Statistical Analysis

Bianca Sofia Pinolini

31 January 2020

To carry out the exercises I developed codes using `Python 3.6.8`, `c` and `c++`, with the help of the scientific software toolkit `ROOT`. I used the open-source web application `Jupyter Notebook` and the platform to perform interactive data analysis in the CERN cloud `SWAN` (Service for Web based ANalysis).

Exercise 1

Task

Build a generator of random numbers distributed with the Landau distribution and discuss over the randomness tests usable for the generated data sample.

1.1 Theoretical Introduction

A random number generator (RNG) is a device that generates a patternless sequence of numbers. RNG's can be divided into two main cathegories:

- true hardware random-number generators (HRNG), which generate genuinely random numbers by measuring some physical phenomenon that is expected to be random, as the cosmic background radiation or a radioactive decay.
- pseudo-random number generators (PRNG) which generate numbers which look random, but are actually deterministic.

In our project we focus on the PRNG, which produce sequences not truly random, but completely determined by an initial value, called "seed". As a result, the entire seemingly random sequence can be reproduced if the seed value and the generating algorithm are known.

As these PRNG-produced sequences are not truly random, the need for a randomness test arises to analyse the distribution of a data set in order to see whether it can be described as random.

1.2 Middle-Square Method (1946)

First of all, we consider the John von Neumann's middle-square method, poor of quality and of historical interest only. To generate a sequence of n-digit numbers, a n-digit seed is required. Then the algorithm proceeds as follows:

1. The seed is squared;
2. If the result has fewer than 2n digits, leading zeroes are added to compensate;
3. The middle n digits of the result would be the next number in the sequence.

This process is then repeated to generate more numbers.
This algorithm has several weak points. First of all, the period (i.e. the number of loops) is limited, since for a generator of n-digit numbers, the period can be no longer than 8^n, with some exceptions. Moreover, if the value of n is odd then there will not necessarily be a uniquely defined "middle n-digits" to select from. It is acceptable to pad the seeds with zeros to the left in order to create an even valued n-digit, but in such way additional work must be done to have the algorithm work. Another weakness of the Middle-Square algorithm is that it breaks in several cases, for example:

- If the middle n digits are all zeroes, the generator then outputs zeroes forever.

- If the first half of a number in the sequence is zeroes, the subsequent numbers will be decreasing to zero.
- When $n = 4$, with the seeds 0100, 0540, 2500, 3792, 7600.
- When $n = 2$, none of the possible seeds generates more than 14 iterations without reverting to 0, 10, 50, 60, or a 24-57 loop.

We proved that the method gets stuck with the seed 0540 with a code shown in the *codes & further results* dispensation.
Obviously this is not an effective or efficient method, but it is very important from an historical point of view, and so it was worth implementing it just to see how it works.

1.3 Linear Congruential Generator (1958)

A linear congruential generator (LCG) is an algorithm that calculates a list of pseudo-random numbers through a discontinuous piece-wise linear equation. The method was invented by W. E. Thomson and A. Rotenberg and it represents one of the oldest and best-known PRNG algorithms.

The generator is defined by the following recurrence relation:

$$X_{n+1} = (a \cdot X_n + c) \mod m \tag{1.1}$$

where all the parameters are integer constants that specify the generator and are defined as follows:

- $a \in (0, m)$ is the multiplier;
- $X_0 \in [0, m)$ is the seed;
- $c \in [0, m)$ is the increment;
- $m \in (0, +\infty)$ is the modulus.

A benefit of LCG's is that with appropriate choice of parameters, the period is known and long. On the other hand, the algorithm is extremely sensitive to the choice of the parameters m and a.
The longest period obtainable is $\mathcal{T} = m$ and it is achieved imposing $c \neq 0$ and with the parameters chosen *ad hoc*.

For our LCG we choose the parameters used by the GNU C library `glibc`, namely:

$$m = 2^{32} \qquad a = 1103515245 \qquad c = 12345. \tag{1.2}$$

In this way we could produce 2^{32} pseudo-random numbers, but for our purpose we produce "only" $3 \cdot 10^5$ numbers.
To make a first check of the randomness of the generated data sample, we plot in Figure 1.1 the LCG-generated numbers in a histogram to see whether the distribution is uniform as it should be. Moreover, by way of example, we show in Figure 1.2 a set of 1000 LCG-generated space coordinates in a 3D scatter plot[1].

[1] This figure has been rotated to different angles to see whether some patterns would come up. We observed a uniformly distribution of points in any angle of analysis.

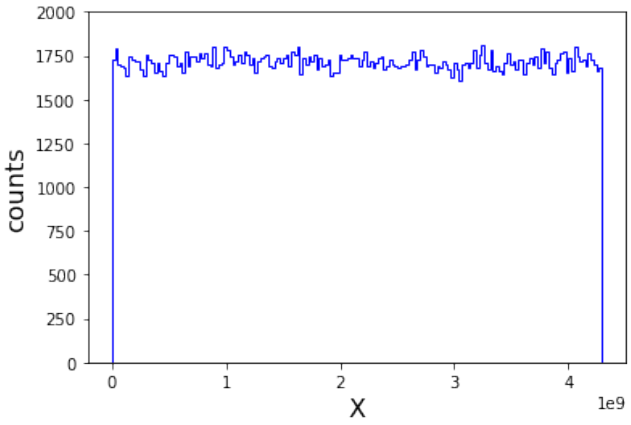


Figure 1.1: Histogram showing the LCG-generated data sample as first check of the randomness of the numbers.

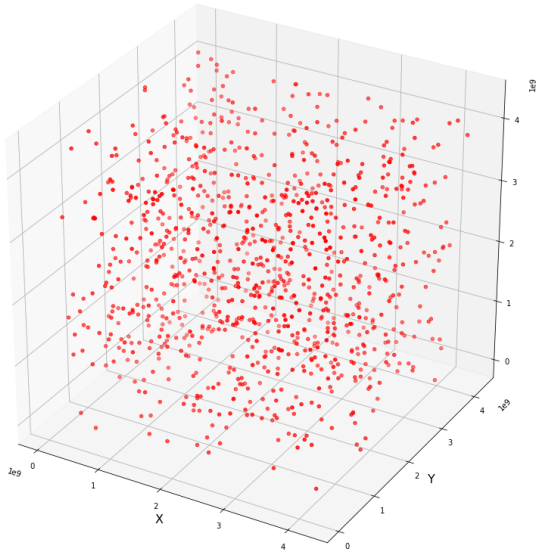


Figure 1.2: An example of 1000 space coordinates pseudo-randomly generated with the linear congruential generator implemented by us.

1.3.1 Randomness Test

Once we have understood how the LCG works and we have generated $3 \cdot 10^5$ pseudo-random numbers, we can use some randomness test to analyse the set of LCG-generated data in order to see whether it can be described as random.
For this purpose we use the `diehard` test, a battery of statistical tests for measuring the quality of a random number generator. They were developed by George Marsaglia over several years and first published in 1995.
As can be seen in the *codes & further results* section, we obtain that all the 113 tests in the library are passed, but 3 of these tests gave as result "weak". Since the results are very good, we could use directly this generator to generate the Landau distribution, but we want to deepen our knowledge about the modern PRNG's.

1.4 The `xoroshiro128+` Algorithm (2016)

We introduce the family of pseudo-random number generators called `xorshift`, whose name comes from the two bitwise operations these algorithms are based on, namely the exclusive or (`xor`) and the logical bit-shifts (`shift`).

David Blackman and Sebastiano Vigna developed in 2016 the `xoroshiro` algorithm in which another operation is used: the bitwise rotation (`ro`) or circular bit-shift. The difference between the logical bit-shift and the circular one lies in the bits that are shifted outside the size of the variable. Indeed, while in the former zeros are inserted to replace the bits that remained empty, the latter puts there the discarded bits. See the Figure 1.3 for an 8-bit example.

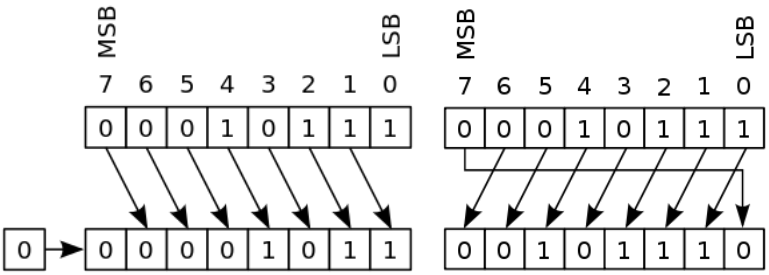


Figure 1.3: An example of logical bit-shift (on the left) and circular bit-shift (on the right) performed on a 8-bit number.

In particular, we used the `xoroshiro128+` algorithm, which requires two initial 64-bit positive integers s_0 and s_1 as seed. The first pseudo-random number generated by the algorithm is the sum of the two seeds $s_0 + s_1$. The subsequent upgrading process is the following:

1. First s_1 is replaced by its `or` combination with s_0.

$$s_0 \vee s_1 \to s_1$$

2. Next, s_0 is rotated by a-bits on the left and it is combined with s_1 through a `or` operation. The resulting number is then combined with s_0 shifted to the left by b-bits, through another `or` operation. The result is then replaced to s_0.

$$[rot(s_0, a) \vee s_1] \vee shift(s_0, b) \to s_0$$

3. s_1 is then replaced by its left-rotated version by c bits.

$$rot(s_1, c) \to s_1$$

4. Finally, the two numbers are added up (hence the `128+` in the name) to obtain the next number PR-generated.

$$\text{New Number} = s_0 + s_1$$

The algorithm – as can be seen above – has three free parameters, namely a, b and c. We chose to use the "best parameters" according to the authors of the algorithm, namely $a = 24$, $b = 16$ and $c = 37$. The same line of reasoning led us to use as seeds $s_0 = 1103527590$ and $s_1 = 2524885223$.

We chose to write the code in C because, unlike python, this programming language is straightforward to use for bit manipulation of fixed size variables.
Nevertheless we had to implement by hand a function for circular shift, since it is not present in any C library. In particular, this function performs a logic-shift of the original 64-bit number by N bits and then by $64 - N$ bits in the other direction, generating two new numbers. Then it combines these numbers with the `or` operator. Figure 1.4 shows an 8-bit example.

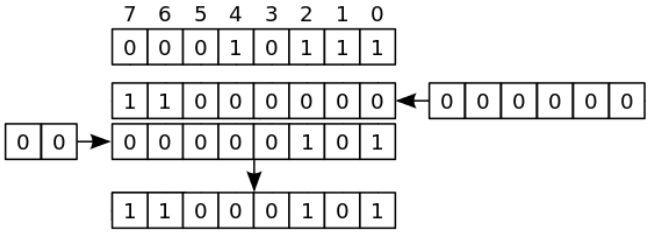


Figure 1.4: Circular shift on a 8-bit number. In the first row, the original number. Second and third rows contain respectively a 6-bit logical shift to the left and a 2-bits logical shift to the right. The last row is the combination of second and third ones through a `or` operation

.

The code written to implement the `xoroshiro128+` algorithm is shown in the *codes & further results* section[2]. As done with the LCG algorithm, we produce $3 \cdot 10^5$ numbers.

1.4.1 Randomness Test

We can then feed the `diehard` with the numbers generated with the `xoroshiro128+` algorithm, to see whether they are patternless. Finally it comes out that the `xoroshiro128+` generated numbers passed all the tests except for one that gave as result "weak". We can then state that this algorithm is better than the LCG.

1.5 The Rejection Sampling

After having implemented three different algorithms and seeing that both the LGC and the `xoroshiro128+` are very good, we decide to use the latter, that gives better results and is the most up-to-date. We then use it to generate pseudo-random numbers distributed with the Landau distribution, which describes the energy loss by ionisation of a charged particle in a thin layer of matter.
We can use an approximation for the Landau distribution, that reads

$$\mathcal{L}(x) = \frac{1}{\sqrt{2\pi}} \exp\left\{\left(-\frac{(x + e^{-x})}{2}\right)\right\} \tag{1.3}$$

This function has been already implemented in the Python-based ecosystem `SciPy` with the name of `moyal.pdf`.

To generate pseudo-random numbers distributed with the Landau distribution, we use the *rejection-sampling* technique. In computational statistics, rejection sampling is a way to simulate random samples from a distribution. In particular, the steps we have followed are the following:

1. Define a Landau distribution function $\mathcal{L}(x)$ with a chosen domain $\mathcal{D}$ and consequently codomain $\mathcal{C}$;
2. Generate a random number $x_r \in \mathcal{D}$;
3. Generate another random number $y_r \in \mathcal{C}$;
4. If $y_r < \mathcal{L}(x_r)$, then x_r is accepted, otherwise is rejected.

In particular, for the points 2. and 3., we built two uniform distributions within $\mathcal{D}$ and $\mathcal{C}$ by means of `xoroshiro128+`; we then obtained the two numbers x_r and y_r picking them up from these uniform distributions.
Finally we obtain the histogram shown in Figure 1.5, fitted with the Landau function.

[2]From now on, the reference to the *codes & further results* section will be implicit. All the implemented codes were written there, together with further results that could have hindered a fluid reading of the report. We recommend reading the two documents in parallel.

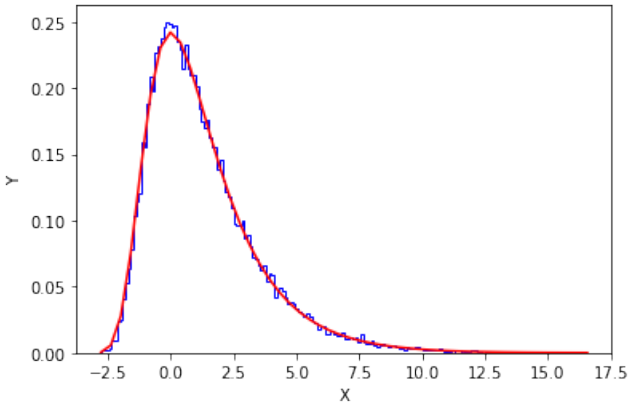


Figure 1.5: Histogram showing the Landau distribution obtained with a rejection-sampling technique performed with the pseudo-random numbers generated through the `xoroshiro128+` algorithm. The fit returns $\tilde{\chi}^2 \simeq 0.9964$, $p = 0.4896$.

Exercise 2

Task

Implement a program for calculating the Euler-Mascheroni constant using multiple methods and discuss the influence of the various types of calculation errors (algorithmic, analytic and round-off) on the result.

2.1 Theoretical Introduction

The Euler-Mascheroni constant γ is a mathematical constant recurring in analysis and number theory. It is defined as the limiting difference between the harmonic series and the natural logarithm:

$$\gamma = \lim_{n\to\infty}\left(-\ln n + \sum_{k=1}^{n}\frac{1}{k}\right) \tag{2.1}$$

The numerical value of the Euler-Mascheroni constant to 57 decimal places is

$$\gamma \simeq 0.57721566490153286060651209008240243104215933593992359880 5 \tag{2.2}$$

Euler in 1734 calculated the constant's value to 6 decimal places. In 1781, he calculated it to 16 decimal places. Mascheroni in 1790 attempted to calculate the constant to 32 decimal places, but made some errors.

Nowadays, to compute γ some summation algorithms are used. To analyse them, we first have to deepen our knowledge about numerical analysis.

2.1.1 Numerical Analysis

Numerical analysis is the study of algorithms that use numerical approximation for the problems of mathematical analysis. The overall goal of numerical analysis is the design and analysis of techniques to give approximate but accurate solutions to several problems, as the actual computation of γ.
The field of numerical analysis predates the invention of modern computers by many centuries. Numerical methods often depended on hand interpolation formulas applied to data from large printed tables. Since the mid 20th century, computers calculate the required functions instead, but many of the same formulas nevertheless continue to be used as part of the software algorithms.

There are some fundamental characteristics that influence the quality of numerical mathematics algorithms:

- The speed of execution,
- The precision of the results,
- The portability on other machines.

These points are not independents of each other. Think for example of double precision calculations, which allow us to increase precision, but that drastically slow down execution speed.

Concerning the precision of the results, the ability of an algorithm to keep errors under control, greatly influences the reliability of the outcome. There are several ways in which errors can be introduced in the solution of the problem. Let's analyse some of them.

2.1.2 Round-Off Error

A round-off error is the difference between the result produced by a given algorithm using exact arithmetic and the result produced by the same algorithm using finite-precision, rounded arithmetic. Rounding errors are due to inexactness in the representation of real numbers and the arithmetic operations done with them.
The error introduced by attempting to represent a number using a finite string of digits is a form of round-off error called *representation error*. The increase of digits allowed in a representation reduces the magnitude of possible round-off errors, but any representation limited to finitely many digits will still cause some degree of round-off error.
Since the computers use finitely many digits to represent real numbers (which in theory have infinitely many digits), the estimate of the computation error represents nowadays one of the main goals of numerical analysis.

Floating-Point Number System

The floating-point number system is the most efficient in representing real numbers, so it is widely used in modern computers. While the real numbers are infinite and continuous, a floating-point number system is finite and discrete. In general, a floating-point number system $\mathcal{F}$ is characterised by the following integers:

- $B \geq 2$: base
- $t \geq 1$: precision
- $p \in [-m, M]$: exponent

The following is the set of the floating-point numbers:

$$\mathcal{F}(t, B, m, M) = \pm B^p \sum_{i=1}^{t} d_i B^{-i} \tag{2.3}$$

where $d_1 \neq 0$ and $d_i \in [0, B-1]$ are called digits of the representation. The quantity $\sum_{i=1}^{t} d_i B^{-i}$ is the mantissa. An approximation of a real number x is thus given by:

$$x \simeq \text{sign}(x) B^p \sum_{i=1}^{t} d_i B^{-i} \tag{2.4}$$

If we choose, for example, $B = 10$ to write $\pi = 3.1415\ldots$ with precision $t = 3$, we have that the digits of the representation are $d_1 = 3$, $d_2 = 1$, $d_3 = 4$ and $p = 1$, since before the comma there is only one digit. So we can write:

$$\begin{aligned} \pi \simeq & B^p(d_1 B^{-1} + d_2 B^{-2} + d_3 B^{-3}) = \\ = & 10^1(3 \cdot 10^{-1} + 1 \cdot 10^{-2} + 4 \cdot 10^{-3}) = \\ = & 10 \cdot (0.3 + 0.01 + 0.004) = \\ = & 10 \cdot (0.314) = \\ = & 3.14 \end{aligned} \tag{2.5}$$

In the `IEEE` standard the base is binary, i.e. $B = 2$. The two most commonly used levels of precision for floating-point numbers are:

- Single precision: Representation at 32 bits
- Double precision: Representation at 64 bits.

1 bit is for the sign (0 stands for +, 1 for -), 8 or 11 bits for p, 23 or 52 bits for the digits of the representation. Obviously, in double precision the number of significant digits is higher (17 more) but the computation time is tripled and the memory occupation is higher too. Then, one has to analyse the problem and tell what is the best choice to make in order to deal with it.

The IEEE standard uses as rounding rule the *round-to-nearest*, in which the approximate value of a real value x is set to the nearest floating point number. When there is a tie, the floating-point number whose last stored digit is *even* (namely 0, in binary) is used. This ensures that it is not rounded up or down systematically, and so it allows to avoid the possibility of an unwanted slow drift in long calculations due simply to a biased rounding.

Loss of Significance

Performing floating-point arithmetic may lead to round-off error in the final result. This effect is called "Loss of Significance" and it is caused by the fact that rounding multiple times can cause error to accumulate.
For example, when two floating-point numbers are summed, their decimal points are lined up and added, and then the result is stored again as a floating-point number. So, even if the addition itself can be done in very high precision, the result must be rounded back to the specified precision, which may lead to round-off error.
To give an idea of what happens, let's consider a base-10 example. Consider two numbers with precision $t = 3$:

$$a = 0.948 \qquad b = 0.638 \tag{2.6}$$

To sum them, one can perform the addition with a higher precision, namely $t = 4$:

$$c = a + b = 1.586 \tag{2.7}$$

but then he has to take a step back and return to $t = 3$, and so he has to round c:

$$c = 1.59 \tag{2.8}$$

In such way, the round-off error is 0.4.

2.1.3 Analytic and Algorithmic Errors

If one wants to calculate the value of x, but x is real and so it cannot be computed in a finite number of iterations, one can approximate its value $\tilde{x}$. The *analytic error* is then:

$$\epsilon_{an} = \frac{x - \tilde{x}}{x} \tag{2.9}$$

and it shows how much the true value x is different from $\tilde{x}$. This problem is therefore due to the discrete model used. For example, to compute the value of e^{-3}, one can do two different things:

$$e^{-3} = 1 - 3 + \frac{3^2}{2!} + \dots \tag{2.10}$$

$$e^{-3} = \frac{1}{e^3} = \frac{1}{1 + 3 + \frac{3^2}{2!} + \dots} \tag{2.11}$$

The two algorithms seem equal, but in actual fact the second way of computing e^{-3} returns a smaller analytic error.

As seen in the subsection 2.1.2, in every floating-point representation of an algorithm, every arithmetic operation could introduce a round-off error. In such way, the final value obtained through the algorithm $x^\star$ may not be exactly equal to the value $\tilde{x}$. Let's thus define the *algorithmic error* as:

$$\epsilon_{alg} = \frac{x^\star - \tilde{x}}{\tilde{x}} \tag{2.12}$$

This error is then generated by the accumulation of local errors relative to each arithmetic floating-point operation. This is the error we will analyse in order to complete this exercise.

Stability of an Algorithm

In numerical analysis, the numerical stability is a generally desirable property of the algorithms. The concern is the growth of round-off errors up to lead to totally wrong results. A unstable algorithm would be useful on a infinite-precision computer, but in our imperfect PCs, we need to use stable algorithms. So, in short words, the algorithms must maintain ϵ_{alg} under control.
Moreover, there are some problems called "poorly conditioned" in which small fluctuations in initial data cause a large deviation of final answer from the exact solution. Some algorithms may damp out the small fluctuations (errors) in the input data; others might magnify such errors. Calculations that can be proven not to magnify approximation errors are called *numerically stable*. One of the common tasks of numerical analysis is to try to select algorithms which are *robust* – that is to say, do not produce a wildly different result for very small change in the input data.

2.2 Floating-Point Summation

Let's now show some solutions to suppress the loss of significance due to the summation of floating-point numbers in the computation of the Euler-Mascheroni constant γ as defined in 2.1.

2.2.1 Sorted Summation

Precision is lost most quickly when the magnitude of the two numbers to be added is most different. Let's make an example. If there are not enough bits of mantissa to resolve a difference of one part in ten billion, then:

$$1.0 + 1.0 \cdot 10^{10} = 1.0 \cdot 10^{10} \tag{2.13}$$

but in this way, if one performs a summation as follows:

$$(1.0 + 1.0 \cdot 10^{10}) + (-1 \cdot 10^{10}) = 1.0 \cdot 10^{10} + (-1.0 \cdot 10^{10}) = 0 \tag{2.14}$$

even if the true value of this summation is 1.0. In this example, all the information from one summand was lost because two of the summands were much larger than the other. This problem is known by the name of *catastrophic cancellation*.

With summation, one of the operands is always the sum so far, and in general it is much larger than the numbers being added to it. This effect can be minimised by adding the numbers from the smallest in magnitude to the largest. To do so, one has to sort the numbers by magnitude before summing. This method is called Sorted Summation.
In our very problem, to compute the summation in the equation 2.1, the sorting is inverted, since the series goes from $k = 1$ to $k = N$ with $N >> 0$ and so the numbers $\frac{1}{k}$ are naturally sorted from the largest to the smallest number. The first solution to avoid the loss of significance consists in the inversion of the order of the floating-point numbers in the summation, and so in summing from $k = N$ to $k = 1$.

2.2.2 Pairwise Summation

Another approach to keep the operands of each addition similar in magnitude is to add them in pairs. Pairwise summation of a sequence of numbers works by recursively breaking the sequence into two halves, summing each half, and adding the two sums: a *divide and conquer* algorithm.

2.2.3 Kahan Summation

An even better way to sum many numbers is the Kahan Summation. To explain how this algorithms works, the best way is a working base-10 example.
Suppose we are using 6-digit decimal floating-point arithmetic, and at some point we have the variable

$$sum = 10000.0$$

Say, the next two numbers to add are

$$\pi = 3.14159 \quad e = 2.71828$$

Adding the two numbers sum = 10005.85987 before rounding and sum = 10005.9 after rounding. With a plain summation, the exact first result would be sum = 10003.14159, which rounds to 10003.1. The second addition would give sum = 10005.81828, which rounds to 10005.8. This is not correct. However, with a *compensated summation*, we can get the correct rounded result of 10005.9. Let's see how.
For the first summation, the algorithm works as follows:

```
y = pi - c   = 3.14159 - 0.00000
t = sum + pi = 10000.0 + 3.14159
             = 10003.14159
             = 10003.1
c = (t - sum) - pi = (10003.1 - 10000.0) - 3.14159
                   = 3.10000 - 3.14159   (*)
                   = -0.0415900
sum = t = 10003.1
```

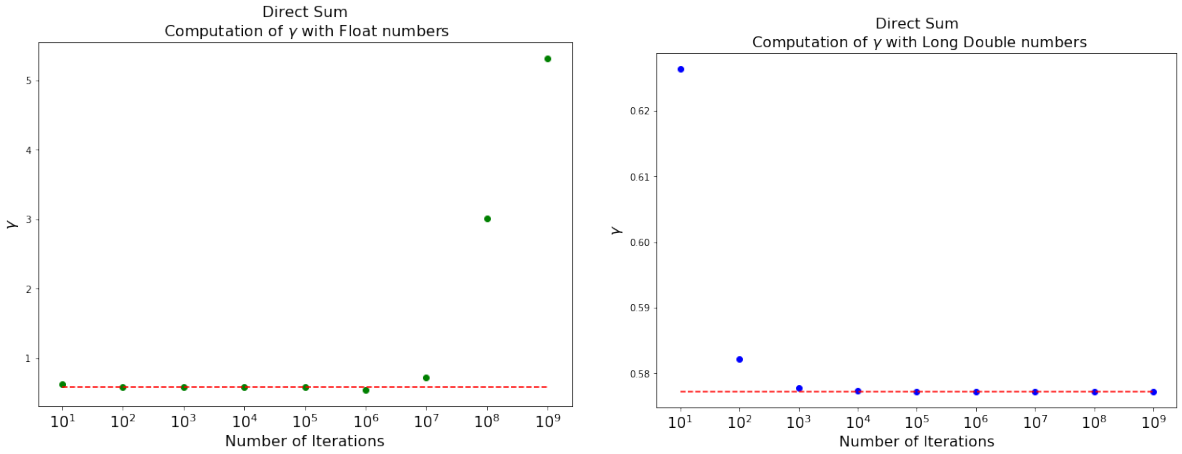
Concerning the addition of `e`, then:

```
y = e - c   = 2.71828 - (-0.0415900)
            = 2.75987
t = sum + e = 10003.1 + 2.75987
            = 10005.85987
            = 10005.9
c = (t - sum) - e = (10005.9 - 10003.1) - 2.75987
                  = 2.80000 - 2.75987   (*)
                  = 0.040130
sum = t = 10005.9
```

The two steps highlighted with $(\star)$, the differences between the assimilated part of π and e and their original values are computed. For π, this difference is negative, while for e is positive. This means that this inequality between the input value and the read value fluctuates around zero, and time by time this value is added or subtracted to the sum. In such way, the final result is approximated correctly.

2.3 Results

After having introduced from a theoretical point of view the problem, the time has come to roll up our sleeves and analyse the results we have achieved with these 4 summation algorithms – Direct Summation, Sorted Summation, Pairwise Summation and Kahan Summation. In particular, we analyse their performance computing the algorithmic error ϵ_{alg} at different number of iterations. By way of example, we show below the resulting plots obtained with the Direct Summation algorithm.



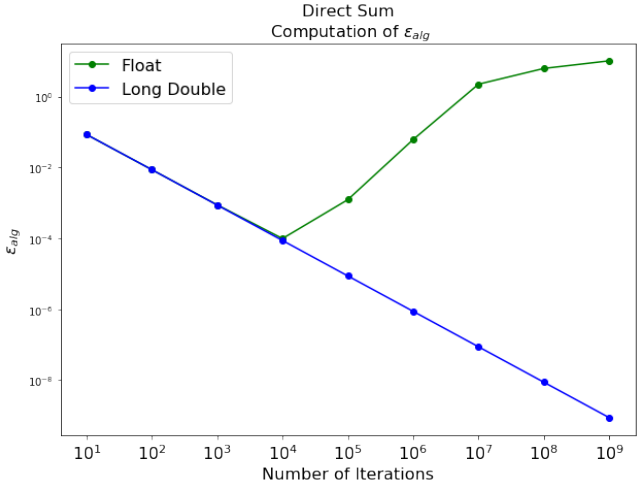


Figure 2.1: Computation of γ and ϵ_{alg} with the Direct Summation algorithm with both Float and Long Double numbers. The correct value of γ is represented with a red dashed line.

The goal of this exercise is however to see how to minimise the algorithmic error in the computation of γ and, to do so, we compare ϵ_{alg} obtained with the four algorithms with different numbers of iterations.

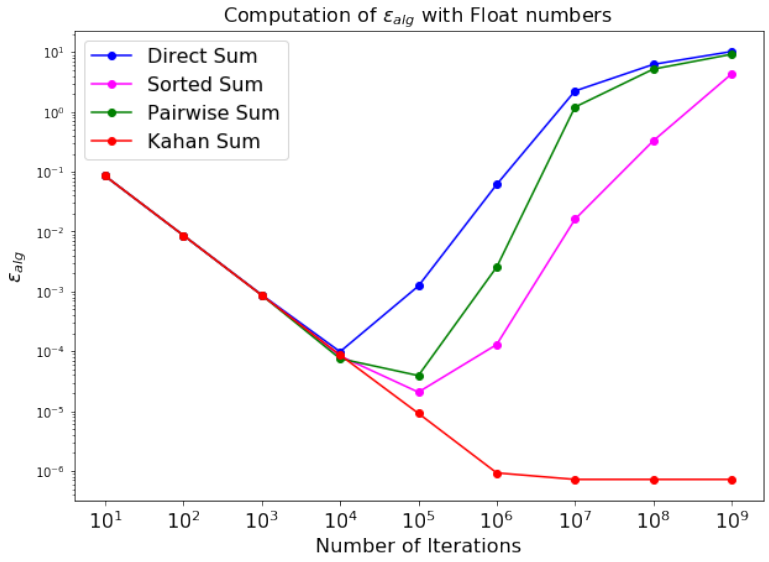


Figure 2.2: Computation of ϵ_{alg} with the 4 summation algorithms in the legend, with Float numbers in function of the number of iterations.

As can be seen in Figure 2.2, with Float numbers after 10^4 iterations the algorithmic error ϵ_{alg} begins to grow for three algorithms, namely the Direct Summation, the Sorted Summation and the Pairwise Summation. Anyway, with the last two of them, ϵ_{alg} grows more softly with respect to the error computed with the Direct Summation, that grows very steeply.
With the Kahan Summation algorithm, ϵ_{alg} begins to grow only after 10^6 iterations, and very very slightly.
From this first result, it is clear that concerning the algorithmic error, the Kahan Summation algorithm is the best one, but requires several times more arithmetic operations with respect to the other algorithms.

Let's analyse now ϵ_{alg} with Long Double numbers. As can be seen from the Figure 2.3, the Long Double numbers are not affected by any representation error.

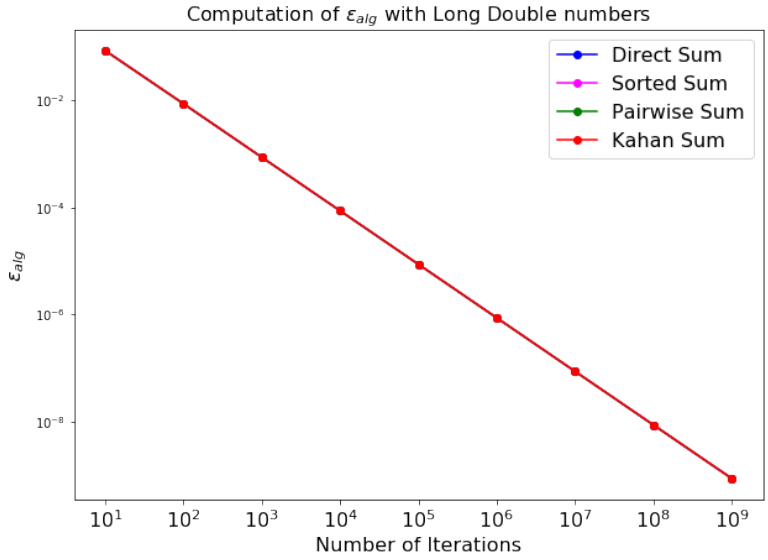


Figure 2.3: Computation of ϵ_{alg} with the 4 summation algorithms in the legend, with Long Double numbers.

Concerning the value of γ obtained through the Long Double numbers, we have that after 10^9 iterations:

$$\gamma = 0.5772156654 \tag{2.15}$$

that is equal to γ as defined in (2.2) up to the eighth decimal place.

In conclusion, the two best summation algorithms are the Pairwise and the Kahan summation ones. In particular, pairwise summation is nearly as good as the Kahan summation, while having much lower computational cost. In fact, it can be implemented so as to have nearly the same cost (and exactly the same number of arithmetic operations) as naive summation. For this very reason, Pairwise summation is the default summation algorithm in `NumPy`.

Exercise 3

Task

A vector meson ($J^P = 1^-$) decays into two pseudo-scalar mesons ($J^P = 0^+$). Knowing that (θ, ϕ) are the polar and azimuthal angles of the positive particle, the angular decay distribution is given by:

$$\mathcal{F}(\theta, \phi) = \frac{3}{4\pi}\Big[\frac{1-\alpha}{2} + \frac{3\alpha - 1}{2}\cos^2\theta - \beta \sin^2\theta \cos 2\phi - \sqrt{2}\,\gamma \sin 20 \cos\phi\Big] \tag{3.1}$$

1. *Generate a MC data-sample of* 50000 *events with (θ, ϕ) distributed according to the function 3.1 with parameters $\alpha = 0.65$, $\beta = 0.06$ and $\gamma = -0.18$.*
2. *Using the generated dataset, estimate the value of the parameters α, β and γ and their errors with the Maximum Likelihood method.*
3. *Represent the generated data in a histogram with an appropriate number of bins knowing that the experimental apparatus has a resolution of 1% in θ and ϕ, and repeat the parameter estimation with the Least Squares method.*
4. *Bearing in mind that a scalar meson has an isotropic decay distribution, test the hypothesis of a vector decay vs. a scalar decay.*

3.1 Generation of the MC data-sample

In order to generate a MC data-sample, we use the RooFit library, which provides a toolkit for modelling the expected distribution of events in a physics analysis. It has been originally developed for the BaBar collaboration, a particle physics experiment at the SLAC.
In order to use it, we have to link every mathematical concept with the associated RooFit class:

Mathematical Concept	RooFit class
variable x	`RooRealVar`
space point $\vec{x}$	`RooArgSet`
pdf $f(x)$	`RooAbsPdf`

First of all, we generate the pdf $\mathcal{F}(\theta, \phi)$ as defined in the equation 3.1 and we obtain the result shown in Figure 3.1. Once we have generated the pdf, we proceed to generate the MC events by means of the method `generate` of the class `RooAbsPdf`. We can see the results of this procedure in Figure 3.2. Notice that the binning of the MC dataset is required to have a more intelligible representation of the two shapes, but the data are actually unbinned.

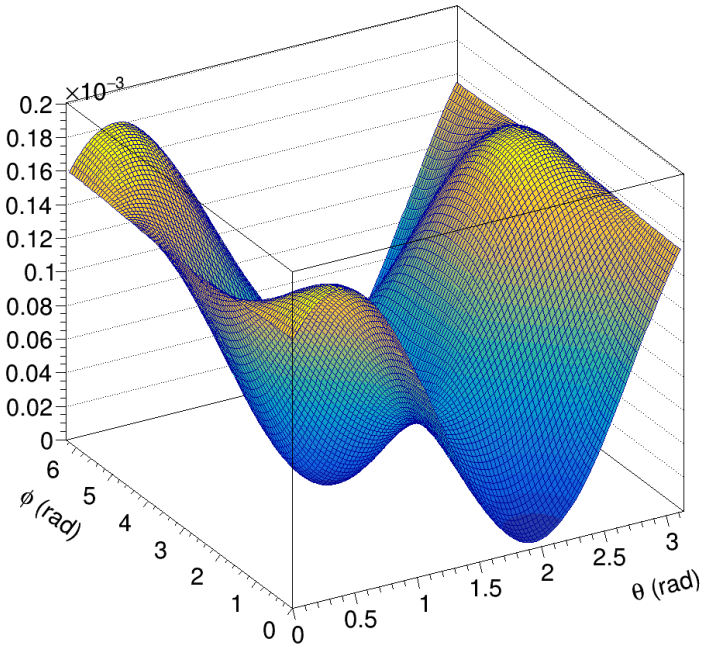


Figure 3.1: Representation of the two dimensional pdf $\mathcal{F}(\theta, \phi)$.

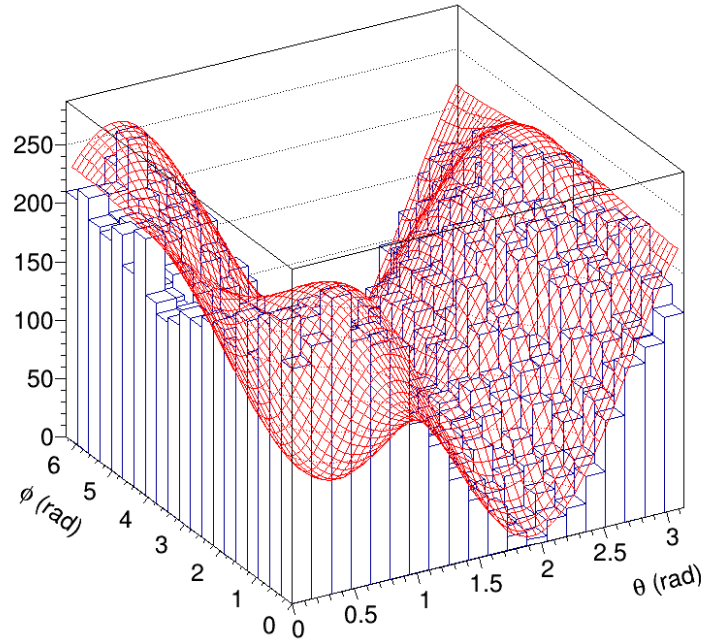


Figure 3.2: 2D pdf with fixed parameters (in red) and generated MC events (in blue).

3.2 Estimation of the Parameters with the ML Method

Now that we have generated a dataset of 50000 MC events, we can reverse the process all the way back and, starting from the events themselves, derive the parameters. Firstly we can use the Maximum Likelihood (ML) method.
To do so, it comes again to our help RooFit with the method `fitTo` of the class `RooAbsPdf`. In fact, in this way we can fit a pdf to a given dataset. The results of the fit are stored in an object called `RooFitResult`, a container class that helds the input and output of a pdf fit to a dataset. In particular, it contains what we are interested in, namely the values of the parameters with errors.
Let's see how these parameters are computed. Suppose that $\mathcal{F}$ as defined in the equation 3.1 depends only from the parameter α, for simplicity. The likelihood function for this problem would be then defined as:

$$L(\alpha) = \prod_{i=1}^{N} \mathcal{F}(x_i|\alpha) \tag{3.2}$$

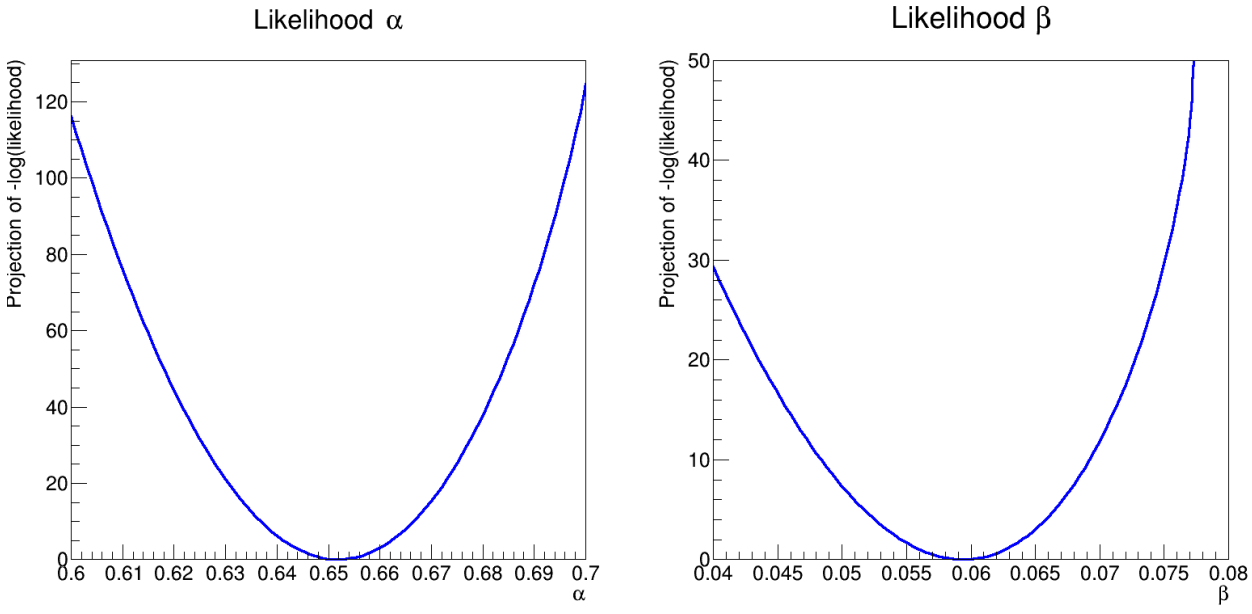
where x_i are the N measurements of the random variable x, and are distributed according to $\mathcal{F}$. In our very case, we know that $N = 50000$. The likelihood function gives the probability to obtain the measured values $\{x_i\}$ given the parameter α.
The Maximum Likelihood principle states that the best estimate of α is given by the estimator $\hat{\alpha}$, which maximises $L(\alpha)$:

$$\hat{\alpha} = \max_{\alpha} L(\alpha) \tag{3.3}$$

In practice, the *negative-log-likelihood* (NLL) is used:

$$l(\alpha) = -\ln L(\alpha) = -\sum_{i=1}^{N} \ln \mathcal{F}(x_i|\alpha) \tag{3.4}$$

In Figure 3.3 the NLL of the three parameters are shown and the best estimate of them are computed minimising these functions.



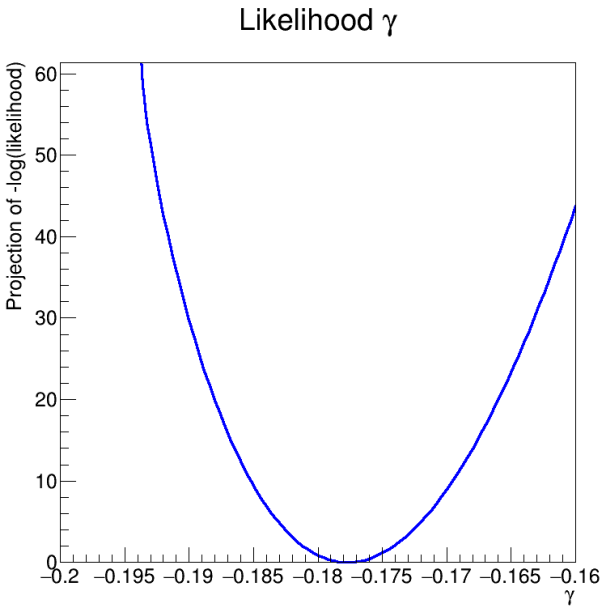


Figure 3.3: The plots containing the likelihood functions of the three parameters α, β and γ. The best estimation of these parameters is performed minimising these functions through the Minuit algorithm.

In such way, the following results are obtained.

```
RooFitResult:

    minimized FCN value: 141656,
    estimated distance to minimum: 413.9822e-05

    covariance matrix quality: Full, accurate covariance matrix
    Status : MINIMIZE=0 HESSE=0
```

Floating Parameter	FinalValue +/- Error
alpha	6.5186e−01 +/- 3.45e−03
beta	5.9386e−02 +/- 2.73e−03
gamma	−1.7771e−01 +/- 1.99e−03

3.3 Estimation of the Parameters with the LS Method

First of all we have to group the generated data in appropriate bins. Bearing in mind that the experimental apparatus has a resolution of 1% in θ and ϕ we decide to use 100 bins for both.
After having binned the data, we follow the same procedure as explained for the ML method, this time using the `chi2fitTo` method of the `RooAbsPdf` class.
Let's give again a brief explanation of the LS method, considering for simplicity one single parameter α. Consider then N measurements of a random variable x_i distributed according to $\mathcal{F}$. In short words, the parameters are obtained minimising the distance between the succession of x_i and $\mathcal{F}$. To obtain a single optimised curve and not a beam, it is needed a number of experimental point greater than the number of parameters on which the curve depends – the problem is generally said to be *overdetermined*.

With this second method we obtain the following results.

```
RooFitResult:

    minimized FCN value: 9736.5,
    estimated distance to minimum: 0.00010663

    covariance matrix quality: Full, accurate covariance matrix
    Status : MINIMIZE=0 HESSE=0
```

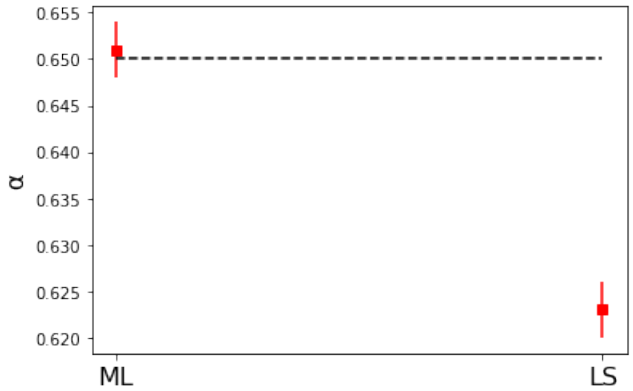
Floating Parameter	FinalValue +/- Error
alpha	6.2336e−01 +/− 3.16e−03
beta	5.7322e−02 +/− 2.66e−03
gamma	−1.6237e−01 +/− 1.89e−03

3.4 Comparison Between ML and LS

We now show a summary of the results obtained with the ML and the LS method compared to the real values of the three parameters.

Parameter	**True Value**	**Maximum Likelihood**	**Least Squares**
α	0.65	0.651 ± 0.003	0.623 ± 0.003
β	0.06	0.059 ± 0.003	0.057 ± 0.003
γ	-0.18	-0.178 ± 0.002	-0.162 ± 0.002

As can be seen from figure 3.4, the Maximul Likelihood method gives us better results with respect to the Least Squares one. In fact, all three parameters obtained with the ML are compatible within 1σ with the real values, while only the value of β calculated with the LS is compatible.



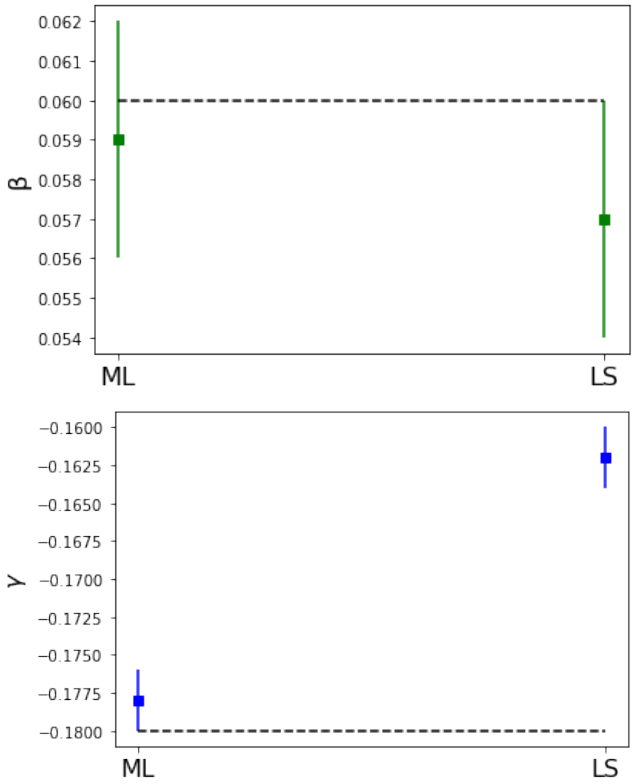


Figure 3.4: Comparison between the values of the three parameters α, β and γ obtained with the ML and the LS methods and the true values (black dashed lines).

3.5 The Likelihood-Ratio Test

We now want to perform a hypothesis test comparing the two hypothesis of a vector decay vs. scalar decay. The former is represented by the well familiar $\mathcal{F}(\theta, \phi)$ defined in the equation 3.1, while the latter has an isotropic decay distribution described by the following pdf:

$$\mathcal{G}(\theta, \phi) = \frac{1}{4\pi} \tag{3.5}$$

We obtain $\mathcal{G}$ starting from $\mathcal{F}$ by imposing $\alpha = \frac{1}{3}$, $\beta = 0$ and $\gamma = 0$. A representation of this pdf is shown in Figure 3.5.

In order to assess the goodness of fit of these two competing statistical models we can use the likelihood-ratio test, the oldest classical approaches to hypothesis testing. In particular, we use it because both the two models have no unknown parameters and the Neyman–Pearson lemma demonstrates that in this case the test has the highest power among all competitors.
We then define two different hypothesis:

- H_0: $\alpha = \frac{1}{3}$, $\beta = 0$ and $\gamma = 0$;
- H_1: $\alpha = 0.65$, $\beta = 0.06$ and $\gamma = -0.18$.

H_0 is also said "null hypothesis" and defines the hypothesis of uniform distribution $\mathcal{G}$ of the scalar decay, while H_1 stands for the vector decay hypothesis $\mathcal{F}$.

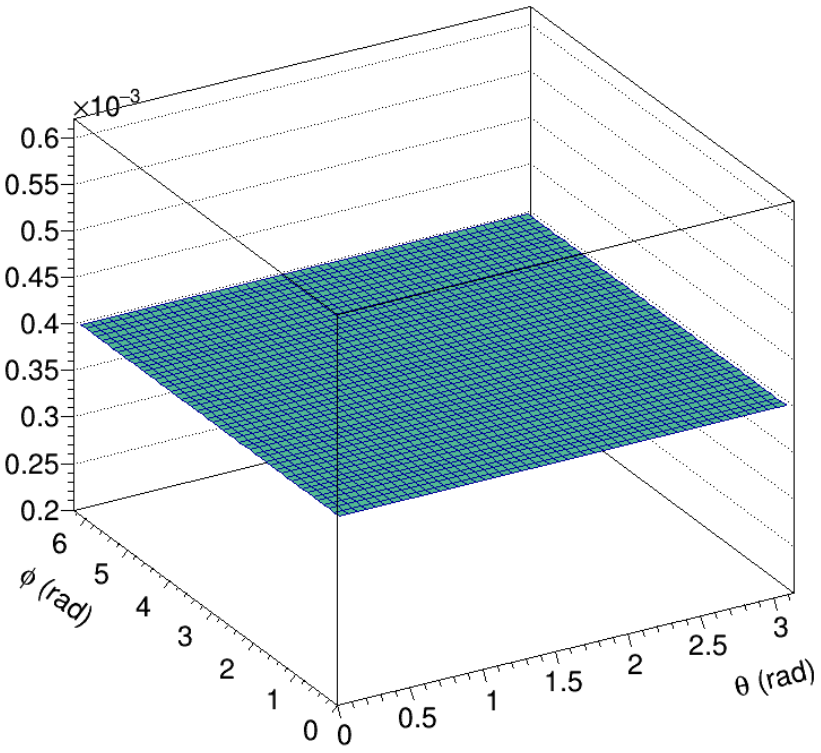


Figure 3.5: Representation of the two dimensional pdf $\mathcal{G}(\theta, \phi)$.

The ratio of the two likelihood is defined as follows:

$$\lambda(\theta, \phi) = \frac{\mathcal{L}(H_0|\theta, \phi)}{\mathcal{L}(H_1|\theta, \phi)} \tag{3.6}$$

Low values of the likelihood ratio mean that the observed result was much less likely to occur under the null hypothesis as compared to the alternative. High values of the statistic λ mean that the observed outcome was nearly as likely to occur under the null hypothesis as the alternative, and so the null hypothesis cannot be rejected. In particular, the likelihood-ratio test provides the decision rule as follows:

- if $\lambda > c$ do not reject H_0;
- if $\lambda < c$ reject H_0;

So the likelihood-ratio test rejects the null hypothesis if the value of the statistic λ is too small.
The value c is usually chosen to obtain a specified significance level α, via the relation:

$$\mathcal{P}(\lambda < c|H_0) = \alpha \tag{3.7}$$

which represents the probability that H_0 is rejected despite H_0 is true (type-I error). The value of c (and thus of α) depends on what probability of Type I error is considered tolerable.

Since $\lambda > 0$, we can define its logarithm $\log \lambda \in (-\infty, +\infty)$, that is monotonically increasing. In fact, it is more convenient to work with a logarithmic transformation of the likelihood functions, known as the "log-likelihood function" l. In fact, given independent events, since the overall likelihood is given by the product of the likelihoods of the individual events, the log-likelihood is the sum of their log-likelihood:

$$l(H_0|\theta, \phi) = \sum_{i=0}^{n} l_i(H_0|\theta, \phi); \qquad l(H_1|\theta, \phi) = \sum_{i=0}^{n} l_i(H_1|\theta, \phi); \tag{3.8}$$

In conclusion, by logarithmically transforming the equation (3.6) it is easier to evaluate the statistic λ:

$$\log \lambda(\theta, \phi) = l(H_0|\theta, \phi) - l(H_1|\theta, \phi) \tag{3.9}$$

We therefore have to compute for each coordinate $(\theta^\star, \phi^\star)$ the values $\mathcal{F}(\theta^\star, \phi^\star)$ and $\mathcal{G}(\theta^\star, \phi^\star)$. Then we compute the difference between their logarithms and finally we obtain

$$\log \lambda(\theta, \phi) = -6346.7 \quad \Rightarrow \quad \lambda(\theta, \phi) = e^{\log \lambda(\theta, \phi)} = e^{-6346.7} \simeq 0$$

We then conclude that the H_0 has to be rejected with a very high significance, and so that the hypothesis of scalar decay is not to be taken into consideration. This makes sense, since the data have been produced starting from the pdf of a vector decay.

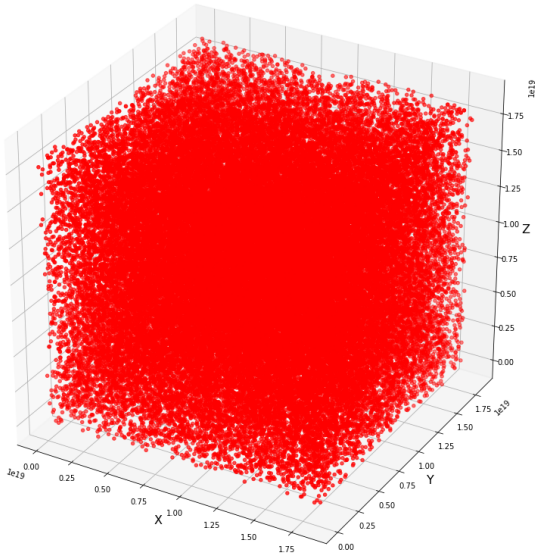
Exercise 4

Task

Generate with a MC method N charged particles with a uniform distribution in space and uniform momentum $p \in [0, 10]$ *GeV.*
Prove that the dip of the distribution $< p_T > (p_L)$ *at low* p_L *values is a kinematic effect.*

4.1 Generation of MC Events

In order to generate some particles with uniform distribution in space and with uniform momenta, we can use the `xoroshiro128+` algorithm widely described in the first exercise. With it we produce four samples, each composed by 60000 numbers. One of the samples will give the momenta, while the other three the space coordinates.
Since the task requires the momenta distributed in $p \in [0, 10]$ GeV we normalise one of the samples so that we obtain the correct range. Finally we obtain the results shown in Figure 4.1.



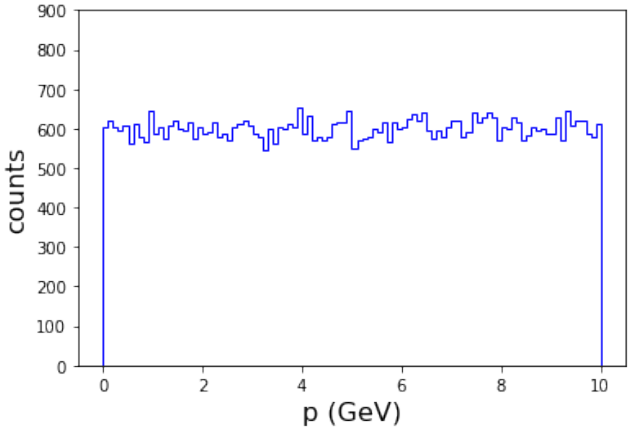


Figure 4.1: Monte Carlo simulation of 60000 particles uniformly distributed in space and with uniform momentum $p \in [0, 10]$ GeV.

Once we have generated these particles, we can deepen the simulation about the momentum by projecting it in p_L and p_T. To do so, we generate another sample of 60000 random numbers to generate θ, and since $\theta \in [0, 2\pi]$, we normalise these number to obtain the right range.
Once we have θ, we can calculate p_L and p_T in the following way:

$$p_L = |p \cdot \cos\theta| \qquad p_T = |p \cdot \sin\theta| \tag{4.1}$$

In such way we obtain the result shown in Figure 4.2.

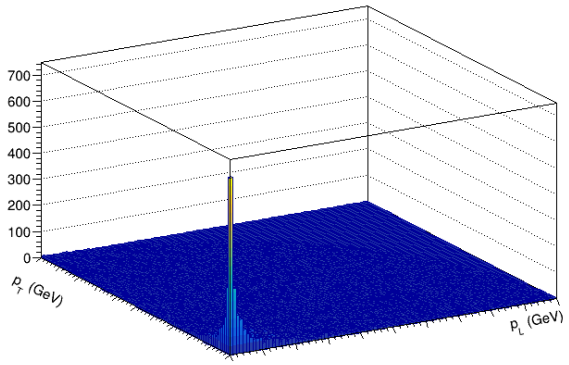


Figure 4.2: 2D histogram showing p_L and p_T of the events generated through the MC method.

We show below by way of example a sample of 10 particles with all their properties.

	X	Y	Z	P	θ	PL	PT
0	1.103528e+09	5.080650e+17	1.271024e+19	5.366743	2.744146	4.948418	2.077280
1	4.112170e+18	1.578383e+19	2.353373e+18	9.765948	3.908518	7.031948	6.776831
2	1.841057e+19	7.429007e+18	1.336695e+19	3.032928	5.396148	1.915936	2.351136
3	1.664838e+19	1.828035e+19	6.710494e+18	4.489837	2.720398	4.097430	1.835675
4	6.287101e+18	1.219377e+19	2.297079e+18	4.015651	4.693456	0.076025	4.014931
5	9.491289e+18	3.727635e+18	3.428383e+17	7.084756	5.185926	3.230908	6.305157
6	1.381415e+19	1.773815e+19	2.567772e+18	2.532706	1.058779	1.240866	2.207907
7	4.180057e+18	4.783356e+18	9.276700e+18	7.129019	0.519479	6.188545	3.539043
8	1.430281e+19	1.197381e+19	9.614016e+18	9.931401	1.345465	2.218962	9.680337
9	1.094761e+19	6.039125e+18	1.663762e+19	9.955091	2.719346	9.080742	4.079701

4.2 The $< p_T > (p_L)$ Distribution

We now want to show the distribution of $< p_T >$ in function of p_L by averaging the values of p_T for every bin of p_L. To do so, we use the `ROOT` method called `Profile`. In this way we obtain the plot shown in Figure 4.3.

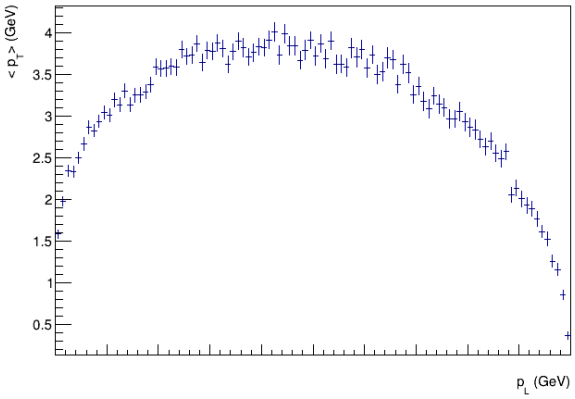


Figure 4.3: Plot of $< p_T >$ in function of p_L of the events generated through the MC method with θ uniform.

To explain the shape shown in Figure 4.3 we look at the pdf of p_L and p_T:

$$f_{p_T}(p_T) = \frac{1}{20\pi}\left[\ln\left(10 + \sqrt{p_T^2 + 100}\right) + \ln 10\right] \tag{4.2}$$

$$f_{p_L}(p_L) = \frac{1}{20\pi}\left[\ln\left(10 + \sqrt{p_L^2 + 100}\right) + \ln 10\right] \tag{4.3}$$

Notice that f_{p_T} and f_{p_L} have the same functional form. First of all we can check whether this statement is correct by looking at the shape of the histograms of p_T and p_L obtained with our data.

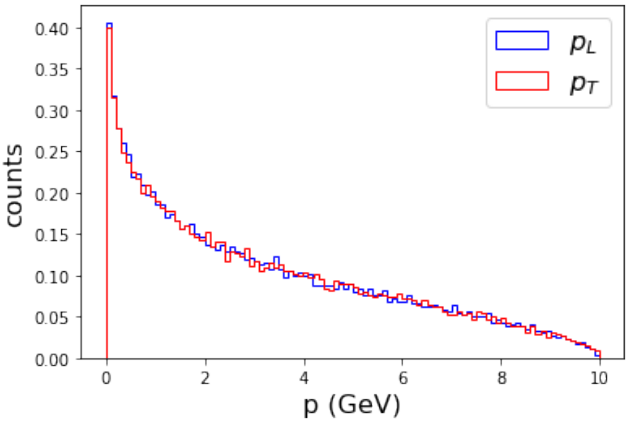


Figure 4.4: Normalised histograms of p_L and p_T of the events generated through the MC method with θ uniform.

In Figure 4.4 it can be seen that the two histograms are completely overlapped, and so the fact that the two equations 4.2 and 4.3 are equal is correct.
Moreover we fit the histogram of p_T with the eq. 4.2 and we obtain the plot shown in Figure 4.5.

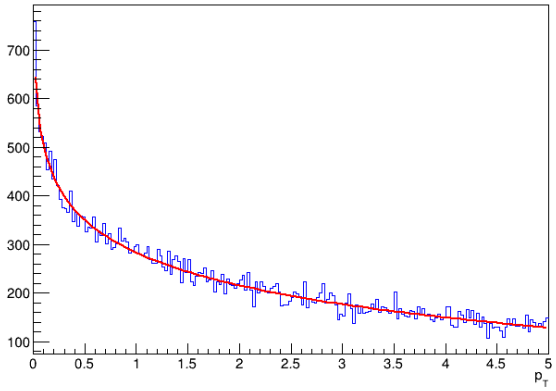


Figure 4.5: Fit of the histogram of p_T obtained with the data with the pdf of p_T shown in the eq. 4.2. f_{p_T} and the histogram obtained with the data are in good agreement since $\tilde{\chi}^2 = 1.001$ and $p = 0.467$.

As can be seen from the plots in Figure 4.4, for both p_T and p_L, the values at low momenta are widely preferred. In fact, counting the number of particles in some specific momentum intervals we found that:

- in [0,2] GeV:
  $N_p = 11979 \sim 12000$
  $N_{p_L} = 25299 \sim 25000$
  $N_{p_T} = 25070 \sim 25000$;
- in [8,10] GeV:
  $N_p = 12042 \sim 12000$
  $N_{p_L} = 3376 \sim 3000$
  $N_{p_T} = 3398 \sim 3000$;

All the 3000 particles with high p_L can only have low p_T and vice verse – although the resulting p would be greater then 10 GeV. In this way it is explained the decreasing part on the right of the plot in Figure 4.3, which reaches 0 at $p_L = 10$.
However, we want to describe the left part of the plot, so we have to analyse the 25000 particles with low values of p_L:

→ All the 12000 particles with low p have both p_L and p_T low, so 12000 of the low-p_L particles, have also low p_T values.

→ All the 3000 particles with high p_T can only have low p_L, and so 3000 of the low-p_L particles have high p_T.

So, not all the 25000 particles with low p_L have low p_T, but 3000 of them have high values of p_T. This cause a sensible increase of $< p_T >$ at low p_L. Moreover, we are left with $25000 - 3000 - 12000 = 10000$ particles with low p_L, that can have p_T variable between 2 and 8 GeV. Even these values rise the dip at low p_L.
In this way the shape of the distribution $< p_T > (p_L)$ shown in Figure 4.3 is explained.

Exercise 5

Task

Calculate through the MC method the integral

$$I = \int_0^1 e^x dx \tag{5.1}$$

and discuss in detail the variance reduction techniques.

5.1 Theoretical introduction

The Monte Carlo integration is a non-deterministic approach to the multidimensional definite integration. It is non-deterministic since it uses random variables and, consequently, each integration provides a different outcome. Each outcome is an approximation of the real value, and it is possible to determine an error related to it.

Let $f(x)$ be the integrand function, so that

$$I = \int_\Omega f(x)dx \tag{5.2}$$

In the naive Monte Carlo approach, we extract N points uniformly in Ω.

$$x_1, x_2, \ldots, x_N \in \Omega. \tag{5.3}$$

From each x_i we compute $f(x_i)$ and we average these values:

$$Q_N = \frac{1}{N}\sum_{i=1}^{N} f(x_i) \tag{5.4}$$

Thanks to the law of large numbers, we know that:

$$\lim_{N\to\infty} Q_N = I \tag{5.5}$$

If one repeats this process several times, more values of Q_N will be available to have a better estimation of I. The central limit theorem states that, since $x_1, x_2, \ldots, x_N$ are identically distributed, then both the quantity Q_N and its variance are asymptotically distributed as gaussians for large values of N. In such way, one can obtain a good estimation of I averaging on all the Q_N, while the standard deviation of I is given by:

$$\sigma_I = \frac{\kappa}{\sqrt{N}} \tag{5.6}$$

where κ depends on the variance of the single computation of Q_N.

Here lies advantage of the Monte Carlo method over the deterministic methods concerning multidimensional integration: the standard deviation goes as $\frac{1}{\sqrt{N}}$, and so the error decreases as N increases. The same cannot be said for the deterministic methods, in which the error does not depend on the sample's dimension.
Concerning the value of κ, we will see how it can be reduced thanks to some variance reduction algorithms based on smarter sampling methods: Changing the way we extract the random numbers $x_1, x_2, \ldots, x_N$, the value of κ changes.

5.2 Crude MC

This algorithm is simply the application of the naive Monte Carlo method. To evaluate the integral defined in the equation 5.1, we perform the following steps:

1. Get a random number x from a uniform distribution in the integration range $(0, 1)$;
2. Evaluate $f(x) = e^x$;
3. Repeat steps 1 and 2 for an arbitrary number N of steps;
4. Determine the average of the N values of $f(x)$ obtained. We call this average Q_N.
5. Determine the variance:
$$\sigma^2 = \frac{\sum_{i=1}^{N}(f(x)_i - Q_N)^2}{N-1} \tag{5.7}$$
6. Determine then the variance on the mean:
$$\hat{\sigma}^2 = \frac{\sigma^2}{N} \tag{5.8}$$

Repeating this algorithm 1000 times with $N = 128$, we obtain the results shown in Figure 5.1.

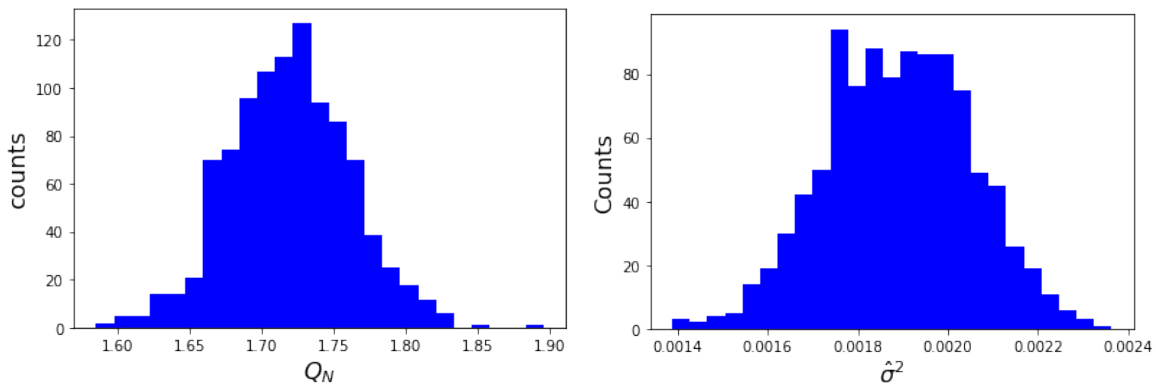


Figure 5.1: Histograms containing the values of 1000 computations of Q_N (left) and their variances $\hat{\sigma}^2$ (right) with the Crude MC method, using $N = 128$ extractions.

The best estimation of the integral I is obtained averaging the Q_N, while to obtain σ_I we compute the standard deviation on the mean:

$$\tilde{\sigma} = \sqrt{\frac{\sum_{i=1}^{1000}(Q_{Ni} - I)^2}{999}} \qquad \sigma_I = \frac{\tilde{\sigma}}{\sqrt{1000}} \tag{5.9}$$

In such way we obtain:

$$I = 1.7190 \pm 0.0014 \tag{5.10}$$

We then want to control if this result can be obtained in a easier way. To do so, we compute σ_I using the `np.var()` method directly on the `NumPy` array containing the values of Q_N. The results are actually the same, and the code is way shorter. From here on, then, we use this second way to compute σ_I.
In order to show how the result changes varying N, we repeat the algorithm with N going from 100 to 2500, with steps of 200. With a fit of these data we can find the coefficient κ of the equation 5.6, as shown in Figure 5.2. In such way, we find:

$$\kappa = 0.485 \pm 0.003 \tag{5.11}$$

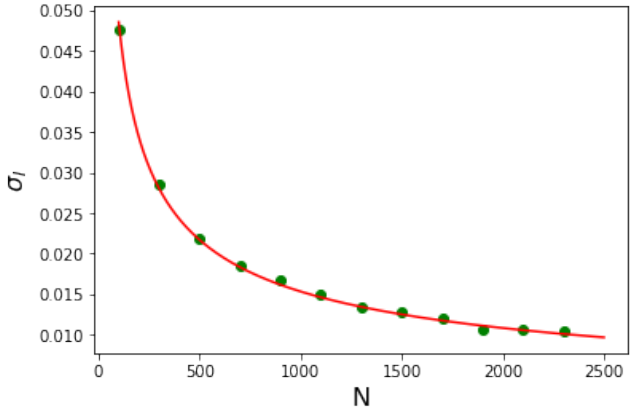


Figure 5.2: Values of σ_I varying with N (green dots) and fitting function $\frac{\kappa}{\sqrt{N}}$ (red line) obtained with the Crude MC method.

5.3 Stratified Sampling

The stratified sampling is the first example of variance reduction technique. It consists in splitting the domain of integration in more equal intervals, and extracting uniformly the points in each of them.

In this project, we divide the integration domain into two intervals, so that the algorithm is the same as before, but now it must be performed twice with random numbers generated respectively in $[0, 0.5]$ and in $[0.5, 1]$.
The result will be given by the average over the two Q_N obtained in each interval:

$$Q_N^{strat} = \frac{Q_N^1 + Q_N^2}{2} \tag{5.12}$$

Its variance, will thus be:

$$\hat{\sigma}^{2\,strat} = \frac{\sigma_1^2 + \sigma_2^2}{4} \tag{5.13}$$

It can be shown that the stratified sampling can never result in higher variance than pure random sampling. In fact, stratified sampling is asymptotically better, since the error reduces linearly with the number of samples.

We repeat the process 1000 times and find the best values as before. Figure 5.3 shows the results obtained with $N = 128$.

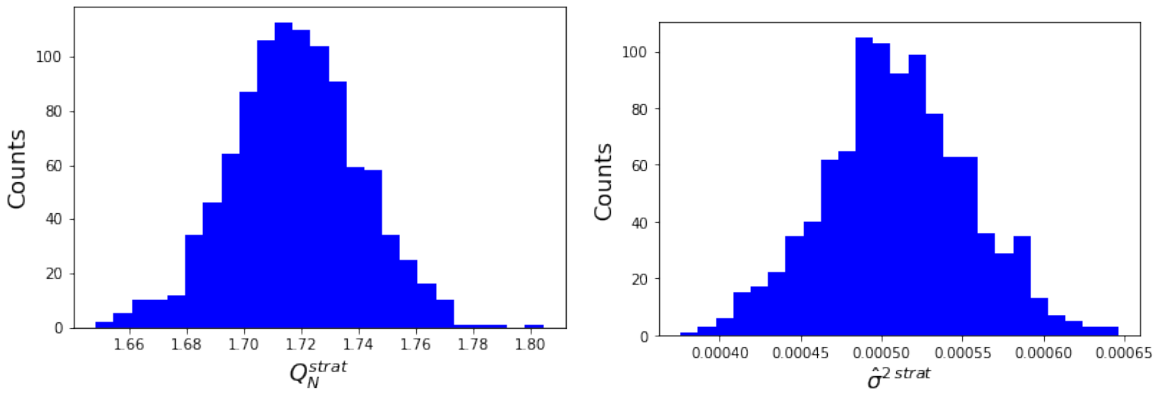


Figure 5.3: Histograms containing the values of 1000 computations of Q_N^{strat} (left) and their variances $\hat{\sigma}^{2\,strat}$ (right) with the Stratified Sampling method, using $N = 128$ extractions.

With such method we find:

$$I = 1.7179 \pm 0.0007 \tag{5.14}$$

We then fit in Figure 5.4 the behaviour of the standard deviation σ_I with respect to the number of extraction N. In such way we find:

$$\kappa = 0.2545 \pm 0.0012 \tag{5.15}$$

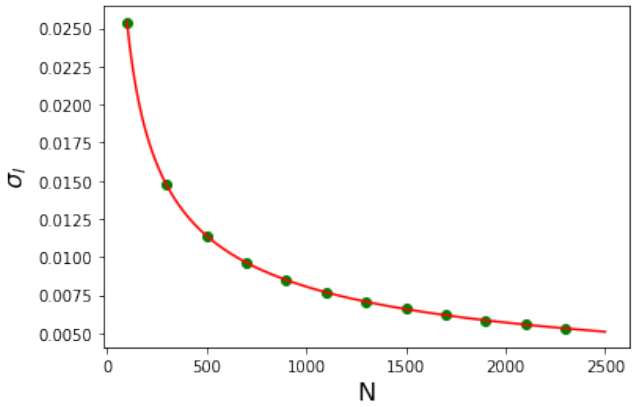


Figure 5.4: Values of σ_I varying with N (green dots) and fitting function $\frac{\kappa}{\sqrt{N}}$ (red line) obtained with Stratified Sampling method.

5.4 Importance Sampling

In the end we see the importance sampling method, that is the most powerful in reducing the variance. Instead of extracting numbers from a uniform distribution, we extract them from a distribution that has a similar shape as the integrand function. Intuitively, importance sampling attempts to place more samples where the contribution of the integrand is high, or "important".

The mathematical concept to implement this method is a simple change of variable:

$$I = \int_0^1 f(x)dx = \int_0^1 \frac{f(x)}{w(x)} w(x)dx \tag{5.16}$$

where $w(x)$ is the sampling distribution. Then, if $w(x)$ is normalised in the interval [0, 1], we can just extract N numbers x_i under this distribution and evaluate the integral using the following estimator:

$$Q_N^{imp} = \frac{1}{N} \sum_{i=1}^{N} \frac{f(x_i)}{w(x_i)} \tag{5.17}$$

For this exercise, we use two different sampling pdfs which have been computed by hand:

$$w_1(x) = \frac{1}{e} [1 + 2(e-1)x] \tag{5.18}$$

$$w_2(x) = \frac{1}{e+1.5} \left[2.5 + 2.5(e-1)x^{1.5}\right] \tag{5.19}$$

The algorithm is analogous to that for crude MC, with the only difference in the first point, with numbers generated according to w_1 and w_2. In particular, we expect a better result with w_2, since it is more similar to the integrand with respect to w_1, as shown in Figure 5.5.

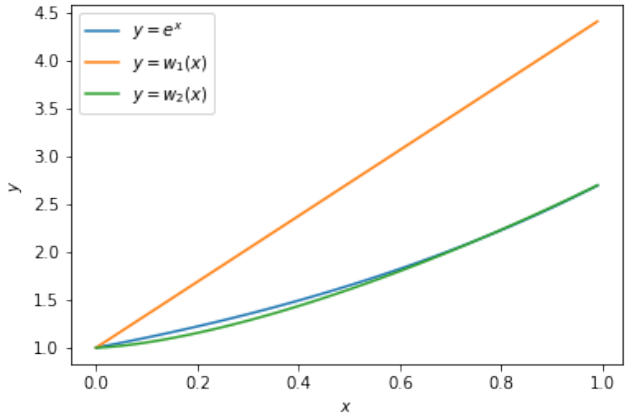


Figure 5.5: Comparison between the sampling functions and the integrand function e^x

5.4.1 Generating Algorithms

In order to generate values distributed as w_1 and w_2 we use two different methods:

1. The first one is called "inverse function method" and can be used when we can easily invert the pdf. If we want to extract numbers following the distribution $w(x)$, we need first to compute its cumulative function:
$$W(x) = \int_0^x w(t)dt \tag{5.20}$$
and to compute its inverse $W^{-1}(x)$.
Next, we extract uniform random numbers x_i in the codomain: $x_i \in W$. The values of $W_i^{-1} = W^{-1}(x_i)$ are then distributed according to $w(x)$.
We use this method for w_1 while for w_2, which is not so easy to be inverted, we use a second and less efficient method.

2. The second method is the rejection-sampling method, widely described in the section 1.5. It consist in extracting pairs of random numbers $x_i \in \mathcal{D}$ and $y_i \in \mathcal{C}$, where $\mathcal{D}$ and $\mathcal{C}$ are respectively the domain of integration the codomain. Then we compute the value of $w(x_i)$ and compare it with y_i: If we find $y_i < w(x_i)$ we keep x_i.

We repeat the process until we get the needed number of values of x_i, that are distributed as desired.

5.4.2 w_1 sampling function

We sample the points using the function 5.18 and we obtain the results shown in Figure 5.6.

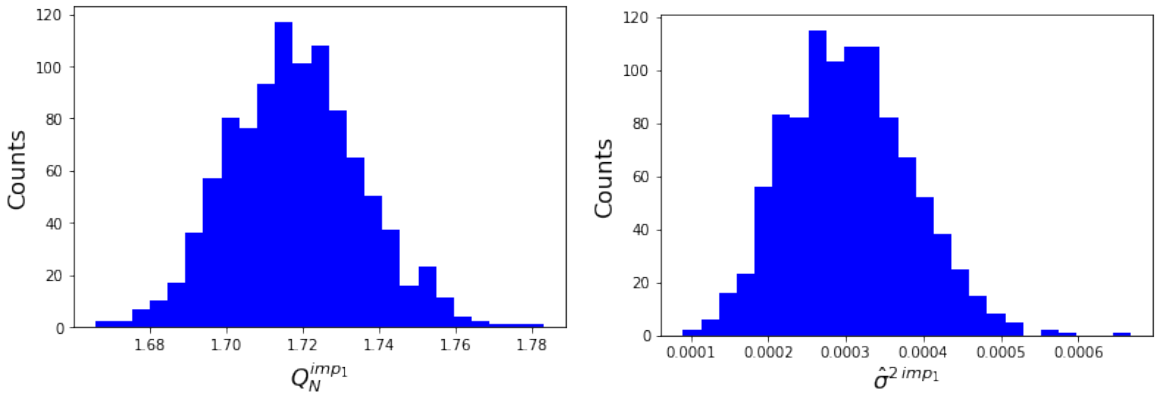


Figure 5.6: Histograms containing the values of 1000 computations of $Q_N^{imp_1}$ (left) and their variances $\hat{\sigma}^{2\,imp_1}$ (right) with the Importance Sampling method with w_1, using $N = 128$ extractions.

With the Importance Sampling method we have the following result:

$$I = 1.7181 \pm 0.0005 \tag{5.21}$$

We then fit in Figure 5.7 the behaviour of the standard deviation σ_I with respect to the number of extraction in each computation. In this way we find:

$$\kappa = 0.1984 \pm 0.0002 \tag{5.22}$$

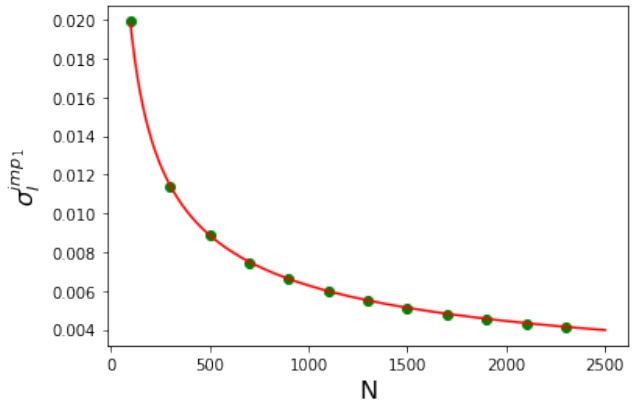


Figure 5.7: Values of σ_I varying with N (green dots) and fitting function $\frac{1}{\sqrt{N}}$ function (red line) obtained with Importance Sampling method with w_1.

5.4.3 w_2 sampling function

We sample the points using the function 5.19 and we obtain the results shown in Figure 5.8.

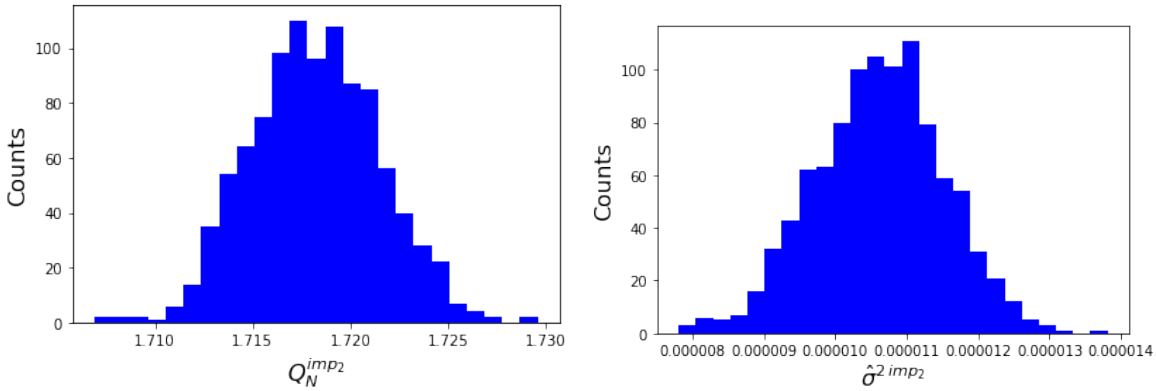


Figure 5.8: Histograms containing the values of 1000 computations of $Q_N^{imp_2}$ (left) and their variances $\sigma^{2\,imp_2}$ (right) with the Importance Sampling method with w_2, using $N = 128$ extractions.

This computation gives the following result:

$$I = 1.7182 \pm 0.0001 \tag{5.23}$$

We finally fit in Figure 5.9 the behaviour of the standard deviation σ_I with respect to the number of extraction in each computation. With this second sampling function we find:

$$\kappa = 0.0372 \pm 0.0002 \tag{5.24}$$

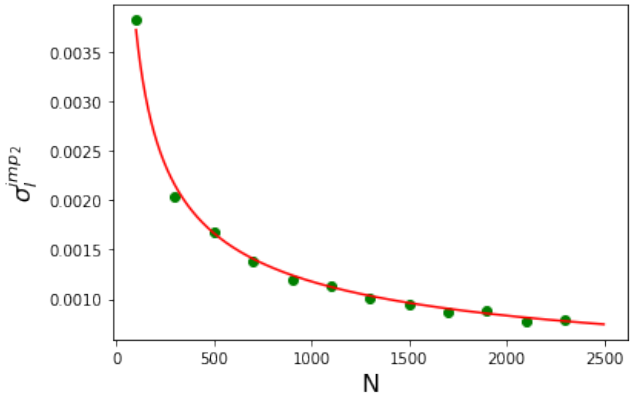


Figure 5.9: Values of σ_I varying with N (green dots) and fitting function $\frac{\kappa}{\sqrt{N}}$ (red line) obtained with Importance Sampling method with w_2.

5.5 Comparison

Finally we show the results found using each method. Figure 5.10 shows the comparison between the behaviours of σ_I in function of N with each algorithm.

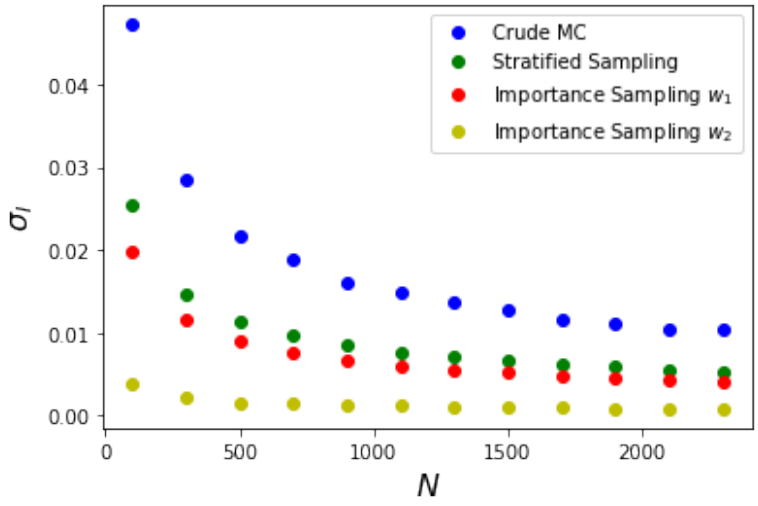


Figure 5.10: Behaviour of σ_I as function of the number of extractions N with every MC method analysed.

The reduction in the value of σ_I can be also seen in Figure 5.11, that shows the narrowing of the gaussian of the 1000 values of Q_N obtained with every method.

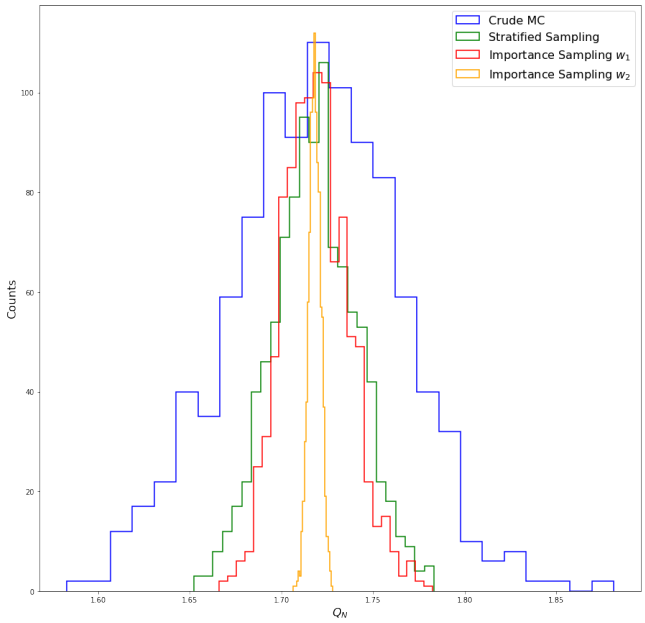


Figure 5.11: Histograms containing the values of 1000 computations of Q_N using $N = 128$ extraction with every MC method analysed

.

Another comparison can be done on the values of κ obtained with each algorithm.

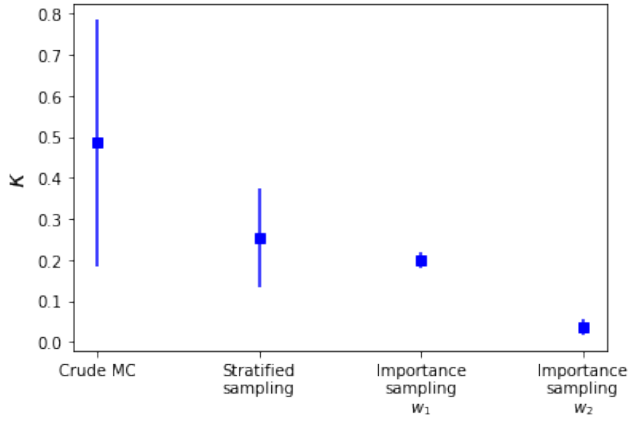


Figure 5.12: Results of κ obtained with every MC method used with relative standard deviations multiplied by 100 to be visible.

Finally, the results of the integration I are shown in Figure 5.13.

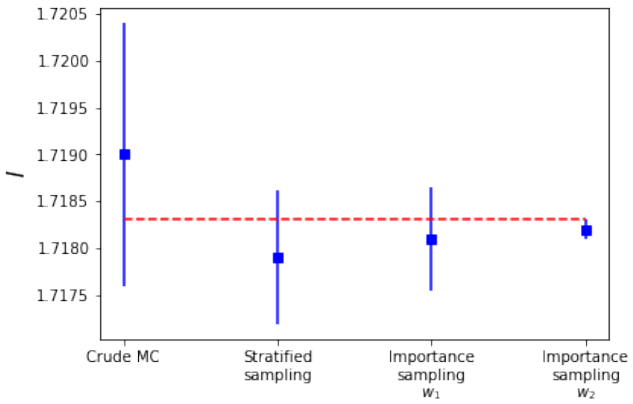


Figure 5.13: Results of I obtained with every MC method analysed with relative standard deviations (blue) together with the expected value of $I = 1.7183$ (red).

.

It is clear that, thanks to the results shown so far, with the chosen strata and sampling functions, the reduction of the variance is remarkable, especially for the importance sampling using w_2.
However, all of them are compatible within 1σ with the result of the integral (5.1) found with WolframAlpha:

$$I \simeq 1.71828182845905 \tag{5.25}$$

Exercise 6

Task

1. *Consider the diffraction caused by a circular aperture of width d and simulate the spectrum of the intensity I on a screen at distance L >> d from the source. For a sample of 50000 events plot a histogram of I with a bin width $\Delta\theta$ suitably chosen (where θ is the diffraction angle).*
2. *Apply a gaussian smearing with $\sigma = c \cdot \Delta\theta$ with $c \in (0, 1)$ and discuss about the effect of c on the resolution.*
3. *Applying one of the regularisation techniques seen during the lectures, unfold the experimental distributions so-built, and discuss about the reconstructed distribution.*

6.1 Theoretical Introduction

Any measurement is affected by the finite resolution of the particle detectors. This causes the observed spectrum of events to be "smeared" or "blurred" with respect to the true one. The unfolding problem is to estimate the true spectrum using the smeared observations.
So we have the true, particle-level spectrum and the smeared, detector-level spectrum grouped into two histograms. We introduce the detector *response matrix* A, such that A_{ij} is the probability that an event from truth bin j is found in the reconstructed bin i. The expected number of events in the bin i is thus given by:

$$\mu_i^{rec} = \sum A_{ij} \nu_j^{truth}$$

6.2 Intensity Spectrum Generation

Python is an easy open source software that can be used to simulate various optical phenomena. Here we are asked to simulate the diffraction pattern resulting from a uniformly illuminated circular aperture. The bright central region is known as the Airy Disk, which together with the series of concentric rings around is called the Airy Pattern.
The intensity profile of the Airy Pattern can be calculated using the Fraunhofer diffraction for a circular aperture far away from the aperture. It can be derived with the following formula:

$$I(\theta) = \left(\frac{2J_1(ka \sin\theta)}{ka \sin\theta} \right)^2 \tag{6.1}$$

where:

- $I(\theta)$ is the intensity at a given angle,
- θ is the diffraction angle,
- a is the radius of the aperture,
- $k = \frac{2\pi}{\lambda}$, λ is the wavelength of the light,
- J_1 is the Bessel function of the first kind of order one.

This analysis applies only to the far field, namely at a distance much larger than the width of the slit.
We can therefore simulate these particles thanks to the Python library `Scipy.special` and finally we obtain a dataset of 50000 particles distributed with the pdf 6.1 as shown in Figure 6.1.

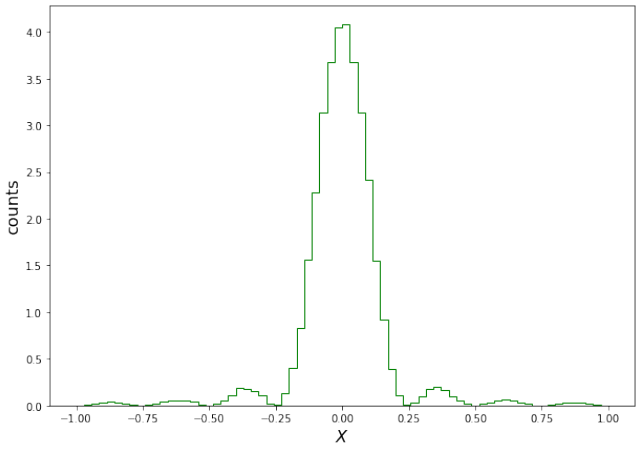


Figure 6.1: Representation of the intensity spectrum of 50000 events generated with the Scipy library. In particular, we have used $a = 2\,\mu m$ and $\lambda = 500\,nm$. The binning of the histogram has chosen to be equal to 70.

6.3 Gaussian Smearing

After having generated 50000 MC events, we apply a gaussian smearing with the parameter

$$\sigma = c \cdot \Delta\theta$$

Since the number of bins has chosen to be 70, the bin width is about $\Delta\theta = 0.03$.
To apply the gaussian smearing, we replace every generated θ, with a random number chosen in the gaussian that has that value of θ as mean value and standard deviation $\sigma = 0.03 \cdot c$.
We then see how the resolution changes when c varies within $(0, 1)$. As can be seen from Figure 6.2, the greater is c, the worse the resolution is. In fact, the major effect of the smearing can be seen in the mean peak: with a consistent smearing, the height of the peak decreases a lot. Moreover, a minor effect is noticed on the secondary peaks, that flatten out a bit when a smearing is applied.
The effect, however, can be appreciated with higher values of c. For example, using $c = 2$ we obtained the results shown in Figure 6.3.

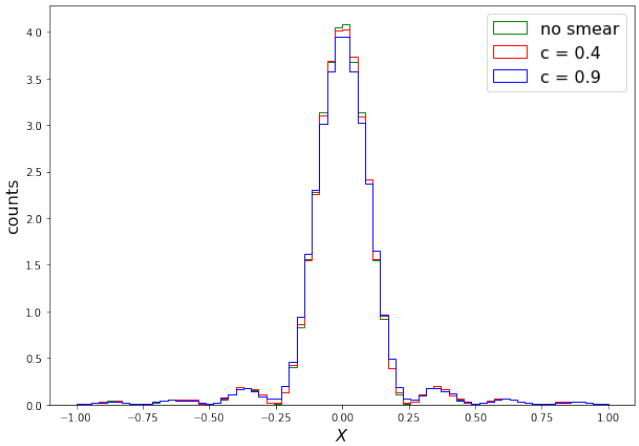


Figure 6.2: Representation of the intensity spectrum with a gaussian smearing applied.

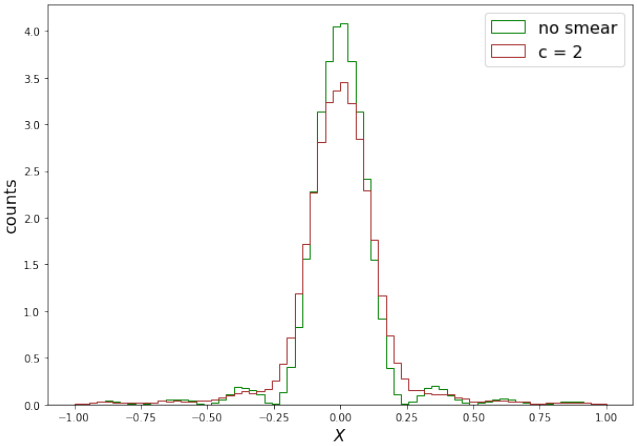


Figure 6.3: Representation of the intensity spectrum with a gaussian smearing with a higher σ applied.

6.4 Unfolding

PyUnfold uses the `iterative_unfold` function to perform iterative unfoldings. In addition, we'll use Logger callback, that writes test statistics information for each unfolding iteration.
We consider as observed data the ones generated with a Gaussian smearing characterised by $c = 0.9$. In addition to the observations themselves, the uncertainty of the observations is also needed to perform an unfolding. Here, we'll assume the errors are simply Poisson counting errors. In the i bin, the error is then given by the square root of the counting rate in that very bin $\epsilon_i = \sqrt{N_i}$.

Not every sample from our true distribution will lead to a measured observation. The probability that a sample from our true distribution is observed is quantified via our detection efficiencies. For now, we'll assume uniform efficiencies of 1 and an uncertainty on our efficiencies of 0.01.

The next step is to construct a response matrix that encapsulate our detector bias and resolution. This can be done by making a 2-dimensional histogram of the true vs. observed distributions. As with the uncertainties on our observed data, we'll assume Poisson counting errors for our response matrix. Doing so we obtain the result shown in Figure 6.4.
However, there is still one more step to do to obtain the response matrix. What's needed for unfolding is not this 2-dimensional counts histogram, but the "normalised response matrix", that speaks in probability language (Figure 6.5). The difference between the two matrices is due to the fact that each column in the normalised response matrix is scaled by the number of samples in that column. So only the relative structure of each column of the 2D response histogram is present in the normalised response matrix.

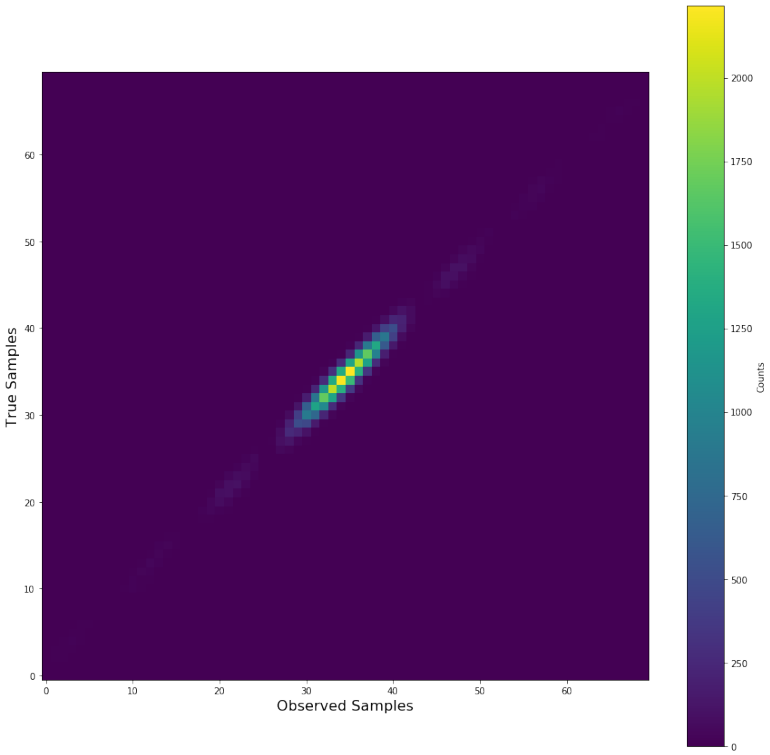


Figure 6.4: 2-dimensional histogram representing the true vs. observed distributions.

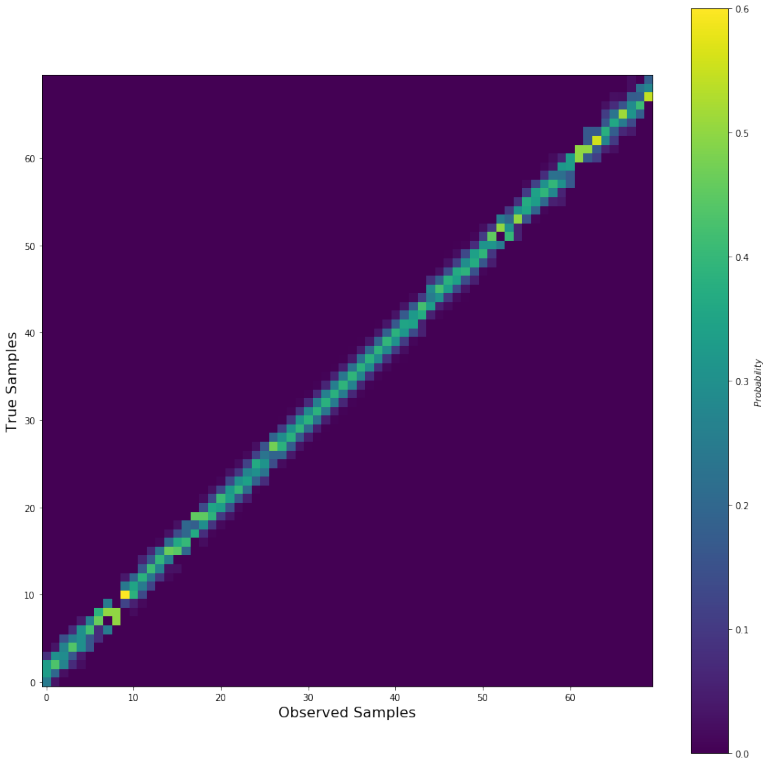


Figure 6.5: Representation of the normalised response matrix.

Now we have everything we need to perform our unfolding:

- Observed distribution to unfold;
- Detection efficiencies;
- Normalised response matrix.

These are all input parameters into the PyUnfold `iterative_unfold` function, that returns a dictionary that contains, among the other things, the unfolded distribution with its statistical and systematic uncertainties. Finally, we can see how the true and unfolded distributions compare to one another in Figure 6.6. As can be seen, the unfolded distribution is consistent with the true distribution.

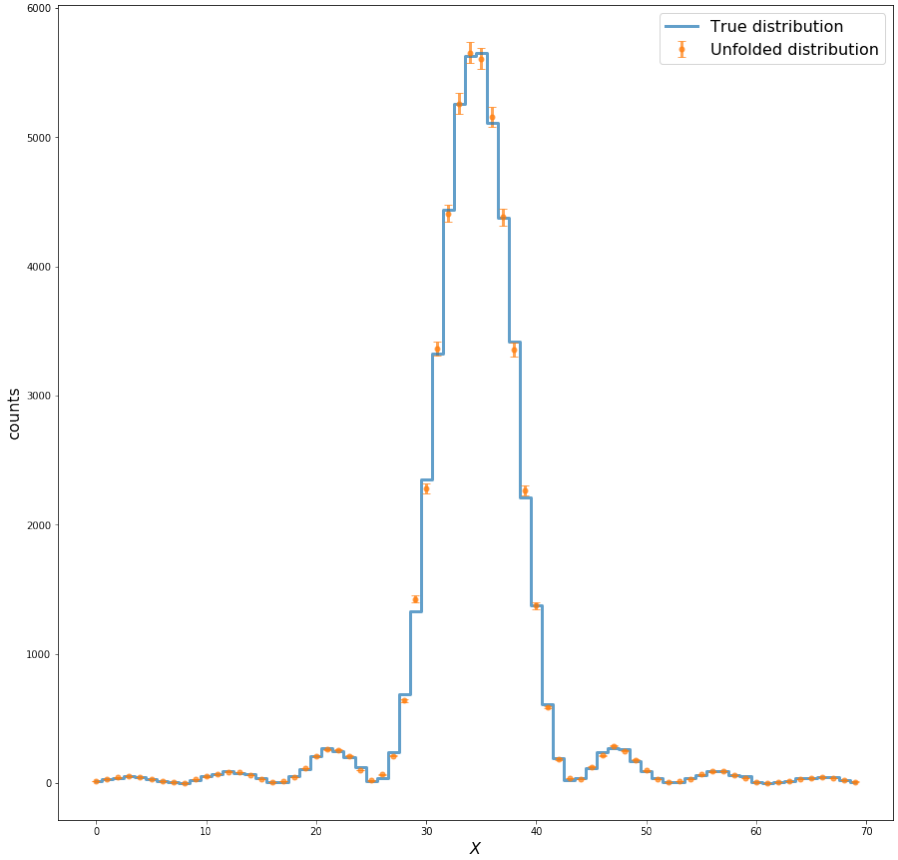


Figure 6.6: True, observed and unfolded distributions compare to one another.

Exercise 7

Task

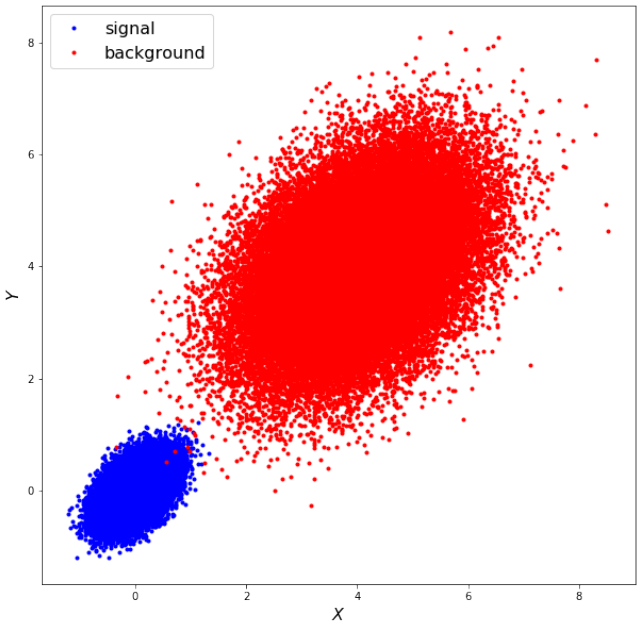
Generate two classes of events:

- *the signal with a 2D gaussian pdf centred in* $(x, y) = (0, 0)$*, with* $\sigma_x = \sigma_y = 0.3$ *and* $\rho = 0.5$*;*
- *the background, with a 2D gaussian pdf centred in* $(x, y) = (4, 4)$*,* $\sigma_x = \sigma_y = 1$ *and* $\rho = 0.4$*.*

Use one of the MVA method seen during the lectures and characterise the result in terms of signal purity and background rejection.

7.1 Generation of MC Events

First of all, we generate 50000 signal events and 50000 background events with the two 2D gaussians described in the task using the numpy function `np.random.multivariate_normal`. The results are shown in Figure 7.1.



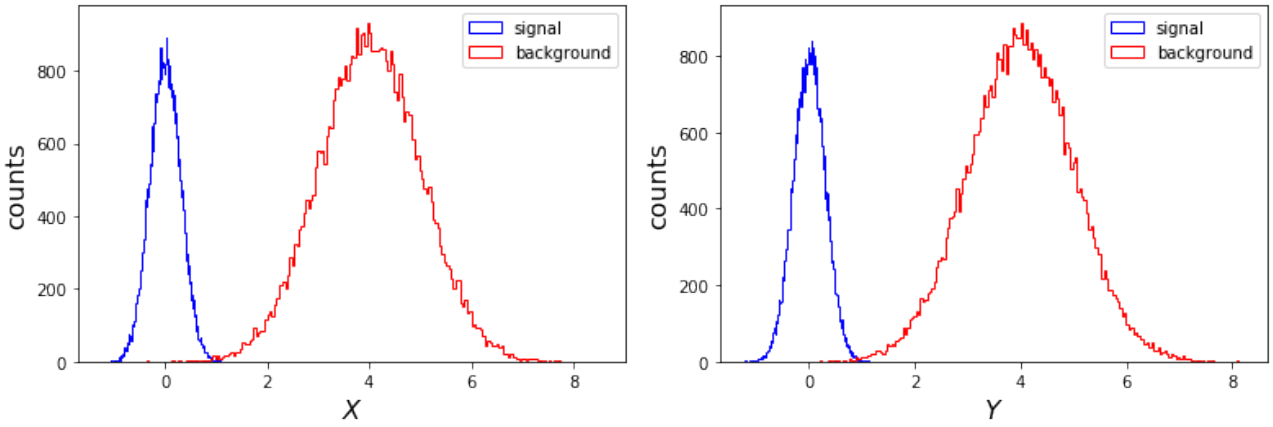


Figure 7.1: The plots showing the two 2D gaussian functions representing the signal (blue) and the background (red).

7.2 Theoretical Introduction on Machine Learning

To implement the exercise we used a Machine Learning (ML) algorithm which is part of the ML branch known as "supervised learning". But let's take a step back and see which are the three main branches of ML:

- Supervised Learning: we provide the algorithm with the targets, i.e. the desired outputs, and the algorithm learns to produce outputs as close as possible to them. Classification and Regression are two Supervised Learning methods.
- Unsupervised Learning: The model does not have targets but it has to find an underlying logical dependency in the provided data. Clustering is a Unsupervised Learning method.
- Reinforcement Learning: The model aims to get the reward or avoid punishment (ex. GANs).

In every Machine Learning method, a training is usually performed on a data set. Subsequently the results of the model predictions have to be verified with another set containing new data. The model optimises the parameters on the train set, but it must be able to generalise on unknown data. So it must not be built only to work on the train set – a condition known as "overtraining".

Let's focus on the Supervised Learning algorithms. Mathematically, the problem of Supervised Learning can be reduced to:

- I have one or more independent variables X ("features") and a dependent variable Y ("target"),
- I have to find a function f such that $f(X) = Y$.

To have them work we have to define a target function, which measures the predictive power of our model, and so it measures how well the outputs of the model correspond to the targets. In supervised learning the target function is called "loss function", and it must be minimised. In this project we use as optimisation algorithm the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm, an iterative method for solving unconstrained nonlinear optimisation problems.

We use the Binary Classification, and so we try to build a model that predicts the values of a binary target Y starting from the values of one or more continuous features X. Since more features X are considered, we need to make sure that they are included in the common range of values. To do so, we can then use two methods:

- Normalisation: We bring all the data in a range between 0 and 1,
- Standardisation: We convert the data into a normal distribution with mean 0 and standard deviation 1.

In our project, we decide to standardise our data.

When we work with 2 features and 2 classes, we can represent them in a scatter plot and the problem of classification is summed up in finding the equation of the line that better divides the two classes.

Once we have obtained our model, we can check its quality using two metrics:

- Accuracy: Counts how many of the classifications made by the model are correct. It's a percentage value, so it goes from 0 to 1 and the higher the better.
- Negative Log-Likelihood: Takes into account the probability that the classification is wrong. This value also goes from 0 to 1, but in this case the less the better.

Another way to check the model quality is the confusion matrix: Comparable to the Accuracy, it is a table in which the predicted values are on the abscissa and the true values on the ordinate. A perfect confusion matrix should have 1 on the main diagonal and 0 otherwise. It shows the false positives and false negatives.

7.2.1 Exercise Implementation

To implement the classification, we use the `scikit-learn` library implemented in Python. First of all, we can represent our data in a DataFrame containing the two coordinates of all the points and the binary target:

- 1.0 if the event belongs to the signal,
- 0.0 if it belongs to the background.

In such way we are left with a 3-column dataset, containing the 2 features x and y and the target s_b. Below, an example of 5 points in the DataFrame.

	x	y	s_b
0	0.074097	-0.314378	1.0
1	0.175280	0.149286	1.0
2	0.022915	-0.056733	1.0
3	0.098645	0.257120	1.0
4	0.291148	-0.144283	1.0

We divide the dataset into train set and test set using the `train_test_split` function, allocating the 70% of the data in the train set and the 30% in the test set. Then we want to standardise our data, and so we use the function `transform` of the `StandardScaler` method.

Now we are ready to create our classification model. We will implement the logistic regression, which returns not only the class in output but also the probability the probability of how correct the classification is. Using the method `LogisticRegression` we implement our model. Subsequently, we want to evaluate how good it actually is, and to do so we can use the test set and the two metrics shown above, namely the Accuracy and the Negative Log-likelihood. Both functions are implemented in `scikit-learn`. We obtain as result:

```
accuracy = 1.0
log loss = 0.0
```

These values are actually perfect.

Then we want to see all the values classified incorrectly in our dataset. To do so, we build a new DataFrame with the target values of test and predicted by the model. We find 0 false negative and 6 false positive. We can therefore build our confusion matrix:

B pred	0	15067
S pred	14927	6
	S test	**B test**

We can therefore state that the signal purity is:

$$sig\_purity = \frac{S\,test}{S\,pred} = 99.95\%$$

while the background rejection is equal to

$$bkg\_rejection = \frac{B\,pred}{B\,test} = 99.96\%$$

Finally we can see the decision boundary that the model has learnt in Figure 7.2.

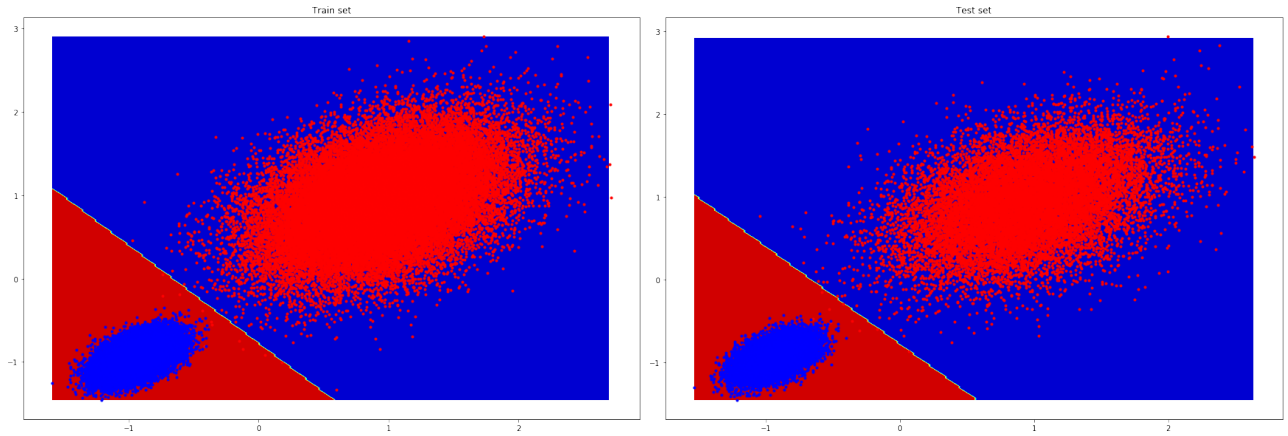


Figure 7.2: The plots showing the two 2D gaussian functions representing the signal (blue) and the background (red) along with the decision boundary predicted by the model. The left plot represents the train set, while the right one the test set. In this latter, the 6 false positive can be seen.