_2 + K_3) \\
v_{i+1} &= v_i + \tfrac{1}{6}(M_0 + 2M_1 + 2M_2 + M_3)
\end{aligned}
\tag{8.14}
$$

8.6 Fourth-Order Runge-Kutta Method for Second-order Differential Equations

If the initial value problem involves a second-order differential equation $y'' = f(x, y)$, the problem can be turned in a system of two first-order equations as in Eq. 8.1 definig $y' = v$. At this point, we can take advantage of the result of the last section with the identification $F(x, y, v) = v$ and $G(x, y, v) = f$, from which we have

$$K_0 = hv_i K_1 = h(v_i + M_0/2)K_2 = h(v_i + M_1/2)K_3 = h(v_1 + M_2) \quad .$$

We can now substitute the K_i functions in the M_i obtaining the algorithm

$$
\begin{aligned}
M_0 &= hf(x_i, y_i, v_i) \\
M_1 &= hf(x_i + \tfrac{h}{2}, y_i + h\tfrac{v_i}{2}, v_i + \tfrac{M_0}{2}) \\
M_2 &= hf(x_i + \tfrac{h}{2}, y_i + h\tfrac{v_i}{2} + h\tfrac{M_0}{4}, v_i + \tfrac{M_1}{2}) \\
M_3 &= hf(x_{x+1}, y_i + hv_i + h\tfrac{M_i}{2}, v_i + M_2)
\end{aligned}
\qquad (8.15)
$$

$$
\begin{aligned}
y_{i+1} &= y_i + hv_i + \tfrac{1}{6})(M_0 + M_1 + M_2) \\
v_{i+1} &= v_i + \tfrac{1}{6}(M_0 + 2M_1 + 2M_2 + M_3)
\end{aligned}
$$

8.7 Taylor Expansion Methods

This class of methods is based on the direct Taylor expansion of f and therefore they require the analytic calculation of the derivative. For example, in the second-order initial value problem $y'' = f(x, y, y')$, we need to calculate the function and its first derivative and we can do it by direct Taylor expansion

$$y_{i+1} = y_i + hy'_i + \frac{1}{2}h^2 y''_i + ... \qquad (8.16)$$

$$y'_{i+1} = y'_i + hy''_i + \frac{1}{2}h^2 y^{(3)}_i + ... \qquad (8.17)$$

The interesting observation is that the two equations above share most of the coefficients $y, y', ..$, and therefore these have to be calculated only once.

8.8 Stability Analysis

Integration methods for differential equations can accumulate an error at each iteration and it should be made sure that this will not happen during the computation. Unfortunately this task is not easy to solve in general and often the program itself has to contain checks for helping pin down such situations. When the error amplifies during the computation, the algorithm is **unstable**. In order to see how an instability might arise, we can analyze a simple case which illustrates the problem.

Let's consider the initial value problem

$$\begin{aligned} y' &= -\lambda y \\ y(0) &= a \quad , \end{aligned}$$

with exact solution $y(x) = ae^{-\lambda x}$.
Applying Euler's method $y(x+h) = y(x) + hy'(x)$ we have

$$y(x+h) = (1-\lambda h)y(x) \quad . \tag{8.18}$$

If $|1-\lambda h| > 1$, the method is unstable, since $|y|$ will simply increase at each iteration. Therefore, the stability condition is

$$|1-\lambda| \leq 1 \Rightarrow h \leq \frac{2}{\lambda} \quad . \tag{8.19}$$

In the case of a system of differential equations $\bar{y}' = -A\vec{y}$ where A is a constant matrix with positive eigenvalues, what drives the stability is the *maximum eigenvalue* of A λ_{max} and therefore $h \leq 2/\lambda_{max}$.
Given the simple results obtained, it seems that the step h has to be small enough and therefore, a general rule of thumb if the algorithm displays instability, is to reduce the value of h (with compatibility of rounding errors).
Another characteristics of linear systems of equations is **stiffness**: there is stiffness, if one component of the vector $\bar{y}$ varies much faster with a variation of x with respect to the other components. The treatment of stiffness is complex and it will not be presented here, but again an analysis of the eigenvalues of A can reveal stiffness problems, for example if one eigenvalue is much larger than all the others.

8.9 Adaptive-Mesh Methods

In the previous section, we have seen that the grid, or mesh width must be appropriately chosen. In general, if the mesh is too coarse, we can have large truncation errors, while a very fine mesh needs more computation. The optimal situation would be to use a coarse mesh when the solution does not change rapidly, and a finer one in the other case.
The main idea of this class of methods is to compare an algorithm at two consecutive precision levels, for example fourth-order and fifth-order Runge-Kutta formulas and then compare their difference as estimate of the error we are making with a certain step h

$$E_i(h) = y_{RK5}(x_i+h) - y_{RK4}(x_i+h) \quad . \tag{8.20}$$

We can now consider as measure of the error over an interval I $e(h) = \max_I E_i(h)$ or the root mean square of the step errors E_i.
Since the truncation error of the fourth-order Runge-Kutta method is $\mathcal{O}(h^5)$, if we compare two errors for two choices of steps h_1 and h_2 we have

$$\frac{e(h_1)}{e(h_2)} \approx \frac{h_1^5}{h_2^5} \quad , \tag{8.21}$$

and if our tolerance with the choice h_2 is ϵ, then $e(h_2) = \epsilon$ and we have to choose

$$h_2 = h_1 \left(\frac{\epsilon}{e(h_1)} \right)^{\frac{1}{5}} \quad . \tag{8.22}$$

After the computation of the last estimate, the algorithm has two choices, depending on the result:

1. $h_2 \geq h_1$: The previous step width h_1 was already good since the error was below tolerance and we can keep the corresponding result.

2. $h_2 < h_1$: we discard the current result corresponding to a step h_1 and repeat it with step h_2.

It is common to use a weaker form of Eq. 8.22

$$h_2 = \alpha h_1 \left(\frac{\epsilon}{e(h_1)} \right)^{\frac{1}{5}} \quad , \tag{8.23}$$

with $\alpha < 1$ for being less aggressive in the decision to repeat the step and save computational time.

8.10 Application: Predator-Prey Model

One of the first ecological mathematical models is the Lotka-Volterra model, or predator-prey model [1]. The model describes two competing populations X and Y described by the number of their individuals x and y, respectively. Populatinon X just naturally grows and dies because it is a "prey" (e.g. gazelles) of the "predator" (lions) population Y, which in turns multiply eating elements of X and dies if X is

[1] Alfred James Lotka (1880-1949), Vito Volterra (1860-1940)

not enough. The situation can be modeled with a system of non-linear differential equations:

$$\begin{cases} \dot{x} = \frac{dx}{dt} = ax - bxy \quad , \\ \dot{y} = \frac{dy}{dt} = -cy + dxy \quad , \end{cases} \tag{8.24}$$

with positive parameters a, b, c, d. The first equation describes the change in time of the population X, which growth is described by the parameter a, while it decreases with the interaction with population Y, as described by the non-linear term bxy. Population Y naturally decreases $(-cy)$, while it increases with the interaction term (+dxy, or "lions eat gazelles").
Setting $\dot{x} = \dot{y} = 0$ we can find the two equilibria of the system (where the populations do not change)

$$\begin{cases} x = 0 \\ y = 0 \end{cases} \qquad \begin{cases} x = \frac{a}{b} \\ y = \frac{c}{d} \end{cases} \tag{8.25}$$

From a stability analysis, it turns out that the $x = y = 0$ "extintion" solution is a saddle point and thus unstable. The other solution corresponds to a stable state where both populations oscillate but never reduce to zero.
We can investigate the dynamics of this system numerically integrating the differential equations. In Fig. 8.2 the RK4 method was used for solving the model and thetypical oscillatory dynamics of the two populations is showed. Another point of view for the dynamics of the system is the phase space representation, where the x and y values are reported. The shape of the phase space can be derived in this case dividing formally the two equations obtaining

$$\frac{dx}{dy} = \frac{x(a - by)}{y(-c + d \cdot x)} \Rightarrow \frac{-c + d \cdot x}{x} dx = \frac{a - by}{y} dy \quad . \tag{8.26}$$

Integrating the two sides of the last equation (and setting the arbitrary constant to zero) we obtain an explicit equation for the "orbits" in the phase space

$$d \cdot x + by - c \ln x - a \ln y = 0 \quad . \tag{8.27}$$

This equation describes closed curves in the phase space followed by the two populations. Since these curves are invariant (do not depend on time) the function $H(x, y) = d \cdot x + by - c \ln x - a \ln y$ can be interpreted as the "energy", or the

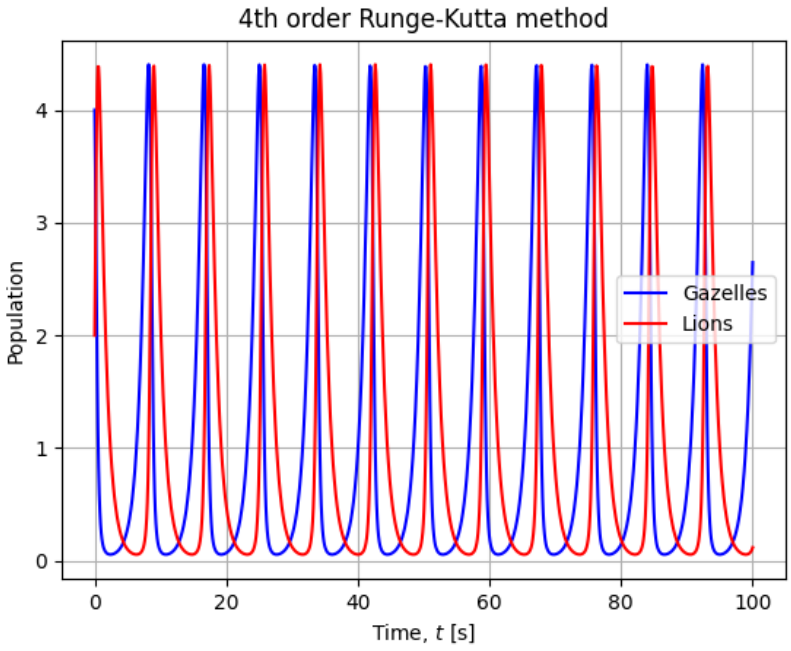

Figure 8.2: Integration of the Lotka-Volterra model for $a = b = c = d = 1$ with the RK4 algorithm.

Hamiltonian of the system and it is straightforward to verify that the Poisson brackets

$$\dot{x} = \{x, H\} \tag{8.28}$$
$$\dot{y} = \{y, H\} \tag{8.29}$$

reproduce the original Lotka-Volterra equations. It is also possible to derive the original equations using the canonical hamiltonian equations but first we have to recover canonical variables, since x and y are not (i.e. $\{x, y\} \neq 1$). This can be done introducing the caonlically conjugated variables $p = \ln x$ and $q = \ln y$. Using now H=H(p,q), the canonical equations $\dot{q} = \partial H/\partial q$ and $\dot{p} = -\partial H/\partial p$ can be applied.
It is interesting to investigate the difference between integration schemes with different precision. To this aim, we integrate the Lotka-Volterra system with the RK2 and RK4 algorithms and check their difference at large integration times. The results are showed in Fig. 8.3. The left panes shows the relative percentage difference

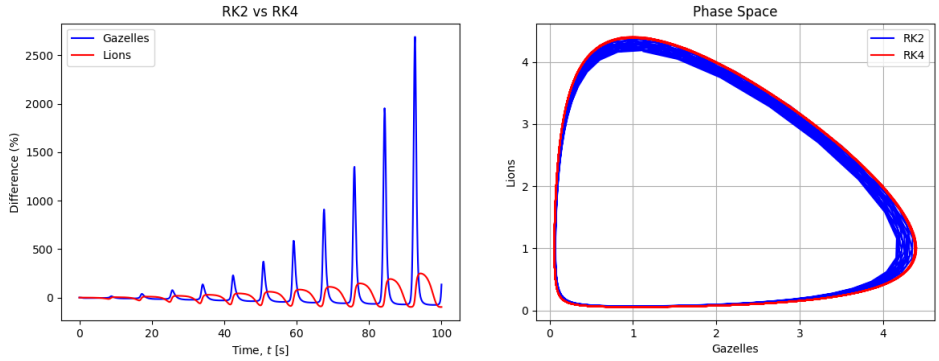

Figure 8.3: **Left:** Percentage difference between the RK2 and RK3 solution for the two populations. **Right:** Phase space (X vs Y) of the two integration methods.

between the solutions of RK2 and RK4 as function of time and we can note that the difference tends to grow. In the right panel the same effect is demonstrated with the phase space portrait (X vs Y): while the RK4 solution remains along a closed line, the RK2 solution changes its orbit over long times. The code used for generating the figures is reported in Code 8.1.

```python
import numpy as np

def LotkaVolterra(X, t, a, b, c, d):
    x, y = X
    dotx =  a*x - b*x*y
    doty = -c*y + d*x*y
    return np.array([dotx, doty])

#2th order Runge-Kutta method
def RK2(f, X0, t, a,  b, c, d):
    dt = t[1] - t[0] ; N = len(t)
    X  = np.zeros([N, len(X0)]) ; X[0] = X0
    for i in range(N-1):
        K0 = f(X[i], t[i],a,b,c,d)
        K1 = f(X[i]+dt*K0,t[i]+dt,a,b,c,d)
        X[i+1] = X[i] + dt/2.0 * (K0 + K1)
    return X

#4th order Runge-Kutta method
def RK4(f, X0, t, a,  b, c, d):
    dt = t[1] - t[0] ; N = len(t)
    X  = np.zeros([N, len(X0)]) ; X[0] = X0
    for i in range(N-1):
        K0 = f(X[i], t[i], a,  b, c, d)
        K1 = f(X[i]+dt/2.0* K0,t[i]+dt/2.0, a,b,c,d)
        K2 = f(X[i]+dt/2.0* K1,t[i]+dt/2.0, a,b,c,d)
        K3 = f(X[i]+dt    * K2,t[i]+dt, a,  b, c,d)
        X[i+1] = X[i]+dt/6.0*(K0+2.0*K1+2.0*K2+K3)
    return X

#Model Parameters
a = 1.0 ; b = 1.0 ; c = 1.0 ; d = 1.0
x0 = 4. ; y0 = 2. ; N = 1000 ; tmax = 100.
t = np.linspace(0.,tmax, N) ; X0 = [x0, y0]
Xrk2 = RK2(LotkaVolterra, X0, t, a,  b, c, d)
Xrk4 = RK4(LotkaVolterra, X0, t, a,  b, c, d)
```

Listing 8.1: RK2 and RK4 for the Predator-Prey Model

8.11 Boundary Value Problems

After the treatment of initial value problems, we consider now the following

$$\begin{cases} y'' = f(x, y, y') \\ y(a) = \alpha \\ y(b) = \beta \end{cases} \tag{8.30}$$

boundary value problem, where the two conditions are not anymore the "initial time" values for the function and its derivative, but two known values of the function at the *boundary* of an interval for x. The conditions $y(a) = \alpha$ and $y(b) = \beta$ are called *boundary conditions* and physically they can be thought with a mechanical analogy as *initial and final positions*.

8.12 Central Difference Method for Boundary Value Problems

The generic second-order differential equation can be written as

$$y'' + P(x)y' + Q(x)y = R(x) \quad , \tag{8.31}$$

where P, Q, and R are continuous functions of x in an interval $[a, b]$ where we know the boudary values $y(a) = \alpha$ and $y(b) = \beta$. Such an equation rarely can be solved analitically, and a numerical solution is required. A *sufficient* (but not necessary) condition for an unique solution is $Q(x) \leq 0$ over $[a, b]$.
We can use directly the central difference approximations of the first and second derivatives

$$\frac{y_{i+1} - 2y_i + y_{i-1}}{h^2} + P(x_i)\frac{y_{i+1} - y_{i-1}}{h} + Q(x_i)y_i = R(x_i) \quad , \tag{8.32}$$

and rearranging the terms

$$\underbrace{[2 - hP(x_i)]}_{A_i} y_{i-1} + \underbrace{\left[-4 + 2h^2 Q(x_i)\right]}_{B_i} y_i + \underbrace{[2 + hP(x_i)]}_{C_i} y_{i+1} = \underbrace{2h^2 R(x_i)}_{D_i} \tag{8.33}$$

for $i = 1, 2, .., n - 1$. If we consider also the known values at $i = 0$ and $i = n$ and see the index i as the column-index of a matrix which is *tridiagonal*. Rewriting the

previous equation as $Ay_{i-1} + By_i + Cy_{i+1} = 0$, the system of equations to solve is

$$\begin{pmatrix} A_1 & B_1 & C_1 & 0 & 0 & 0 & ... \\ 0 & A_2 & B_2 & C_2 & 0 & 0 & ... \\ 0 & 0 & A_3 & B_3 & C_3 & 0 & ... \\ 0 & 0 & 0 & A_4 & B_4 & C_4 & ... \\ .. & .. & .. & .. & .. & .. & ... \end{pmatrix} \cdot \begin{pmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ ... \end{pmatrix} = \begin{pmatrix} D_1 \\ D_2 \\ D_3 \\ D_4 \\ ... \end{pmatrix} \quad , \tag{8.34}$$

which is fast to solve for example with the LU decomposition algorithm.
An useful theorem about the convergence of the method is the following:

Theorem 9. *Assume P, Q, R are continuous on an interval $I = [a, b]$, $Q(x) \leq 0$ on I, and M is a constant such that $|P(x)| \leq M$ on I.*
If $Mh = M(b-a)/N < 2$, then the solution given by the central value approximation exists and it is unique.

The result of the last theorem implies that the number of points N in which we divide the interval is subject to the condition

$$N > M\frac{(b-a)}{2} \quad , \tag{8.35}$$

and although the central difference scheme is more precise than a forward/backward scheme, if M is large, the number of equations might grow to the point to be a too large numerical problem to treat on a computer efficiently.

8.13 Upwind Difference Method

In order to overcome the limitation given by the previous theorem, we develop a new scheme which tries always to be consistent with Theorem 3 for a tridiagonal system, in order to ensure the existence of a solution.
Since Theorem 3 requires negative diagonal elements and positive off-diagonal elements, we will choose for each equation either backward or forward approximations for the first derivatives for enhancing these characteristics. The **upwind difference method** is described by the following equations

$$\frac{y_{i+1}-2y_i+y_{i+1}}{h^2}+P_i\frac{y_{i+1}-y_i}{h}+Q_iy_i=R_i \quad , \quad P_i \geq 0 \quad ,$$
$$\frac{y_{i+1}-2y_i+y_{i+1}}{h^2}+P_i\frac{y_i-y_{i-1}}{h}+Q_iy_i=R_i \quad , \quad P_i \leq 0 \quad .$$

which can be rewritten without an `if` condition as

$$\frac{y_{i+1}-2y_i+y_{i+1}}{h^2}+$$
$$\frac{(|P_i|+P_i)y_{i+1}-2|P_i|y_i+(|P_i|-P_i)y_{i-1}}{h}+Q_iy_i=R_i \quad ,$$

for $i = 1, 2, ..., n-1$ and $P_i = P(x_i)$ (and analogously for Q_i and R_i).
Adding to the results of this procedure the known boundary values, it yields the solution. The procedure does not have restrictions on h like the central difference, although it is less precise.
The following theorem is valid for the upwind difference scheme:

Theorem 10. *If P, Q, and R are continuous and $Q \leq 0$ in the interval $[a, b]$, the upwind difference method has a solution which is also unique.*

8.14 Leapfrog method for second order differential equations

In many applications, equations of motion based on Newton's law have to be solved. The general form of such equations is

$$\ddot{x} = F(x) \quad , \tag{8.36}$$

where $\ddot{x} = d^2x/dt^2$ and F are physically interpreted as the acceleration and the position-dependent force (divided by the mass), respectively. We also define the

"velocity" $v = dx/dt$.
A convenient method for the solution of Eq. 8.36 is the Leapfrog method, which has the following advantages:

- it is a second-order method
- it obeys time-reversal.

The idea of the method is to evaluate the space evolution on a grid with time spacings $h, 2h, 3h, ...$ and the velocity evolution on a "staggered" grid $h/2, 3/2h, 5/2h, ...$
Considering the finite difference approximation for the velocity:

$$v_{i+1/2} = \frac{x_{i+1} - x_i}{h} \quad . \tag{8.37}$$

Since we would like to evaluate the function F (the "force") at the space point x, we can discretize the second derivative using the velocity

$$F(x_{i+1}) = \frac{v_{i+3/2} - v_{i+1/2}}{h} \quad . \tag{8.38}$$

The last two equations lead to the **leapfrog** integration scheme

$$\begin{aligned} x_{i+1} &= x_i + hv_{i+1/2} \\ v_{i+3/2} &= v_{i+1/2} + hF(x_{i+1}) \end{aligned} \tag{8.39}$$

The starting position and velocity x_0 and v_0 must be provided, and $v_{1/2} = v_0 + F(x_0)h/2$. The last system of equations can be conveniently re-written without half-indices in the following way. In the first equation for x_{n+1}, we can eliminate $v_{n+1/2}$ with the derivative

$$F(x_i) = \frac{v_{i+1/2} - v_i}{h/2} \Rightarrow v_{i+1/2} = v_i + F(x_i)/2 \quad . \tag{8.40}$$

Shifting the second equation back by 1/2:

$$v_{i+1} = v_i + hF(x_{i+1/2}) = v_i + h\frac{F(x_i) - F(x_{i+1})}{2} \quad . \tag{8.41}$$

where we replaced $F(x_{i+1/2})$ with its average over the two neighboring points. Substituting Eq. 8.40 and 8.41 in the equations 8.39:

$$\begin{aligned} x_{i+1} &= x_i + hv_i + \tfrac{h^2}{2}F(x_i) \\ v_{i+1} &= v_i + \tfrac{h}{2}(F(x_{i+1}) + F(x_i)) \end{aligned} \tag{8.42}$$

It looks like the method requires two evaluations of F, but actually storing the force value at the previous step allows a single evaluation/step.

8.15 Leapfrog Method: Application to the Damped Oscillator

The Leapfrog method can be applied quite straightforwardly to many equations of motion. Here we present a slightly more difficult case: the damped harmonic oscillator described by the equation

$$\ddot{x} + \gamma\dot{x} + \omega_0^2 x = \frac{f}{m} \quad , \tag{8.43}$$

where γ is a damping term, ω_0 the harmonic frequency, f a force, and m the mass. Rewriting the last equation in the form studied in the previous section

$$\ddot{x} = F(x, \ddot{x}) - \gamma\dot{x} - \omega_0^2 x + \frac{f}{m} \quad , \tag{8.44}$$

with the key difference that now F is not only a function of x, but also of the velocity $\dot{x} = v$. The first Leapfrog equation in 8.42 does not present problems, while the force is evaluated at x_i. The second equation requires $F(x_{i+1}, v_{i+1})$ and therefore it cannot be calculated directly. What we can do is to calculate the sum of forces in the second leapfrog equation

$$v_{i+1} = v_i + \frac{h}{2}\left[\frac{f_i + f_{i+1}}{m} - \omega_0^2(x_i + x_{i+1}) - \gamma(v_i + v_{i+1})\right] \quad . \tag{8.45}$$

Rearranging the terms and collecting all the v_{i+1} factors on the left

$$v_{i+1} = v_i\left(\frac{2-\gamma h}{2+\gamma h}\right) + \frac{h}{2+h\gamma}\left[\frac{f_i + f_{i+1}}{m} - \omega_0^2(x_i + x_{i+1})\right] \quad . \tag{8.46}$$

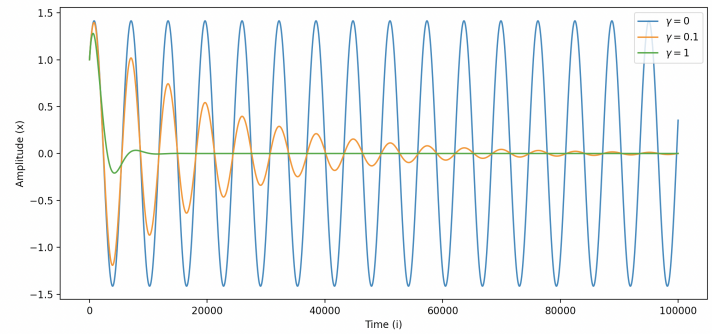

Figure 8.4: Damped harmonic oscillator for 3 values of γ (see legend) and $\omega_0^2 = 1$.

The last equation can be now used as second Leapfrog equation together with the first $x_{i+1} = x_i + hv_i + \frac{h^2}{2}F(x_i)$.
An example result with $f = 0$ (no forcing) is showed in Fig. 8.15 and the corresponding code in Code 8.2.

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

def LeapFrog(x0,v0,h,N,gamma,omega):

    x = np.zeros(N)
    v = np.zeros(N)
    x[0] = x0 ; v[0] = v0

    for i in range(0,N-1):
        F = -gamma*v[i] - omega*x[i]
        x[i+1] = x[i] + v[i]*h + 0.5*F*h*h
        g1 = 2-h*gamma ; g2 = 2+h*gamma
        v[i+1] = v[i]*g1/g2 + (h/g2) * (-omega*(x[i]+x[i
    +1]))

    return x,v

N = 100000

gamma = 0.0 ; omega = 1
x,v = LeapFrog(1,1,0.001,N,gamma,omega)
```

Listing 8.2: Leapfrog Algorithm for the Damped Harmonic Oscillator

8.16 The Numerov Method

This method, precise at O(h^6)[2] is due to B.V. Numerov (1891—1941) and can solve second order differential equations without a first-order term

$$\frac{d^2y}{dx^2} = -g(x)y(x) + s(x) \quad . \tag{8.47}$$

In the following, we will use the notation $f(x_i) = f_i$ for the various functions defined on a discrete set of points. Considering an equally-spaced set of points x_i

[2]More precisely, the method should be $O(h^5)$, since the error grows like h at each step. Usually $O(h^6)$ is quoted since functions are Taylor-expanded up to sixth order. A careful analysis reveals that Numerov's method is as precise as fourth-order Runge-Kutta.

on the x-axis with $x_i - x_{i+1} = h$, we can expand the functions y_{i+1} and y_{i-1}

$$\begin{aligned} y_{i+1} &= y_i + hy'_n + \tfrac{h^2}{2!}y''_n + \tfrac{h^3}{3!}y'''_n + \tfrac{h^4}{4!}y_n^{(iv)} + \tfrac{h^5}{5!}y_n^{(v)} + O(h^6) \\ y_{i-1} &= y_i - hy'_n + \tfrac{h^2}{2!}y''_n - \tfrac{h^3}{3!}y'''_n + \tfrac{h^4}{4!}y_n^{(iv)} - \tfrac{h^5}{5!}y_n^{(v)} + O(h^6) \end{aligned} \tag{8.48}$$

Summing the last two expansions:

$$y_{i+1} - 2y_i + y_{i-1} = h^2 y''_n + \frac{h^4}{12}y_n^{(iv)} \quad . \tag{8.49}$$

In the last equation, we can substitute y''_i with Eq. 8.47. The term $y_n^{(iv)}$ can be calculated using the central difference method for the second derivative (which can be obtained also subtracting the Taylor expansions up to the second order) applied to the second derivative

$$\frac{dy^4}{dx^4} = \frac{y''_{i+1} - 2y''_i + y_{i-1}}{h^2} \quad , \tag{8.50}$$

where again we can insert Eq. 8.47 with the correct indices.
After substituting in Eq. 8.49 the expressions for the second and fourth derivatives

$$\begin{aligned} y_{i+1} - 2y_i + y_{i-1} = h^2(-g_i y_i + s_i) + \\ + \tfrac{h^2}{12}\left((-g_{i+1}y_{i+1} + s_{i+1} + 2g_i y_i - 2s_i - g_{i-1}y_{i-1} + s_{i-1}\right) \end{aligned} \tag{8.51}$$

Rearranging the terms bringing all the y on the left we obtain Numerov's algorithm:

$$\begin{aligned} \left(1 + \tfrac{h^2}{12}g_{i+1}\right)y_{i+1} - 2\left(1 - \tfrac{5h^2}{12}g_i\right)y_i + \left(1 + \tfrac{h^2}{12}g_{i-1}\right)y_{i-1} = \\ = \tfrac{h^2}{12}(s_{i+1} + 10s_i + s_{i-1}) \end{aligned} \tag{8.52}$$

The last formula can be slighly rewritten noticing that

$$2\left(1 - \frac{5h^2}{12}g_i\right) = 12 - 10\left(1 + \frac{h^2}{12}g_i\right) \quad .$$

This allows to define in a numerical code the function $G(i) = 1 + \frac{h^2}{12} g_i$ reducing the Numerov equation to

$$\begin{aligned} G(i+1)y_{i+1} - [12 - 10G(i)]\, y_i + G(i-1)y_{i-1} = \\ = \tfrac{h^2}{12}(s_{i+1} + 10s_i + s_{i-1}) \end{aligned} \tag{8.53}$$

8.17 Application to the Schrödinger Equation: Particle in a Box Potential

The one-dimensional Schrödinger equation of a particle in a potential V(x) is

$$\left[-\frac{\hbar^2}{2m}\frac{d}{dx} + V(x) \right] \psi(x) = E\psi(x) \quad . \tag{8.54}$$

Defining

$$k^2(x) = \frac{2m}{\hbar}\left[E - V(x)\right] \quad , \tag{8.55}$$

the Schrödinger equation becomes

$$\psi''(x) = -k^2(x)\psi(x) \quad , \tag{8.56}$$

which is in the Numerov's form with $g(x) = k^2(x)$ and $s(x) = 0$. Eq. 8.52 specifying the Numerov's procedure becomes (with the usual notation $\psi_i = \psi(x_i)$)

$$\psi_{i+1} = \frac{1}{1 - \frac{h^2}{12}k_{i+1}^2}\left[\psi_i\left(2 + \frac{10h^2}{12}k_i^2\right) - \psi_{i-1}\left(1 - \frac{h^2}{12}k_{i-1}^2\right)\right] \tag{8.57}$$

Let's test the algorithm in a case where the analytic solutions are known. We consider the simple potential given by the "square well"

$$V(x) = \begin{cases} 0 & \text{if} \quad 0 < x < L \\ \infty & \text{otherwise} \end{cases} \tag{8.58}$$

with boundary conditions $\psi(0) = \psi(L) = 0$.
In this case, the potential is constant and the Schrödinger equation to solve is

$$\psi''(x) + k^2\psi(x) = 0 \quad , \tag{8.59}$$

which has general solutions $\psi(x) = A\sin(kx) + B\cos(kx)$. Taking into account the boundary conditions, the solutions are

$$\psi(x)_n = A\sin(n\pi x) \quad , \tag{8.60}$$

where we introduced the subscript n which labels the different solutions. The constant A can be fixed with the normalization condition $\int |\psi| dx = 1$ which results in $A = \sqrt{2}$. Since $k^2 = n\pi$, we find the energies associated to the different wave functions (the *eigenvalues*):

$$E_n = \frac{n\pi\hbar}{2m} \quad . \tag{8.61}$$

8.18 Application to the Schrödinger Equation: The Hydrogen Atom

We apply now Numerov's algorithm to the classical quantum mechanical problem of a particle in a central potential. This case is interesting since it shows different numerical problems to tackle, from numerical stability to the functional eigenvalue problem.
The Schrödinger equation for the hydrogen atom (an electron in a central Coulomb potential) is

$$\left[-\frac{\hbar^2}{2m}\nabla^2 - \frac{Ze^2}{4\pi\epsilon_0 r}\right]\psi(\bar{r}) = E\psi(\bar{r}) \quad . \tag{8.62}$$

Using spherical coordinates[3] (r, θ, phi), the wavefunction ψ can be rewritten as the product of one radial and one angular function: $\psi(\bar{r}) = R(r)\cdot Y(\theta,\phi)$. Moreover, we define a new function $u(r) = R(r)/r$. We consider here only the radial function $u(r)$ introducing the variables

$$x = r/a_o \quad ; \quad a_0 = \frac{4\pi\epsilon_0\hbar^2}{me^2} \approx 0.529 \text{ Å} \quad , \tag{8.63}$$

where a_0 is the Bohr radius. We introduce also the Rydberg energy E_0 (the energy of the ground state) and the corresponding normalized energy $\mathcal{E}$

$$E_0 = \frac{\hbar^2}{2ma_0} \approx 13.605 \text{ eV} \quad ; \quad \mathcal{E} = \frac{E}{E_0} \quad . \tag{8.64}$$

[3] Also the Laplacian ∇^2 should be written in spherical coordinates.

Substituting Eq. 8.63 and Eq. 8.64 in Eq. 8.62 and considering only the function $u(r)$ we obtain [4]

$$\frac{d^2u(x)}{dx^2} - \left[\frac{l(l+1)}{x^2} - \frac{2Z}{x} - \mathcal{E}\right] u(x) = 0 \quad , \tag{8.65}$$

which is in the Numerov form with $s = 0$. The integer number l labels the quatum state's angular momentum.
The boundary conditions are $u(0) = 0$ and $u(\infty) = 0$.
The problem defined in Eq. 8.65 is of a new kind with respect to what we analyzed until now: we have to find at the same time the wavefunction $u(r)$ *and* the value for the constant $\mathcal{E}$. This is actually an *eigenvalue problem* where $u(r)$ are the eigenfunctions and $\mathcal{E}$ the eigenvalues.
First, we are going to solve Eq. 8.65 with the Numerov's method, given a value of $\mathcal{E}$:

$$u_{n+1} = \frac{[12 - 10G_n]\, u_n - G_{n-1}u_{n-1}}{G_{n+1}} \quad , \tag{8.66}$$

where $G_n = (1 + h^2/12)g_n$ (see Eq. 8.53). An important numerical observation at this point is related to the boundary conditions. Since at large x the function must approach zero and at the finite x value $u(0) = 0$, it is better to apply Eq. 8.66 **backwards**, starting from a large value of x and then going towards $x = 0$: this procedure will be numerically more stable. In fact, starting from $x = 0$ the numerical error might accumulate and $u(\infty) = 0$ could be difficult to reach.
Having now a (stable)procedure for integrating the Schrödinger equation, we have to find the correct value of $\mathcal{E}$ corresponding to a given function $u(x)$.
The idea is to start from a guess value $\mathcal{E}_0$, calculate the wavefunction $u(x)$ and check if it crosses the x-axis at $x = 0$ (i.e. respects the boudary condition u(0)=0). If it is not the case, another $\mathcal{E}_1 = \mathcal{E}_0 + \delta E$ is chosen and u recalculated.
Since the solution is found on a discrete mesh of points, it is not easy to check with precision if $u(0) = 0$. In order to improve the check, a *linear extrapolation* to zero using the last two (backwards) calculated points u_0 and u_1 can be used:

$$u(0) = u_0 + \frac{u_1 - u_0}{h}(0 - x_0) \quad . \tag{8.67}$$

In summary, the algorithm is the following:

[4] For obtaining the final radial equation, the method of separation of variables should be used. The complete treatment of the hydrogen atom can be found in most quantum mechanics books and here we just quote the result for concentrating on the numerical algorithm.

1. Start with a guess energy $\mathcal{E}_0$. The guess energy should be smaller than the smallest potential energy. In the variables we choose, this means $E < -Z^2$

2. Integrate with Numerov's method the Schrödinger radial equation with $\mathcal{E} = \mathcal{E}_0$ obtaining $u_1(x)$.

3. Extrapolate the solution and obtain $u_1(0)$.

4. Increase the energy $\mathcal{E}_1 = \mathcal{E}_0 + \delta E$ and recalculate $u(x)$ again, extrapolate and obtain $u_2(0)$.

5. If $u_1(0) \cdot u_2(0) > 0$ go back and repeat the previous step. If $u_1(0) \cdot u_2(0) \leq 0$, continue.

6. Now we are sure that our solution is in the interval $\mathcal{E}_1 < \mathcal{E} < \mathcal{E}_2$, we can use a root finding algorithm to find $\mathcal{E}$ with good precision.

The previous algorithm is implemented in Code 8.3/8.4.

```python
import numpy as np
from scipy.integrate import simps
from scipy.optimize import brentq
import matplotlib.pyplot as plt

#Numerov's algorithm (backwards for better stability)
def WaveFunctionZero(E, r0, n=1, l=0, Z=1, du=0.01):

    ur = np.zeros(len(r0))
    ur[-1] = 0.0 ; ur[-2] = du

    h   = r0[1] - r0[0]
    h12 = h**2 / 12.0
    gn  = E + 2.0*Z/r0 - l*(l+1)/r0**2
    fn  = 1. + h12 * gn

    for i in range(gn.size-3, -1, -1):
        ur[i] = ((12 - 10*fn[i+1]) * ur[i+1] - ur[i+2] *
    fn[i+2])/fn[i]

    # normalization
    ur /= np.sqrt(simps(ur**2, x=r0))

    # Linear extrapolation to u(r=0)
    u0 = ur[0] + (ur[1] - ur[0]) * (0 - r0[0]) / h

    return u0
```

Listing 8.3: Energy Eigenvalues of the Hydrogen Atom (1/2).

```
r0 = np.linspace(1e-6, 50, 5000) #Radius mesh

#Quantum numbers
N=1  #Principal quantum number (not used here)
L=0  #Angular quantum number

E_low = E_up = -1.0
dE = 0.10
u1 = WaveFunctionZero(E_low, r0, n=N,l=L)

#Increase E until crossing the x-axis
while True:
    E_up += dE
    u2 = WaveFunctionZero(E_up, r0, n=N,l=L)
    if u1 * u2 < 0: break

#Find the root of the wavefunction between E_low and
   E_up using Brent's algorithm
E = brentq(WaveFunctionZero, E_low, E_up, args=(r0, N, L
   , 1, 0.001))
print("Energy = ",E*13.605, " eV")
```

Listing 8.4: Energy Eigenvalues of the Hydrogen Atom (2/2).

The function `WaveFunctionZero` applies Numerov's algorithm, calculates the wavefunction, and extrapolates to u(0). The main part of the code starts from a value for the energy and then increases it until $u_1(0) \cdot u_2(0) > 0$. Finally, Brent's algorithm (in its `scipy` library implementation) is used for calculating the precise final result.

CHAPTER 9 | Elliptic Equations

9.1 Introduction

Many phenomena are described by second order partial differential equations. Introducing the notation $\partial f/\partial x = f_x$, a general form of such equations is

$$\begin{aligned} a(x,y,f,f_x,f_y)f_{xx} + \\ 2b(x,y,f,f_x,f_y)f_{xy} + \\ c(x,y,f,f_x,f_y)f_{yy} + \\ f(x,y,f,f_x,f_y) = 0 \quad . \end{aligned} \tag{9.1}$$

It is generally assumed that $f_{xy} = f_{yx}$ and

$$a^2 + b^2 + c^2 \neq 0 \quad , \tag{9.2}$$

such that a second order derivative will always be present.
A basic classification of the possible forms of Eq. 9.1 is given by the evaluation of the *discriminant* $\Delta = b^2 - ac$

- $\Delta < 0$ **Elliptic Equations:** In this class of equations we have for example Laplace's Equation

  $$\nabla^2 f = f_{xx} + f_{yy} = 0 \quad , \tag{9.3}$$

  where f is a *potential function* in many physics applications. In particular, the functions solving this equations are called *harmonic*.

- $\Delta = 0$ **Parabolic Equations:** The most important example of this class of equations is the *diffusion equation*

  $$f_{xx} - f_y = 0 \quad . \tag{9.4}$$

- $\Delta > 0$ **Hyperbolic Equations:** Typical hyperbolic equations are the *wave equations*

$$f_{xx} - f_{yy} = 0 \quad . \tag{9.5}$$

We will start the discussion of partial differential equations from the elliptic ones.

9.2 Boundary Value Problems for Elliptic Equations

We will concentrate on the solution of the Laplace problem in a boundary given its relevance in applications such as finding the electric potential in a given region of space. This problem is commonly known as **Dirichlet Problem** and is defined as finding a function $f(x, y)$ with the following conditions

1. Ω is an open bounded simply connected set
2. Γ is the boundary of Ω and it is piecewise continuously differentiable
3. f is harmonic: $\nabla^2 f(x, y) = 0$
4. f continuous on $\Omega \cup \Gamma$
5. $g(x, y)$ is a continuous function on Γ
6. f identical to g on Γ

In can be proven, that the Dirichlet problem has one single solution, although only in very simple cases it can be constructed analitically: most of the times, a numerical treatment is needed.

9.3 Dirichlet Problem on a Rectangle

This is one of the simplest cases we can consider, where Γ describes the perimeter of a rectangle and Ω is the set of points contained in the rectangle. Let's define the sides of the rectangle as the sets $[a_1, b_1]$ and $[a_2, b_2]$. Inside these sets, we can define the grids of n and m points equispaced by h in one direction and k in the other

$$\begin{aligned} x_i &= a_1 + ih \quad , \quad h = (b_1 - a_1)/n \\ y_i &= a_2 + ik \quad , \quad k = (b_2 - a_2)/m \end{aligned}$$

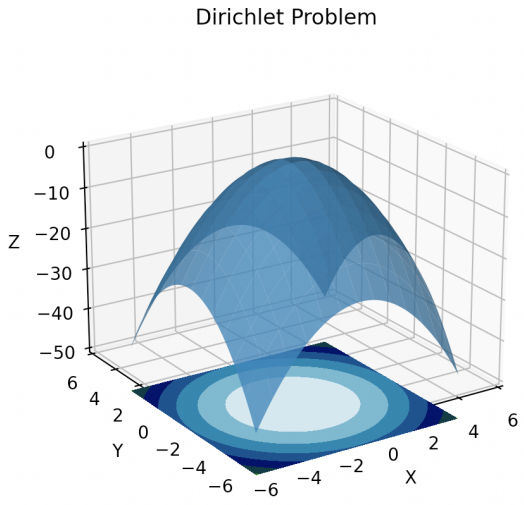

Figure 9.1: Visualization of the two-dimensional Dirichlet problem: the function $f(x,y) = -x^2 - y^2$ is defined on a set Ω and the border Γ is a square. On this square, a function $g(x,y) = 0$ defines the boundary values of the problem.

We can also introduce the following nomenclature

- The points $(x_0, y_j), (x_n, y_j)$ with $j = 1, 2., ..m-1$ and $(x_i, y_0), (x_i, y_m)$ with i=1,2,..,n-1 are the *boundary points*.
- The points (x_i, y_j) with i=1,2,..n-1, j=1,2,..,m-1 are called *interior points*.
- The points $D = (x_0, y_0), E = (x_n, y_0), F(x_n, y_m), G(x_0, y_n)$ are the *corner points*.
- Every sub-rectangle delimited by the points (x_i, y_j), (x_{i+1}, y_i), (x_i, y_{j+1}), (x_{i+i}, y_{j+1}) is called a *cell*.
- For a function defined on such a grid, we define the shorter notation $f(x_i, y_j) = f_{i,j}$.

The Laplace equation can be directly approximated on the previously defined grid applying the central difference scheme for second order derivatives

$$\frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{h^2} + \frac{f_{i,j+1} - 2f_{i,j} + f_{i,j-1}}{k^2} = 0 \quad . \tag{9.6}$$

Very often, in this kind of problems $h = k$: eliminating the grid spacing and rearranging the terms we can see how a single grid point $f_{i,j}$ depends from the neighboring ones

$$f_{i,j} = \frac{1}{4}\left(f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-1}\right) \quad . \tag{9.7}$$

This last equation suggests an iterative updating procedure for approximating the solution: given the values on the grid (which are the initial boundary values and guess values for the interior points) at a step k, we can find new values at the next step $k+1$ with what is sometimes called **Jacobi iteration**:

$$f_{i,j}^{(k+1)} = \frac{1}{4}\left(f_{i+1,j}^{(k)} + f_{i-1,j}^{(k)} + f_{i,j+1}^{(k)} + f_{i,j-1}^{(k)}\right) \quad . \tag{9.8}$$

The following code 9.1 implements the solution of the Dirichlet problem with Jacobi iterations.

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

def LaplaceIterative(g,Tinitial,\
                     lenX,lenY,delta,maxIter):

    # Set meshgrid
    X, Y = np.meshgrid(np.arange(0, lenX), np.arange(0,
    lenY))

    # Set initial matrix
    T = np.zeros((lenX, lenY))
    T.fill(Tinitial)

    # Set Boundary condition
    for i in X[0,:]: T[i,0] = g(i,0)
    for i in X[0,:]: T[i,lenY-1] = g(i,lenY-1)
    for i in Y[:,0]: T[0,i] = g(0,i)
    for i in Y[:,0]: T[lenX-1,i] = g(lenX-1,i)

    #Jacobi Iteration
    for iteration in range(0, maxIter):
        for i in range(1, lenX-1, delta):
            for j in range(1, lenY-1, delta):
                T[i, j] = 0.25 * \
                (T[i+1][j] + T[i-1][j] + \
                 T[i][j+1] + T[i][j-1])

    return T

#Example Boundary condition function
def g(x,y): return (x-1)**2 - (y-2)**2

Tinitial = 30 #Starting guess value
lenX = 50;lenY = 100 #Grid dimensions
delta = 1 #Grid width
maxIter = 500 #Maximum number of iterations

#Iterative solution
T = LaplaceIterative(g,Tinitial,lenX,lenY,delta,maxIter)
```

Listing 9.1: Dirichlet Problem

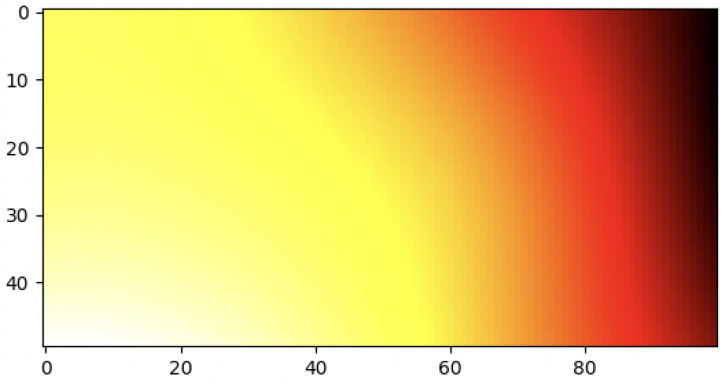

Figure 9.2: Graphical representation of the output matrix from code 9.1.

In Fig 9.3, the output of the code 9.1 is showed. The output of the code is a matrix T, which is the solution of the Dirichlet problem on a rectangle. The boundary condition is described by the function $g(x,y) = (x-1)^2 - (y-1)^2$: this equation is used only for fixing the initial values on the grid points which belong to the border of the rectangle. The interior points can be initialized with an initial guess.
The same problem can be also solved in principle with matrix algorithms. Eq. 9.6, in the simplified case where $h = k$, can be viewed as a linear system of equations. For example, considering $n = 3$ and $m = 4$

$$\begin{cases} f_{21} - 2f_{11} + f_{01} + f_{12} - 2f_{11} + f_{10}=0 \\ f_{31} - 2f_{21} + f_{11} + f_{22} - 2f_{21} + f_{20}=0 \\ f_{22} - 2f_{12} + f_{02} + f_{13} - 2f_{12} + f_{11}=0 \\ f_{32} - 2f_{22} + f_{12} + f_{23} - 2f_{22} + f_{21}=0 \\ f_{23} - 2f_{13} + f_{03} + f_{14} - 2f_{13} + f_{12}=0 \\ f_{33} - 2f_{23} + f_{13} + f_{24} - 2f_{23} + f_{22}=0 \end{cases} \tag{9.9}$$

which reduces to

$$\begin{cases} -4f_{11} & +f_{21} & +f_{12} & & & & = & -f_{01} - f_{10} \\ f_{11} & -4f_{21} & & +f_{22} & & & = & -f_{20} - f_{31} \\ f_{11} & & -4f_{12} & +f_{22} & +f_{13} & & = & -f_{02} \\ & f_{21} & f_{12} & -4f_{22} & & +f_{23} & = & -f_{32} \\ & & f_{12} & & -4f_{13} & +f_{23} & = & -f_{14} - f_{03} \\ & & & f_{22} & +f_{13} & -4f_{23} & = & -f_{33} - f_{24} \end{cases} \tag{9.10}$$

The numbers on the right-hand of the equations are known since they correspond to the boudary of Ω and are fixed by calculating the function $g(x,y)$. In matrix form $Ax = b$, the problem is block-diagonal

$$A = \begin{pmatrix} -4 & 1 & 1 & 0 & 0 & 0 \\ 1 & -4 & 0 & 1 & 0 & 0 \\ 1 & 0 & -4 & 1 & 1 & 0 \\ 0 & 1 & 1 & -4 & 0 & 1 \\ 0 & 0 & 1 & 0 & -4 & 1 \\ 0 & 0 & 0 & 1 & 1 & -4 \end{pmatrix} \quad , \tag{9.11}$$

and can be solved with methods described in Chap. 3.

Chapter 10 | Parabolic Equations

10.1 Definition of the problem

The typical parabolic equation is the **heat equation**

$$\frac{\partial^2 \phi}{\partial x^2} - \frac{\partial \phi}{\partial y} = 0 \quad , \tag{10.1}$$

or in a more compact form $\phi_{xx} - \phi_y = 0$. In common applications, y has the physical meaning of time and the equation is written as $\phi_{xx} - \phi_t = 0$.
Two problems can be defined for this differential equation

- **Initial Value Problem:** Find ϕ satisfying Eq. 10.1 with - $-\infty < x < +\infty$, $t \geq 0$
  - $\phi(x, 0) = f(x)$ at $t = 0$ for $-\infty < x < +\infty$.
- **Initial-Boundary Value Problem:** Given $a > 0, g_1(0) = f(0), g_2(0) = f(a)$,
  Find ϕ satisfying Eq. 10.1 with
  - $t \geq 0, 0 \leq x \leq a$
  - satisfies Eq. 10.1 in $0 < x < a$, $t > 0$. - satisfies the *initial condition*: $\phi(x, 0) = f(x), 0 \leq x \leq a$
  - satisfies the *boundary condition*: $\phi(0, t) = g_1(t), \phi(a, t) = g_2(t), t \geq 0$.

10.2 Explicit Method for the Initial-Boundary Problem for the Heat Equation

Solutions of the heat equation have the min-max property.
Using a formalism similar to that used for elliptic equations, we can directly dis-

cretize the (x,t) space and rewrite the parabolic equation as

$$\frac{\phi_{i,j+1} - \phi_{i,j}}{k} = \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{h^2} \quad , \tag{10.2}$$

which can be rewritten as

$$\phi_{i,j+1} = \phi_{i,j} + \frac{k}{h^2}(\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}) \quad , \tag{10.3}$$

and setting $\alpha = k/h^2$ we have

$$\phi_{i,j+1} = \alpha\phi_{i+1,j} + (1 - 2\alpha)\phi_{i,j} + \alpha\phi_{i-1,j} \quad . \tag{10.4}$$

The last equation can be solved applying the following algorithm:

1. Setting $j = 0$, calculate $\phi_{i,1}$ for each $i = 1, 2, .., n-1$
2. Use the results of step 1. and set $j = 1$, solving for $\phi_{i,2}$.
3. Continue the procedure for $j = 2, 3, .., m-1$ using the boundary conditions for the boudary grid points.

The obtained set of values $\phi_{i,j}$ at each grid point represent the numerical solution of the parabolic equation.

10.3 Explicit Central Difference Method

The Initial-Boundary Problem for the general Parabolic Equation consists in finding the solution of

$$\phi_t = P(x,t)\phi_{xx} + Q(x,t)\phi_x + R(x,y)\phi + S(x,y) \quad , \tag{10.5}$$

in the region $0 < x < a$, $t > 0$ with:

- For $a > 0$ the functions P(x,u), Q(x,t), R(x,t), S(x,t) are bounded and continuous functions in $0 \leq x \leq a$, $t > 0$,
- $\phi(x,0) = f(x)$, $0 \leq x \leq a$

- $\phi(0,t) = g_1(t), \phi(a,t) = g_2(t), t \geq 0.$
- $g_1(0) = f(0), g_2(0) = f(a)$

The explicit method of discretization for finding the values of the function ϕ in the inner grid points is the following

$$\frac{\phi_{i,j+1} - \phi_{i,j}}{k} = P_{ij}\frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{h^2} + Q_{ij}\frac{\phi_{i+1,j} - \phi_{i-1,j}}{2h} + R_{ij}\phi_{i,j} + S_{ij} \quad , \tag{10.6}$$

where we used a forward derivative in time and a central derivative in space. Defining $\alpha = k/h^2$ and $\beta = k/2h$, the last equation can be rewritten as

$$\begin{aligned}\phi_{i,j+1} = \\ &(\alpha P_{ij} - \beta Q_{ij})\phi_{i-1,j} + (1 + kR_{ij} - 2\alpha P_{ij})\phi_{i,j} + \\ &(\alpha P_{ij} + \beta Q_{ij})\phi_{i+1,j} + kS_{ij} \quad .\end{aligned}$$

The last equation can be used to explicitly generate new values of ϕ at increasing time values. This method is known as **explicit central difference method**.

10.4 Implicit Central Difference Method

This method uses a backward derivarive in time

$$\phi_t = \frac{\phi_{i,j} - \phi_{i,j-1}}{k} \quad , \tag{10.7}$$

and the central first and second derivatives

$$\phi_x = \frac{\phi_{i+1,j} - \phi_{i-1,j}}{2h} \quad , \tag{10.8}$$

$$\phi_{xx} = \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{h^2} \quad . \tag{10.9}$$

We condider now a "mildly" non-linear parabolic equation of the form

$$\phi_t = P(x,t)\phi_{xx} + Q(x,t)\phi_x + F(x,t,\phi) \quad , \tag{10.10}$$

to arrive to the form (see previous section for an analogue calculation)

$$\begin{aligned}(\alpha P_{ij} - \beta Q_{ij})\phi_{i-1,j} - (1 + 2\alpha P_{ij})\phi_{i,j} + \\ (\alpha P_{ij} + \beta Q_{ij})\phi_{i+1,j} + kF(x_i, t_j, \phi_{i,j}) = 0 \quad .\end{aligned} \tag{10.11}$$

The previous result can be used for generating a numerical solution at each grid point. Defining

$$\begin{aligned}f_1(\phi_{1,j}) &= \phi_{1,j-1} + kF(x_1, t_j, \phi_{1,j}) + (\alpha P_{1j} - \beta Q_{1j})\phi_{0,j} \quad , \\ f_i(\phi_{i,j}) &= \phi_{i,j-1} + kF(x_1, t_j, \phi_{1,j}) \quad , i = 2, 3, 4, .., n-2 \quad , \\ f_{n-1}(\phi_{n-1,j}) &= \phi_{n-1,j-1} + kF(x_{n-1}, t_j, \phi_{n-1,j}) + (\alpha P_{n-1,j} - \beta Q_{n-1,j})\phi_{n,j} \quad ,\end{aligned} \tag{10.12}$$

we can write consecutively Eq. 10.11 for i=1,2,..,n-1 :

$$\begin{aligned}-(1 + 2\alpha P_{1j})\phi_{1,j} + (\alpha P_{1j} + \beta Q_{1j})\phi_{2,j} + f_1(\phi_{1,j}) = 0 \\ (\alpha P_{2j} - \beta Q_{2j})\phi_{1,j} - (1 + 2\alpha P_{2j})\phi_{2,j} + (\alpha P_{2j} + \beta Q_{2j})\phi_{3,j} + f_1(\phi_{2,j}) = 0 \\ \ldots \quad \ldots \quad \ldots \\ \ldots \quad \ldots \quad \ldots \\ (\alpha P_{n-1,j} - \beta Q_{n-1,j})\phi_{n-2,j} - (1 + 2\alpha P_{n-1,j})\phi_{n-1,,j} + f_{n-1}(\phi_{n-1,j}) = 0\end{aligned}$$

The previous system has one equation for each "space" point. Starting with the "time" $j = 1$ and inserting the known boudary values in Eqs. 10.12, the system can be solved obtaining $\phi_{1,1}, \phi_{2,1}, \phi_{3,1}, ..., \phi_{n-1,1}$. The solution at $j = 1$ can be used to find the $j = 2$ solution adn so on. This procedure is known as **implicit central difference method**.

10.5 The Crank-Nicolson Method

The mathod outlined before has rather good accuracy in the space variable. We describe now a method for increasing also the accuracy in the time variable. In general, symmetric discrete derivatives achieve a higher accuracy. Following this

idea, we consider a time derivative defined at the point $C = (i, j - 1/2)$ which is *between* $A = (i, j)$ and $B = (i, j - 1)$

$$\phi_t(x_i, t_{j-1/2}) = \frac{\phi_{i,j} - \phi_{i,j-1}}{k} \quad . \tag{10.13}$$

Considering then the central second derivatives in A and B

$$\phi_{xx}(x_i, t_j) = \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{h^2} \quad ,$$
$$\phi_{xx}(x_i, t_{j-1}) = \frac{\phi_{i+1,j-1} - 2\phi_{i,j-1} + \phi_{i-1,j-1}}{h^2} \quad ,$$

we can assume as second derivative at the point C the average of the second derivatives in A and B:

$$\phi_{xx}(x_i, t_{j-1/2}) \approx \frac{1}{2}\left[\phi_{xx}(x_i, t_j) + \phi_{xx}(x_i, t_{j-1})\right] \quad . \tag{10.14}$$

Following the same idea for the first derivative ϕ_x and substituting in Eq. 10.10

$$\begin{aligned} \frac{\phi_{i,j} - \phi_{i,j-1}}{k} = \\ P_{i,j-1/2}\left[\frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{2h^2} + \frac{\phi_{i+1,j} - 2\phi_{i,j-1} + \phi_{i-1,j-1}}{2h^2}\right] \\ + Q_{i,j-1/2}\left[\frac{\phi_{i+1,j} - \phi_{i-1,j}}{4h} + \frac{\phi_{i+1,j-1} - \phi_{i-1,j-1}}{4h}\right] \\ + F\left(x_i, t_{j-1/2}, \frac{\phi_{i,j} - \phi_{i,j-1}}{2h}\right) \quad . \end{aligned} \tag{10.15}$$

Eq. 10.15 is known as the **Crank-Nicolson method**. This method has higher accuracy as the explicit and implicit methods.

10.6 Reaction-Diffusion Systems

Many interesting physical and chemical systems can be modeled with the so-called reaction-diffusion equation, which is a parabolic equation of the form

$$\frac{\partial n}{\partial t} - \underbrace{D\frac{\partial^2 n}{\partial x^2}}_{\text{Diffusion}} + \underbrace{nL(n) = G(n)}_{\text{Reaction}} \quad , \tag{10.16}$$

where we are considering here the simplest one spacial dimension case and a single *concentration* n. The function $n(x,t)$ might describe the concentration of a chemical reactant or some other physical quantity. The term containing the second spacial derivative is the *diffusion* term and D is called *diffusion constant*. The diffusion term accounts for the diffusion of the species, while the *loss term* L(n) and the *gain term* G(n) model the "reaction" part of the system. In summary, the equation describes the time and spacial evolution of a certain species which diffuese in an environment which can "react" to the diffusion reinforcing or diminishing the local population of the diffusing species.
A system can be composed from more species n_i, leading to a system of reaction-diffusion equations and also the number of dimensions can be higher than one. For example, in more spacial dimensions, the second derivative becomes the Laplacian $\frac{\partial^2}{\partial x^2} \longrightarrow \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + ...$

10.7 Bidimensional Systems: Turing Instability

We introduce now a class of reaction-diffusion systems which was proposed by A. Turing for explaining morphogenesis: the pattern creation in various biological systems.
In order to understand in simple terms this phaenomenon, we first consider the linear system of ordinary differential equations

$$\begin{cases} \frac{du}{dt} = au + bv \\ \frac{dv}{dt} = cu + dv \end{cases} \quad , \tag{10.17}$$

for the two functions $u = u(t)$ and $v = v(t)$, while a, b, c, d are real constant parameters. If we assume

$$T = a + d < 0 \quad , \quad D = ad - bc > 0 \quad , \tag{10.18}$$

we have (0,0) as an attractive stable point for the system. An equivalent statement is that the matrix of the parameters has two eigenvalues $\lambda_{1/2}$ with negative real parts. This can be checked directly calculating the characteristic polynomial which results in

$$\lambda^2 - \lambda T + D = 0 \quad \Rightarrow \quad \lambda_{1/2} = \frac{1}{2}(T \pm \sqrt{T^2 - 4D}) \quad . \tag{10.19}$$

If we now promote the system 10.17 to a reaction-diffusion system

$$\begin{cases} \frac{du}{dt} - \sigma_u \Delta u = au + bv \\ \frac{dv}{dt} - \sigma_v \Delta v = cu + dv \quad , \end{cases} \tag{10.20}$$

what we expect is that the friction caused by diffusion will strenghten the stability of (0,0) even more. Surprisingly, this might not happen, as explained by the

Theorem 11 (Turing Instability Theorem). *Consider the system 10.20 within a domain Ω of R^d and the coefficients a, b, c, d following the conditions in Eq. 10.18 and $a > 0$, $d < 0$, $\sigma_v > 0$. Then, for sufficiently small values of σ_u, the stationary state (0,0) is linearly unstable and only a finite number of eigenmodes of the system are unstable.*

Since $a > 0$, the component u is called the *activator*, while since $d < 0$, v is called the *inhibitor*. The competition between these two components in the particular settings of the latter theorem give rise to an instability which forms interesting spatial patterns with applications to biology problems. Another phaenomenon happening in reaction-diffusion systems is the formation of waves. The relationship with Turing instability can be traced to the following observations:

- **Turing instability:** short range activator, long range inhibitor
- **Travelling waves**: long range activator, short range inhibitor.

The smaller diffusion coefficient of the inhibitor makes it a long-range degree of freedom, preventing the formation of waves, but allowing Turing instability.

10.8 A non-linear Turing Instability Example

As example for a non-linear model displaying Turing instability, we use the FitzHugh–Nagumo model, which is a bi-dimensional version of the one-dimensional Hodgkin–Huxley

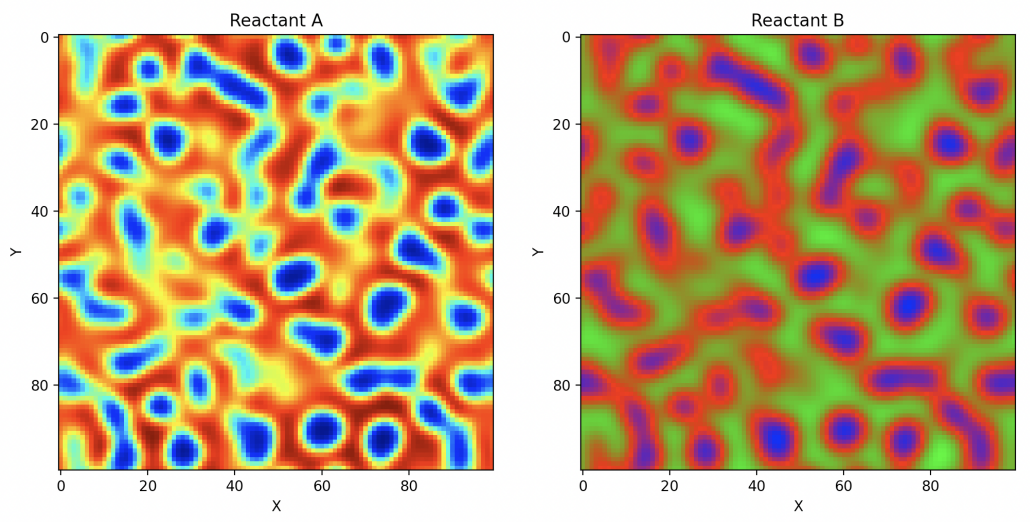

Figure 10.1: Turing instability patterns from the FitzHugh–Nagumo model.

model for the neuronal acivity. The model is defined by

$$\begin{cases} \frac{du}{dt} - \sigma_u \Delta u = u(1-u^2) - v + \alpha \\ \frac{dv}{dt} - \sigma_v \Delta v = \beta(u-v) \end{cases} , \tag{10.21}$$

where α and β are parameters of the model. A result of the spatial patterns resulting from the model are showed in Fig. 10.1. The choice of the parameters is $\alpha = -0.005$, $\beta = 10$, and $\sigma_u = 1$, $\sigma_v = 100$ and the code 10.1 implements an explicit finite-difference scheme in two dimensions.
Note how the 2-dimensional Laplacian ($\Delta = \partial_{xx} + \partial_{yy}$) is implemented in the code using the `np.roll` vector with automatic periodic boundary conditions.
In two spacial dimensions, the finite difference scheme for the Laplacian is (compare with the code 10.1)

$$\Delta f = \partial_{xx} f + \partial_{yy} f \approx \frac{f_{i+1,j} - 2f_{i,j} + f_{i-1,j}}{dx^2} + \frac{f_{i,j+1} - 2f_{i,j} + f_{i,j-i}}{dy^2} = \frac{1}{h^2}\left(-4f_{i,j} + f_{i+1,j} + f_{i-1,j} + f_{i,j+1} + f_{i,j-i}\right) \quad .$$

where we choose the same grid spacing in both spatial directions (dx=dy=h).

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

#2D Laplacian with periodic boundary conditions
def laplacian2D(a, dx):
    return (
        - 4 * a
        + np.roll(a,1,axis=0) + np.roll(a,-1,axis=0)
        + np.roll(a,+1,axis=1)+ np.roll(a,-1,axis=1)
    ) / (dx ** 2)

#Diffusion coefficients and parameters
Da, Db , alpha, beta = 1, 100, -0.005 , 10
#Reaction terms (FitzHugh-Nagumo model)
def Ra(a,b): return a - a ** 3 - b + alpha
def Rb(a,b): return (a - b) * beta
#Discretization and steps
X, Y = 100 , 100
dx, dt, steps = 1, 0.001, 1000
#Random initialization
a , b  = np.random.random((X,Y)),np.random.random((X,Y))

t=0
for i in range(steps):
    t = t + dt
    La, Lb = laplacian2D(a, dx), laplacian2D(b, dx)

    #Reaction-Diffusion System
    delta_a = dt * (Da * La + Ra(a,b))
    delta_b = dt * (Db * Lb + Rb(a,b))

    a,b = a + delta_a , b + delta_b

fig, ax = plt.subplots(nrows=1, ncols=2, figsize=(12,6))
ax[0].imshow(a, cmap='jet');ax[1].imshow(b, cmap='brg')
ax[0].set_title("Reactant A");ax[1].set_title("Reactant
    B")
ax[0].set_xlabel("X");ax[0].set_ylabel("Y")
ax[1].set_xlabel("X");ax[1].set_ylabel("Y")
plt.show()
```

Listing 10.1: Turing Instability

Chapter 11 | Financial Applications

11.1 Introduction: the Ito Formula

A fundamental result in the theory of stochastic processes is the **Ito Formula** in the framework of Ito calculus which states how to differentiate a function Y which depends from a stochastic process X and the time variable t.
We consider the stochastic process defined in the interval [0,T] by the stochastic differential equation

$$dX_t = \underbrace{F(t, X_t)dt}_{\text{Drift}} + \underbrace{G(t, X_t)dW_t}_{\text{Diffusion}} \quad , \tag{11.1}$$

which is a formal way for expressing the integral

$$X_t = X_0 + \int_0^t F(t')dt' + \int_0^t G(t)dW_{t'} \tag{11.2}$$

Ito's formula gives an expression for the differential of $Y_t = Y_t(t, X_t)$:

$$dY_t = \left[\partial_t Y_t + F(t)\partial_x Y_t + \frac{1}{2}G^2(t)\partial_{xx}Y_t\right]dt + G(t)\partial_x Y_t dW_t \quad . \tag{11.3}$$

It is interesting to note the presence of the second derivative. Intuitively, such term must be present since if we do not expand up to second order, we do not take into account the "variance" of the stochastic process but only the "drift".
Another useful form of Ito's formula is

$$dY_t = \partial_t dt + \partial_x Y dX_t + \frac{1}{2}\partial_{xx} Y (dX_t)^2 \quad , \tag{11.4}$$

taking into account the commutative multiplication rules $dW_t \cdot dW_t = dt$, $dt \cdot dW_t = 0$, and $dt \cdot dt = 0$.
For the D-dimensional case where there are X_t^i (i=1,..,D) stochastic processes we have

$$dY_t = \left[\partial_t Y + \sum_i F_i \partial_{x_i} Y_t + \frac{1}{2}\sum_{i,j} G_i G_j \partial^2_{x_i x_j} Y_t\right] dt + \left[\sum_i G_i \partial_{x_i} Y_t\right] dW_t \tag{11.5}$$

As example for the application of Ito's formula, we choose the geometric stochastic process

$$dS = \mu S dt + \sigma S dW \quad , \tag{11.6}$$

which will play an important role in the following sections. In this process, $F = \mu S$ and $G = \sigma S$. The equation can be rewritten as

$$d(\ln S) = \mu dt + \sigma dW \quad , \tag{11.7}$$

and applying Ito's formula to $Y = \ln S$

$$dY = \left[\mu \frac{1}{S} S - \frac{1}{S^2}\sigma^2 S^2\right] dt + \sigma \frac{1}{S} S dW = \left(\mu - \frac{\sigma^2}{2}\right) dt + \sigma dW \quad . \tag{11.8}$$

The last equation can be directly integrated

$$\ln S_t - \ln S_0 = \left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W \quad , \tag{11.9}$$

and therefore the solution of the stochastic differential equation for the geometric process is

$$S_t = S_0 e^{\left(\mu - \frac{\sigma^2}{2}\right) t + \sigma W} \quad , \tag{11.10}$$

where $W \sim N(0,t) = \sqrt{t}N(0,1)$ with $N(\alpha,\beta)$ the normal distribution with mean α and root mean square β.

11.2 The Black-Scholes Equation

The option pricing formula was derived by F. Black and M. Scholes in 1973 and independently by R.C. Merton in the same year. The formula solves the problem of pricing an European (put or call) option. The hypotheses under the derivation of the formula are

1. The short-term interest rate is constant and equal to r. It is possible to borrow at that rate.

2. The strike price X is known and constant in time.

3. The stock price S follows the geometric process

$$\frac{dS}{S} = \mu dt + \sigma dW \quad , \tag{11.11}$$

   where W is a Wiener process, μ the (constant) expected yield, and σ the (constant) volatility.

4. The stock does not pay dividends.

5. The marked has no transaction costs and there are no limits to short selling.

6. There is no arbitrage opportunities.

Let us consider a *call* option with price $c = c(S,t)$. The call gives the owner the right (but not the obligation) to buy the underlying financial product (e.g. a stock) at a certain price (the *strike price*) within or at a certain date. (A *put* option is similar but gives the right to sell).
The *intrinsic value* of a option is the maximum between zero and the option value if would be used right away. For a call option

$$c = \max(S - X, 0) \quad . \tag{11.12}$$

The last equation shows that the value is not zero only if the underlying asset has a value higher than the strike price. This means also that the option owner will exercise his right to buy only if $S > X$. For a put option

$$c = \max(X - S, 0) \quad . \tag{11.13}$$

Going back to the Black-Scholes equation, the main idea behind its derivation is the construction of a portfolio where stocks and the option are combined in a way

to eliminate the risk. No risk means constantly compensating for the variation of the stock price. The value V of such a portfolio is

$$V = S - \frac{1}{\Delta}c \quad , \tag{11.14}$$

where S is the stock price and c is the option price. $1/\Delta$ is fraction of options bought with respect to the stocks. A variation of the value can be expressed differentiating

$$dV = dS - \frac{1}{\Delta}dc \quad . \tag{11.15}$$

Since the differential dc(t,S) depends from the stochastic variable S following a geometric process, from **Ito's Lemma** we have

$$dc = \left[\frac{\partial c}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 c}{\partial S^2}\right] dt + \frac{\partial c}{\partial S} dS \quad . \tag{11.16}$$

Substituting the last result in Eq. 11.15 and noticing that $\frac{\partial c}{\partial S} = \Delta$ we have

$$dV = -\frac{1}{\Delta}\left[\frac{\partial c}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 c}{\partial S^2}\right] dt \quad . \tag{11.17}$$

If the portfolio is really risk-free, its yield must be equal to the fixed interest rate r (yield = $V \cdot r \cdot t$) and thus its instantaneous change is $V \cdot r \cdot dt$:

$$-\frac{1}{\Delta}\left[\frac{\partial c}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 c}{\partial S^2}\right] dt = \left(S - \frac{1}{\Delta}c\right) r \cdot dt \quad . \tag{11.18}$$

Simplifying dt and rearranging the terms with the definition of Δ we obtain the Black-Scholes equation

$$\frac{\partial c}{\partial t} = rc - rS\frac{\partial c}{\partial S} - \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 c}{\partial S^2} \quad . \tag{11.19}$$

which is a **parabolic equation**. The equation can be solved analytically or with a numerical scheme. In more complex cases, where volatility or other parameters are not constant (and maybe even follow a different stochastic process), Monte Carlo methods can be used.

11.3 Analytic Solution

The Black-Scholes equation has an analytic solution which can be derived in different ways. One possibility is to reduce it to a parabolic equation form where the solution is known. Introducing the substitution

$$c(S,t) = e^{r(t-T)}Y \quad , \tag{11.20}$$

where the new function Y has the following dependency

$$Y(x,y) = \tag{11.21}$$
$$Y\left[\frac{2}{\sigma^2}(r-\frac{\sigma^2}{2})(\ln\frac{S}{X}-(r-\frac{\sigma^2}{2})(t-T)) \quad , \quad -\frac{2}{\sigma^2}(r-\frac{\sigma^2}{2})^2(1-T)\right]$$

With this substitution, dividing by $e^{r(t-T)}$ and then by $2\frac{r}{\sigma^2}-1$ and $-r+\sigma^2/2$, the Black-Scholes equation becomes

$$\frac{\partial Y}{\partial y} = \frac{\partial^2 Y}{\partial x^2} \quad , \tag{11.22}$$

which is the diffusion equation with unitary diffusion coefficient and it has a known Gaussian solution. With the used substitution, the value of the option becomes

$$Y(u,0) = \begin{cases} X(e^{u/(2r/\sigma^2)}-1) & u \geq 0 \\ 0 & u < 0 \end{cases} \tag{11.23}$$

where $u = (\frac{2r}{-1})\ln\frac{S}{X}$.
Rewriting the parabolic equation problem with the new variable u and calling z the other one we have

$$Y(u,0) = \begin{cases} \frac{\partial Y}{\partial z} = \frac{\partial^2 Y}{\partial u^2} \\ Y(u,0) = g(u) \end{cases} . \tag{11.24}$$

which has the Gaussian solution

$$Y(u,z) = \int_{-\infty}^{+\infty} \frac{1}{\sqrt{4\pi z}} e^{-(u-\xi)^2/4z} g(\xi) d\xi \quad . \tag{11.25}$$

Setting $q = (\xi - u)/\sqrt{2z}$ and using Eq. 11.24, the solution becomes

$$Y(u,z) = \int_{-u/\sqrt{2z}}^{+\infty} e^{-q^2/2} X\left(e^{(\sqrt{2z}q+u)/(2r/\sigma^2-1)}-1\right) dq \quad . \tag{11.26}$$

Substituting into the first variable redefinition (Eq. 11.20) we obtain

$$c(S,t) = SN(d_1) - Xe^{-r(T-t)}N(d_2) \quad , \tag{11.27}$$

with

$$d_1 = \frac{\ln \frac{S}{X} + (r + \frac{1}{2}\sigma^2)(T-t)}{\sigma\sqrt{T-t}} \tag{11.28}$$

$$d_2 = d_1 - \sigma\sqrt{T-t} \quad . \tag{11.29}$$

The function N is the distribution function of a standard normal distribution

$$N(y) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{y} e^{-\frac{1}{2}x^2} dx \quad . \tag{11.30}$$

The code 11.1 implements the analytic solution of the Black-Scholes equation for the european call and put options and plots their value as function of the underlying stock. From this point on, many similar finantcial instruments can be modeled along similar lines. One example are american options, where the option can be used at any time point before the expiry date. Other finantial instruments can be reduced to the option case and other variables like volatility and interest rate can be made stochastic. Most cases cannot be treated analitycally and a simulation or numeric approach must be followed.

```python
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt

N = norm.cdf

def BS_call(S, X, T, r, sigma):
    d1 = (np.log(S/X) + (r + sigma**2/2)*T) / (sigma*np.
    sqrt(T))
    d2 = d1 - sigma * np.sqrt(T)
    return S * N(d1) - X * np.exp(-r*T)* N(d2)

def BS_put(S, X, T, r, sigma):
    d1 = (np.log(S/X) + (r + sigma**2/2)*T) / (sigma*np.
    sqrt(T))
    d2 = d1 - sigma* np.sqrt(T)
    return X*np.exp(-r*T)*N(-d2) - S*N(-d1)

X = 100      #strike price
r = 0.1      #interest rate
T = 1        #maturity
sigma = 0.3 #volatility

S = np.arange(60,140,0.1)

calls = [BS_call(s, X, T, r, sigma) for s in S]
puts = [BS_put(s, X, T, r, sigma) for s in S]

plt.plot(S, calls, label='European Call Value')
plt.plot(S, puts, label='European Put Value')

plt.xlabel('$S_0$')
plt.ylabel('Option Value')
plt.legend()
plt.show()
```

Listing 11.1: Analytic Solution of the BS Equation

11.4 Numerical schemes for the Black-Scholes Equation

We would like to find a numercal scheme for solving the Black-Scholes (or similar) equation. The stock value S will be equivalent to the x variable treated in the section before. Defining a discretized grid, $S_i = ih$ (i=0,..,N) and $t_j = jk$ (j=0,..,M), we would like to find a solution to Eq. 11.19. Since at this point we do not differentiate between call or put options, we will call the generic option price $v = v(S,t)$. Rearranging the terms, and using the usual notation for the derivatives, the equation to solve is

$$\partial_t + rS\partial_S v + \frac{1}{2}\sigma^2 S^2 \partial_{SS} v = rv \quad . \tag{11.31}$$

The time derivative can be approximated as

$$\partial_t v = \frac{v_{i,j+1} - v_{i,j}}{k} \quad , \tag{11.32}$$

while for the first derivative with respect to S we use the central scheme

$$\partial_S v = \frac{v_{i+1,j} - v_{i-1,j}}{2h} \quad . \tag{11.33}$$

The second derivative is as usual

$$\partial_{SS} v = \frac{v_{i+1,j} - 2v_{i,j} + v_{i-1,j}}{2h} \quad . \tag{11.34}$$

We remark again that for central differences (first and second derivatives) the error is of order $O(h^2)$ while for forward/backward differences it is $O(h)$.
Defining

$$\alpha_i = \left(\frac{r}{2} + \frac{\sigma^2}{2} i^2\right) k \quad , \tag{11.35}$$

$$\beta_i = (r + \sigma^2 i^2) k \quad , \tag{11.36}$$

$$\gamma_i = \left(\frac{\sigma^2}{2} i^2 - \frac{r}{2} i\right) k \quad , \tag{11.37}$$

and substituting the discretizations in Eq. 11.31 we obtain the finite difference equation

$$-\alpha_i v_{i+1,j} + (1 + \beta_i) v_{i,j} - \gamma_i v_{i-1,j} = v_{i,j+1} \quad . \tag{11.38}$$

This is an **implicit** method, since knowing the boudary values of $v_{i,M}$ for each i and $v_{N,j}$ for each j, it is possible to calculate the interior grid points. In order to solve the implicit scheme the whole system of equations must be solved for each time step.
It is also possible to derive an **explicit** method approximating the first derivative with a central scheme centered on the time step $j+1$:

$$\partial_S v = \frac{v_{i+1,j+1} - v_{i-1,j+1}}{2h} \quad . \tag{11.39}$$

The corresponding numerical scheme is

$$\alpha_i v_{i+1,j+1} + (1+\beta_i) v_{i,j+1} + \gamma_i v_{i-1,j+1} = v_{i,j} \quad . \tag{11.40}$$

In this method, the values of v at the time j depend to the values at time $j+1$. Since the value of v is known at the final times, all the values can be calculated with a procedure which goes backwards in time.
When we explained the Crank-Nicolson scheme, we considered a point between two time steps for improving the time accuracy. A similar procedure can be done here taking the **weighted average** of the implicit and explicit schemes. Taking $\theta \in [0,1]$ as the weight we obtain

$$\begin{aligned}(1-\theta)\alpha_i v_{i+1,j+1} + [1-(1-\theta)\beta_i]\, v_{i,j+1} + (1-\theta)\gamma_i v_{i-1,j+1} = \\ -\theta\alpha_i v_{i+1,j} + (1+\theta\beta_i) v_{i,j} - \theta\gamma_i v_{i-1,j} \quad .\end{aligned} \tag{11.41}$$

For $\theta = 0$ and $\theta = 1$ we recover the explicit and the implicit method, respectively. The choice $\theta = 1/2$ corresponds to the **Crank-Nicolson** method, where also the error in the time discretization is reduced to $O(k^2)$.

Code 11.2 implements the explicit finite difference scheme for solving the Black-Scholes equation in the case of a european call option. For too large time steps, the explicit scheme becomes unstable. The stability condition is

$$\frac{\sigma^2 S^2 k}{h^2} \leq 1 \quad . \tag{11.42}$$

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

#Forward-time, central-space EXPLICIT scheme
def FTCS_European_Call(C, N, M, dt, r, sigma):

    i = numpy.arange(1,M)
    for n in range(N): #Loop over time steps
        #[1:-1]: all elements but first and last
        C[1:-1]=\
        0.5*(sigma**2 * i**2 * dt - r*i*dt)*C[0:-2] \
        + (1 - sigma**2* i**2 *dt - r*dt)*C[1:-1] \
        + 0.5*(sigma**2 * i**2 * dt + r*i*dt)*C[2:]
    return C

#Call option features
T = 0.25      # expiry time
r = 0.1       # interest rate
sigma = 0.4   # volatility
X = 10.       # strike price
S_max = 4*X   # upper bound of the stock price

#Numerical scheme
N = 2000        # number of time steps
M = 200         # number of space grids
dt = T/N        # time step
s = numpy.linspace(0, S_max, M+1) # grid for S

# Initial condition
C = s - X

#Boundary condition, clip to zero outside range
C = numpy.clip(C, 0, S_max-X)
C0 = FTCS_European_Call(C, N, M, dt, r, sigma)

plt.plot(C)
plt.xlabel("Stock Price S"); plt.ylabel("Option Price C"
    )
plt.show()
```

Listing 11.2: Explicit Scheme for the Black-Scholes Equation

11.5 Monte Carlo Approach

The stochastic differential equation for the geometric stochastic process can be solved analytically, yielding the solution

$$S(t) = S(0)e^{(r-\frac{\sigma^2}{2})t+\sigma\sqrt{t}Z} \quad , \tag{11.43}$$

where Z is a random number sampled from a Gaussian distribution with zero mean and unit root mean square ($\mathcal{N}(0,1)$). For calculating the value of the option (let's consider an European call) at the initial time, we can discount back the final value with the interest rate

$$S(0) = e^{-rT}S(T) \quad . \tag{11.44}$$

We can simulate different realizations of S(T) using Eq. 11.43 generating N random numbers from $\mathcal{N}(0,1)$ and then calculate the initial prices $S_i(0)$ (i=1,..,N) with Eq. 11.44.
The Monte Carlo value V_{MC} of the option price is thus the average

$$V_{MC} = \frac{1}{N}\sum_{i=1}^{N} S_i(0) \quad , \tag{11.45}$$

and the error on the estimate can be evaluated with the variance

$$\sigma = \frac{\sqrt{\frac{1}{N}\sum_i (S_i - V_{MC})^2}}{\sqrt{N}} \quad . \tag{11.46}$$

Fig. 11.1 shows the results of the Monte Carlo calculation (blue markers). Every marker is the result of a calculation with a fixed number of samples N. The red lines represent the estimate of the variance. The lines are calculated adding and subtracting the result of Eq. 11.46 multiplied by a factor of 3 in order to obtain the so-called 3σ belt around the data. The lines show the expected $1/\sqrt{N}$ evolution of the Monte Carlo error.
This method can be extended to situations where the parameters of the Black-Scholes model are not constant or following a stochastic process themselves. Another modification is the adoption of a stochastic process different than the geometric one. In these cases, an analytic solution of the stochastic equation might not be available, and the stochastic process has to be fully simulated, for example with an Euler-like procedure. For example, a simple numerical scheme for simulating a stochastic process is the **Euler-Maruyama method**

$$S_{i+1} = a(S_i, t_i)\Delta t + b(S_i, t_i)\sigma\sqrt{t_i}z_i \quad , \tag{11.47}$$

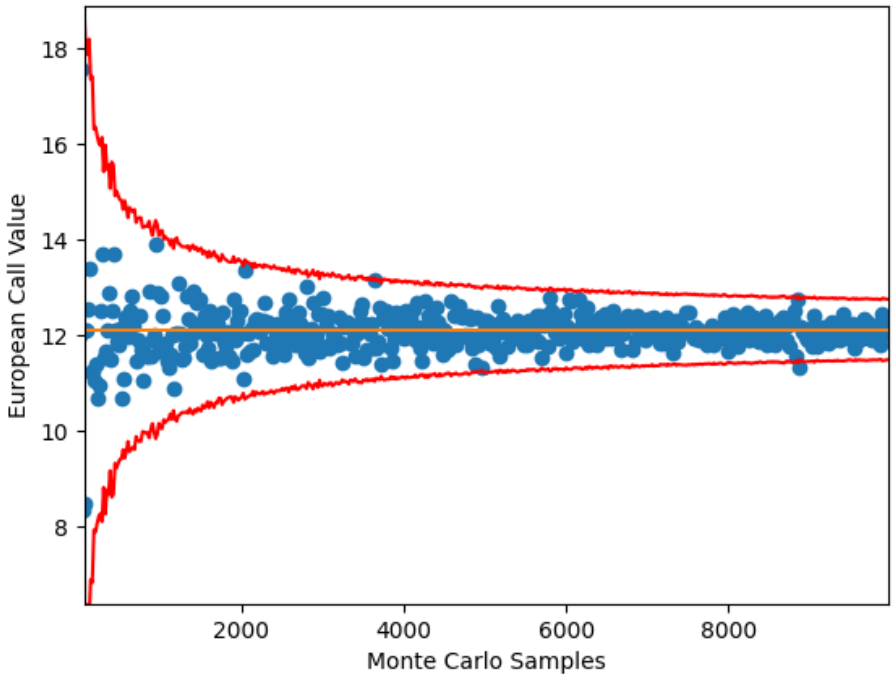

Figure 11.1: Monte Carlo estimation of an European call with an increasing number of samples (blue markers). The red lines represent the 3σ contours calculated with Eq. 11.46 and the orange horizontal line the analytic calculation.

where $z_i \sim \mathcal{N}(0,1)$. More precise schemes exist if higher order expansions of the derivatives are used. For example, in the **Milstein method**, an expansion up to second order is used.

```
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt

r     = 0.1 #risk-free interest rate
S0    = 100 #starting stock value
sigma = 0.3 ; X=110 #volatility;strike
T = 1 ; t = 0 #duration ; current time
np.random.seed(1) #set the random seed
N = 1000000       #MC samples

def EuropeanCallValue(S0, X, r, sigma, Z, T):
    ST = S0*np.exp((r-sigma**2/2)*T+sigma*np.sqrt(T)*Z)
    return np.exp(-r*T)*np.maximum(ST-X, 0)

def EuropeanCallMC(S0, X, r, sigma, T, N):
    V = np.zeros(N) ; Z = norm.rvs(size=N)
    V = EuropeanCallValue(S0,X,r,sigma,Z,T)
    Vmean = np.mean(V); Verr = np.std(V)/np.sqrt(N)
    return Vmean,Verr

#Analytic Solution
d_1 = (np.log(S0/X)+(r+sigma**2/2)*(T-t))/(sigma*np.sqrt
    (T-t))
d_2 = d_1 - sigma*np.sqrt(T-t)
analytic=S0*norm.cdf(d_1)-X*np.exp(-r*(T-t))*norm.cdf(
    d_2)

#Plotting MC solution vs number of samples
Nmax = 10000 ; step = 20
Vmean = np.zeros((int)(Nmax/step))
Verr = np.zeros((int)(Nmax/step))
n = np.zeros((int)(Nmax/step)) ; i=0
for N in range(1,Nmax,step):
    Vmean[i],Verr[i] = EuropeanCallMC(S0,X,r,sigma,T,N)
    n[i] = N ; i += 1

plt.plot(n,Vmean,marker='o',linestyle='none')
plt.plot(n,analytic+Verr*3,'r-',marker='')
plt.plot(n,analytic-Verr*3,'r-',marker='')
plt.plot(n,(np.zeros((int)(Nmax/step)) + 1)*analytic)
plt.xlabel('Monte Carlo Samples')
plt.ylabel('European Call Value')
plt.show()
```

Listing 11.3: Black-Scholes Monte Carlo

CHAPTER 12 | Hyperbolic Equations

12.1 Introduction

Hyperbolic equations are also known as wave equations, since they describe various types of waves, including electromagnetic waves. The simplest hyperbolic equation is

$$\phi_{xx} - \phi_{yy} = 0 \quad , \tag{12.1}$$

where in many applications the variable y is identified with the time t. Similarly to the parabolic (heat) equation, there are two studied problems for the hyperbolic equation: the **initial value problem (or Cauchy problem)** and the **initial boundary problem**.

12.1.1 The Cauchy Problem

The initial value or Cauchy problem consists in finding the function $\phi(x,t)$ satisfying the hyperbolic equation which is defined and continuous in $(-\infty,\infty)$ and $t > 0$ and satisfying the *initial conditions*

$$\begin{cases} \phi(x,0) & = f_1(x) \quad , \quad -\infty < x < \infty \\ d\phi/dt(x,0) & = f_2(x) \quad , \quad -\infty < x < \infty \quad . \end{cases} \tag{12.2}$$

The Cauchy problem is defined on a semi-infinite strip of the (x,t) plane and it is possible to solve it with an analytic formula due to D'Alembert. Introducing the rotation of axes

$$\begin{cases} \xi = x + t \quad , \\ \psi = x - t \quad , \end{cases} \tag{12.3}$$

the wave equation transforms to

$$\frac{d^2\phi}{d\xi d\psi} = 0 \quad . \tag{12.4}$$

Integrating the last equation with respect to ψ,

$$\frac{d\phi}{d\xi} = F_1(\xi) \quad , \tag{12.5}$$

where F_1 is some differentiable function. Integrating now with respect to ξ,

$$\phi = \int_0^{\xi} F_1(z)dz + G_2(\psi) \quad . \tag{12.6}$$

If we now define

$$G_1(\xi) = \int_0^{\xi} F_1(z)dz \quad , \tag{12.7}$$

we have that the solution must be of the general form

$$\phi = G_1(\xi) + G_2(\psi) = G_1(x+t) + G_2(x-t) \quad . \tag{12.8}$$

Since the functions G_1 and G_2 are differentiable, from the last form we can compute

$$\frac{d\phi}{dy} = \frac{\partial G_1}{\partial \xi}\frac{\partial \xi}{\partial t} + \frac{\partial G_2}{\partial \psi}\frac{\partial \psi}{\partial t} = \frac{\partial G_1}{\partial \xi} - \frac{\partial G_2}{\partial \psi} \quad . \tag{12.9}$$

From Eq. 12.8 and Eq. 12.9 and remembering the initial conditions, we have

$$\begin{array}{l} \phi(x,0) = G_1(x) + G_2(x) = f_1(x) \Rightarrow G_1'(x) + G_2'(x) = f_1'(x) \quad , \\ \frac{d\phi}{dt} = G_1'(x) - G_2(x) = f_2(x) \quad , \end{array} \tag{12.10}$$

where the "prime" indicated a derivative with respect of the only dependent variable. Solving the last equations for $G_{1/2}'$:

$$\begin{array}{l} G_1'(x) = \frac{1}{2}\left[f_1'(x) + f_2(x)\right] \\ G_2'(x) = \frac{1}{2}\left[f_1'(x) - f_2(x)\right] \quad , \end{array} \tag{12.11}$$

and integrating we have

$$\begin{array}{l} G_1(x) = \frac{1}{2}\left[f_1(x) + \int_0^x f_2(z)dz\right] \\ G_2(x) = \frac{1}{2}\left[f_1'(x) - \int_0^x f_2(z)dz\right] \quad . \end{array} \tag{12.12}$$

Remembering the general form Eq. 12.8, we can combine the last equations in

$$\phi(x,t) = \frac{1}{2}\left[f_1(x+y) + \int_0^{x+y} f_2(z)dz\right] + \left[f_1(x-y) + \int_0^{x-y} f_2(z)dz\right] \quad . \tag{12.13}$$

The last equation can be rewritten as

$$\phi(x,t) = \frac{1}{2}\left[f_1(x+y) + f_1(x-y) + \int_{x-y}^{x+y} f_2(z)dz\right] \quad , \qquad (12.14)$$

which is the **D'Alembert solution**. While the solution looks simple, the integral might not be analythical, but in that case it can be solved numerically. Given the existence of this solution, a numerical treatment of the Cauchy case will not be pursued.

12.1.2 The Initial Boundary Problem

In this case, a parameter $a > 0$ is specified, together with the continuous functions

$$\begin{cases} g_1(t) & , \quad t \geq 0 \\ g_2(t) & , \quad t \geq 0 \\ f_1(x) & , -\infty \leq x \leq \infty \\ f_2(x) & , -\infty < x < \infty \end{cases} \qquad (12.15)$$

The problem consists in finding a function $\phi(x,t)$ satisfying the hyperbolic equation and which is continuous in $t \geq 0$ and $0 \leq x \leq a$ and the *initial and boudary conditions*

$$\textbf{Initial Conditions:} \begin{cases} \phi(x,0) = f_1(x) & , \quad 0 \leq x \leq a \\ d\phi/dt(x,0) f_2(x) & , \quad 0 < x < a \end{cases} \qquad (12.16)$$

$$\textbf{Boundary Conditions} \begin{cases} \phi(0,t) = g_1(t) & , \quad t \geq 0 \\ \phi(a,t) = g_2(t) & , \quad t \geq 0 \end{cases} \qquad (12.17)$$

In order to avoid "corner" discontinuities, it is assumed

$$\begin{cases} g_1(0) = f_1(0) & , \\ g_2(0) = f_1(a) & . \end{cases} \qquad (12.18)$$

The solution to boundary conditions problems can be given as series of functions. Most of the times, non-linear cases have to be treated numerically. In the next sections, we will look at some algorithms for the solution of this problem.

12.2 Explicit Method for the Initial Boundary Problem

Considering a grid of points with width $h = a/n$ for the space coordinate x and $k = b/m$ for the "time" coordinate t (we are assuming $0 \leq x <\leq a$ and $0 \leq t <\leq b$), where n and m are choosen number of points in the two directions, we can discretize the hyperbolic equation with central differences

$$\phi_{xx} = \phi_{tt} \Rightarrow \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{h^2} = \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{k^2} \quad . \tag{12.19}$$

Rearraging the terms

$$\phi_{i,j+1} = 2\phi_{i,j} - \phi_{i,j-1} + \frac{k^2}{h^2}\left(\phi_{i+1,j} - \phi_{i,j} + \phi_{i-1,j}\right) \quad . \tag{12.20}$$

The last equation represents an explicit formula for $\phi_{i,j+1}$, but there is still a problem. It is possible to generate $\phi_{i,j+1}$ knowing (fixing a "space" i) the function at the two previous "times" j and $j-1$. This means that we cannot apply the equation to the first time value, but only starting from the second. We solve this issue approximating the first time value with a Taylor series

$$\phi_{i,1} \approx \phi_{i,0} + k\phi_t(x_i, 0) + \frac{k^2}{2}\phi_{tt}(x_i, 0) \quad , \tag{12.21}$$

which can be rewritten as

$$\phi_{i,1} \approx \phi_{i,0} + k\phi_t(x_i, 0) + \frac{k^2}{2}\phi_{xx}(x_i, 0) \quad , \tag{12.22}$$

using the wave equation. Introducing now a finite difference scheme, and remembering the boudary conditions $\phi(x, 0) = f_1(x), \phi_t(x, 0) = f_2(x)$ the last equation becomes

$$\phi_{i,1} \approx f_1(x_i) + kf_2(x_i) + \frac{k^2}{2}\frac{f_1(x_{i+1}) - 2f_1(x_i) + f_1(x_{i-1})}{h^2} \quad . \tag{12.23}$$

Equations 12.20 and 12.23 define an explicit finite difference approximation scheme for the hyperbolic equation with initial boundary conditions.
A general condition for the stability of the method is given by the **Courant-Friedrichs-Lewy stability condition**, which results in

$$k \leq h \quad , \tag{12.24}$$

thus "temporal" steps should be smaller or equal to spatial steps.

12.3 Implicit Method for the Initial Boundary Problem

The following implicit method results stable for any choice of the steps h and k, but it is implicit and thus requires the solution of a linear system, which in this specific case is tri-diagonal.
The idea is to discretize the "spatial" second derivative of the hyperbolic equation with the average of two second derivatives at two symmetric times around the reference point (x_i, t_j)

$$\phi_{xx} \approx \frac{1}{2}\left[\frac{\phi_{i+1,j+1} - 2\phi_{i,j+1} + \phi_{i-1,j+1}}{h^2} + \frac{\phi_{i+1,j-1} - 2\phi_{i,j-1} + \phi_{i-1,j-1}}{h^2}\right] \quad . \tag{12.25}$$

The "time" second derivative keeps the usual discretization

$$\phi_{yy} \approx \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{k^2} \quad . \tag{12.26}$$

Substituting the last two discretizations into the wave equation $\phi_{xx} - \phi_{tt} = 0$ yields

$$\begin{aligned} \phi_{i-1,j+1} - 2\left(1 + \frac{h^2}{k^2}\right)\phi_{i,j+1} + \phi_{i+1,j+1} = \\ -\phi_{i-1,j-1} - \phi_{i+1,j-1} - 4\frac{h^2}{k^2}\phi_{i,j} + 2\left(1 + \frac{h^2}{k^2}\right)\phi_{i,j-1} \quad . \end{aligned} \tag{12.27}$$

The last equation represens a linear system of $n-1$ equations with the $n-1$ unknowns $\phi_{i,j+1}$ ($i = 1, 2, .., n-1$), if the boundary conditions are taken into account. This system is tri-diagonal and diagonally dominant with negative diagonal elements and non-negative off-diagonal elements, thus it is always possible to find an unique solution.
The solution algorithm proceeds as follows:

1. Choose values for h and k
2. Use the boundary and initial conditions: $\phi_{i,0} = f_1(x_1), \phi_{0,j} = g_1(t), \phi_{n,j} = g_2(t)$ with $i = 0, 1, 2, .., n$, and $j = 1, 2, .., m$.
3. At the point $t_1 = k$, compute $\phi_{i,1}$ ($i = 1, 2, .., n-1$) with the expansion formula of Eq. 12.23.
4. At each time t_j ($j = 1, 2, .., m-1$) solve the tri-diagonal system 12.27 finding the values $\phi_{i,j+1}$ ($i = 1, 2, .., n-1$).

12.4 The Lax-Wendroff Method

We describe here an explicit solution method for a particular hyperbolic equation (or systems of equations) which often appears in physical problems related to gases and which is relevant in applications The system we consider is in *conservative form*

$$\frac{\partial \phi}{\partial t} + \frac{\partial F(\phi)}{\partial x} = 0 \quad , \tag{12.28}$$

since the function F represents conseved quantities (like mass or energy).
The problem is to determine the functions v and u at every point (x,t).
We consider the Taylor expansion around the time point $j + 1/2$ fixing the space point $i + 1/2$

$$\phi_{i+1/2,j+1/2} \approx \phi_{i+1/2,j} + k\frac{\partial \phi(x_{i+1/2}, t_j)}{\partial t} \quad . \tag{12.29}$$

Substituting the hyperbolic equation

$$\phi_{i+1/2,j+1/2} \approx \phi_{i+1/2,j} - k\frac{\partial F}{\partial x}|_{i+1/2,j} \quad . \tag{12.30}$$

If we now introduce a finite difference scheme for the spatial derivative of F symmetric around $x_{i+1/2}$ and replace $\phi_{i+1/2,j}$ with the average of the function again between two symmetric points around $x_{i+1/2}$ we obtain the first step of the **Lax-Wendroff** scheme

$$\phi_{i+1/2,j+1/2} = \frac{\phi_{i+1,j} + \phi_{i,j}}{2} - \frac{k}{2h}\left(F(\phi_{i+1,j}) - F(\phi_{i,j})\right) \quad . \tag{12.31}$$

An interesting observation is that rearranging the terms involving ϕ, a difference scheme of the second spatial derivative can be constructed: this physically means introducing a diffusion (or friction) term into the system which might explain the stability of the method.
The second and last step, is to determine ϕ on a grid of points of the form x_i, t_{j+1} obtaining

$$\phi_{i,j+1} = \phi_{i,j} - \frac{k}{h}\left(F(\phi_{i+1/2,j+1/2}) - F(\phi_{i-1/2,j+1/2})\right) \quad . \tag{12.32}$$

The Lax-Wendroff method is (Neumann) stable if $|k/h| \leq 1$.

CHAPTER 13 | The Navier-Stokes Equations

13.1 Introduction

The Navier-Stokes Equations govern the motion of all the fluids in the hydrodynamical approximation. These equation are particularly complex since they are non-linear and there are not existence and uniqueness theorems availavble in general for characterizing the solutions. The numerical treatment of these complicated equations is thus central in many applications.
The Navier-Stokes equation for a Newtonian incompressible fluid of density ρ and viscosity ν is

$$\rho\left(\frac{\partial}{\partial t}+\vec{v}\cdot\vec{\nabla}\right)=-\nabla\vec{P}+\nu\nabla^2\vec{v} \quad , \tag{13.1}$$

where $\vec{v}$ is the velocity, and P the pressure. Physically, the last equation represents momentum conservation.
Mass conservation requires

$$\frac{\partial\rho}{\partial t}=-\nabla\cdot\vec{v} \quad . \tag{13.2}$$

and incompressibility results in $\nabla\cdot\vec{v}=0$. Newtonian fluids are fluids for which the viscous stresses arising from its flow are at every point linearly correlated to the local strain rate. In other words, the viscosity ν is a constant and this simplifies the stress tensor of the fluid. The equation is already complicated with the latter approximations and we will use them in the following numerical treatments. Moreover, the Navier-Stokes equations cannot be classified into the three categories (elliptic, parabolic, hyperbolic) which we discussed in the previous chapters.

13.2 2-Dimensional Flow

For simplicity, here we will consider only the Navier-Stokes equations in two dimensions, which is already a challenging problem. Still, the two-dimensional treatment has many applications, for example in the dynamics of shallow waters. Writing Eq. 13.1 explicitly in two dimensions we have

$$\begin{aligned} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} &= -\frac{\partial P}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \\ \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} &= -\frac{\partial P}{\partial y} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) \quad , \end{aligned} \tag{13.3}$$

where v and u are the velocity components along x and y, respectively.
If the fluid is incompressible, mass conservation (Eq. 13.2) implies the additional constraint for the velocity components

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \quad . \tag{13.4}$$

The three latter equations form a system where the unknowns are the functions $u(x,y,t)$, $v(x,y,t)$, $P(x,y,t)$. The **initial-boundary condition problem** for the 2-dimensional Navier-Stokes flow is defined as follows.

- Find $u(x,y,t)$, $v(x,y,t)$, $P(x,y,t)$ on a domain Ω with border Γ
- with initial conditions: $u(x,y,0) = u_0(x,y)$,$v(x,y,0) = v_0(x,y)$ continuous with $(x,y) \in \Omega$,
- boundary conditions: $u(x,y,t) = u_\gamma(x,y,t)$, $v(x,y,t) = v_\gamma(x,y,t)$, continuous with $(x,y) \in \Gamma$ and $t \geq 0$.

13.3 A general Finitie-Difference Scheme

We indicate with indices i,j the discrete spatial coordinates of the finite difference scheme, and with the index k the time, adopting the notation $v^k_{i,j}$ ($i = 1,2,..n$, $j = 1,2,..,m$) for the value of the velocity v (u,v in the two dimensional case). The pressure follows the same notation: $P^k_{i,j}$. The space grid has steps Δx, while the time grid has steps Δt, so that $t_k = k\Delta t$.

13.3.1 Boundary conditions

The velocities are defined at the middle point of the border of a "cell", while the pressure is defined at its centre.
The initial conditions at $t_0 = 0$ are

$$\begin{aligned} u^0_{i+1/2,j} &= u_0(x_i, y_{j-1/2}) \quad , i = 0, 1, .., n \quad , \quad j = 1, 2, .., m \\ v^0_{i,j+1/2} &= v_0(x_{i-1/2}, y_j) \quad , i = 1, 2, .., n \quad , \quad j = 0, 1, .., m \quad , \end{aligned} \tag{13.5}$$

In this way, we know the *tangential* (u component) and the *normal* (v component) valocities at time zero at each point of the cell.
Note that there are no initial conditions for the pressure and they have to be derived later.
Now we can make use of the boundary conditions and we start with the **top and bottom boundaries for the tangential velocities**. Since we decided to define the velocity components on the "half points" of the grid (i+1/2, j+1/2, ...) and the boundary conditions are instead known on the borders (i=0, j=0, i=a, j=b), we can consider the averages

$$\begin{aligned} \frac{u^k_{i+1/2,0} + u^k_{i+1/2,1}}{2} &= u_\Gamma(x_i, 0, t^k) \quad , \quad i = 1, 2, .., n-1 \quad , \\ \frac{u^k_{i+1/2,m} + u^k_{i+1/2,m+1}}{2} &= u_\Gamma(x_i, b, t^k) \quad , \quad i = 1, 2, .., n-1 \quad , \end{aligned} \tag{13.6}$$

from which we have

$$u^k_{i+1/2,0} = 2u_\Gamma(x_i, 0, t_k) - u^k_{i+1/2,1} \quad , \tag{13.7}$$

$$u^k_{i+1/2,m+1} = 2u_\Gamma(x_i, b, t_k) - u^k_{i+1/2,m} \quad . \tag{13.8}$$

The last formulas define also the initial conditions which in Eq. 13.5 were not fully determined. For the **left and right boundaries for the tangential** components we have

$$v^k_{0,j+1/2} = 2v_\Gamma(0, y_j, t_k) - v^k_{1,j+1/2} \quad , \tag{13.9}$$

$$v^k_{n+1,j+1/2} = 2v_\Gamma(a, y_j, t_k) - v^k_{n,j+1/2} \quad . \tag{13.10}$$

For the **normal velocity boundary conditions** at time t_{k+1} we take

$$\begin{aligned} u^{k+1}_{1/2,j} &= u_\Gamma(0, y_{j-1/2}, t_{k+1}) \quad , \quad u^{k+1}_{n+1,j} = u_\Gamma(a, y_{j-1/2}, t_{k+1}) \quad , j = 1, 2, .., m \\ v^{k+1}_{i,1/2} &= v_\Gamma(0, x_{i-1/2}, t_{k+1}) \quad , \quad v^{k+1}_{i,m+1/2} = v_\Gamma(x_{i-1/2}, b, t_{k+1}) \quad i = 1, 2, .., n \end{aligned} \tag{13.11}$$

13.3.2 Numerical approximation scheme

We now introduce an approximation scheme for Eq. 13.3 starting with the first one

$$\frac{u^{k+1}_{i+1/2,j} - u^k_{i+1,j}}{\Delta t} + u^k_{i+1/2,j}\frac{u^k_{i+3/2,j} - u^k_{i-1,j}}{2\Delta x} + \bar{v}^k_{i+1/2,j}\frac{u^k_{i+1/2,j+1} - u^k_{i+1/2,j-1}}{2\Delta y} = -\frac{P^k_{i+1,j} - P^k_{i,j}}{\Delta x} + \nu\left(\frac{u^k_{i+3/2,j} - 2u^k_{i+1/2,j} + u^k_{i-1/2,j}}{(\Delta x)^2} + \frac{u^k_{i+1/2,j+1} - 2u^k_{i+1/2,j} + u^k_{i+1/2,j-1}}{(\Delta y)^2} + \right) . \tag{13.12}$$

The approximation scheme expresses velocities at time t_{k+1} and pressures at times t_k. Rearranging the terms and introducing the finite difference operator F, the last equation can be compactly rewritten as

$$u^{k+1}_{i+1/2,j} = Fu^k_{i+1/2,j} - \Delta t\frac{P^k_{i+1,j} - P^k_{i,j}}{\Delta x} \quad , \tag{13.13}$$

with $i = 1, 2, .., n-1$, $j = 1, 2, .., m$.
The same can be done with the second Navier-Stokes equation

$$v^{k+1}_{i,j+1/2} = Gv^k_{i,j+1/2} - \Delta t\frac{P^k_{i,j+1} - P^k_{i,j}}{\Delta y} \quad , \tag{13.14}$$

with another operator G. Explicitly, the two operators are

$$Fu^k_{i+1/2,j} = u^k_{i+1/2,j} - \Delta t \left[u^k_{i+1/2,j} \frac{u^k_{i+3/2,j} - u^k_{i-1,j}}{2\Delta x} + \bar{v}^k_{i+1/2,j} \frac{u^k_{i+1/2,j+1} - u^k_{i+1/2,j-1}}{2\Delta y} \right] + \nu \Delta t \left(\frac{u^k_{i+3/2,j} - 2u^k_{i+1/2,j} + u^k_{i-1/2,j}}{(\Delta x)^2} + \frac{u^k_{i+1/2,j+1} - 2u^k_{i+1/2,j} + u^k_{i+1/2,j-1}}{(\Delta y)^2} + \right) \quad , \tag{13.15}$$

$$Gv^k_{i,j+1/2} = v^k_{i,j+1/2} - \Delta t \left[\bar{u}^k_{i,j+1/2} \frac{v^k_{i+1,j+1/2} - v^k_{i-1,j+1/2}}{2\Delta x} + v^k_{i,j+1/2} \frac{v^k_{i,j+3/2} - v^k_{i,j-1/2}}{2\Delta y} \right] + \nu \Delta t \left(\frac{v^k_{i+1,j+1/2} - 2v^k_{i,j+1/2} + v^k_{i-1/2,j+1/2}}{(\Delta x)^2} + \frac{v^k_{i,j+3/2} - 2v^k_{i,j+1/2} + v^k_{i,j-1/2}}{(\Delta y)^2} + \right) \quad . \tag{13.16}$$

The operators contain the "averages" $\bar{u}$ and $\bar{v}$. The reason is that (for example) v is not defined at the center of a vertical side of a cell, thus it has to be calculated

$$\bar{v}^k_{i+1/2,j} = \frac{1}{4} \left(v^k_{i,j+1/2} + v^k_{i,j-1/2} + v^k_{i+1,j+1/2} + v^k_{i+1,j-1/2} \right) \quad . \tag{13.17}$$

The same happens at the center of an horizontal side for the u component:

$$\bar{u}^k_{i,j+1/2} = \frac{1}{4} \left(u^k_{i+1/2,j} + u^k_{i-1/2,j} + u^k_{i+1/2,j+1} + u^k_{i-1/2,j+1} \right) \quad . \tag{13.18}$$

The outlined scheme is *explicit*, but still it cannot be used right away, since the values of the pressure $P^k_{i,j}$ are not fixed. In order to complete the scheme, we have to take into account the incompressibility condition of Eq. 13.4 which in its discretized form is

$$\frac{u^{k+1}_{i+1/2,j} - u^{k+1}_{i-1/2,j}}{\Delta x} + \frac{v^{k+1}_{i,j+1/2} - v^{k+1}_{i,j-1/2}}{\Delta y} = 0 \quad , \tag{13.19}$$

for $i = 1, 2, .., n$ and $j = 1, 2, .., m$.
Inserting the boundary conditions Eq. 13.11 in the discretized Navier-Stokes equations 13.13, 13.14 and in the incompressibility condition Eq. 13.19, we obtain

- A linear algebric system of $2nm - n - m$ equations,
- nm unknown pressure values at time t_k,
- $m(n-1)$ unknown velocities of the u component at time t_{k+1}.
- $n(m-1)$ unknown velocities of the v component at time t_{k+1}.

Solving this system for the previously listed unknowns, provides values for the successive time step.

CHAPTER 14 | The Fourier Transform

14.1 Introduction

The (continuous) Fourier transform (FT) is defined as

$$f(\nu) = F_t[f(t)](\nu) = \int_{-\infty}^{+\infty} f(t)e^{-2\pi i\nu t}dt \quad . \tag{14.1}$$

The discrete version of the prevous integral is

$$F_n = \sum_{k=0}^{N-1} f_k e^{-2\pi ink/N} = \sum_{k=0}^{N-1} f_k w_N^{kn} \quad , \tag{14.2}$$

where we introduced the complex number $w_N^{kn} = e^{-2\pi ink/N}$ for simplifying the notation. In the discretized formula (discrete Fourier transform, DFT), the variable t (usually the time) becomes the discrete variable $t_k = k\Delta t$ with $k = 0, 1, .., N-1$, while the "frequency" ν becomes the discrete variable n.
The discrete *inverse* transform is

$$f_k = \frac{1}{N}\sum_{n=0}^{N-1} F_n e^{2\pi ink/N} \quad . \tag{14.3}$$

Sometimes, it is convenient to rewrite the DFT in matrix form

$$\begin{bmatrix} F_0 \\ F_1 \\ ... \\ ... \\ F_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & .. & 1 \\ 1 & w & w^2 & .. & w^{N-1} \\ 1 & w^2 & w^4 & .. & w^{2N-2} \\ .. & .. & .. & .. & .. \\ 1 & w^{N-1} & w^{2(N-1)} & .. & w^{(N-1)^2} \end{bmatrix} \times \begin{bmatrix} f_0 \\ f_1 \\ ... \\ ... \\ f_{N-1} \end{bmatrix} \tag{14.4}$$

From the last expression, it is clear that the DFT has a time-complexity which scales as $O(N^2)$. The FT is important since it recovers periodicities in the input

function and the strenght of each periodic component.
Since the transform contains a complex exponential, in general even if the input is real, the transform can be complex. If f_k are all real numbers the following relation holds

$$F_{N-n} = \bar{F}_n \quad , \tag{14.5}$$

which implies that an input periodic function will be transformed into peaks which are "doubled" into complex positive and negative frequancies.

14.2 Fast Fourier Transform

In the following, we describe one of the existing algorithms for reducing the computational complexity of the Fourier transform. These algorithms are known under the generic name of Fast Fourier Transform (FFT) methods. The following algorithm is called **Cooley-Tukey Algorithm**.
Defining the (discrete) FT as

$$X(k) = \sum_{n=0}^{N-1} x(n) w_n^{kn} \quad , \quad n,k = 0,..,N-1 \quad , \tag{14.6}$$

where

$$w_N^{kn} = e^{-\frac{2\pi}{N} j} \quad . \tag{14.7}$$

In the last expression, j is the imaginary unit and it will remain "hidden" into w so that we will use j as an index in the next calculations (committing a small abuse of notation for clarity).
We assume now that N can be written as the product of two other integers $N = n_1 n_2$.
Now (this is a major point) we rewrite the input vector x and the output vector X as *matrices* with the following definition

$$n = n_1 i + j \quad \text{with} \quad i = 0,..,n_2 - 1 \quad j = 0,..,n_1 - 1 \quad . \tag{14.8}$$

$$k = n_2 a + b \quad \text{with} \quad i = 0,..,n_1 - 1 \quad j = 0,..,n_2 - 1 \quad . \tag{14.9}$$

With the latter definitions, Eq. 14.6 becomes

$$X(n_2 a + b) = \sum_{i=0}^{n_2-1} \sum_{j=0}^{n_1-1} x(n_1 i + j) w_n^{(n_2 a+b)(n_1 i+j)} \quad . \tag{14.10}$$

In order to simplify the exponent of the complex number w, we use the observations $e^{-2\pi} = 1$ and

$$w_N^{n_1} = w_{n_1 n_2}^{n_1} = e^{-\frac{2\pi}{n_1 n_2} j n_1} \quad \Rightarrow \quad w_N^{n_1} = w_{n_2} \quad . \tag{14.11}$$

and Eq. 14.10 can be rewritten as (swapping sums and factors)

$$X(n_2 a + b) = \underbrace{\sum_{j=0}^{n_1-1} \underbrace{\sum_{i=0}^{n_2-1} x(n_1 i + j) w_{n_2}^{ib}}_{\text{For each j,} n_2-\text{lenght DFT}} \underbrace{w_N^{bj}}_{\text{Twiddle factor}} w_{n_1}^{aj}}_{\text{For each i,} n_1-\text{lenght DFT}} \quad . \tag{14.12}$$

After the last manipulation, Eq. 14.12 shows that the original DFT is reduced to two nested DFTs. The inner DFT has length n_2 (times the so-called *twiddle factors*) and when done, it is used n_1 times in the outer DFT, reducing the complexity of the full DFT.
The DFT implies naively N^2 computations. The scheme we derived previously instead is based on n_1 DFTs of length n_1 (complexity $n_1 \times n_2^2$) and n_2 DFTs of lenght n_2 (complexity $n_2 \times n_1^2$), plus N multiplications by the twiddle factors. In total, the number of computations is

$$n_1 n_2^2 + n_2 n_1^2 + N = N(n_1 + n_2 + 1) < N^2 \tag{14.13}$$

The last estimate shows the computational advantage of the FFT scheme with respect to the normal DFT. The steps of the Cooley-Tukey FFT algorithm are visualized schematically in Fig. 14.1 for $n_1 = 4$ and $n_2 = 3$. The reduction we proved can be extended breaking n_1 and n_2 further in their factors (if possible) and then repeating the Cooley-Tukey scheme until factorization is not possible anymore. The procedure then starts from the shorter DFTs and proceeds backwards by recursion to the larger DFTs. There are other FFT algorithms, for example, the **Good-Thomas algorithm** (sometimes called Prime Factor Algorithm) which does not involve the twiddle factors, thus eliminating N multiplications.

14.3 Aliasing

There are some problems connected to the discretized form of the FT. If we would like to sample a signal with a certain frequency, we cannot do it reliably if the

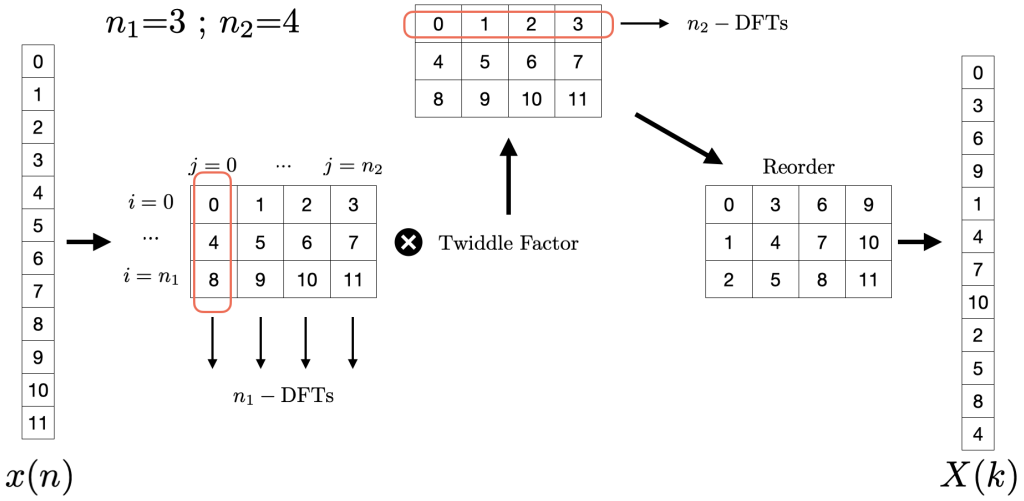

Figure 14.1: Steps involved in the calculation of the fast Fourier transform according to the Cooley-Tukey algorithm.

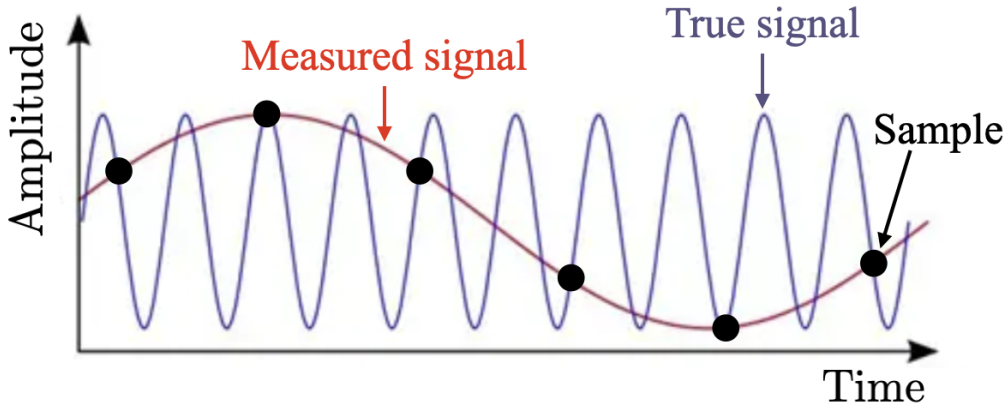

Figure 14.2: Example of the aliasing effect: the samples are measured with a frequency which is lower than the frequency of the signal. The result is the extraction of a wrong frequency.

sample frequency is smaller than the frequency of the signal itself. If this happens, we will measure a wrong frequency. This effect is called **aliasing** (See Fig. 14.2).
We can look at aliasing with a concrete example, where we try to sample a signal $A(t)$ which has a single frequency of f=4 Hz so $A(t) = \sin(2\pi f t)$. If we sample the signal N times with a lower frequency f_s, for example 3 Hz, we measure A(t) at times $t_k = k/f_s$ (k=0,1,..,N-1). The samples are

$$A(t_k) = \sin(2\pi f t_k) = \sin(8\pi \frac{k}{f_s}) = \sin(2\pi + \frac{2\pi k}{3}) = \sin(2\pi t_k) \quad . \tag{14.14}$$

The last step of the calculation shows that if we sample with 3 Hz frequency the 4 Hz signal, we will reconstruct a 1 Hz signal: we have aliasing.
If we try to sample at a higher rate, for example $f_s = 6\ Hz$ we will measure a 2 Hz signal (compute again Eq. 14.14) and aliasing is thus still there.
If we increase the sampling such that we match the signal frequancy $f_s = f$, it is easy to conclude that we will measure a frequency equal to zero.
This hints to the idea that $f_s > f$: in order to measure a certain frequency, we must sample at a higher frequency, but how much, al least?
The answer is provided by the **sampling theorem**:

$$f_s > f_{max} \quad , \tag{14.15}$$

that is: in order to avoid aliasing, we have to use a sampling frequency more than twice as high than the maximum frequency present in the signal. Thinking the other way around, if we sample with frequency f_s, this means that the maximum frequency we can reliably extract without introducing aliasing is $f_{max} = f_s/2$: this maximum frequency is also called **Nyquist frequency**.

List of Figures

3.1 Output of the previous code. The steps (bllue line) of the algorithm are plotted over a contour map of the quadratic function $L(x) = \frac{1}{2}x^T Ax - b^T x$. 38

4.1 Linear interpolation of a discrete function. 51
4.2 Parabolic interpolation of a discrete function. 52
4.3 Spline interpolation of a discrete function. 56
4.4 Linear fit of a set of 30 data points generated with function $f(x) = a + bx + RDM$ and parameters $a = 0$, $b = 1$. RDM is a random number in the range [-2,2]. 64

6.1 Graphical representation of the Romberg integration algorithm steps where two trapezoidal integrations on different grids are merged through Richardson extrapolation. 84
6.2 **(Left)** Generated 1000 random points in the unit square. The ratio between the square and circle area is equal to the ratio of the number of points landing in the circle (orange) and the total number of generated points. **Right:** Value of π estimated with Monte Carlo integration on a circle as a function of the number of generated points N. The error decreases as $1/\sqrt{N}$. 94

8.1 Dependence of the error in the presence of round-off as a function of the grid spacing h. The value $h_{optimal}$ is the best trade-off between discretization and round-off errors. 105
8.2 Integration of the Lotka-Volterra model for $a = b = c = d = 1$ with the RK4 algorithm. 114

8.3 **Left:** Percentage difference between the RK2 and RK3 solution for the two populations. **Right:** Phase space (X vs Y) of the two integration methods. 115
8.4 Damped harmonic oscillator for 3 values of γ (see legend) and $\omega_0^2 = 1$. 122

9.1 Visualization of the two-dimensional Dirichlet problem: the function $f(x, y) = -x^2 - y^2$ is defined on a set Ω and the border Γ is a square. On this square, a function $g(x, y) = 0$ defines the boundary values of the problem. 133
9.2 Graphical representation of the output matrix from code 9.1. 136

10.1 Turing instability patterns from the FitzHugh–Nagumo model.146

11.1 Monte Carlo estimation of an European call with an increasing number of samples (blue markers). The red lines represent the 3σ contours calculated with Eq. 11.46 and the orange horizontal line the analytic calculation. 160

14.1 Steps involved in the calculation of the fast Fourier transform according to the Cooley-Tukey algorithm. 180
14.2 Example of the aliasing effect: the samples are measured with a frequency which is lower than the frequency of the signal. The result is the extraction of a wrong frequency. . . 180

List of Tables

7.1 Coefficients of the numerical derivatives up to fourth order. 101