

# Analisi Statistica Dei Dati

Wahid Redjeb: 809147  
Giacomo Boldrini: 800692  
a.a. 2018/2019

---

Tutti i codici riportati nelle seguenti pagine sono integralmente visionabili ai seguenti link:

GitHub: Giacomo Boldrini

GitLab: Wahid Redjeb

---

# Indice

<b>1 Esercizio 1</b>	<b>4</b>
1.0.1 Costruzione del generatore . . . . .	4
1.0.2 Classe prng . . . . .	5
1.0.3 Test dieharder . . . . .	6
1.1 Algoritmo xorshiro128p . . . . .	9
1.1.1 Test dieharder . . . . .	11
1.2 Distribuzione di Landau . . . . .	14
1.3 Classe xorshiro128p . . . . .	16
1.4 Codice creazione Landau e analisi . . . . .	18
<b>2 Esercizio 2</b>	<b>21</b>
2.1 Rappresentazione dei numeri sul calcolatore . . . . .	21
2.1.1 Sistema posizionale . . . . .	21
2.1.2 Rappresentazione fixed point . . . . .	21
2.1.3 Rappresentazione floating point . . . . .	22
2.2 Sorgenti di errore . . . . .	22
2.2.1 Roundoff error . . . . .	22
2.2.2 Errore di troncamento . . . . .	23
2.2.3 Stabilità . . . . .	23
2.3 Calcolo della costante . . . . .	24
2.3.1 Studio degli errori . . . . .	24
2.3.2 Algoritmo sommatoria Kahan . . . . .	26
2.3.3 Pairwise summation . . . . .	28
2.4 Risultati . . . . .	30
2.5 Codice per l'analisi dei metodi . . . . .	33
<b>3 Esercizio 3</b>	<b>37</b>
3.1 Creazione Pdf e generazione eventi Monte Carlo . . . . .	37
3.2 Unbinned ML fit . . . . .	40
3.3 Likelihood plots . . . . .	42
3.4 Esperimenti Monte Carlo ripetuti . . . . .	44
3.5 Binning dei dati e fit $\chi^2$ . . . . .	48
3.6 Comparazione risultati . . . . .	51
3.7 Test ipotesi con distribuzione uniforme . . . . .	51
3.8 Likelihood ratio . . . . .	54

<b>4 Esercizio 4</b>	<b>57</b>
4.1 Generazione degli eventi . . . . .	57
4.2 Distributione $\langle P_T \rangle$ . . . . .	60
4.3 Distribuzione di probabilità del $P_T$ . . . . .	62
<b>5 Esercizio 5</b>	<b>67</b>
5.1 Descrizione delle classi . . . . .	68
5.2 Crude MC . . . . .	72
5.3 Tecniche di riduzione della varianza . . . . .	79
5.3.1 Stratified Sampling . . . . .	79
5.3.2 Importance sampling . . . . .	86
5.3.3 Antithetic Variates . . . . .	94
5.4 Comparazione finale . . . . .	97
5.5 Analisi con generatore xorshiro . . . . .	99
5.5.1 Crude MC xorshiro . . . . .	99
5.5.2 Stratified Sampling xorshiro . . . . .	102
5.5.3 Importance sampling xorshiro . . . . .	104
5.5.4 Antithetic Variates xorshiro . . . . .	106
5.6 Risultati xorshiro . . . . .	107
<b>6 Esercizio 6</b>	<b>108</b>
6.1 Generazione $I(\theta)$ Monte Carlo . . . . .	110
6.2 Smearing gaussiano . . . . .	115
6.3 Unfolding della distribuzione sperimentale . . . . .	118
6.3.1 Bayes Approach . . . . .	122
6.3.2 Bin-by-Bin Unfolding . . . . .	126
6.3.3 SVD unfolding . . . . .	128
6.4 Comportamento ad alte $\sigma$ . . . . .	131
<b>7 Esercizio 7</b>	<b>135</b>
7.1 Modelli Deterministici . . . . .	139
7.2 Discriminante quadratico . . . . .	141
7.3 Percettrone . . . . .	148
7.4 Support vector machine . . . . .	154
7.5 ANN . . . . .	165

# Capitolo 1

## Esercizio 1

Il primo esercizio chiedeva di costruire un generatore di numeri casuali distribuiti secondo una distribuzione di Landau e di discutere i possibili tests di casualità utilizzabili per i numeri generati.

### 1.0.1 Costruzione del generatore

Una sequenza di numeri casuali è impredicibile e perciò irriproducibile. Possiamo prima di tutto distinguere tra

- Numeri casuali ("Truly random numbers")
- Numeri pseudo-casuali ("Pseudo random numbers")

I primi possono essere generati da un processo fisico random, come ad esempio un decadimento radioattivo o i tempi di arrivo dei raggi cosmici.

I secondi sono sequenze di numeri generate da algoritmi computazionali eseguiti su calcolatori. Una sequenza di numeri pseudocasuali è ottenuta con una ben precisa formula matematica che fornisce una sequenza indistinguibile da una di numeri casuali. Queste relazioni vengono chiamate *generatori* di numeri pseudocasuali, in inglese *pseudo random number generator (prng)*. I prng per essere considerati tali devono essere soggetti a test di *randomness* e avere un periodo. Di seguito si descrivono due algoritmi implementati per la generazione di numeri pseudo casuali.

### Metodo lineare congruente

Il primo algoritmo utilizzato viene chiamato generatore lineare congruente (LCG). Questo è basato sulla seguente relazione di ricorrenza:

$$X_{n+1} = (aX_n + c) \cdot \text{mod}(m)$$

dove

- $X_n$  è l' $n$ -esimo numero casuale generato;
- $m$  è il modulo, con  $m > 0$ ;
- $a$  è il parametro moltiplicatore;

- $c$  è l'incremento;
- $X_0$  è il seed, con  $0 < X_0 < m$ .

Il periodo di un LCG è al più  $m$  e per alcune scelte di  $a$  può essere molto più piccolo. Un LGC ha un periodo completo se e solo se:

- $c$  e  $m$  sono coprimi
- $a - 1$  è divisibile per tutti i fattori primi di  $m$
- $a - 1$  è un multiplo di 4 se  $m$  è un multiplo di 4

Come parametri sono stati scelti quelli utilizzati per la funzione `rand()` implementata in C++. Questa funzione utilizza l'architettura di Donald Knuth, come descritta in questa pagina [Wik].

I parametri scelti sono quindi i seguenti:

- $a = 6364136223846793005$ ;
- $c = 1442695040888963407$ ;
- $m = 2^{64}$

Il generatore è stato implementato in una classe chiamata `prng`, si rimanda il lettore alla sezione di Listing per l'implementazione dei metodi.

## 1.0.2 Classe `prng`

Header file

```
#ifndef PRNG_H
#define PRNG_H

//definisco la variabile randtype.
//unsigned (solo 0 o var positive), long long -> 64bit di memoria.
typedef unsigned long long int randtype;

class prng{

private:
    randtype a;
    randtype c;
    randtype x;

public:
    prng(); //constructor
    ~prng();

    randtype randInt(); //numero casuale
    double rand(double min=0., double max =1.); //numero casuale tra 0 e 1
}
```

```

};

#ifndef PRNG_H
#define PRNG_H

#include "prng.h"
#include <climits>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

//Constructor

prng::prng(){
    a = 6364136223846793005U;
    c = 1442695040888963407U;
    x = 1.;
}

prng::~prng(){}
randtype prng::randInt(){
    x = (a * x + c);
    return x;
}

//genera numeri casuali in [min,max]
double prng::rand(double min, double max){

    return (double)(max-min) * ((double) randInt()) /
        ((double) ULLONG_MAX ) + min;
}

```

### 1.0.3 Test dieharder

Una volta creato il generatore si effettuano dei test di randomness per studiare la bontà del generatore. Per questo scopo si è sfruttata la *testing suite dieharder*.

I test diehard sono dei test statistici che misurano la qualità di un generatore di numeri casuali.

Per interfacciarsi con dieharder e poter effettuare tutti i test bisogna utilizzare file di grandi dimensioni (>2GB). Tuttavia, per ovviare a questo problema, è possibile utilizzare il *piping*.

Per effettuare i test si implementa quindi il seguente codice per poter sfruttare il piping.

```
#include "prng.h"
#include <climits>
```

```

#include <iostream>

int main(int argc, char** argv){

    prng gen;

    while(1){

        randtype rnd_num = gen.randInt();
        std::cout.write(reinterpret_cast<char*>(&rnd_num), sizeof rnd_num);

    }

    return 0;
}

```

Per quanto riguarda l'algoritmo LCG si ottengono i seguenti risultati:

```

>./die_lgc |dieharder -a -g 200
=====
#           dieharder version 3.31.1 Copyright 2003 Robert G. Brown      #
=====
rng_name      |rands/second|   Seed   |
stdin_input_raw| 2.43e+07 |2211827448|
=====
test_name    |ntup| tsamples |psamples| p-value |Assessment
=====
diehard_birthdays| 0|     100| 100|0.50737037| PASSED
    diehard_operm5| 0| 1000000| 100|0.06833705| PASSED
diehard_rank_32x32| 0|     40000| 100|0.75095569| PASSED
    diehard_rank_6x8| 0|    100000| 100|0.00000000| FAILED
    diehard_bitstream| 0| 2097152| 100|0.00000000| FAILED
        diehard_opsol| 0| 2097152| 100|0.00164948| WEAK
        diehard_oqso| 0| 2097152| 100|0.00000000| FAILED
        diehard_dna| 0| 2097152| 100|0.00000000| FAILED
diehard_count_1s_str| 0| 256000| 100|0.63288882| PASSED
diehard_count_1s_byt| 0| 256000| 100|0.00000000| FAILED
    diehard_parking_lot| 0|    12000| 100|0.35982598| PASSED
    diehard_2dsphere| 2|     8000| 100|0.31311681| PASSED
    diehard_3dsphere| 3|     4000| 100|0.42586843| PASSED
    diehard_squeeze| 0|    100000| 100|0.62811155| PASSED
    diehard_sums| 0|      100| 100|0.14828954| PASSED
    diehard_runs| 0|    100000| 100|0.01881563| PASSED
    diehard_runs| 0|    100000| 100|0.11081012| PASSED
    diehard_craps| 0|    200000| 100|0.40802577| PASSED
    diehard_craps| 0|    200000| 100|0.88815500| PASSED

```

marsaglia_tsang_gcd	0	10000000	100 0.80024006	PASSED
marsaglia_tsang_gcd	0	10000000	100 0.30190217	PASSED
sts_monobit	1	100000	100 0.05100784	PASSED
sts_runs	2	100000	100 0.18129480	PASSED
sts_serial	1	100000	100 0.80976811	PASSED
sts_serial	2	100000	100 0.10764077	PASSED
sts_serial	3	100000	100 0.00203334	WEAK
sts_serial	3	100000	100 0.04826766	PASSED
sts_serial	4	100000	100 0.00045636	WEAK
sts_serial	4	100000	100 0.04285806	PASSED
sts_serial	5	100000	100 0.00000295	WEAK
sts_serial	5	100000	100 0.00170849	WEAK
sts_serial	6	100000	100 0.00000000	FAILED
sts_serial	6	100000	100 0.00000002	FAILED
sts_serial	7	100000	100 0.00000000	FAILED
sts_serial	7	100000	100 0.00000823	WEAK
sts_serial	8	100000	100 0.00000000	FAILED
sts_serial	8	100000	100 0.00000048	FAILED
sts_serial	9	100000	100 0.00000000	FAILED
sts_serial	9	100000	100 0.00000000	FAILED
sts_serial	10	100000	100 0.00000000	FAILED
sts_serial	10	100000	100 0.00000000	FAILED
sts_serial	11	100000	100 0.00000000	FAILED
sts_serial	11	100000	100 0.00000000	FAILED
sts_serial	12	100000	100 0.00000000	FAILED
sts_serial	12	100000	100 0.00000000	FAILED
sts_serial	13	100000	100 0.00000000	FAILED
sts_serial	13	100000	100 0.00000000	FAILED
sts_serial	14	100000	100 0.00000000	FAILED
sts_serial	14	100000	100 0.00000000	FAILED
sts_serial	15	100000	100 0.00000000	FAILED
sts_serial	15	100000	100 0.00000000	FAILED
sts_serial	16	100000	100 0.00000000	FAILED
sts_serial	16	100000	100 0.00000000	FAILED
rgb_bitdist	1	100000	100 0.57831363	PASSED
rgb_bitdist	2	100000	100 0.00000000	FAILED
rgb_bitdist	3	100000	100 0.00000000	FAILED
rgb_bitdist	4	100000	100 0.00000000	FAILED
rgb_bitdist	5	100000	100 0.00000001	FAILED
rgb_bitdist	6	100000	100 0.00000017	FAILED
rgb_bitdist	7	100000	100 0.36170970	PASSED
rgb_bitdist	8	100000	100 0.99974691	WEAK
rgb_bitdist	9	100000	100 0.36874390	PASSED
rgb_bitdist	10	100000	100 0.90721720	PASSED
rgb_bitdist	11	100000	100 0.10196733	PASSED
rgb_bitdist	12	100000	100 0.00280730	WEAK
rgb_minimum_distance	2	10000	1000 0.35398627	PASSED

rgb_minimum_distance	3	10000	1000 0.85473125	PASSED
rgb_minimum_distance	4	10000	1000 0.35565112	PASSED
rgb_minimum_distance	5	10000	1000 0.05918343	PASSED
rgb_permutations	2	100000	100 0.82743915	PASSED
rgb_permutations	3	100000	100 0.73804656	PASSED
rgb_permutations	4	100000	100 0.64362494	PASSED
rgb_permutations	5	100000	100 0.35801069	PASSED
rgb_lagged_sum	0	1000000	100 0.74591474	PASSED
rgb_lagged_sum	1	1000000	100 0.16837719	PASSED
rgb_lagged_sum	2	1000000	100 0.15315890	PASSED
rgb_lagged_sum	3	1000000	100 0.46004558	PASSED
rgb_lagged_sum	4	1000000	100 0.24961485	PASSED
rgb_lagged_sum	5	1000000	100 0.84405686	PASSED
rgb_lagged_sum	6	1000000	100 0.15267119	PASSED
rgb_lagged_sum	7	1000000	100 0.05572633	PASSED

Come si può osservare molti dei test falliscono e altri sono deboli. Si passa quindi allo studio di un nuovo generatore più robusto.

## 1.1 Algoritmo xorshiro128p

Una trattazione formale dell'algoritmo xorshiro è visionabile nel seguente articolo [BV18].

L'algoritmo *xorshiro* è basato su trasformazioni lineari ottenute combinando una rotazione, uno shift e nuovamente una rotazione (da cui il nome), definendo quindi la seguente matrice.

$$\chi_{2w} = \begin{pmatrix} R^a + S^b + I & R^c \\ S^b + I & R^c \end{pmatrix}$$

L'algoritmo implementato per lo svolgimento dell'esercizio è *xorshiro128plus*, che prevede anche un'operazione di somma. Il generatore è stato implementato in una classe *xorshiro* strutturata nel seguente modo:

```
#ifndef XORSHIRO_
#define XORSHIRO_

#include "prng.h"

typedef long long unsigned int randtype;

class xorshiro
{
private:
    prng rnd;
    int a;
    int b;
```

```

int c;
randtype s[2];

public:
xorshiro();
~xorshiro();

randtype rotl(const randtype, int k);
randtype xorshiro128p();
double rand(double min = 0., double max = 1.);
double landau(double x, double mu = 0., double sigma = 1.);
double landau_tc(double mu = 0, double sigma = 1,
                  double xmin = 0, double xmax = 1,
                  double ymin = 0, double ymax = 0.5);
};

#endif //XOROSHIRO_

```

I metodi necessari per i test di randomness, oltre che al costruttore e al distruttore della classe, sono spiegati dal seguente codice:

```

#include "xorshiro.h"
#include "prng.h"
#include <cmath>
#include <climits>

xorshiro::xorshiro(){

    a = 53;
    b = 22;
    c = 41;

    s[0] = rnd.randInt();
    s[1] = rnd.randInt();

}

xorshiro::~xorshiro(){}
randtype xorshiro::rotl(const randtype x, int k){

    return (x << k) | (x >> (64 - k));
}

randtype xorshiro::xorshiro128p(){

    const randtype s0 = s[0];
    randtype s1 = s[1]; //not constant, must be modifiable var
}

```

```

const randtype sum = s0 + s1;

s1 ^= s0; //that's why s1 is not a const value.

//rotation
s[0] = rotl(s0,a) ^ s1 ^ (s1 << b);
s[1] = rotl(s1,c);

return sum;
}

double xorshiro::rand(double min, double max){

    return (double)(max-min) * ((double) xorshiro128p())
        / (double)(ULLONG_MAX) + min;
}

```

### 1.1.1 Test dieharder

I test dieharder sull'algoritmo xorshiro128p danno i seguenti risultati:

```

>./die_xorshiro128p |dieharder -a -g 200
=====
#           dieharder version 3.31.1 Copyright 2003 Robert G. Brown      #
=====
rng_name    |rands/second|   Seed   |
stdin_input_raw| 4.40e+07 |3364727400|
=====
test_name    |ntup| tsamples |psamples| p-value |Assessment
=====
diehard_birthdays| 0| 100| 100| 0.98126052| PASSED
diehard_operm5| 0| 1000000| 100| 0.78025793| PASSED
diehard_rank_32x32| 0| 40000| 100| 0.90584103| PASSED
diehard_rank_6x8| 0| 100000| 100| 0.55660772| PASSED
diehard_bitstream| 0| 2097152| 100| 0.51009734| PASSED
diehard_opso| 0| 2097152| 100| 0.35749432| PASSED
diehard_oqso| 0| 2097152| 100| 0.29327913| PASSED
diehard_dna| 0| 2097152| 100| 0.37856837| PASSED
diehard_count_1s_str| 0| 256000| 100| 0.99742415| WEAK
diehard_count_1s_byt| 0| 256000| 100| 0.81791107| PASSED
diehard_parking_lot| 0| 12000| 100| 0.91461926| PASSED
diehard_2dsphere| 2| 8000| 100| 0.99089191| PASSED
diehard_3dsphere| 3| 4000| 100| 0.85795911| PASSED
diehard_squeeze| 0| 100000| 100| 0.12095467| PASSED
diehard_sums| 0| 100| 100| 0.37261963| PASSED
diehard_runs| 0| 100000| 100| 0.64922085| PASSED
diehard_runs| 0| 100000| 100| 0.95149882| PASSED

```

diehard_craps	0	200000	100 0.15316267	PASSED
diehard_craps	0	200000	100 0.34924854	PASSED
marsaglia_tsang_gcd	0	10000000	100 0.81857868	PASSED
marsaglia_tsang_gcd	0	10000000	100 0.29503721	PASSED
sts_monobit	1	100000	100 0.65153908	PASSED
sts_runs	2	100000	100 0.27481384	PASSED
sts_serial	1	100000	100 0.01461527	PASSED
sts_serial	2	100000	100 0.46734519	PASSED
sts_serial	3	100000	100 0.48648681	PASSED
sts_serial	3	100000	100 0.98963137	PASSED
sts_serial	4	100000	100 0.64301838	PASSED
sts_serial	4	100000	100 0.89852194	PASSED
sts_serial	5	100000	100 0.19750874	PASSED
sts_serial	5	100000	100 0.83489901	PASSED
sts_serial	6	100000	100 0.93203538	PASSED
sts_serial	6	100000	100 0.61317107	PASSED
sts_serial	7	100000	100 0.14559841	PASSED
sts_serial	7	100000	100 0.00462871	WEAK
sts_serial	8	100000	100 0.00705352	PASSED
sts_serial	8	100000	100 0.15125225	PASSED
sts_serial	9	100000	100 0.01937396	PASSED
sts_serial	9	100000	100 0.64320684	PASSED
sts_serial	10	100000	100 0.02083360	PASSED
sts_serial	10	100000	100 0.45416045	PASSED
sts_serial	11	100000	100 0.79554638	PASSED
sts_serial	11	100000	100 0.19009721	PASSED
sts_serial	12	100000	100 0.73648562	PASSED
sts_serial	12	100000	100 0.96027437	PASSED
sts_serial	13	100000	100 0.79380811	PASSED
sts_serial	13	100000	100 0.04123945	PASSED
sts_serial	14	100000	100 0.88879512	PASSED
sts_serial	14	100000	100 0.63273366	PASSED
sts_serial	15	100000	100 0.98278591	PASSED
sts_serial	15	100000	100 0.99476012	PASSED
sts_serial	16	100000	100 0.70980744	PASSED
sts_serial	16	100000	100 0.42176374	PASSED
rgb_bitdist	1	100000	100 0.87265845	PASSED
rgb_bitdist	2	100000	100 0.11088463	PASSED
rgb_bitdist	3	100000	100 0.18331334	PASSED
rgb_bitdist	4	100000	100 0.44503287	PASSED
rgb_bitdist	5	100000	100 0.08937414	PASSED
rgb_bitdist	6	100000	100 0.35648138	PASSED
rgb_bitdist	7	100000	100 0.91873276	PASSED
rgb_bitdist	8	100000	100 0.22582863	PASSED
rgb_bitdist	9	100000	100 0.90407848	PASSED
rgb_bitdist	10	100000	100 0.99902493	WEAK
rgb_bitdist	11	100000	100 0.93730670	PASSED

rgb_bitdist	12	100000	100 0.05609507	PASSED
rgb_minimum_distance	2	10000	1000 0.15502286	PASSED
rgb_minimum_distance	3	10000	1000 0.07993638	PASSED
rgb_minimum_distance	4	10000	1000 0.59363378	PASSED
rgb_minimum_distance	5	10000	1000 0.68598771	PASSED
rgb_permutations	2	100000	100 0.55010395	PASSED
rgb_permutations	3	100000	100 0.37418808	PASSED
rgb_permutations	4	100000	100 0.23735035	PASSED
rgb_permutations	5	100000	100 0.76508980	PASSED
rgb_lagged_sum	0	1000000	100 0.97726506	PASSED
rgb_lagged_sum	1	1000000	100 0.93407297	PASSED
rgb_lagged_sum	2	1000000	100 0.63215632	PASSED
rgb_lagged_sum	3	1000000	100 0.00507557	PASSED
rgb_lagged_sum	4	1000000	100 0.25558872	PASSED
rgb_lagged_sum	5	1000000	100 0.04601191	PASSED
rgb_lagged_sum	6	1000000	100 0.65581081	PASSED
rgb_lagged_sum	7	1000000	100 0.96978297	PASSED
rgb_lagged_sum	8	1000000	100 0.98467498	PASSED
rgb_lagged_sum	9	1000000	100 0.94372569	PASSED
rgb_lagged_sum	10	1000000	100 0.30911804	PASSED
rgb_lagged_sum	11	1000000	100 0.22809154	PASSED
rgb_lagged_sum	12	1000000	100 0.40984619	PASSED
rgb_lagged_sum	13	1000000	100 0.27834763	PASSED
rgb_lagged_sum	14	1000000	100 0.60972668	PASSED
rgb_lagged_sum	15	1000000	100 0.23318474	PASSED
rgb_lagged_sum	16	1000000	100 0.87286224	PASSED
rgb_lagged_sum	17	1000000	100 0.20404857	PASSED
rgb_lagged_sum	18	1000000	100 0.26858278	PASSED
rgb_lagged_sum	19	1000000	100 0.60090761	PASSED
rgb_lagged_sum	20	1000000	100 0.80925008	PASSED
rgb_lagged_sum	21	1000000	100 0.09072868	PASSED
rgb_lagged_sum	22	1000000	100 0.34456013	PASSED
rgb_lagged_sum	23	1000000	100 0.04574526	PASSED
rgb_lagged_sum	24	1000000	100 0.56361439	PASSED
rgb_lagged_sum	25	1000000	100 0.55949903	PASSED
rgb_lagged_sum	26	1000000	100 0.42225639	PASSED
rgb_lagged_sum	27	1000000	100 0.63271753	PASSED
rgb_lagged_sum	28	1000000	100 0.27315087	PASSED
rgb_lagged_sum	29	1000000	100 0.99010427	PASSED
rgb_lagged_sum	30	1000000	100 0.96857797	PASSED
rgb_lagged_sum	31	1000000	100 0.76848282	PASSED
rgb_lagged_sum	32	1000000	100 0.45937636	PASSED
rgb_kstest_test	0	10000	1000 0.14351080	PASSED
dab_bytedistrib	0	51200000	1 0.12172326	PASSED
dab_dct	256	50000	1 0.92094552	PASSED
Preparing to run test	207.	ntuple = 0		
dab_filltree	32	15000000	1 0.64672656	PASSED

```

dab_filltree| 32| 15000000|      1|0.76558752| PASSED
Preparing to run test 208. ntuple = 0
    dab_filltree2| 0| 5000000|      1|0.38879382| PASSED
    dab_filltree2| 1| 5000000|      1|0.58519990| PASSED
Preparing to run test 209. ntuple = 0
    dab_monobit2| 12| 65000000|      1|0.33282359| PASSED

```

Come si può osservare dai risultati, il generatore passa tutti i test eccetto 3 di questi per cui risulta debole. Si conclude che questo generatore è sicuramente più robusto rispetto all'algoritmo LCG. Si decide di utilizzare l'algoritmo xorshiro128p per generare numeri casuali.

## 1.2 Distribuzione di Landau

Un modello che descrive le fluttuazioni di energia persa ( $\Delta$ ) da parte di una particella carica con velocità  $\beta = v/c$ , che attraversa uno strato di materiale sottile di spessore  $d$  è dato dalla distribuzione di probabilità di Landau.

$$f(\Delta; \beta) = \frac{1}{\xi} \phi(\lambda)$$

Dove,

$$\phi(\lambda) = \frac{1}{\pi} \int_0^\infty \exp(-u \ln u - \lambda u) \sin(\pi u) du$$

con:

$$\begin{aligned} \lambda &= \frac{1}{\xi} \left[ \Delta - \xi \left( \ln\left(\frac{\xi}{\epsilon}\right) + 1 - \gamma_E \right) \right] \\ \xi &= \frac{2\pi N_A e^4 z^2 \rho \Sigma Z d}{m_e c^2 \Sigma A \beta^2} \\ , \quad \epsilon &= \frac{I^2 \exp(\beta^2)}{2_e c^2 \beta^2 \gamma^2} \end{aligned}$$

Più semplicemente possiamo scriverla come:

$$L(x) = \frac{1}{\pi} \int_0^\infty e^{-t \log t - xt \sin \pi t} dt \quad (1.1)$$

Soltanente viene riportata come distribuzione shiftata da una costante  $\mu$  e normalizzata da una costante  $\sigma$ :

$$L(x; \mu, \sigma) = L\left(\frac{x - \mu}{\sigma}\right) \quad (1.2)$$

Per l'implementazione della funzione viene utilizzato il metodo `TMath::Landau(x, mu, sigma)` del framework ROOT.

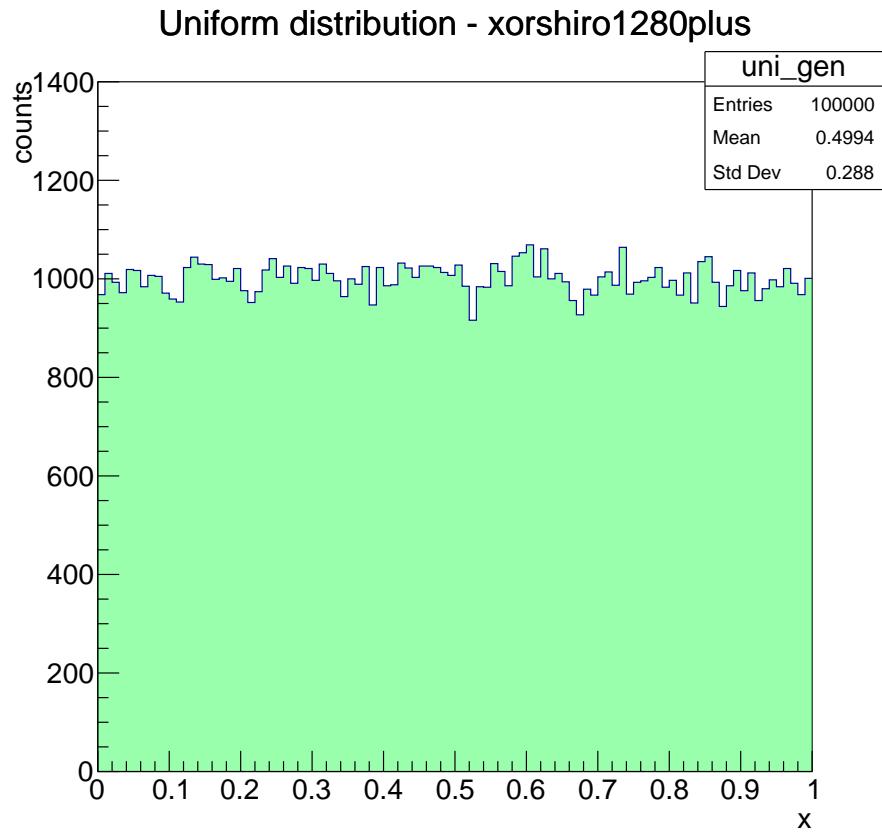


Figura 1.1: Distribuzione uniforme ottenuta con xorshiro128plus

Per generare numeri casuali che seguano questa distribuzione viene utilizzato il metodo *Accept or Reject*. In particolare si genera un numero casuale  $x_{rand}$  distribuito uniformemente entro un determinato range  $[x_{min}, x_{max}]$  che rappresenta il supporto della pdf. A questo punto si estraе un nuovo numero casuale  $y_{rand}$  nel range  $[0, \max_{[x_{min}, x_{max}]} pdf(x)]$ , ovvero nel codominio della pdf.  $x_{rand}$  seguirà la pdf e quindi verrà tenuto se:

$$y_{rand} < pdf(x_{rand})$$

Si ottengono i seguenti risultati:

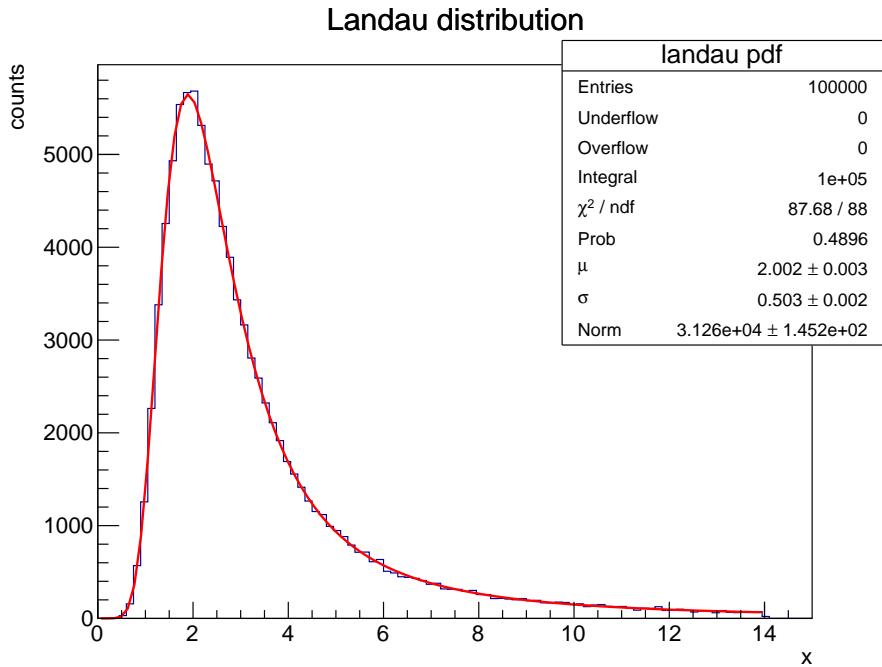


Figura 1.2: Distribuzione di Landau fittata

Il fit è eseguito per verificare che la distribuzione ottenuta fosse quella desiderata.

### 1.3 Classe xorshiro128p

Header

```
#ifndef XORSHIROGEN_
#define XORSHIROGEN_
#include "prng.h"
#include "xorshiro.h"

class xorshiroGen{

private:
    xorshiro uniform; //oggetto della classe prng
    double pi;

public:
    xorshiroGen();
    ~xorshiroGen();\
    double landau(double x, double mu = 0., double sigma = 1.);
    double landau_tc(double mu = 0, double sigma = 1,
                     double xmin = 0, double xmax = 1,
                     double ymin = 0, double ymax = 0.5);
};

}
```

```
#endif
```

Definizione dei metodi classe

```
#include <climits>
#include "prng.h"
#include "prngGen.h"
#include "xorshiro.h"
#include "xorshiroGen.h"
#include <cmath>
#include "TMath.h"

xorshiroGen::xorshiroGen(){
    pi = atan(1)*4;
}

xorshiroGen::~xorshiroGen(){};

double xorshiroGen::landau(double x, double mu, double sigma){
    return TMath::Landau(x,mu,sigma);
}
//Hit or miss (try and catch)
double xorshiroGen::landau_tc(double mu, double sigma,
                               double xmin, double xmax,
                               double ymin, double ymax){

    double x_rand = uniform.rand(xmin,xmax);
    double y_rand = uniform.rand(ymin,ymax);
    double y = landau(x_rand, mu, sigma);

    while(y_rand > y){

        x_rand = uniform.rand(xmin,xmax);
        y_rand = uniform.rand(ymin,ymax);
        y = landau(x_rand,mu,sigma);
    }

    return x_rand;
}
```

Utilizzo della classe nel main per la generazione della landau

```

#include <iostream>
#include "prng.h"
#include "prngGen.h"
#include "xorshiro.h"
#include "xorshiroGen.h"

int main(int argc, char ** argv){

    int N = 100;
    double m0 = 2.;
    double gamma = 0.5;

    if(argc > 1){

        N = std::stoi(argv[1]);
    }

    xorshiroGen landau;

    for(int i = 0; i < N; i++){

        std::cout << landau.landau_tc(m0,gamma,0,10,0,10) << std::endl;
    }

    return 0;
}

```

## 1.4 Codice creazione Landau e analisi

```

double landau_fun(double x, double mu, double sigma){
    return TMath::Landau(x,mu,sigma);
}

void line(){
    std::cout << "\n ===== \n" << std::endl;
}

void landau_analysis(string file_name = "../data/landau_hm.txt"){

    //string file_name = "../data/landau.txt"; //data under study
    string save_path = "../analysis/landau_hm.pdf"; //canvas print path

    std::cout << "File: " << file_name << std::endl;

```

```

    std::fstream landau_file;
    landau_file.open(file_name);

    //Histograms parameters
    double min = 0;
    double max = 15;
    double bins = 100;

    //Canvas
    TCanvas* c1 = new TCanvas("landau_c","landau_c",1000,800);

    //Histo
    TH1D* landau_histo = new TH1D("landau pdf","landau pdf",bins,min,max);

    double entry;

    while(landau_file >> entry){
        landau_histo -> Fill(entry);
    }

    int real_entries = landau_histo->GetEntries() - landau_histo->GetBin(0)
    - landau_histo->GetBin(bins+1);

    double over = landau_histo->GetBin(bins+1);
    double under = landau_histo->GetBin(0);
    double norm = real_entries*(max-min)/bins;
    std::cout << "Norm: " << norm << std::endl;
    std::cout << "Overflow: " << over << std::endl;
    std::cout << "Underflow: " << under << std::endl;

    //TF1
    TF1 * landau_pdf = new TF1("landauPDF","[2]*landau_fun(x,[0],[1])",min,14);

    landau_pdf->SetParameters(2,0.5);
    landau_pdf->SetParName(0,"#mu");
    landau_pdf->SetParName(1,"#sigma");
    landau_pdf->SetParName(2,"Norm");

    gStyle->SetOptStat(1110011);
    gStyle->SetOptFit(1111);

    landau_histo->Fit("landauPDF","R");
    landau_histo->SetTitle("Landau distribution");
    landau_histo->GetXaxis()->SetTitle("x");
    landau_histo->GetYaxis()->SetTitle("counts");

```

```

landau_histo->Draw();

c1->Print(save_path.c_str(),"pdf");

//Hip test
double chi_sq = landau_pdf->GetChisquare();
double dof = landau_pdf->GetNDF();
double p = landau_pdf->GetProb();

line();
std::cout << "Chi Square tilde: " << chi_sq / dof << " Probability:
" << p << std::endl;
line();
}

```

# Capitolo 2

## Esercizio 2

Il secondo esercizio chiede di implementare un programma per il calcolo della costante di Eulero-Mascheroni  $\gamma$  utilizzando diversi metodi. In seguito, viene richiesto di discutere l'influenza dei vari tipi di errore di calcolo.

**La costante  $\gamma$ :** La costante di Eulero-Mascheroni è una costante matematica ricorrente in analisi e in teoria dei numeri, definita come la differenza tra la serie armonica e il logaritmo naturale:

$$\gamma = \lim_{n \rightarrow \infty} \left( \sum_{k=1}^n \frac{1}{k} - \ln n \right) \simeq 0,57721566490153286060651209008240243 \quad (2.1)$$

### 2.1 Rappresentazione dei numeri sul calcolatore

Per la seguente descrizione è stato utilizzato [Pre+07]

#### 2.1.1 Sistema posizionale

Consideriamo un sistema avente una base  $\beta \geq 2$  e consideriamo un numero reale  $x$  con un numero finito di cifre  $x_k$  con  $k = -m, \dots, n$ , si esprime  $x$  in base  $\beta$  come:

$$x_\beta = (-1)^s \left( \sum_{k=-m}^n x_k \beta^k \right) \quad (2.2)$$

Questa viene chiamata *rappresentazione di  $x$  nella base  $\beta$*

#### 2.1.2 Rappresentazione fixed point

Un metodo per rappresentare numeri reali attraverso la rappresentazione binaria è la rappresentazione in virgola fissa.

Consideriamo un sistema che ha a disposizione  $N$  bit, questi vengono utilizzati nel seguente modo:

- 1 bit per il segno del numero da rappresentare

- I bit per rappresentare la parte intera del numero,  $I < N - 1$
- D bit per rappresentare la parte decimale del numero,  $D = N - (I + 1)$

Con questo metodo l'intervallo di numeri interi rappresentabili è  $[-2^{I-1}, 2^{I-1}]$ , mentre l'intervallo rappresentabile dalla parte decimale è  $[0, \frac{1}{2^D}]$ . Lo svantaggio è che il numero massimo e minimo di rappresentabili è limitato.

### 2.1.3 Rappresentazione floating point

Si assuma di avere a disposizione  $N$  bit con cui rappresentare un numero, un metodo più adatto per rappresentare numeri decimali è dato dalla rappresentazione in virgola mobile (floating point), data dalla seguente:

$$x = (-1)^s(0.a_1a_2\dots a_t) \cdot \beta^e = (-1)^s m \beta^{e-t} \quad (2.3)$$

con  $t \in N$  numero delle cifre significative  $a_i$  permesse, con  $0 \leq a_i \leq \beta - 1$ , e  $m = a_1a_2\dots a_t$  intero chiamato *mantissa* tale per cui  $0 \leq \beta^t - 1$  e  $e$  intero chiamato *esponente*. Chiaramente anche l'esponente può variare entro un intervallo finito di valori ammissibili,  $L \leq e \leq U$ , (tipicamente  $L < 0$  e  $U > 0$ ). A questo punto le  $N$  posizioni di memoria sono distribuite tra il segno (una posizione), le cifre significative ( $t$  posizioni) e le cifre per l'esponente (le rimanenti  $N - t - 1$  posizioni). Tipicamente su un computer ci sono due possibili formati per la rappresentazione floating point:

- Singola precisione,
- Doppia precisione.

In particolare la precisione singola utilizza una rappresentazione con  $N = 32$  bits,



Figura 2.1: Rappresentazione con precisione singola

mentre quella doppia utilizza  $N = 64$  bits.



Figura 2.2: Rappresentazione con precisione doppia

## 2.2 Sorgenti di errore

### 2.2.1 Roundoff error

L'aritmetica tra numeri rappresentati in virgola mobile non è esatta. Il più piccolo (in termini di ordine di grandezza) numero in virgola mobile, che aggiunto a

un numero floating 1.0, produce un nuovo numero floating-point diverso da 1.0 è chiamato *accuratezza di macchina*  $\epsilon_m$ . Utilizzando lo standard IEEE 754 (che fissa vincoli sui valori di  $\beta$ ,  $t$ , L e U per evitare sviluppo di formati diversi) un `float` ha  $e_m \simeq 1.19 \cdot 10^{-7}$ , mentre con i `double` si arriva a  $e_m = 2.22 \cdot 10^{-16}$ . Ogni operazione aritmetica tra numeri in virgola mobile deve essere pensata come un'introduzione di un errore addizionale di almeno  $\epsilon_m$ . Questo tipo di errore viene chiamato *roundoff error*. E' importante ricordare che  $\epsilon_m$  non è il più piccolo numero rappresentabile dal calcolatore, questo numero dipende da quanti bits ci sono disponibili per l'esponente, mentre  $e_m$  dipende da quanti bits sono disponibili per la mantissa. Gli errori di roundoff si accumulano incrementando il numeri di operazioni di calcolo. Se eseguo  $N$  operazioni di questo tipo posso ottenere un errore pari a  $\sqrt{N}\epsilon_m$ , se gli errori di roundoff sono sia dall'alto che dal basso ( $\sqrt{N}$  deriva dal random-walk). Tuttavia questa stima potrebbe essere sbagliata, infatti molto frequentemente accade che le regolarità dell'operazione o l'architettura del calcolatore possano portare ad un accumulo degli errori di roundoff, portando ad un errore pari a  $N\epsilon_m$ .

### 2.2.2 Errore di troncamento

Gli errori di roundoff sono caratteristici della componente hardware del calcolatore. Esiste un altro, diverso, tipo di errore caratteristico del programma o dell'algoritmo utilizzato che è indipendente dalla componente hardware. Diversi algoritmi calcolano approssimazioni discrete di quantità continue. Per esempio un integrale è computato numericamente valutando una funzione su un set discreto di punti al posto di ogni punto nel suo dominio. La discrepanza tra la risposta corretta e la risposta ottenuta dal calcolo pratico è chiamata *errore di troncamento*. Questo tipo di errore potrebbe persistere anche su un ipotetico calcolatore perfetto che possiede una rappresentazioni infinitamente accurata e senza errori di roundoff. A differenza dell'errore di roundoff, che non può essere controllato dal programmatore, l'errore di troncamento è sotto il suo pieno controllo. Di questo tipo di controllo se ne occupa l'analisi numerica.

### 2.2.3 Stabilità

Questo problema consiste nel fatto che piccole perturbazioni sui dati iniziali possono dare luogo a perturbazioni dello stesso ordine di grandezza sulla soluzione. Riassumendo, dato un problema fisico dove la soluzione è chiamata con  $x_t$ , attraverso metodi numeri che portano ad ottenere come risultato  $x_n$ , l'errore totale  $e = x_t - x_n$  può essere separato in errore del modello matematico  $e_{math}$  e errore computazionale  $e_c = x - x_n$ , con  $F(x, d) = 0$  modello del problema fisico. Si ha quindi  $e = e_{math} + e_c$ . A sua volta l'errore computazionale  $e_c$  risulta dalla combinazione dell'errore di discretizzazione (di troncamento o analitico) che è legato al fatto che un problema può essere calcolato numericamente solo se è descritto da un algoritmo che risolve il problema approssimato  $F_n(x_n, d_n) = 0$ , e dell'errore di roundoff, legato alla rappresentazione finita dei numeri all'interno di un calcolatore. Si ha quindi  $e_c = e_{math} + \epsilon_m$ .

Per quanto riguarda il calcolo della costante di Eulero-Mascheroni, possiamo di-

sinteressarci dell'errore dovuto al modello, in quanto la sua definizione è completamente di natura matematica.

## 2.3 Calcolo della costante

Riprendendo 2.1 notiamo che abbiamo bisogno di calcolare una somma e successivamente una differenza. Il problema principale che potrebbe portare ad un accumulo di errori è dato dal calcolo della somma.

Per studiare gli errori di roundoff si decide di studiare il comportamento degli errori di roundoff utilizzando diverse rappresentazioni dei decimali. Per questo si costruisce una classe sfruttando i `template` di C++ che permettono di utilizzare una stessa funzione con rappresentazioni diverse.

Successivamente si osserva il comportamento degli errori di troncamento utilizzando algoritmi diversi per il calcolo della somma. In particolare si implementano tre tipi diversi di algoritmi:

- Somma diretta
- Algoritmo di sommatoria di Kahan
- Sommatoria a coppie.

### 2.3.1 Studio degli errori

Assumendo l'aritmetica dei numeri in virgola mobile come quella descritta nello standard IEEE 754. Il set dei numeri in virgola mobile è denotato con  $F$  che è un sottoinsieme dei numeri reali definito come segue:

$$F = \left\{ y \in R \mid y = \pm \beta^e \sum_{k=1}^t d_k \beta^{-k} \right\}$$

Dove ricordiamo che  $\beta \geq 2$  è la base,  $t \geq 1$  è la precisione e  $e_{min} \leq e \leq e_{max}$  è l'esponente. Denotando con  $fl(\cdot)$  il risultato di un'operazione floating, dove le operazioni nella parentesi sono eseguite con la precisione con cui si lavora, si ha che le operazioni floating point, in accordo con lo standard IEEE 754, soddisfano il seguente modello:

$$fl(x \cdot y) = (x \cdot y)(1 + \delta), |\delta| \leq u, \cdot = +, -, *, /, x, y \in F \quad (2.4)$$

Dove  $u$  è chiamata unità di roundoff:

$$u = \frac{1}{2} \beta^{1-t}$$

#### Somma diretta

L'algoritmo per il calcolo della serie armonica tramite somma diretta è stato implementato in una classe `sum` con la seguente struttura:

```

#ifndef SUM_
#define SUM_


#include <iostream>
#include <cmath>
#include <numeric>
#include <limits>
#include <iomanip>
#include <fstream>
#include <gsl/gsl_math.h>

template <typename float_num, typename int_num>
class sum{

private:
    float_num sum_in;
    float_num c;

public:
    sum(){sum_in = 0;};
    ~sum(){};
    float_num norm_sum(int_num N = 100){

        sum_in = 0;
        for(int i = 1; i < N; i++){
            sum_in += 1. / (float_num)i;
        }
        return sum_in;
    };
}

```

La classe è stata poi utilizzata studiando il comportamento dell'algoritmo per due diversi tipi di variabili: `float` e `long double`. Per fare ciò è stata creata una seconda classe `gamma_const` che effettua le operazioni prima descritte.

La classe ha la seguente struttura:

```

#ifndef GAMMA_CONST_
#define GAMMA_CONST_


class gamma_const{

public:
    gamma_const();
    ~gamma_const();

    float gamma_constFI(int n = 100);
}

```

```

    double gamma_constLDI (int n = 100);

};

#endif //GAMMA_CONST_

```

I metodi della classe `gamma_const` sono implementati come segue. Si noti il linking della libreria `sum` per l'algoritmo somma:

```

#include "sum .h"
#include "gamma_const.h"
#include <cmath>
#include <iostream>
#include <math.h>

gamma_const::gamma_const(){}
gamma_const::~gamma_const(){}

long double gamma_const::gamma_constLDI(int n){

    sum <long double,int> sumLDI;
    return sumLDI.norm_sum(n) - log(n);;
}

float gamma_const::gamma_constFI(int n){

    sum <float,int> sumFI;
    return sumFI.norm_sum(n) - log(n);
}

```

**Analisi dell'errore** Si può dimostrare che l'errore associato utilizzando questo metodo soddisfa la seguente diseguaglianza. Detta  $S_n = \sum_{i=1}^n x_i$  la sommatoria "reale", e detta  $\hat{S}_n$  la somma calcolata numericamente, si ha:

$$E_n = S_n - \hat{S}_n \quad (2.5)$$

$$|E_n| \leq (|x_1| + |x_2|)\gamma_{n-1} + \sum_{i=3}^n |x_i|\gamma_{n-1+i} \quad (2.6)$$

dove:

$$\gamma_n := \frac{nu}{1-nu} \quad (2.7)$$

### 2.3.2 Algoritmo sommatoria Kahan

Come detto precedentemente, rappresentando un numero reale in virgola mobile con precisione finita, la rappresentazione differisce dal valore vero di una certa

quantità, corrispondente all'errore di arrotondamento. Eseguendo una semplice sommatoria di numeri reali, utilizzando le rispettive rappresentazioni in virgola mobile, il totale ottenuto presenta un certo errore dato dalla somma algebrica dei singoli errori di arrotondamento. L'algoritmo di sommatoria di Kahan riduce significativamente l'errore totale, utilizzando una variabile che permette la compensazione di questo errore (una variabile che accumula gli errori piccoli). L'algoritmo è stato implementato attraverso una classe che ha la seguente struttura:

```
#ifndef SUM_
#define SUM_


#include <iostream>
#include <cmath>
#include <numeric>
#include <limits>
#include <iomanip>
#include <fstream>
#include <gsl/gsl_math.h>

template <typename float_num, typename int_num>
class sum{

private:
    float_num sum_in;
    float_num c;

public:
    sum(){sum_in = 0;};
    ~sum(){};

float_num kahan_sum(int_num N = 100){

    sum_in = 0;
    c = 0;
    for(int i = 1; i < N; i++){

        float_num y = 1./((float_num)i) - c;
        float_num t = sum_in + y;
        c = (t-sum_in) - y;
        sum_in = t;
    }
    return sum_in;
};

};
```

Nella classe `gamma_const` è stato quindi implementato un metodo per linkare l'algoritmo sommatoria di Kahan nel modo seguente:

```

long double gamma_const::Kgamma_constLDI(int n){

    sum <long double,long long int> sumDI;
    return sumDI.kahan_sum(n) - log(n);
}

```

**Analisi dell'errore** Knuth (1981) dimostrò che con una variabile che compensa l'errore nell'addizione si ha:

$$\hat{S}_n = \sum_{i=1}^n (1 + \mu_i)x_i, \quad |\mu_i| \leq 2u + O(nu^2) \quad (2.8)$$

Da cui:

$$|E_n| \leq (2u + O(nu^2)) \sum_{i=1}^n |x_i| \quad (2.9)$$

### 2.3.3 Pairwise summation

Un'altro algoritmo che permette di compensare l'errore è quello della sommatoria accoppiata. Sia  $S_n = \sum_{i=1}^n x_i$  la sommatoria da calcolare. L'algoritmo funziona nel modo seguente. Sia  $S = x_1, \dots, x_n$ , si ripete finché  $S$  contiene più di un elemento la seguente operazione:

Rimuovo due numeri  $x$  e  $y$  da  $S$ , aggiungo la loro somma  $x + y$  a  $S$ . Assegno l'elemento rimanente di  $S$  a  $S_n$ .

L'algoritmo è stato implementato attraverso una classe come precedentemente visto. La struttura della classe è la seguente:

```

#ifndef SUM_
#define SUM_

#include <iostream>
#include <cmath>
#include <numeric>
#include <limits>
#include <iomanip>
#include <fstream>
#include <gsl/gsl_math.h>

template <typename float_num, typename int_num>
class sum{

private:
    float_num sum_in;
    float_num c;
public:
    float_num pairwise_sum(int start, int stop){
        if ((stop - start) < 100){ //Solo se numero è abbastanza grande

```

```

    sum_in = 0. ;
    for (int i = start; i<=stop; i++){
        sum_in += 1 / (double)i;
    }
    return sum_in;
}
else{
    int m = floor((stop - start) /2 );
    return pairwise_sum(start, start+m) + pairwise_sum(start+m+1, stop);
}
}
};


```

Come precedentemente visto, viene aggiunto un metodo alla classe `gamma_const` per l'applicazione del metodo di pairwise summation nel calcolo della costante nel modo seguente:

```

long double gamma_const::PWgamma_constLDI(int n){

    sum <long double,long long int> sumPWLDI;

    return sumPWLDI.pairwise_sum(1,n) - log(n);
}

```

**Analisi dell'errore** Esprimendo l'i-esima iterazione del loop come  $T_i = x_{i1} + y_{i1}$ . La somma calcolata soddisfa:

$$\hat{T}_i = \frac{x_{i1} + y_{i1}}{1 + \delta_i}, |\delta_i| \leq u, \quad i = 1, \dots, -1. \quad (2.10)$$

L'errore local è dato da  $\delta_i \hat{T}_i$ .

L'errore totale è la somma degli errori locali, quindi si ha:

$$E_n := S_n - \hat{S}_n = \sum_{i=1}^{n-1} \delta_i \hat{T}_i \quad (2.11)$$

L'errore più piccolo possibile sarà dato da:

$$|E_n| \leq u \sum_{i=1}^{n-1} |\hat{T}_i| \quad (2.12)$$

Essendo  $|\hat{T}_i| \leq \sum_{j=1}^n |x_j| + O(u)$ , si ha anche un limite inferiore debole:

$$|E_n| \leq (n-1)u \sum_{i=1}^n |x_i| + O(u^2) \quad (2.13)$$

L'analisi degli errori trattata fino a questo momento è descritta [J.H02]

## 2.4 Risultati

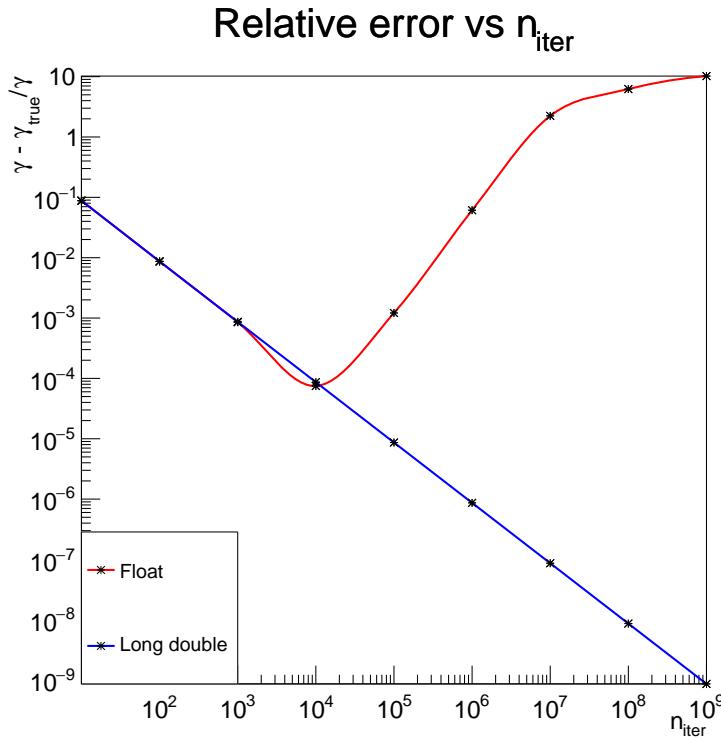


Figura 2.3: Andamento errore relativo rispetto al numero di iterazioni utilizzando tipo float e long double

**Somma diretta** Come si può notare da figura 2.3, utilizzando un tipo float a  $10^4$  iterazioni l'errore relativo incomincia a crescere in maniera molto rapida. Per quanto riguarda i valori della costante si ottengono i seguenti risultati (per i long double).

```
>./um_emconstant.sh
Euler constant: 0.5772156649015328655
Iterazione: 10E+1 Constant computed: 0.5263831609742080672
Iterazione: 10E+2 Constant computed: 0.5722073316515284588
Iterazione: 10E+3 Constant computed: 0.5767155815682080975
Iterazione: 10E+4 Constant computed: 0.5771656640681986595
Iterazione: 10E+5 Constant computed: 0.5772106648931993305
Iterazione: 10E+6 Constant computed: 0.5772151649014500008
Iterazione: 10E+7 Constant computed: 0.5772156149015313965
Iterazione: 10E+8 Constant computed: 0.5772156599015311161
Iterazione: 10E+9 Constant computed: 0.5772156644015335718
```

Si osserva che con  $10^9$  iterazioni si raggiunge la nona cifra decimale.

**Sommatoria di Kahan** Come si può osservare da figura 2.4 a  $10^6$  iterazioni utilizzando il tipo float l'errore relativo smette di decrescere. In confronto alla

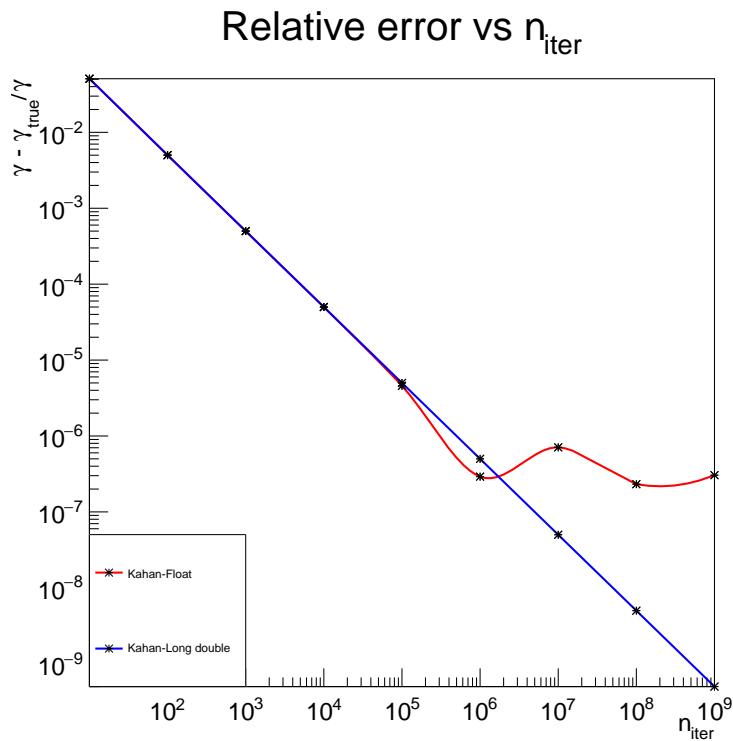


Figura 2.4: Andamento errore relativo rispetto al numero di iterazioni utilizzando tipo float e long double

somma diretta questo algoritmo riesce a compensare l'errore e di evitare un brusco aumento, inoltre l'errore continua a decrescere fino a  $10^6$  iterazioni, due ordini in più rispetto alla sommatoria diretta.

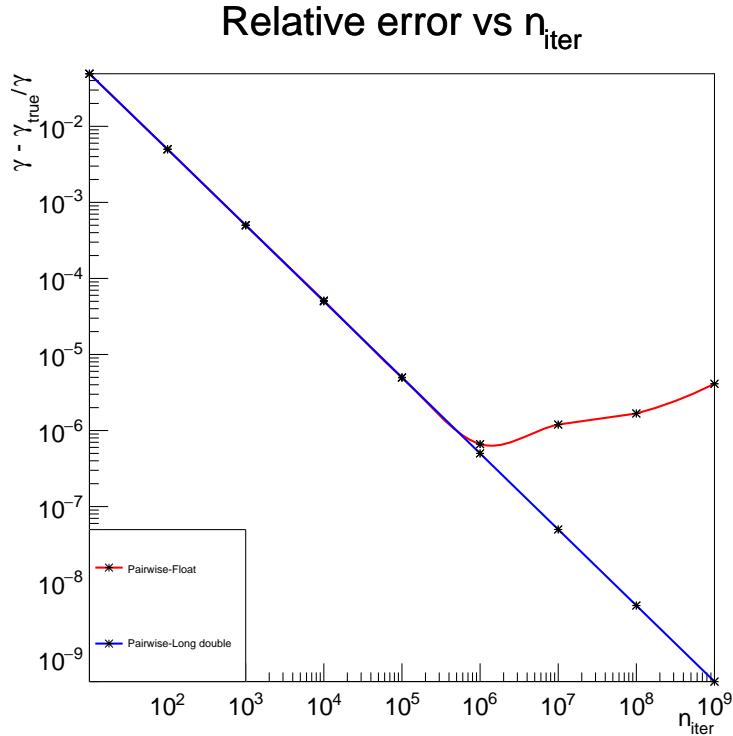


Figura 2.5: Andamento errore relativo rispetto al numero di iterazioni utilizzando tipo float e long double

**Pairwise summation** Come si può osservare da figura 2.5 anche qui, per quanto riguarda il tipo `float`, si ha un aumento dell'errore relativo a partire da  $10^6$  iterazioni. Questo algoritmo permette di compensare l'errore come l'algoritmo di Kahan, tuttavia si nota che a differenza dell'utilizzo di una variabile che controlla l'errore, la sommatoria a coppie porta ad un aumento dell'errore più veloce.

**Confronto** Da figura 2.6 si possono osservare i diversi andamenti. In particolare la sommatoria di Kahan è quella che compensa meglio l'errore con l'utilizzo di tipi `float`, mentre la sommatoria a coppie è quella che presenta un errore relativo minore con l'utilizzo dei `long double`

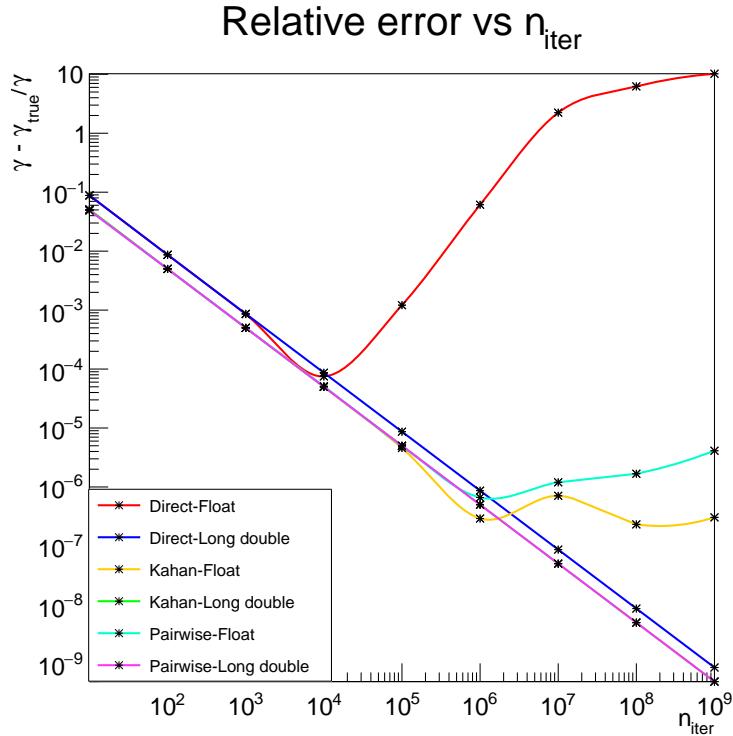


Figura 2.6: Andamento errore relativo rispetto al numero di iterazioni utilizzando tipo float e long double

## 2.5 Codice per l'analisi dei metodi

Viene riportato di seguito il main utilizzato per produrre l'analisi, capace di linkare tutti i metodi visti in precedenza per il calcolo della costante:

```
#include <iostream>
#include "sum_alg.h"
#include "gamma_const.h"
#include <cmath>
#include <numeric>
#include <iomanip> //setPrecision
#include <vector>
#include <gsl/gsl_math.h> //M_EULER
#include <limits>
#include "TCanvas.h"
#include "TMultiGraph.h"
#include "TGraph.h"
#include "TAxis.h"
#include "TStyle.h"
#include "TLegend.h"
#include "TApplication.h"

int main(int argc, char** argv){
```

```

//Object TApplication instance
TApplication* Grafica = new TApplication("",NULL,NULL);

//Object gamma_const instance (class implemented)
gamma_const gammaEM;

std::cout << "Euler constant: " << std::setprecision(std::numeric_limits<long double>::max_digits10+1) << gammaEM.EULER;

int size = 9; //Iterations order of magnitude
//Arrays objects instance, they will contain the results
//of the computation
double* x_axis = new double[size]; //Iterations
double* stdemF = new double[size];
double* stdemLD = new double[size];

double* KstdemF = new double[size];
double* KstdemLD = new double[size];

double* PWstdemF = new double[size];
double* PWstdemLD = new double[size];

//start with the computation of the constant
for(int i = 1; i < size+1; i++){

    double x = pow(10,(double)i);
    std::cout << "Iterazione: " << "10E+" << i << " Constant computed: " <<
    std::setprecision(std::numeric_limits<long double>::max_digits10+1) << gammaEM.EULER;
    x_axis[i-1]= x;

    KemF[i-1]=gammaEM.Kgamma_constFI(x);
    emF[i-1]=gammaEM.gamma_constFI(x);
    KemLD[i-1]=gammaEM.Kgamma_constLDI(x);
    emLD[i-1]=gammaEM.gamma_constLDI(x);
    KstdemF[i-1]=abs(gammaEM.Kgamma_constFI(x) - M_EULER);
    stdemF[i-1]=abs(gammaEM.gamma_constFI(x) - M_EULER)/M_EULER;
    KstdemLD[i-1]=abs(gammaEM.Kgamma_constLDI(x) - M_EULER);
    stdemLD[i-1]=abs(gammaEM.gamma_constLDI(x) - M_EULER)/M_EULER;
    PWemF[i-1]=gammaEM.PWgamma_constFI(x);
    PWstdemF[i-1]=(gammaEM.PWgamma_constFI(x)-M_EULER);
    PWemLD[i-1]=gammaEM.PWgamma_constLDI(x);
    PWstdemLD[i-1]=(gammaEM.PWgamma_constLDI(x)-M_EULER);
}

//Graphs objects instances
TMultiGraph* stdmg = new TMultiGraph();
TLegend * legstd = new TLegend(0.1,0.1,0.3,0.3);

```

```

TGraph * stdgF = new TGraph(size,x_axis,stdemF);
TGraph * stdgLD = new TGraph(size,x_axis,stdemLD);

TGraph * KstdgF = new TGraph(size,x_axis,KstdemF);
TGraph * KstdgLD = new TGraph(size,x_axis,KstdemLD);

TGraph * PWstdgF = new TGraph(size,x_axis,PWstdemF);
TGraph * PWstdgLD = new TGraph(size,x_axis,PWstdemLD);

//Customization of graphs style
stdgF->SetLineWidth(2);
stdgLD->SetLineWidth(2);
stdgF->SetLineColor(kRed);
stdgLD->SetLineColor(kBlue);

KstdgF->SetLineWidth(2);
KstdgLD->SetLineWidth(2);
KstdgF->SetLineColor(kOrange);
KstdgLD->SetLineColor(kGreen);

PWstdgF->SetLineWidth(2);
PWstdgLD->SetLineWidth(2);
PWstdgF->SetLineColor(kTeal);
PWstdgLD->SetLineColor(kMagenta-4);

//add the single graphs in a multigraph
stdmg->Add(stdgF);
stdmg->Add(stdgLD);

stdmg->Add(KstdgF);
stdmg->Add(KstdgLD);

stdmg->Add(PWstdgF);
stdmg->Add(PWstdgLD);

TCanvas* c1 = new TCanvas("Euler-Mascheroni_std","c1",800,800);

legstd->AddEntry(stdgF,"Direct-Float");
legstd->AddEntry(stdgLD,"Direct-Long double");

legstd->AddEntry(KstdgF,"Kahan-Float");
legstd->AddEntry(KstdgLD,"Kahan-Long double");

legstd->AddEntry(PWstdgF,"Pairwise-Float");
legstd->AddEntry(PWstdgLD,"Pairwise-Long double");
stdmg->SetTitle("Relative error vs n_{iter}");

```

```

stdmg->GetXaxis()->SetTitle("n_{iter}");
stdmg->GetYaxis()->SetTitle("#gamma - #gamma_{true}/#gamma");
stdmg->GetYaxis()->SetTitleOffset(1.5);
stdmg->Draw("APC*");
legstd->Draw("SAME");
c1->SetLogx();
c1->SetLogy();
c1->SaveAs("./results/all_euler_masc_std.pdf","pdf");

Grafica->Run();
return 0;

}

```

# Capitolo 3

## Esercizio 3

### 3.1 Creazione Pdf e generazione eventi Monte Carlo

I bosoni vettori  $1^-$  sono caratterizzati da una distribuzione angolare nello spazio delle coordinate  $\theta \in [0, \pi], \phi \in [0, 2\pi]$  che ha la seguente espressione:

$$F(\theta, \phi) = \frac{3}{4\pi} [0.5(1-\alpha) + (0.5)(3\alpha-1)\cos(\theta)^2 - \beta\sin(\theta)^2\cos(2\phi) - \sqrt{2}\gamma\sin(2\theta)\cos(\phi)]$$

I parametri del modello sono teoricamente fissati ai seguenti valori:

$$\alpha = 0.65 , \quad \beta = 0.06 , \quad \gamma = -0.18$$

Generiamo quindi 50000 eventi monte carlo secondo la precedente distribuzione grazie al pacchetto `RooFit` creato appositamente per modellare e analizzare distribuzioni di eventi in fisica delle particelle (originariamente creato per l'esperimento BaBar allo Stanford Linear Accellerator Centre).

Le librerie incluse nell'analisi non sono riportate per alleggerire la descrizione dell'esercizio. Verrà comunque esplicato il funzionamento di ogni oggetto di interesse creato.

Il seguente codice definisce le variabili a valori reali di interesse attraverso il costruttore `RooRealVar` e la distribuzione attraverso il costruttore `RooClassFactory` il quale genera un oggetto `RooAbsPdf`, un'interfaccia astratta per qualsiasi distribuzione di probabilità. La suddetta classe provvede in automatico a normalizzare la funzione in ingresso un metodo ibrido analitico/numerico.

```
int main(){
    //Defining Real variables according to RooFit.
    //Angles physical defined in [0-pi], [0, 2*pi]
    RooRealVar t("t","t",0,M_PI) ;
    RooRealVar p("p","p", 0, 2*M_PI) ;

    //Defining Parameters according to RooFit.
```

```

//Initial value is the 3rd argument, 4th and 5th arg
//are the range of existance of the parameter.
RooRealVar alpha("alpha", "alpha", 0.65, 0.62, 0.66);
RooRealVar beta("beta", "beta", 0.06, 0.05, 0.075);
RooRealVar gamma("gamma", "gamma", -0.18, -.2, -0.16);

//ClassFactory generates a pdf instance and normalize it.
RooAbsPdf* genpdf = RooClassFactory::makePdfInstance("GenPdf",
"(3./(4.*M_PI))*(0.5*(1.-alpha) + (0.5)*(3.*alpha-1)*cos(t)*cos(t)
- beta*sin(t)*sin(t)*cos(2.*p)- sqrt(2.)*gamma*sin(2.*t)*cos(p))",
RooArgSet(t,p,alpha, beta, gamma)) ;

```

La visualizzazione della pdf può essere fatta solo attraverso un istogramma binnato in quanto **RooFit** non supporta ancora metodi efficienti per visualizzazioni in più dimensioni:

```

TH2F* h2_pdf = new TH2F("F(#theta, #phi) distribution",
                        "F(#theta, #phi) distribution", 100, 0, M_PI, 100, 0, 2*M_PI);
genpdf->fillHistogram(h2_pdf, RooArgList(t,p));
h2_pdf->GetXaxis()->SetTitle("#theta [rad]");
h2_pdf->GetYaxis()->SetTitle("#phi [rad]");
h2_pdf->GetXaxis()->SetTitleOffset(2);
h2_pdf->GetYaxis()->SetTitleOffset(2);
h2_pdf->GetZaxis()->SetTitle("F(#theta, #phi)");
h2_pdf->GetZaxis()->SetTitleOffset(1.5);
TCanvas* c_pdf = new TCanvas("c_pdf", "c_pdf", 1000,1000,1000,800);
c_pdf->SetLeftMargin(2);
h2_pdf->Draw("surf");
c_pdf->Draw();

```

Il risultato è il seguente:

## F( $\theta$ , $\phi$ ) distribution

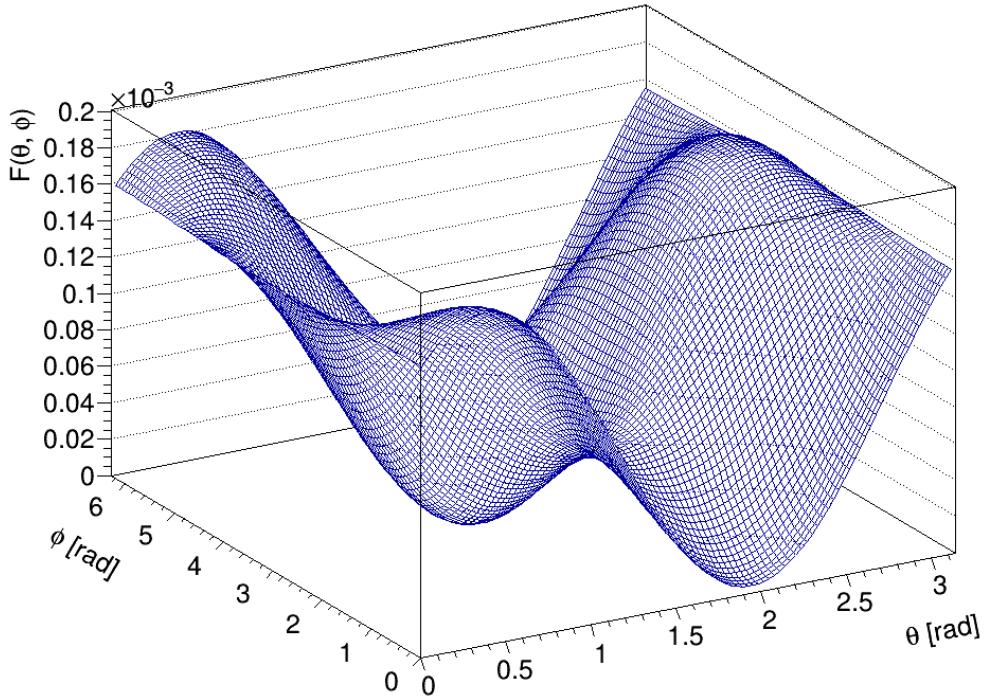


Figura 3.1: Rappresentazione della distribuzione attraverso un istogramma binnato e il metodo `surf`.

Creata la *pdf* possiamo ora generare gli eventi monte carlo. `RooAbsPdf` può farlo grazie ad un metodo built-in della classe chiamato `generate`. `RooArgSet` è un oggetto contenitore per le variabili di interesse  $\theta, \phi$ . I dati generati vengono salvati quindi in un oggetto `RooDataSet` nel quale ogni istanza delle variabili definite dal `RooArgSet` è rappresentata da un oggetto `RooRealVar`:

```
RooDataSet* data = genpdf->generate(RooArgSet(t,p),50000) ;
```

La distribuzione appena creata può essere visualizzata binnando i dati in un `TH2F` e attraverso il metodo `fillHistogram` dell'oggetto `RooDataSet` :

```
//visualizing the events generated through a ROOT TH2F.
TH2F* dh2 = new TH2F("h2", "h2", 100, 0, M_PI, 100, 0, 2*M_PI);
data->fillHistogram(dh2, RooArgList(t,p));
TCanvas*c = new TCanvas("c", "c", 1000,1000,1000,800);
dh2->SetTitle("Binned #theta, #phi distribution");
dh2->GetXaxis()->SetTitle("#theta [rad]");
dh2->GetYaxis()->SetTitle("#phi [rad]");
dh2->Draw("colz");
c->Draw();
```

Il risultato è il seguente:

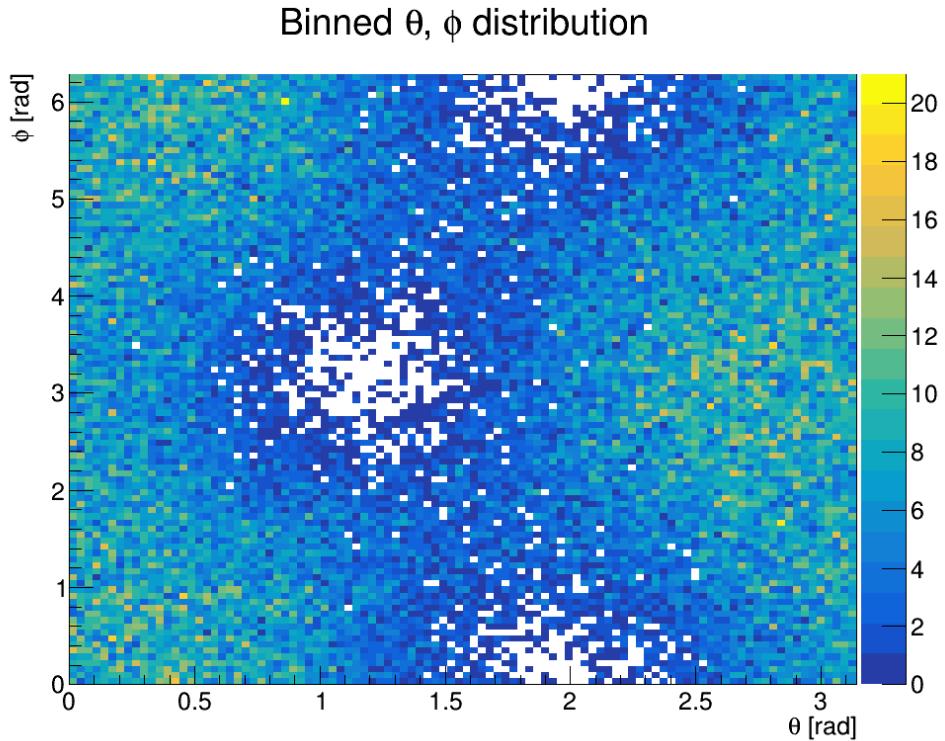


Figura 3.2: Rappresentazione bi-dimensionale dei dati generati. Il binning è totalmente arbitrario ma necessario per la visualizzazione. Il colore è proporzionale al numero di eventi all'interno del bin grazie al metodo `colz`

### 3.2 Unbinned ML fit

L'esercizio chiede quindi di effettuare un fit dei dati appena generati con un metodo maximum likelihood per la stima dei parametri  $\alpha, \beta, \gamma$  della pdf con cui abbiamo generato i dati stessi.

Il fit è inteso come unbinned, infatti i dati all'interno dell'oggetto `RooDataSet` non sono binnati e la procedura in `RooFit` è molto semplice e ha la seguente sintassi:

```
RooFitResult* r_ML = genpdf->fitTo(*data, Save()) ;
```

L'oggetto `RooFitResults` contiene tutte le informazioni relative al fit, effettuato con il metodo `fitTo` della classe `RooAbsPdf`. È importante passare al metodo un oggetto del tipo `RooDataSet` su cui effettuare il fit unbinned ML (analogamente potremmo costruire un oggetto `RooDataHist` e con la stessa sintassi effettuare un binned ML fit). L'opzione aggiuntiva `Save()` permette di salvare i risultati in un oggetto `RooFitResults`.

Essendo i parametri per definizione di `RooRealVar` delle variabili nel loro range di definizione, dopo il fit esse assumeranno il valore stimato. Possiamo quindi plottare il risultato (notare che `Roofit` adatta automaticamente la scala della pdf ai dati. Il fattore di scala non è infatti un parametro del fit in quanto la pdf è normalizzata, per definizione, a 1). Vengono ricavati gli estimatori dei parametri dal fit e viene,

come in precedenza, generata la rappresentazione della pdf fittata ( da notare come vengono ricavati i parametri del fit. Sono semplicemente contenuti nelle variabili che sono state aggiornate dopo l'operazione) :

```
//Retireve statistics:  
auto alpha_est = alpha.getVal();  
auto beta_est = beta.getVal();  
auto gamma_est = gamma.getVal();  
  
TH2F* h2_pdf_ml = new TH2F("F(#theta, #phi) distribution ml"  
    , "F(#theta, #phi) distribution", 100, 0, M_PI, 100, 0, 2*M_PI);  
h2_pdf_ml->SetLineColor(kRed);  
genpdf->fillHistogram(h2_pdf_ml, RooArgList(t,p));
```

Vengono quindi plottati i dati e la distribuzione con il seguente risultato:

```
Unbinned ML fit:  $\alpha = 0.652 \pm 0.003$ ,  $\beta = 0.059 \pm 0.003$ ,  $\gamma = -0.178 \pm 0.002$   
minimized FCN value: 141656.32768, EDM: 0.000040
```

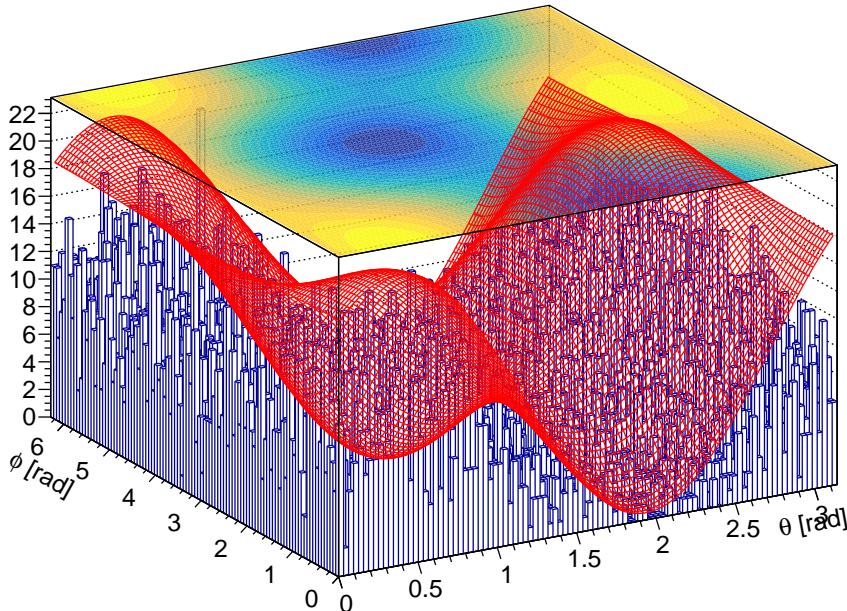


Figura 3.3: Rappresentazione bi-dimensionale dei dati generati e della pdf con parametri ottenuti dal ML unbinned fit. La rappresentazione bidimensionale della pdf fittata viene plottata sopra le distribuzioni come curve di livello grazie al metodo `surf3`

I risultati del fit sono quindi stampati grazie al metodo `Print()` della classe `RooFitResults` e di seguito riportati:

```
==> ML Fit results
```

```
RooFitResult: minimized FCN value: 141656, estimated distance to minimum:
```

```

3.9822e-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

```

Gli errori sono stimati da `Migrad` secondo  $L_{max} - 0.5$ .  
I coefficienti di correlazione dei parametri sono dati dalla matrice di covarianza.  
E' importante che la correlazione sia bassa altrimenti il metodo di minimizzazione `MINUIT` ha difficoltà a convergere e a calcolare gli errori sui parametri:

PARAMETER		CORRELATION COEFFICIENTS		
NO.	GLOBAL	1	2	3
1	0.28417	1.000	-0.256	-0.006
2	0.52164	-0.256	1.000	0.456
3	0.47034	-0.006	0.456	1.000

### 3.3 Likelihood plots

I plot delle likelihood per i singoli parametri possono essere ricavate dalla minimizzazione diretta dell'oggetto `likelihood`. Viene creato un oggetto `RooAbsReal`, che è una classe astratta per manipolazioni di variabili a valori reali, contenete la likelihood della pdf in esame attraverso il metodo `createNLL`. Al metodo vengono passati i dati ed il numero di processori per creare l'oggetto.

Viene quindi istanziato un oggetto `RooMinimizer` (che altro non è che un wrapper della classe `TMinuit` di ROOT) al cui costruttore viene passata la likelihood appena creata. Viene chiamato il metodo `migrad` il quale sceglierà l'omonimo algoritmo di minimizzazione creato da Davidson-Fletcher-Powell. Altri algoritmi di minimizzazione sono presenti nella classe, i quali sono tutti asintoticamente equivalenti. E' stato scelto `migrad` in quanto è il minimizzatore di default in `TMinuit` e l'algoritmo utilizzato nel fit riportato in precedenza. Ricordiamo che sicuramente l'utilizzo di un diverso algoritmo porta inevitabilmente a stime differenti (anche se presumibilmente comparabili entro gli errori) in quanto il nostro campione è finito. In particolare l'algoritmo `Minos` calcola esattamente gli errori sui parametri che, in generale, possono anche essere asimmetrici.

```

RooAbsReal* nll = genpdf->createNLL(*data, NumCPU(4));

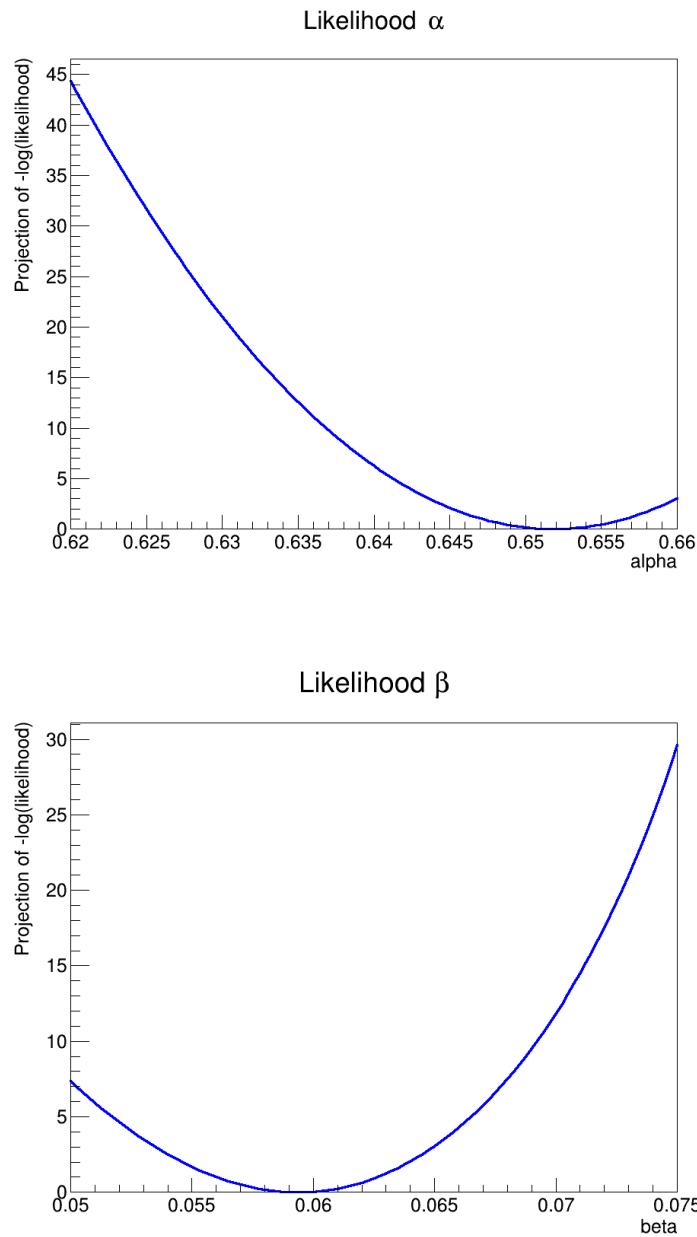
//instantiating minimizer for likelihood
RooMinimizer m(*nll) ;
m.migrad();

```

Viene quindi plottato su un **RooPlot** (un contenitore per oggetti grafici di ROOT) il risultato della computazione e la shape della likelihood per ogni parametro. Ad esempio riportiamo la procedura di plot per il parametro  $\alpha$ :

```
RooPlot* frame1 = alpha.frame(Title("Likelihood #alpha")) ;
nll->plotOn(frame1,ShiftToZero()) ;
```

Il risultato per i tre parametri è il seguente:



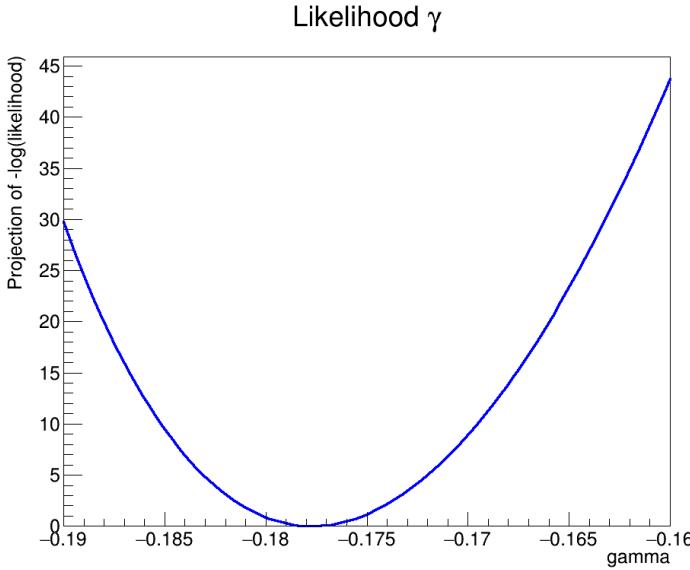


Figura 3.4: Proiezioni di  $-\text{Log}(\text{Likelihood})$  per i parametri  $\alpha, \beta, \gamma$ . Il punto di minimo corrisponde alla migliore stima dei parametri mentre il loro errore viene computato come l'intervallo corrispondente all'ascissa di  $L_{\max} - 0.5$

### 3.4 Esperimenti Monte Carlo ripetuti

Un comodo strumento che `RooFit` fornisce è l'automatica creazione di dataset monte carlo e il relativo unbinned ML fit. Grazie al metodo `generateAndFit` della classe `RooMCStudy`, possiamo generare  $N$  esperimenti ognuno composto da  $n$  eventi e quindi fissare le distribuzioni dei dati con la pdf passata al costruttore dell'oggetto.

Secondo Il Teorema Centrale del Limite (CLT) l'estimatore ML è distribuito secondo una gaussiana e nel limite  $n \rightarrow \infty$  coincide con il valore vero, in altre parole, lo stimatore ML è non biassato.

Possiamo quindi verificare la distribuzione gaussiana degli estimatori effettuando  $N = 1000$  esperimenti ognuno da  $n = 50000$  eventi:

```
//defining a RooMCStudy object
RooMCStudy mcs(*genpdf, RooArgSet(t,p));
//generate & fit 1000 samples of 50000 events each.
//(kTRUE) for saving every estimate in a RooDataSet.
mcs.generateAndFit(1000,50000, kTRUE) ;
```

La routine di generazione dei numeri pseudo-random dipende da un seed interno che garantisce che i dati siano diversi per ogni esperimento.

Il processo di generazione e fit è temporalmente e computazionalmente molto costoso quindi si è scelto di limitare la statistica a soli 1000 esperimenti.

Una volta finita la generazione possiamo leggere e quindi ricavare i risultati dell'esperimento. Questa procedura viene effettuata solo per gli estimatori dei parametri, può essere estesa analogamente a qualsiasi altro parametro del fit (ad esempio agli errori sugli estimatori):

```

//retrieving the results
RooDataSet fitParData = mcs.fitParDataSet();
//defining containers for estimators
RooDataSet all_alpha_est("all_alpha_est", "all_alpha_est", RooArgSet(alpha));
RooDataSet all_beta_est("all_beta_est", "all_beta_est", RooArgSet(beta));
RooDataSet all_gamma_est("all_gamma_est", "all_gamma_est", RooArgSet(gamma));

//cycling and retireving the estimators
for(int i = 0; i < 100; i++){
    const RooArgSet* f = fitParData.get(i);
    alpha = f->getRealValue("alpha");
    beta = f->getRealValue("beta");
    gamma = f->getRealValue("gamma");
    all_alpha_est.add(RooArgSet(alpha));
    all_beta_est.add(RooArgSet(beta));
    all_gamma_est.add(RooArgSet(gamma));
}

```

A questo punto è possibile creare una pdf gaussiana e fittarla con metodo unbinned ML ai dati appena raccolti. Viene riportato solo il caso del parametro  $\alpha$  in quanto la procedura è analoga per i due rimanenti:

```

RooRealVar meana("mean", "mean", 0.65, 0.5, 1) ;
RooRealVar sigmaa("sigma", "sigma", 0.001, 0.0001, 10) ;
RooGaussian gaussa("gauss", "gauss", alpha, meana, sigmaa) ;

gaussa.fitTo(all_alpha_est);

```

Viene quindi plottato su un *RooPlot* frame il risultato del fit assieme ai dati:

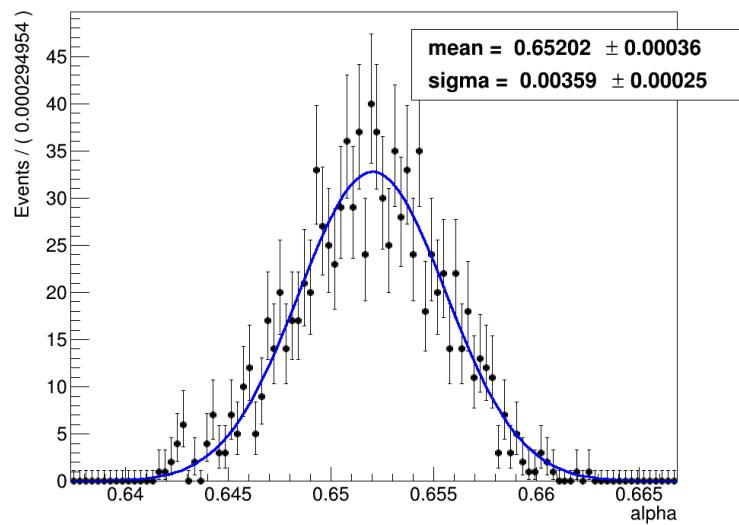
```

//creating a canvas
TCanvas* mc_can = new TCanvas("mc_can", "mc_can", 1000, 1000, 1000, 800);
//Defining a RooPlot and plot the stimations distribution as a histogram
RooPlot* mc_frame1 = mcs.plotParam(alpha) ;
mc_frame1->SetTitle("#alpha MC estimations");
//plotting the fitted pdf
gaussa.plotOn(mc_frame1);
//plotting fit parameters
gaussa.paramOn(mc_frame1, Layout(0.55, 0.99, 0.87));
mc_frame1->Draw() ;
mc_can->Draw();

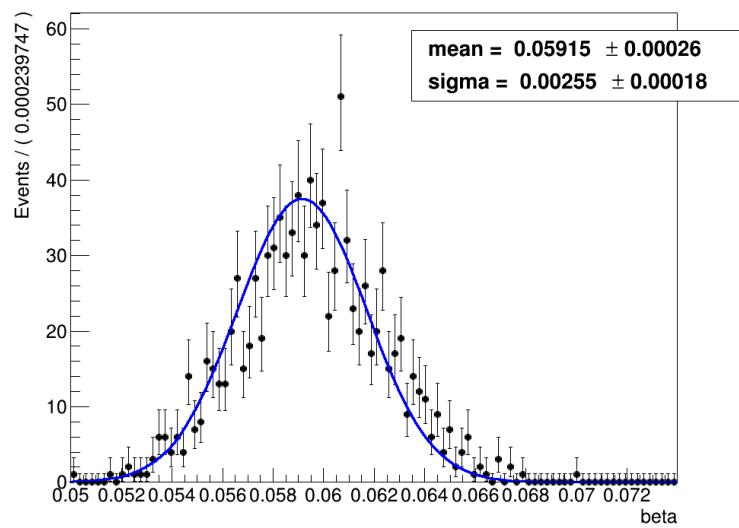
```

Il risultato è il seguente:

$\alpha$  MC estimations



$\beta$  MC estimations



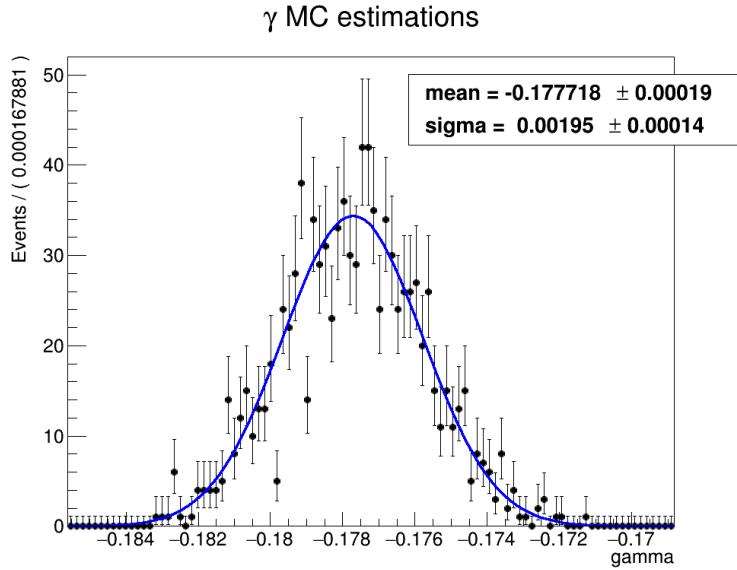
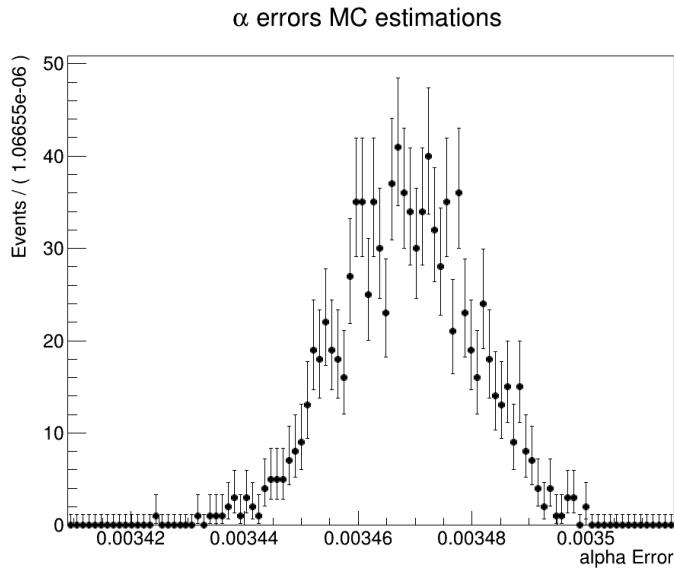


Figura 3.5: Risultati del fit da multipli esperimenti con  $n = 50000$ . Il valore dell'errore sulla miglior stima segue approssimativamente  $\frac{\sigma}{\sqrt{N}}$

Altre distribuzioni importanti che possiamo ricavare dai risultati dei fit sono le distribuzioni degli errori di ogni parametro. Vengono riportate senza un fit gaussiano di seguito:



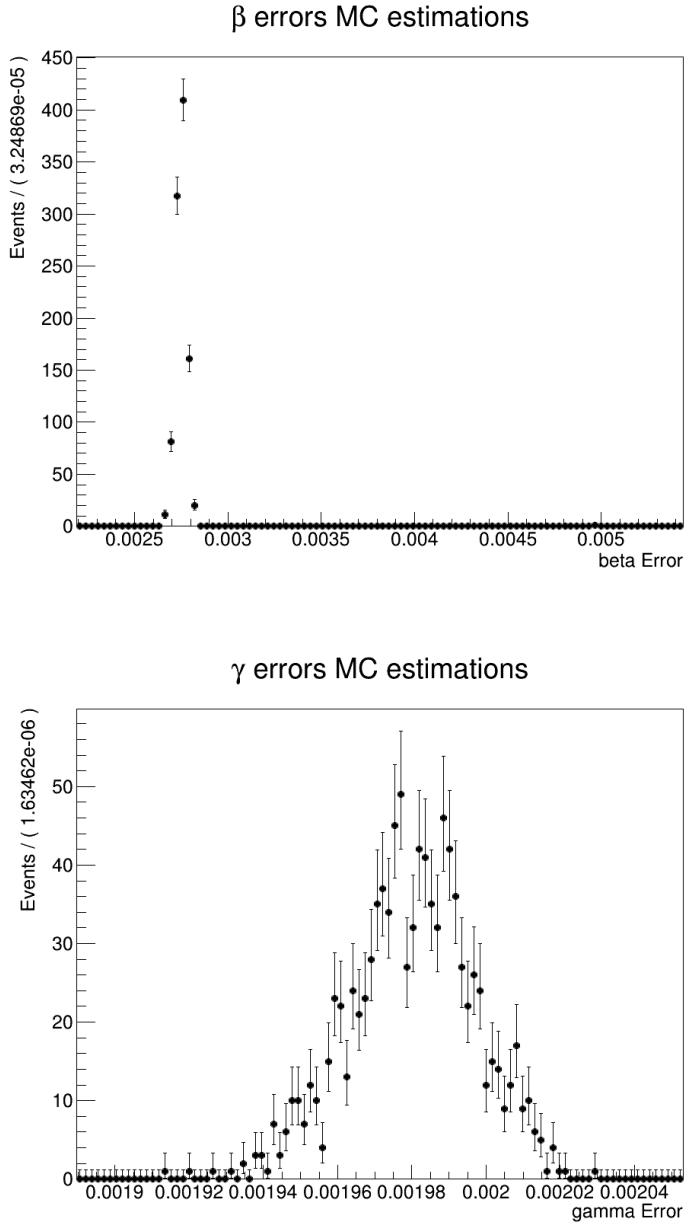


Figura 3.6: Distribuzione degli errori associati ai parametri del fit per ogni esperimento MC.

### 3.5 Binning dei dati e fit $\chi^2$

Il binning dei dati viene eseguito con risoluzione dell'1% quindi in totale si avranno 100 bin per la variabile  $\theta$  e 100 per la variabile  $\phi$ . Viene riempito un oggetto TH2F binnato tramite il metodo `fillHistogram` della classe `RooDataSet`. Per la procedura di fit, `RooFit` necessita di un oggetto `RooDataHist` che viene riempito quindi a partire dall'istogramma bi-dimensionale. Le due soluzioni vengono riportate di seguito:

```
TH2F* dh2_chi = new TH2F("", "", 100, 0, M_PI, 100, 0, 2*M_PI);
data->fillHistogram(dh2_chi, RooArgList(t,p));
```

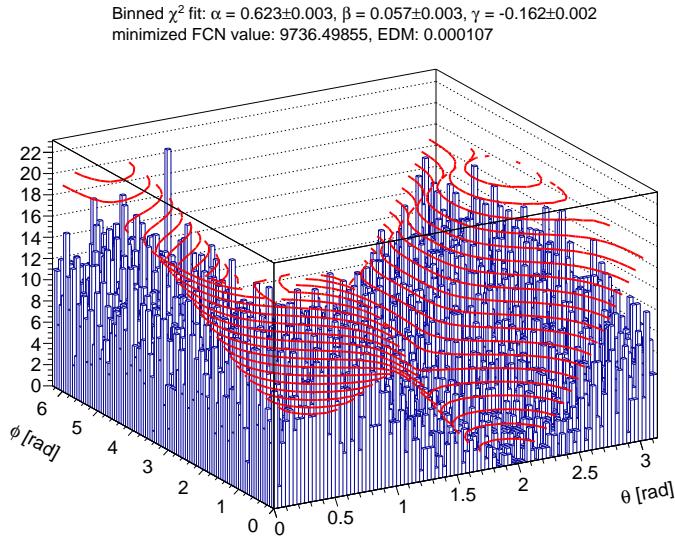
```
RooDataHist data_hist("data_hist",
"binned version of data",RooArgList(t,p),dh2_chi) ;
```

La procedura di fit tramite  $\chi^2$  è analoga a quella per ML. Infatti è presente il metodo `chi2fitTo` della classe `RooAbsPdf`. La formula analitica del  $\chi^2$  è la stessa sia in ROOT che in `RooFit` tuttavia bisogna in particolare prestare attenzione al trattamento dei bin vuoti da parte delle procedure di fit in particolare, i metodi di `RooFit` danno errori se i bin vuoti hanno anche errore pari a zero, ROOT d'altro canto non considera a prescindere i bin vuoti. L'algoritmo usato dalla procedura per la minimizzazione è Migrad e successivamente Hesse. Il codice e i risultati del fit vengono riportati di seguito in modo analogo a quanto fatto per il fit ML.

```
RooFitResult* r_chi2 = genpdf->chi2FitTo(data_hist, Save());
```

Il codice per plottare la distribuzione risultante e il dataset binnato è analogo a quella riportata in precedenza per il fit ML.

Il risultato è riportato di seguito:



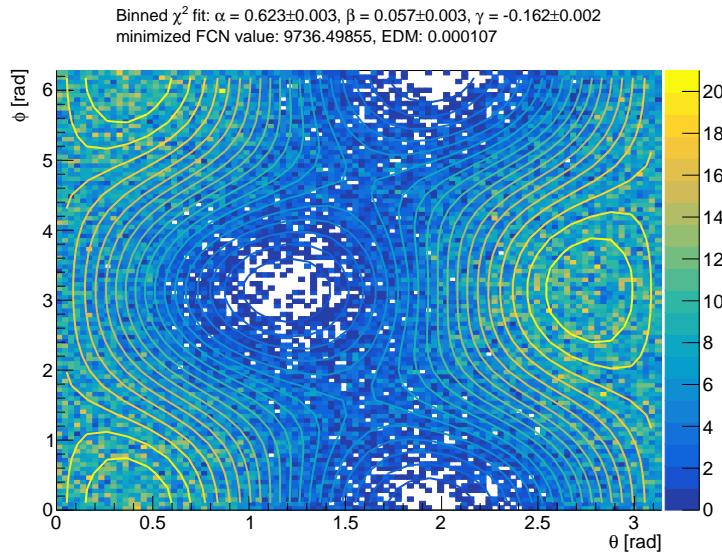


Figura 3.7: Risultati del fit  $\chi^2$  plottati nel primo grafico assieme ai dati binnati con il metodo `cont1` mentre nel secondo grafico gli stessi dati sono plottati con il metodo `colz`

I risultati della minimizzazione sono quindi riportati di seguito attraverso il metodo `Print` della classe `RooFitResults` analogamente a quanto fatto in precedenza. Si omette, per ridondanza, il codice.

`==> Chi2 Fit 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 +/- 3.17e-03  

beta       5.7336e-02 +/- 2.66e-03  

gamma     -1.6235e-01 +/- 1.89e-03
```

Analogamente a quanto fatto in precedenza riportiamo la matrice di correlazione, necessaria per capire se la minimizzazione è andata a buon fine:

PARAMETER CORRELATION COEFFICIENTS				
NO.	GLOBAL	1	2	3
1	0.18131	1.000	-0.179	0.035
2	0.37267	-0.179	1.000	-0.333
3	0.33367	0.035	-0.333	1.000

I parametri del fit binnato con metodo  $\chi^2$  sono sensibilmente peggiori di quelli da ML. Questo è prevedibile in quanto, binnando i dati, perdiamo informazione.

## 3.6 Comparazione risultati

Il fit dei parametri ha avuto buon esito per entrambi i metodi così da poter dichiarare gli estimatori come attendibili. La precedente affermazione è supportata dai parametri *EDM*, dello status della minimizzazione e dalla computazione della matrice di correlazione dei parametri.

Riportiamo quindi in una tabella i risultati dei due metodi:

	M.L.	$\chi^2$	Theory
$\alpha$	0.6518	0.6233	0.6500
$\sigma_\alpha$	0.0035	0.0032	0.0000
$\beta$	0.0593	0.0573	0.0600
$\sigma_\beta$	0.0027	0.0027	0.0000
$\gamma$	-0.1777	-0.1623	-0.1800
$\sigma_\gamma$	0.0020	0.0019	0.0000

Come prevedibile il fit di dati non binnati ottiene la massima precisione nel calcolare i parametri correttamente. Tuttavia il binnaggio è stato necessario a casua di una risoluzione finita nelle misure degli angoli. In questo caso il metodo di stima dei parametri  $\chi^2$  computa stimatori molto vicini ai valori teorici.

In particolare il metodo di fit non binnato ML stima correttamente i parametri  $\alpha$  e  $\beta$  all'interno dei loro errori statistici mentre a  $2\sigma_\gamma$  per il parametro  $\gamma$ .

## 3.7 Test ipotesi con distribuzione uniforme

I bosoni  $0^-$  hanno distribuzione isotropa ovvero ogni punto nel piano  $\theta \in [0, \pi], \phi \in [0, 2\pi]$  ha un egual probabilità di essere misurato.

Per il formalismo delle pdf, la distribuzione deve essere normalizzata nello spazio delle variabili il che porta alla seguente forma analitica:

$$f_{\Theta,\Phi}(\theta, \phi) = \begin{cases} \frac{1}{2\pi^2} & \theta \in [0, \pi], \phi \in [0, 2\pi] \\ 0 & otherwise \end{cases}$$

Possiamo costruire tale pdf grazie al pacchetto `RooFit` nel modo seguente. Viene dichiarata una variabile `RooRealVar` fissata a uno per la somma delle pdf uniformi in  $\theta, \phi$ . Vendono quindi create le pdf uniformi attraverso la classe `RooUniform` passando la variabile di interesse dichiarata all'inizio. Vengono quindi sommate le pdf uniformi creando un oggetto `RooAddPdf` istanza della classe `RooAbsPdf`:

```
RooRealVar scale_uni("scale_uni", "scale_uni", 1.);

RooUniform uni_t("uniformt", "uniformt", t);
RooUniform uni_p("uniformp", "uniformp", p);

//UNIFORM PDF
RooAddPdf unif("unif", "uni_t+uni_p",
               RooArgList(uni_t, uni_p), RooArgList(scale_uni));
```

Analogamente a quanto fatto il precedenza plottiamo la distribuzione appena creata:

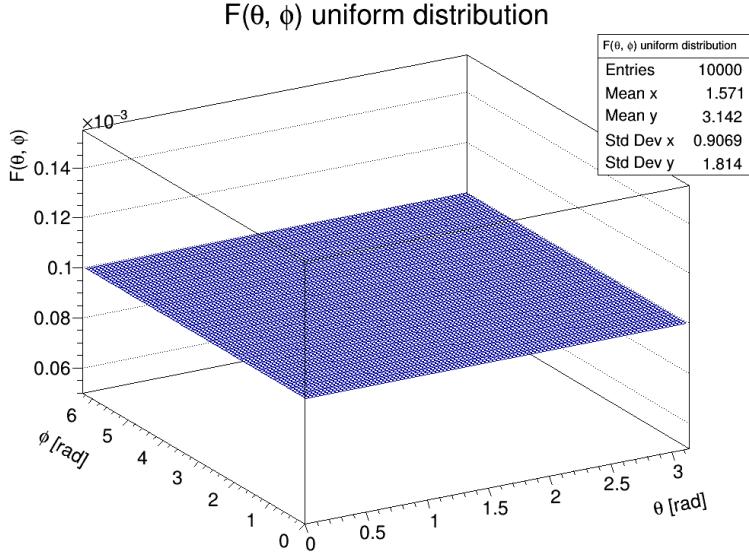


Figura 3.8: Plot bi dimensionale della distribuzione uniforme creata sommando due distribuzioni uniformi ognuna di una variabile rispettivamente  $\theta, \phi$

Un primo test può essere quello di creare una pdf formata dalla somma della pdf uniforme e della pdf data dalla distribuzione dei bosoni  $1^-$  con un parametro di libero di combinazione lineare delle pdf. Un fit viene quindi eseguito ai dati e si osserva l'adattamento dei parametri. Ci aspettiamo ovviamente che il parametro di combinazione delle pdf sia posto a zero mentre i restanti parametri della pdf  $\alpha, \beta, \gamma$  si adattino ai parametri del modello che li ha generati. La procedura è analoga alla precedente e riportata in seguito:

```
RooRealVar for_comp("for_comp", "for_comp", 0.01, 0.001, 1);
RooAddPdf zero_meno("zero_meno", "gen_pdf+uniform",
RooArgList(unif,*genpdf), RooArgList(for_comp));

zero_meno.Print();
```

Il risultato del metodo `Print` evidenzia come il parametro moltiplicativo sia associato alla pdf uniforme:

```
RooAddPdf::zero_meno[ for_comp * unif + [%] * GenPdf ] = 0.163624
```

Viene quindi eseguito un unbinned likelihood fit attraverso il seguente comando:

```
RooFitResult* r = zero_meno.fitTo(*data, Save());
```

I risultati del fit sono i seguenti:

```
====> Fit Results:
```

```

RooFitResult: minimized FCN value: 141656, estimated distance to minimum:
8.10264e-06
covariance matrix quality: Full matrix,
but forced positive-definite
Status : MINIMIZE=0 HESSE=0

Floating Parameter      FinalValue +/- Error
-----
alpha                  6.5487e-01 +/- 2.81e-02
beta                   5.9970e-02 +/- 6.22e-03
for_comp                7.5745e-03 +/- 1.92e-01
gamma                 -1.7939e-01 +/- 1.66e-02

```

Come ci aspettavamo, il parametro di combinazione ha un valore comparabile con zero all'interno del suo errore statistico mentre gli altri parametri si adattano naturalmente alla pdf che li ha generati. Tuttavia è presente un messaggio che ci avverte che l'estimazione della matrice di covarianza è andata a buon fine ma è stata forzata per definire la matrice positiva.

Il metodo non è quindi totalmente attendibile e non può essere preso in considerazione, altri test statistici devono essere indagati. E' comunque possibile plottare l'adattamento della pdf ai dati attraverso il fit precedentemente eseguito:

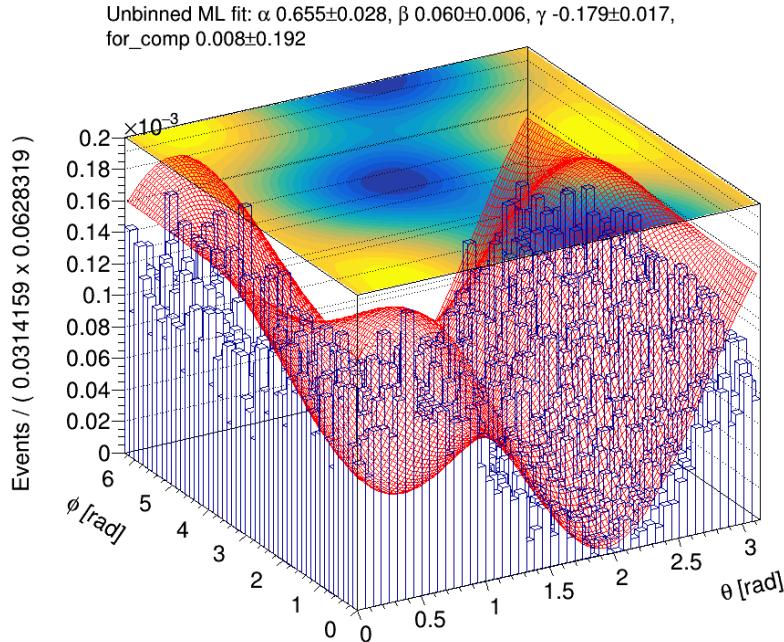


Figura 3.9: Plot della distribuzione *zero\_meno* creata dalla somma della pdf dei bosoni  $1^-$  e  $0^-$  e fittata con metodo ML ai dati generati dalla prima. I risultati sono riportati nel titolo e la pdf dopo il fit è plottata con il metodo **surf3**.

## 3.8 Likelihood ratio

Il Lemma di Neyman-Pearson ci assicura che, in caso di test statistico in cui le pdf sono completamente definite ovvero non ci sono parametri ignoti (ipotesi semplici), allora il rapporto delle likelihood è la statistica con più alto potere di test alla significatività  $\alpha$ . Più precisamente:

Sia  $H_0$  ipotesi nulla ovvero i parametri del modello sono fissati in modo che la pdf sia uniforme,  $\alpha = 1/3, \beta = 0; \gamma = 0$  mentre sia  $H_1$  l'ipotesi alternativa ovvero dove la distribuzione assume parametri della teorica di decadimento di bosoni vettori  $1^-$ ,  $\alpha = 0.65, \beta = 0.06, \gamma = -0.18$ . Il rapporto di verosimiglianza o Likelihood ratio è definito nel modo seguente:

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

La significatività del test statistico  $\alpha$  indica la probabilità di rigettare l'ipotesi  $H_0$  anche quando questa sia vera. L'errore è chiamato errore di tipo I e, scegliendo valori sufficientemente bassi di  $\alpha$ , possiamo decidere la confidenza con cui accettare o rigettare l'ipotesi nulla. L'errore di seconda specie, contrariamente è l'accettazione dell'ipotesi nulla anche quando questa sia falsa e la probabilità di questo avvenimento è definita dal parametro  $\beta$

Sia  $c$  la regione critica per il test, allora il likelihood ratio test supporta la seguente regola decisionale per rigettare o accettare l'ipotesi nulla in favore dell'ipotesi alternativa:

$$\begin{aligned} \text{if } \lambda > c &\rightarrow \text{fail to reject } H_0 \\ \text{if } \lambda < c &\rightarrow \text{reject } H_0 \end{aligned}$$

Il valore  $c$  è computato al fine di ottenere la significatività voluta dal test nel modo seguente:

$$\alpha = P(\lambda(\theta, \phi) \leq c | H_0)$$

In pratica è intuitivo affermare che se l'ipotesi nulla è più compatibile dell'ipotesi alternativa quando valutate sui dati allora il valore della statistica di test  $\lambda$  sarà grande, d'altro canto se l'ipotesi alternativa è quella più probabile allora il valore della statistica sarà prossimo a zero.

Avendo a che fare con numeri possibilmente prossimi a zero, conviene sfruttare il logaritmo della statistica. Essendo il logaritmo una funzione monotona crescente, l'operazione non ha influenza sulle qualità della statistica di test

$$\log(\lambda(\theta, \phi)) = \log\left(\frac{L(H_0|\theta, \phi)}{L(H_1|\theta, \phi)}\right) = \log(L(H_0|\theta, \phi)) - \log(L(H_1|\theta, \phi))$$

Ricordando che la likelihood è la produttoria delle pdf associate ad ogni variabile:

$$L(H_0|\theta, \phi) = \prod_{i=0}^n f_{0-}(\theta_i, \phi_i)$$

$$L(H_1|\theta, \phi) = \prod_{i=0}^n f_{1-}(\theta_i, \phi_i)$$

e dalle proprietà dei logaritmi possiamo scrivere:

$$\log(\lambda(\theta, \phi)) = \dots = \sum_{i=0}^n \log(f_{0-}(\theta_i, \phi_i)) - \sum_{i=0}^n \log(f_{1-}(\theta_i, \phi_i))$$

La procedura per calcolare  $\log(\lambda)$  si basa sul fatto che per ogni valore di  $\theta, \phi$  nel `RooDataSet`, la distribuzione uniforme prevede una probabilità pari a  $\frac{1}{2\pi^2}$  mentre la pdf per la distribuzione dei bosoni  $1^-$  deve essere valutata in ogni punto del dataset. Si prende quindi il logaritmo dei due valori e si sottrae il primo al secondo. Il risultato viene quindi sommato per ottenere la statistica finale.

Il codice per la computazione viene riportato di seguito:

```
double prod_H1; //first sum variable
double prod_H0 ; //second sum variable

long double ln_r = 0.; //final statistic
double unif_prob = 1./(M_PI*2*M_PI); //prob for uniform distribution

//for retrieving height of 1- pdf
RooArgSet* pdf0bs = genpdf->getObservables(*data) ;

//The formula is the following:
//lambda = (prod(f(x/H1))/prod(f(x/H0)))
//log(lambda) = sum(log(f(x/H1))) - sum(log(f(x/H0)))

//cycle on all data
for(int i = 0; i < 50000; i++){

    //retireving the row
    auto t_p_ref = *data->get(i);
    double t_val = t_p_ref.getRealValue("t");

    //retrieving the value of the variables
    double p_val = t_p_ref.getRealValue("p");

    //the probability is uniform for each entry, does not depend on
    //theta and phi.
    prod_H0 = log(unif_prob);

    //Retrieving height of the pdf at the dataset point
    *pdf0bs = *data->get(i);
    prod_H1 = log(genpdf->getVal());

    //subtracting logarithm and summing them to the statistics
    ln_r += prod_H0-prod_H1;

}
```

Il risultato finale di questa computazione è di seguito riportato:

$$\log(\lambda(\theta, \phi)) = -40772.8 \rightarrow \lambda = e^{-40772.8}$$

Si può notare che il valore nella RHS sorpassa di molti ordini di grandezza il limite numerico della macchina che è fissato a 2.22507e-308. L'ipotesi di distribuzione uniforme è quindi da rigettare in favore dell'ipotesi alternativa come previsto.

# Capitolo 4

## Esercizio 4

L'esercizio propone di generare  $N$  particelle con distribuzione di momento uniforme nell'intervallo  $[0, 10] \frac{GeV}{c}$  con metodo Monte Carlo. Bisogna quindi dimostrare che il dip della distribuzione di  $< p_T >$  in funzione di  $P_L$  è dovuto solamente ad effetti cinematici.

### 4.1 Generazione degli eventi

Per la prima parte si è implementata una libreria `generate_mc` in grado di generare le componenti  $P_T, P_L$  del momento di una particella.

La struttura della libreria è la seguente:

```
#ifndef MONTE_CARLO_H
#define MONTE_CARLO_H

struct generation{
    double pl;
    double pt;
};

class generate_mc{

    int seed;

public:
    generate_mc(int s){ seed = s; };
    ~generate_mc(){}
    generation simulation();

};

#endif //MONTE_CARLO_H
```

Viene definita una struct `generation` che conterrà le variabili di interesse  $P_T, P_L$ .

La classe `generate_mc` ha come attributi le seguenti variabili:

- **seed**: il seed di inizializzazione del generatore di numeri pseudo-casuali.

Come metodi invece abbiamo i seguenti:

- **generate\_mc(int s)**: costruttore a cui viene passato un numero intero che verrà salvato nell'attributo della classe. Questo sarà il seed per la generazione.
- **generate\_mc()**: distruttore dell'oggetto.
- **simulation**: simula l'osservazione di una particella. restituisce quindi le componenti del momento  $P_T, P_L$  attraverso la struct **generation**.

Nel main generiamo quindi un totale di  $N = 100000$  eventi nel modo seguente. Viene inizializzato il generatore pseudo-random **srand** di c++. Questo servirà solamente a generare numeri interi casuali che verranno passati come seed alla classe **generate\_mc** la quale usa il generatore di numeri pseudo-casuali **TRandom3** di ROOT.

Cicliamo sul numero di eventi e, per ogni evento, creiamo un oggetto della classe **generate\_mc** passando al costruttore un seed pseudo-casuale, generato tramite la funzione **rand()**. Ricaviamo i valori di interesse dalla struct **generation** all'output del metodo **simulation** e filliamo un istogramma bi dimensionale **TH2F** in  $P_T, P_L$  composto da 100 bin per dimensione.

Il codice è riportato di seguito:

```
int main(){
    int N = 100000;

    TH2F* h2 = new TH2F("h2", "h2", 100, 0, 10, 100, 0, 10);
    for(int i = 0; i < N; i++){
        generate_mc sim(rand());
        generation result = sim.simulation();
        h2->Fill(result.pl, result.pt);

    }
}
```

Il metodo **simulation** è esplicato di seguito.

Si creano due generatori di numeri pseudo-casuali dalla classe **TRandom3** di ROOT. Calcoleremo  $P_T, P_L$  nel seguente modo:

$$P \sim U[0, 10] , \quad \cos(\theta) \sim \cos(U[0, 2\pi])$$

$$P_L = \text{abs}(P \cos(\theta)) , \quad P_T = P \sqrt{1 - \cos^2(\theta)}$$

Avremmo alternativamente potuto campionare direttamente:

$$\cos(\theta) \sim U[-1, 1]$$

Tuttavia, come è evidente dalla forma funzionale del coseno nell'intervallo  $[0, 2\pi]$ , solo un valore di  $\theta$  porta a  $\cos(\theta) = -1$  mentre qualsiasi altro valore di  $\cos(\theta)$  in  $(-1, 1]$  ha due possibili valori di  $\theta$ . Il valore di  $\cos(\theta) = -1$  ha quindi esattamente

metà della probabilità di essere campionato rispetto agli altri, ma questo non è vero dal campionamento  $\cos(\theta) \sim U[-1, 1]$  ove ogni numero nell'intervallo [-1,1] ha una probabilità di  $p = \frac{1}{2\pi}$  di essere campionato. Osserveremo le differenze nelle generazioni nei capitoli seguenti.

Il codice del metodo è quindi riportato di seguito:

```
generation generate_mc::simulation(){

    generation result;

    //generatore causale momento
    TRandom3* gen_p = new TRandom3(seed);
    //generatore causale angoli
    TRandom3* gen_cos = new TRandom3(seed + rand());

    double p = gen_p->Uniform(0,10);
    double coseno = cos( gen_cos->Uniform(0,2*M_PI) );

    result.pl = abs(p*coseno);
    result.pt = p*(sqrt(1-pow(coseno, 2)));

    return result;
}
```

Il risultato della generazione Monte Carlo è plottato su un canvas con metodo `lego` della classe `TH2F`. Il risultato è il seguente:

P<sub>T</sub> versus P<sub>L</sub> distribution for N=100000 particles

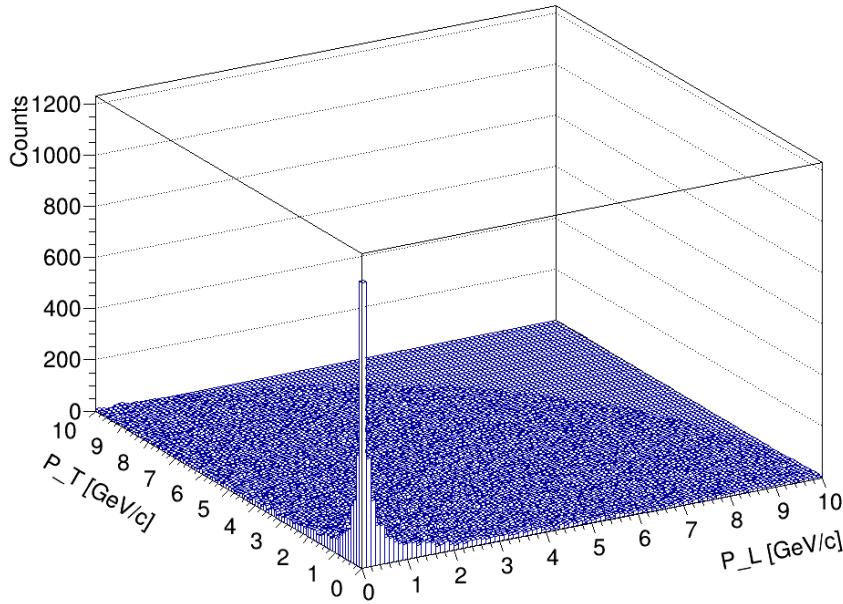


Figura 4.1: Distribuzione bi-dimensionale di  $P_T$  versus  $P_L$  per i dati generati con metodo Monte Carlo.

## 4.2 Distributione $\langle P_T \rangle$

Vogliamo ora calcolare la media dei momenti trasversi in funzione del momento longitudinale ovvero:

$$\langle P_T \rangle = f(P_L)$$

Sia  $k$  il numero di intervalli, uguale per entrambe le dimensioni e sia  $i = 1...k$  un bin generico nella variabile  $P_L$ . Sia  $m = 0...k$  un bin generico nella variabile  $P_T$  e sia  $n_m$  il numero di entrate del bin. Allora possiamo scrivere:

$$f(P_L)_i = \left( \frac{1}{\sum_{m=0}^k n_m} \sum_{m=0}^k n_m (P_T)_m \right)_i$$

Dove la media è relativa al bin  $i$ -esimo in  $P_L$ . Root implementa questo tipo di calcolo in una classe chiamata **TProfile** che esegue esattamente il calcolo descritto precedentemente.

Possiamo inoltre creare un oggetto **TProfile** direttamente dal **TH2F** creato in precedenza tramite il metodo **ProfileX**.

Creiamo quindi il **TProfile** specificando il binnaggio su cui computare la media. Plottiamo quindi il risultato in un canvas:

```
TProfile *pt_m = h2->ProfileX("p_m", 0, 100);
pt_m->GetXaxis()->SetTitle("P_{L} [GeV/mc2]");
pt_m->GetXaxis()->SetTitleOffset(1.2);
pt_m->GetYaxis()->SetTitle("<P_{T}> [GeV/mc2]");
```

```

pt_m->SetTitle("<P_T> Graph in function of P_L for 10000 mc events");

TCanvas*c = new TCanvas("c", "c", 1000, 1000, 1000, 800);
pt_m->Draw();
c->Draw();

```

Il risultato è il seguente:

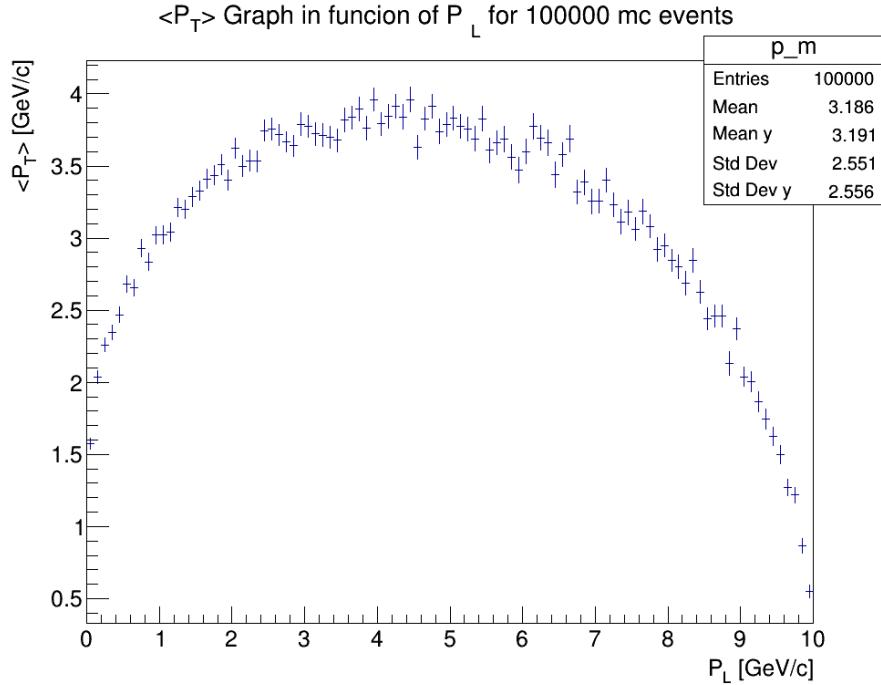


Figura 4.2: Profile dell’istogramma bi dimensionale, eseguendo la media dei  $P_T$  per ogni bin  $P_L$ . L’istogramma creato avrà come entrata  $\langle P_T \rangle$  e come errore sul bin la deviazione standard della media  $\delta \langle P_T \rangle_i = \frac{\sigma}{\sqrt{n_i}}$ .

E’ ovvio dedurre l’andamento dell’istogramma al variare di  $P_L$ . Dato che:

$$P_L \leq P , \quad P_T \leq P$$

$$P \leq 10 \text{GeV}/c$$

Allora:

$$P_L \rightarrow 10 , \quad P_T \rightarrow 0$$

Tuttavia non è per niente scontato l’andamento per  $P_L \rightarrow 0$  in cui osserviamo un dip della distribuzione, con valori di  $P_T$  che tendono a 1.5 molto rapidamente.

Supponiamo di generare valori di  $P$  prossimi agli estremi dell’intervallo:

$$P \sim U[0, 10] \rightarrow 10$$

$$P \sim U[0, 10] \rightarrow 0$$

Allora:

$$P^2 = P_T^2 + P_L^2 = P^2 \sqrt{1 - \cos^2 \theta} + P^2 \cos^2(\theta)$$

$$P \rightarrow 10 , \quad 100 = 100\sqrt{1 - \cos^2\theta} + 100\cos^2(\theta)$$

Essendo  $\cos^2[\theta] \in [0, 1]$  avremo che per  $\cos(\theta)^2 \rightarrow 1$ ,  $P_L \rightarrow P$  ,  $P_T \rightarrow 0$ . Nell'altro limite invece il valore di  $P$  è finito ed è nullo per qualsiasi valore di  $\theta$  generati:

$$\lim_{P^2 \rightarrow 0} P^2\sqrt{1 - \cos^2\theta} + P^2\cos^2(\theta)$$

I due termini tendono a zero indipendentemente da  $\theta$  quindi:

$$P \rightarrow 0 , \quad P_L \rightarrow P , \quad P_T \rightarrow P$$

Che porta al risultato:

$$P \rightarrow 0 , \quad P_L \rightarrow 0, P_T \rightarrow 0$$

Ovviamente la distribuzione non va esattamente a zero in quanto si hanno contributi non nulli da situazioni dove  $P \neq 0$  e la generazione dell'angolo porta a  $\theta \sim (2k + 1)\frac{\pi}{2}, k = 0, 1, \dots$

### 4.3 Distribuzione di probabilità del $P_T$

Si può, come prova, spiegare il picco della distribuzione bi dimensionale di  $P_T, P_L$  a bassi valori di  $P_T, P_L$ . L'argomento è puramente statistico.

Per semplicità chiamiamo la variabile  $P \rightarrow X$ . La distribuzione di  $X$  è uniforme in  $[0, 10]$ :

$$X \sim U[0, 10] , \quad f_X(x) = \begin{cases} \frac{1}{10} & \text{if } x \in [0, 10] \\ 0 & \text{otherwise} \end{cases}$$

Anche la variabile  $\theta$  è generata uniformemente tra  $[0, 2\pi]$ :

$$\theta \sim U[0, 2\pi] , \quad f_\Theta(\theta) = \begin{cases} \frac{1}{2\pi} & \text{if } \theta \in [0, 2\pi] \\ 0 & \text{otherwise} \end{cases}$$

Tuttavia la distribuzione del seno di una variabile casuale distribuita uniformemente non è anche essa uniforme.

L'immagine della distribuzione seno è ovviamente  $[-1, 1]$ . Ricordando che, dalla cdf di una variabile casuale uniforme:

$$F_W(\theta) = Pr(W \leq \theta) = \int_0^W f_\Theta(\theta)d\theta = \frac{W}{2\pi}$$

Chiamiamo  $Y = \sin(\theta)$  allora per  $y \in [0, 1]$ :

$$F_Y(y) = Pr(Y \leq y) = Pr(0 \leq X \leq \arcsin(y), 2\pi \geq \theta \geq \pi - \arcsin(y)) =$$

$$\frac{\arcsin(y) + (2\pi - (\pi - \arcsin(y)))}{2\pi} = \frac{1}{\pi\sqrt{1 - y^2}}$$

Per ottenere la distribuzione del  $P_T$  doppiamo moltiplicare due pdf. La formula per  $Z$  variabile casuale prodotto di due variabili casuali  $Z = XY$  è la seguente:

$$f_Z(z) = \int_{-\infty}^{\infty} f_X(x) f_Y\left(\frac{z}{x}\right) \frac{1}{|x|} dx$$

Nel nostro caso avremo:

$$f_Z(z) = \int_0^{10} \frac{1}{10\pi} \frac{1}{\sqrt{(1 - \frac{z^2}{x^2})}} \frac{1}{|x|} dx = \frac{1}{10\pi} \left[ \log(\sqrt{x^2 - z^2} + x) \right]_{x=|z|}^{x=10}$$

Da cui finalmente:

$$f_{P_T}(p_T) = \frac{1}{5\pi} \log\left(\frac{\sqrt{10^2 - p_T^2} + 10}{|p_T|}\right)$$

Si può controllare che la distribuzione è normalizzata in  $[0,10]$ . La distribuzione risulta divergente a  $\infty$  per  $p_T \rightarrow 0$ . Plottiamo la distribuzione nell'intervallo  $[0,10]$ :

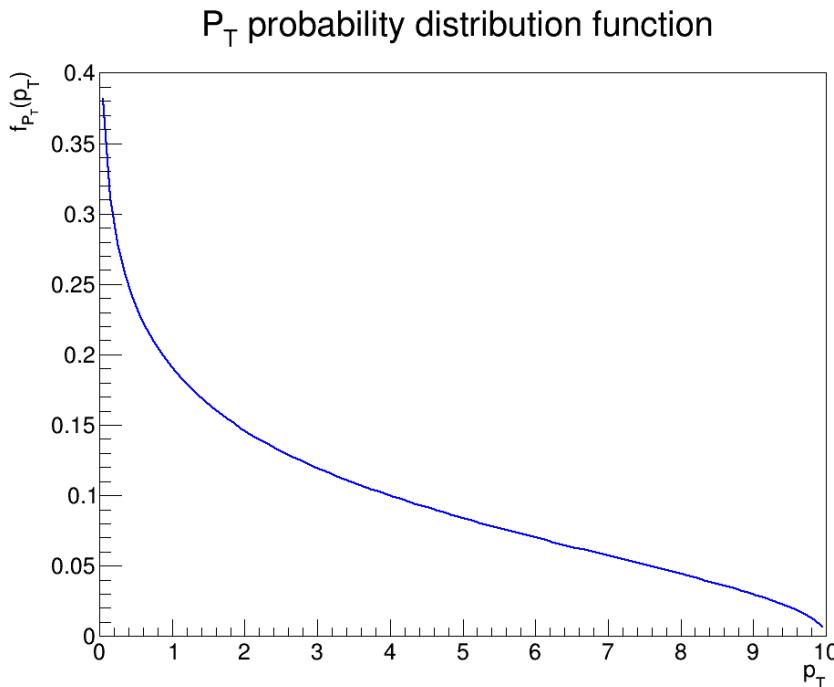


Figura 4.3: pdf del momento trasverso creato dalla produttoria di una variabile casuale uniforme in  $[0,10]$  e del seno di una variabile casuale in  $[0, 2\pi]$

Si potrebbe pensare che generare  $\theta$  uniforme oppure  $\sin(\theta)$  uniforme non cambierebbe nulla dal punto di vista fisico. Abbiamo tuttavia dimostrato che la distribuzione del seno di una variabile casuale distribuita uniformemente è sensibilmente diversa dalla distribuzione uniforme. Proviamo a simulare lo stesso processo ma generando  $\sin(\theta) \sim U[-1, 1]$ .

Per comodità rinominiamo  $P \rightarrow X$ ,  $\sin(\theta) \rightarrow Y$  allora:

$$X \sim U[0, 10] \quad , \quad f_X(x) = \begin{cases} \frac{1}{10} & \text{if } x \in [0, 10] \\ 0 & \text{otherwise} \end{cases}$$

$$Y \sim U[-1, 1] \quad , \quad f_Y(y) = \begin{cases} \frac{1}{2} & \text{if } y \in [-1, 1] \\ 0 & \text{otherwise} \end{cases}$$

La pdf di  $Z = XY$  sarà, in questo caso, il prodotto di due pdf uniformi.

$$\begin{aligned} f_Z(z) &= \int_{-\infty}^{\infty} \frac{1}{|x|} f_{X,Y}(x, \frac{z}{x}) dx = \frac{1}{|x|} f_X(x) f_Y(\frac{z}{x}) dx = \\ &= \frac{1}{10} \int_0^{10} \frac{1}{|x|} f_Y(\frac{z}{x}) dx = \frac{1}{10} \log\left(\frac{10}{|z|}\right) \quad 0 \leq z \leq 10 \end{aligned}$$

Per riassumere:

$$\begin{aligned} \theta &\sim U[0, 2\pi] \quad , \quad f_{P_T}(p_T) = \frac{1}{5\pi} \log\left(\frac{\sqrt{10^2 - p_T^2} + 10}{|p_T|}\right) \\ \sin(\theta) &\sim U[-1, 1] \quad , \quad f_{P_T}(p_T) = \frac{1}{10} \log\left(\frac{10}{|p_T|}\right) \end{aligned}$$

Notiamo che le distribuzioni sono sensibilmente differenti. La fisica e la generazione stessa è diversa per i due metodi. Plottiamo le due distribuzioni a confronto tra [0,10]:

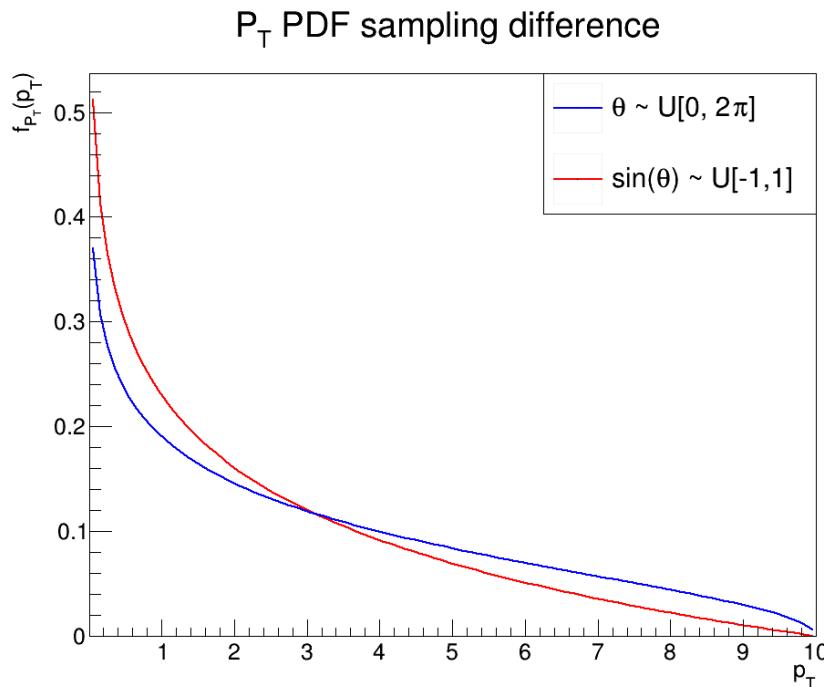


Figura 4.4: Differenza nelle pdf del momento trasverso a seconda della generazione uniforme di  $\theta$  o di  $\sin(\theta)$ .

Proviamo a simulare un esperimento generando il momento uniforme in  $[0,10]$  e il seno uniforme in  $[-1,1]$ .

Nella classe `monte_carlo` definiamo una funzione `simulation_sine` con la seguente struttura:

```

generation generate_mc::simulation_sine(){

    generation result;

    TRandom3* gen_p = new TRandom3(seed); //generatore causale momento
    TRandom3* gen_sin = new TRandom3(seed + rand()); //generatore causale angoli

    double p = gen_p->Uniform(0,10);
    double sine = gen_sin->Uniform(-1,1);

    result.pl = p*sqrt(1-sine*sine);
    result.pt = abs(p*sine);

    return result;

}

```

Nel main lo svolgimento è del tutto analogo al precedente e ci limitiamo a plottare i risultati:

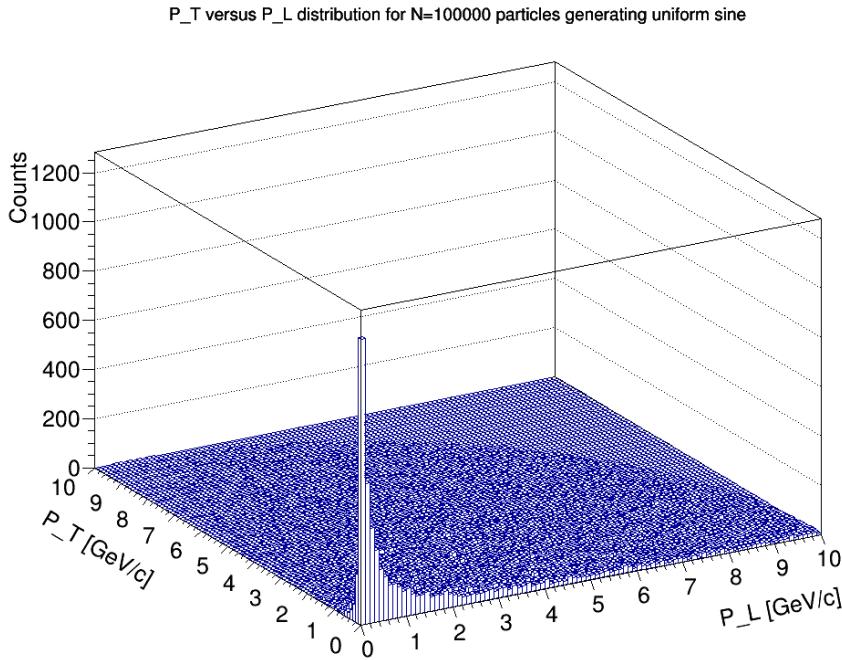


Figura 4.5: Distribuzione bi-dimensionale associata alla generazione di 100000 eventi monte carlo con  $P \sim U[0, 10]$ ,  $\sin(\theta)U[-1, 1]$ .

Notiamo la forte assimetria del grafico nelle sue componenti cartesiane  $P_T, P_L$ . Plottiamo quindi i risultati del **TProfile** in contrapposizione ai precedenti:

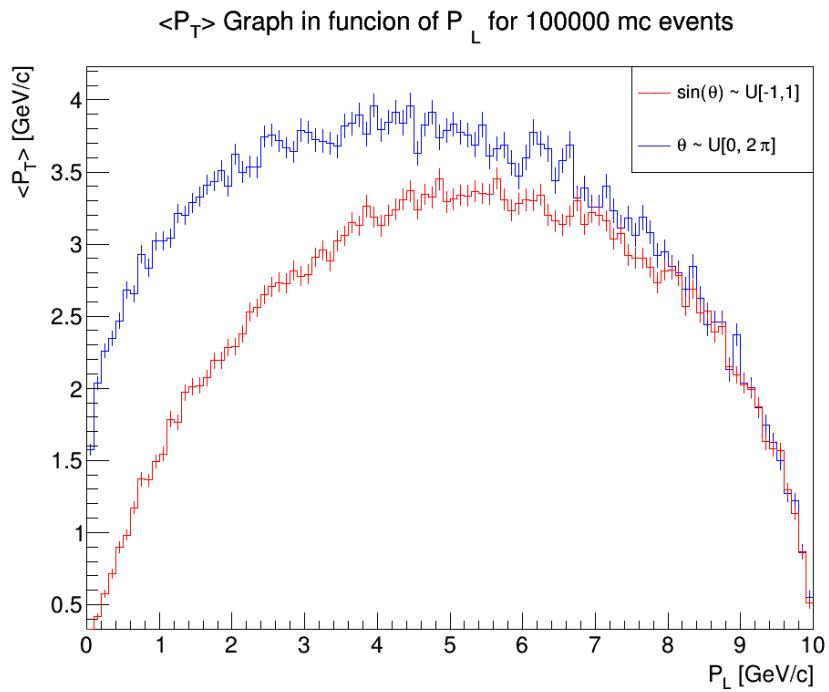


Figura 4.6: Confronto dei valori medi del  $\langle P_T \rangle$  in funzione del momento longitudinale ottenuti con un Profiling degli istogrammi bi dimensionali. Possiamo apprezzare la differenza di distribuzione con le due generazioni.

# Capitolo 5

## Esercizio 5

L'esercizio propone di calcolare il seguente integrale definito, attraverso metodo montecarlo:

$$\int_0^1 e^x dx$$

La cui soluzione analitica è la seguente:

$$e - 1 \simeq 1.71828182846\dots$$

Per introdurre la validità del metodo di seguito spieghiamo il principio dell'integrazione con metodo Montecarlo.

L'obiettivo del metodo è computare integrali definiti in uno spazio  $N$  dimensionale usando generatori di numeri casuali. Sia:

$$I = \int_{\Omega} f(\bar{x}) d\bar{x}$$

Dove  $\Omega$  è un sottospazio di  $R^N$  di volume:

$$V = \int_{\Omega} d\bar{x}$$

Siano  $\bar{x}_1 \dots \bar{x}_n$  n vettori generati uniformemente nel dominio  $\Omega$  allora la legge dei grandi numeri ci permette di scrivere:

$$\frac{1}{n} \sum_{i=1}^n f(\bar{x}_i) \xrightarrow{n \rightarrow \infty} \frac{1}{V} \int_{\Omega} f(\bar{x}) d\bar{x}$$

Quindi matematicamente per legge dei grandi numeri:

$$Q_n = \frac{V}{n} \sum_{i=1}^n f(\bar{x}_i) \approx I$$

$Q_n$  è uno stimatore consistente dell'integrale definito per n abbastanza grande (l'approssimazione diventa un uguaglianza nel limite  $n \rightarrow \infty$ ).

Il rate di convergenza e quindi l'errore dello stimatore possono essere calcolati dalla varianza campionaria:

$$Var(f) = \sigma_n^2 = \frac{1}{N-1} \sum_{i=1}^n (f(\bar{x}_i) - \langle f \rangle)^2$$

$$Var(Q_n) = \frac{V^2}{n^2} \sum_{i=1}^n Var(f) = \frac{V^2 \sigma_n^2}{n}$$

L'errore sullo stimatore  $Q_n$  è quindi:

$$\delta Q_n = \sqrt{Var(Q_n)} = \frac{V \sigma_n}{\sqrt{n}}$$

il cui andamento  $\frac{1}{\sqrt{n}}$  dipende solo dalla dimensione del campione generato nel dominio  $\Omega$  e non dalla dimensione dello spazio. Questo è uno dei vantaggi dei metodi montecarlo per la computazione di integrali in spazi N dimensionali.

## 5.1 Descrizione delle classi

Il primo passo per il calcolo dell'integrale della funzione esponenziale in  $[0,1]$  è generare numeri causali nell'intervallo. Il procedimento adottato involve l'uso della classe `TRandom3` di ROOT capace di generare numeri pseudo-casuali secondo il Mersenne Twister generator. Il periodo del generatore è di  $2^{19937} - 1$ .

ROOT implementa delle macro di test statistici per generatori pseudo-casuali. La classe `TRandom` implementa l'algoritmo LCG e per questo fallisce la maggior parte dei test. `TRandom3` si rivela il generatore più affidabile di numeri pseudo casuali come visionabile superando 94 tests su 94 come visionabile al seguente link [CDF].

Tuttavia drawback della classe `TRandom3` sono conosciute dal team di ROOT e perciò è stata implementata la classe `TRandomMixMax`, con un generatore più robusto. In alternativa avremmo potuto utilizzare il generatore `TRandomMixMax` ma la differenza tra i due generatori, al fine dell'esercizio è pressocchè inosservabile. Come ultima alternativa si sarebbe potuto utilizzare l'algoritmo `xorshiro128p` illustrato nel primo esercizio.

Attraverso una dedita classe siamo capaci di implementare diverse strategie di generazione e storage delle informazioni. La classe `one`, utilizzata nel nostro codice per generare sequenze di numeri causali con vari metodi, è strutturata nel modo seguente:

```
#ifndef ONE_H
#define ONE_H

#include <vector>

class ran{
    int size;
    std::vector<double> arr;
    int seed;
    int interval_elements;
    std::vector<double> strat_arr;

public:
```

```

ran(int s, int se){ size = s ; seed = se ; };
~ran(){}
//methods

int get_size(){ return size ; };
int get_strat_size(){ return interval_elements; };
std::vector<double> get_arr(){ return arr ; };
std::vector<double> get_stratified_arr(){ return strat_arr ; };
void generate();
void generate_interval(std::vector<double> intervals);
void generate_expo();
void generate_par();
void generate_par2();

};

#endif

```

Gli attributi privati prevedono le seguenti variabili:

- **size**: dimensione dei numeri da generare.
- **arr**: vettore della STL dove verranno salvati i numeri generati uniformemente.
- **seed**: intero per l'inizializzazione del TRandom. Utile se vogliamo simulare diversi esperimenti montecarlo.
- **interval\_elements**: numero di generazioni pseudo-random in ogni intervallo nel metodo stratified-sampling per riduzione della varianza.
- **strat\_arr**: vettore della STL dove verranno salvati i numeri generati dalla stratificazione.

I metodi della classe permettono le seguenti operazioni:

- **ran(int s, int se)**: costruttore, vengono passate la dimensione e il seed del generatore.
- **~ran()**: distruttore.
- **int get\_size()**: restituisce la dimensione dei numeri generati.
- **int get\_strat\_size()**: restituisce la dimensione dei numeri generati per ogni intervalli dello stratified-sampling.
- **vector<double> get\_arr()**: restituisce l'array di numeri generati uniformemente.
- **vector<double> get\_stratified\_array**: restituisce array stratificato.

- void **generate()**: funzione che genera i numeri pseudo-causali.
- void **generate\_intervals(vector<double>)**: genera un vettore di numeri pseudocasuali stratificato secondo gli intervalli passati dall'user. il vettore degli intervalli deve contenere dei limitatori equispaziati.
- void **generate\_expo()**: genera numeri casuali distribuiti secondo un esponenziale.
- void **generate\_par()**: genera numeri distribuiti secondo  $2.5x^{1.5}$ .
- void **generate\_par2()**: genera numeri distribuiti secondo  $(x + 1)^{1.5}$ .

La classe **func** è utilizzata invece per lo storage e la manipolazione delle funzioni come ad esempio quella esponenziale. La struttura è la seguente:

```
#ifndef FUNCTION_H
#define FUNCTION_H

#include <vector>

class func{

    int size;
    std::vector<double> x;
    std::vector<double> y;
public:
    func(std::vector<double> x_gen){ x = x_gen; size = x_gen.size() ; };
    ~func(){};
    std::vector<double> compute(std::vector<double> arr);
    std::vector<double> compute_par(std::vector<double> arr);
    compute_par2(std::vector<double> arr);
    double mc_integral();
    double mc_variance();

};

#endif
```

Gli attributi privati prevedono le seguenti variabili:

- **size**: numero di istanze da computare.
- **x**: vettore STL delle ascisse.
- **y**: vettore STL delle ordinate.

I metodi della classe sono descritti di seguito:

- **func(std::vector<double> x\_gen)**: costruttore dell'oggetto, viene passato in input il vettore delle ascisse che sarà salvato di default nel relativo vettore privato dell'oggetto.

- **func()**: distruttore dell'oggetto.
- **compute**: valuta la funzione esponenziale sui punti del vettore delle ascisse  $x$ .
- **compute\_par**: valuta la funzione  $2.5x^{1.5}$
- **compute\_par2**: valuta la funzione  $(x + 1)^{1.5}$
- **mc\_integral**: computa l'integrale dalle operazioni prima svolte.
- **mc\_variance**: computa la varianza dell'estimatore monte carlo.

L'ultima classe è quella che collega le librerie precedenti e produce un'analisi del risultato e i plot significativi `monte_carlo`:

```
#ifndef MONTE_H
#define MONTE_H
#include <vector>

class monte_carlo{
    int events;
    int measures;
    int seed;
    std::vector<double> mc_est;
    std::vector<double> mc_var;
public:
    monte_carlo(int n, int m){ events = n ; measures = m; seed = 0; };
    ~monte_carlo(){};
    void set_seed(int s){seed = s ;}
    void run_simulation();
    void run_stratified_simulation(std::vector<double> interv);
    void run_importance_expo_simu();
    void run_importance_par_simu();
    void run_importance_par_simu_2();
    void run_antithetic_simu();
    std::vector<double> get_var(){return mc_var; } ;
    void plot_estimators(const char* title, const char* output);
    void plot_variance(const char* title, const char* output);
};

#endif
```

Gli attributi privati della classe prevedono le seguenti variabili:

- **events**: numero di esperimenti montecarlo.
- **measures**: numero di valutazioni della funzione per ogni esperimento.
- **seed**: il seed iniziale per la generazione dei numeri casuali.

- **mc\_est**: vettore STD contenente gli estimatori per ogni esperimento.
- **mc\_var**: vettore STD contenente gli estimatori della varianza per ogni esperimento.

I metodi della classe sono descritti di seguito:

- **monte\_carlo(int n, int m)**: costruttore dell'oggetto monte carlo inizializzato con il numero di esperimenti ed il numero di misure per esperimento.
- **monte\_carlo()**: distruttore dell'oggetto.
- **set\_seed(int s)**: metodo per cambiare il seed di inizializzazione di default.
- **run\_simulation()**: Restituisce il vettore degli estimatori con metodo crude monte carlo.
- **run\_stratified\_simulation**: restituisce l'array degli estimatori del monte carlo con stratified sampling come metodo di riduzione della varianza.
- **run\_importance\_expo\_simu**: restituisce l'array degli estimatori del monte carlo con importance sampling di un'esponenziale come metodo di riduzione della varianza.
- **run\_importance\_par\_simu**: restituisce l'array degli estimatori del monte carlo con importance sampling della funzione  $2.5x^{1.5}$  come metodo di riduzione della varianza.
- **run\_antithetic\_simu**: restituisce l'array degli estimatori del monte carlo con metodo di riduzione della varianza il metodo delle variabili antitetiche.
- **plot\_estimators**: produce un plot della distribuzione degli estimatori nei vari esperimenti.
- **plot\_variance**: produce un plot della distribuzione delle varianze degli estimatori nei vari esperimenti.

## 5.2 Crude MC

Per il crude montecarlo è solamente necessario l'utilizzo della funzione `run_simulation` di seguito illustrata.

Nel main viene istanziato un oggetto Monte Carlo. Al costruttore vengono passati il numero di esperimenti e il numero di sample causali per ogni esperimento, valori che vengono fissati rispettivamente a 2000 e 10000:

```
int main(){
    //Setto numero eventi, numero punti random e vettore estimatori finali.
    int events = 2000;
    int measures = 10000;

    monte_carlo mc(events, measures); //Istanza mc con 2000 eventi ognuno da 10000
```

Da notare che questo sarà l'unico oggetto che effettuerà l'analisi con diversi metodi nel seguito esplicati. Per simulare gli esperimenti basta solamente chiamare il metodo `run_simulation()` della classe `monte_carlo` che salva negli attributi `mc_est` e `mc_var` un vettore della libreria standard c++:

```
//Eseguo simulazioni dei dati e ne traggo l'integrale
mc.run_simulation();
```

Il metodo `run_simulation` è illustrato di seguito: Vengono inizializzati un seed da passare al generatore di numeri casuali e il vettore contenente gli estimatori montecarlo. L'attributo `events` viene salvato nella classe `monte_carlo` alla sua costruzione e indica il numero di esperimenti che si vogliono eseguire.

Si cicla quindi sugli esperimenti, viene inizializzato il seed intero e viene costruito un oggetto della classe `ran` chiamato `evento` a cui vengono passati in ordine il numero di eventi pseudo-casuali da generare e il seed. Viene quindi chiamato il metodo `generate()` della classe `ran` che genera numeri pseudo-casuali e li salva in un vettore attributo della classe stessa. Questi numeri vengono quindi restituiti grazie al metodo `get_arr()` della classe `ran`, infatti il vettore generato è salvato come attributo privato, a cui abbiamo accesso solamente tramite metodi dell'oggetto stesso.

Viene quindi costruito un oggetto della classe `func` che istanzia la funzione esponenziale e a cui viene passato l'array delle ascisse ovvero i numeri pseudo-casuali generati in precedenza. Il metodo `compute(array)` della classe `func` permette di valutare la funzione in ogni punto dell'array e restituirne le ordinate sotto forma di vettore.

Si arriva quindi alla computazione dell'integrale e della varianza dell'estimatore. Vengono riportate di seguito le formule:

$$S_n = \frac{1}{n} \sum_{j=0}^n f(x_j)$$

$$\hat{\sigma}^2 = \frac{1}{n} \sum_{j=0}^n (f(x_j))^2 - S_n^2$$

Il codice esegue in un ciclo la sommatoria. Si salva quindi in un vettore interno alla funzione il risultato dell'integrale  $S_n$  e della varianza `variance`. Gli attributi della classe relativi a queste computazioni vengono quindi aggiornati con i vettori appena computati alla fine del ciclo esterno.

```
void monte_carlo::run_simulation(){

    //inizializzo seed e vettore finale.
    srand(time(NULL));
    std::vector<double> sim_res;
    std::vector<double> sim_var;

    //ciclo sul numero di eventi per generare la simulazione.
    for(int i=0; i<events; i++){
```

```

int s = rand();
//creo classe numero casuale. Grazie al seed
//all'interno del ciclo, ci assicuriamo che per ogni
//esperimento i numeri generati saranno diversi
//(o quasi data la pseudo-casualità del generatore)
ran evento(measures, rand());
//genero numeri casuali
evento.generate();

//retrieve vettore numeri causali
std::vector<double> myarr = evento.get_arr();

//definisco oggetto funzione passandogli le x random
func expo(myarr);

//compuo valore delle y, verrà salvato nella classe
std::vector<double> y = expo.compute(myarr);

//compuo integrale e varianza e appendo alla lista finale

double S_n = 0; //integrale
double Error = 0; //varianza
for(int j = 0; j < measures; j++){
    S_n += y[j];
    Error += pow(y[j],2);

}
S_n = S_n/measures;
double variance = Error/measures -pow(S_n, 2);

sim_res.push_back(S_n);
sim_var.push_back(variance);
}

mc_est = sim_res;
mc_var = sim_var;

return;
}

```

La classe `ran` è stata illustrata in precedenza. Di seguito vengono illustrati i metodi utilizzati nel codice precedente. Il metodo `generate` genera numeri casuali in base alla dimensione dell'esperimento. I valori vengono quindi salvati in un vettore attributo della classe

```

void ran::generate(){

//Creo generatore TRandom3. Seed da attributo classe.
TRandom3* gen = new TRandom3(seed);

```

```

//Ciclo su size, attributo della classe.
for(int i = 0; i < size; i++){
    //Push nell'array attributo della classe
    //il numero casuale uniforme.
    arr.push_back(gen->Uniform(0, 1));
}

return; //Void.
}

```

Il metodo esterno prima utilizzato in `run_simulation()` è `compute`. Vengono descritte di seguito le operazioni svolte da questo metodo. Si cicla sulla dimensione dell'array delle ascisse e per ognuno dei valori, si appende all'array finale delle ordinate la funzione valutata nel punto (funzione esponenziale). Viene restituito quindi il vettore delle ordinate e viene salvato in un attributo privato della classe.

```

std::vector<double> func::compute(std::vector<double> arr){

    //Numero di elementi.
    int size = arr.size();
    //Istanzia array finale.
    std::vector<double> final_arr;
    //Ciclo su tutti gli elementi.
    for(int i = 0; i < size; i++){
        //Push nel vettore dell'esponenziale
        //valutata nell'elemento.
        final_arr.push_back(exp(arr[i]));
    }

    y = final_arr; //Salvo nell'attributo della classe.

    return final_arr;
}

```

Viene chiamato dal main il metodo `plot_estimators` e il metodo `plot_variance`. Essi sono metodi generici della classe `monte_carlo` che plottano la distribuzione degli estimatori dell'integrale e della varianza su un canvas. Essi prendono semplicemente l'attributo privato relativo alla variabile di interesse e fillano un oggetto `TH1F` di ROOT. Viene quindi creata una gaussiana per fissare i dati in quanto sappiamo che le distribuzioni di tali estimatori tendono alla normalità per CLT. Il tutto viene stampato e salvato in un'immagine.

Bisogna prestare attenzione che i vettori `mc_est` e `mc_var` sono aggiornati ogni qualvolta si ripete una computazione. I metodi di plot vanno quindi chiamati appena dopo l'operazione, quando la classe ha ancora in memoria il vettore computato e non è stato sovrascritto da un'altra operazione.

Dal main:

```
mc.plot_estimators("Crude MC estimators distribution", "./mc_est_crude.png");
mc.plot_variance("Crude MC variance distribution", "./mc_var_crude.png");
```

La definizione delle funzioni è la seguente. Riportiamo solo il metodo `plot_estimators` in quanto il secondo è identico al primo, solo che al posto di prendere i valori dall'attributo vettore `mc_est`, prende valori dal vettore attributo `mc_var` della classe.

```
void monte_carlo::plot_estimators(const char* title, const char* output){

    int bins = int(sqrt(events));
    auto min = *std::min_element(mc_est.begin(), mc_est.end());
    auto max = *std::max_element(mc_est.begin(), mc_est.end());

    TH1F* h = new TH1F("Mc integral estimator", "Mc integral estimator",
    bins, min, max);
    h->SetTitle(title);
    h->GetXaxis()->SetTitle("MC Estimator");
    h->GetYaxis()->SetTitle("Events");
    h->SetLineWidth(2);
    TCanvas*c = new TCanvas("c", "c", 1000,1000,1000,800);

    for(int i=0; i< events; i++){
        h->Fill(mc_est[i]);
    }

    TF1* fit = new TF1("fit", "gaus", min, max);
    gStyle->SetOptStat(1111);
    gStyle->SetOptFit(1111);
    h->Fit(fit);
    h->Draw("hist");
    fit->Draw("same");
    c->Draw();
    c->SaveAs(output, "png");

    return;
}
```

Il risultato è riportato nell'immagine seguente:

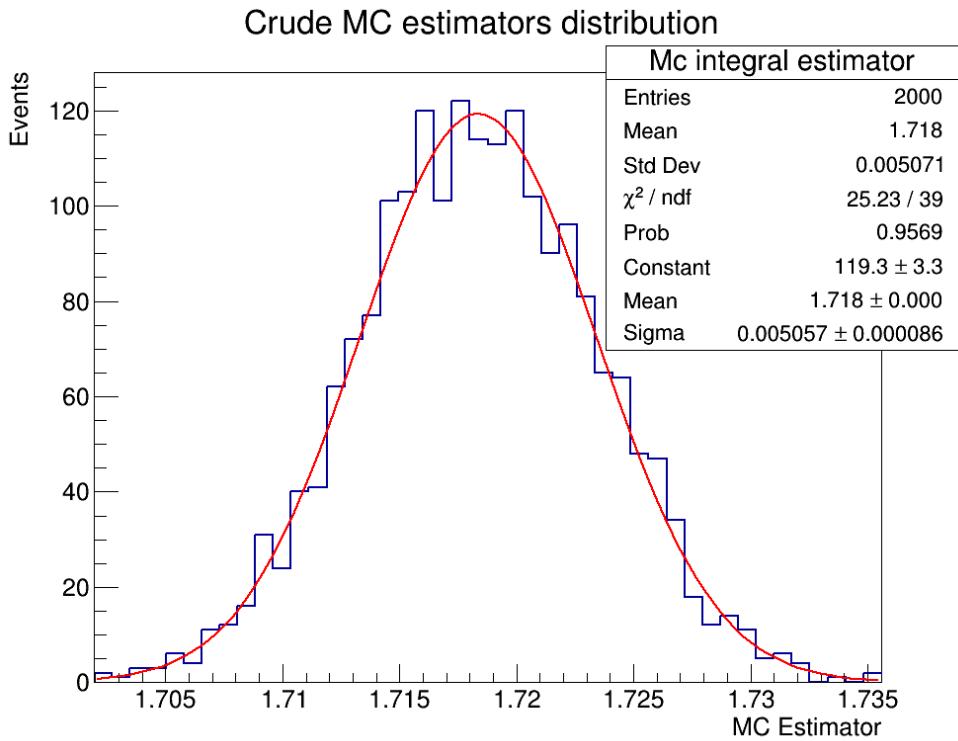


Figura 5.1: Istogramma di 2000 computazioni dell'integrale ognuna eseguita con un sampling di 10000 punti.

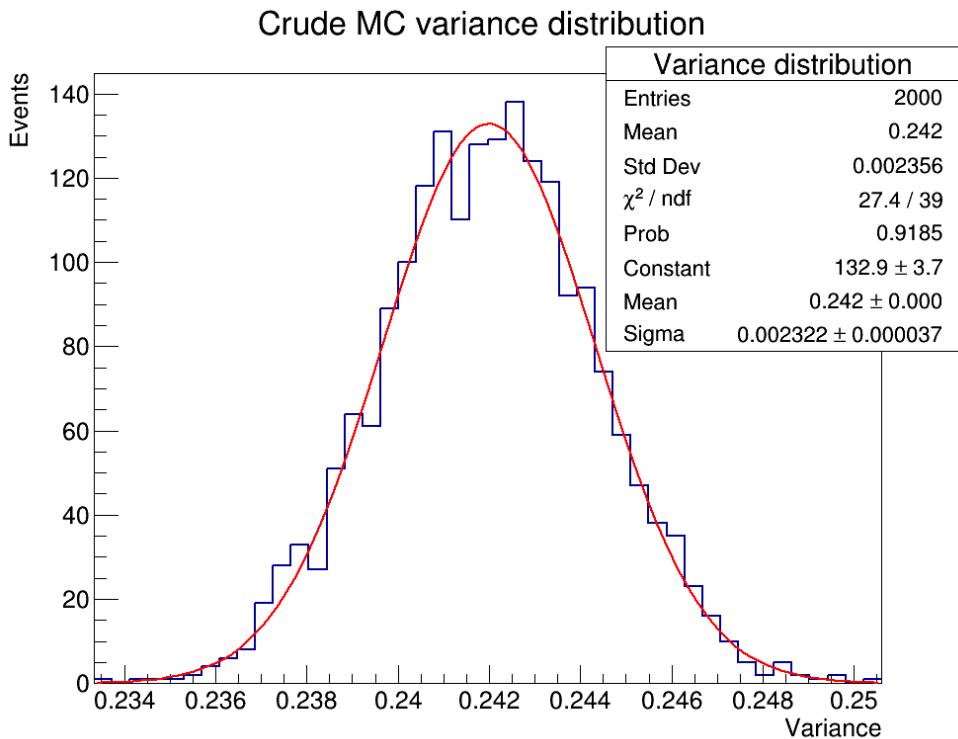


Figura 5.2: Istogramma di 2000 computazioni della varianza dell'estimatore integrale ognuna eseguita con un sampling di 10000 punti.

Al fine di ridurre errori statistici sia sulla computazione dell'integrale che alla computazione della varianza, si prende come miglior stima dell'integrale dal metodo Crude Montecarlo la media delle gaussiane e come loro errore l'errore standard della media ottenuto dividendo la varianza delle distribuzioni per la radice del numero di eventi.

I risultati sono quindi:

$$\hat{I}_{crude} = 1.71835 \pm 0.00005$$

$$\hat{\sigma}_{crude}^2 = 0.241914 \pm 0.000023$$

E' comunque utile un check sulle proprietà dell'estimatore singolo. L'andamento predetto teoricamente è rispettato come si può vedere dai seguenti grafici per l'andamento dell'estimatore, che converge verso il valore vero dell'integrale:

$$\hat{I} - I_{true} \quad (N \rightarrow \infty) \quad 0$$

Mentre la varianza dell'estimatore decresce all'aumentare della grandezza del sample:

$$\hat{\sigma}_I \propto \frac{1}{\sqrt{N}}$$

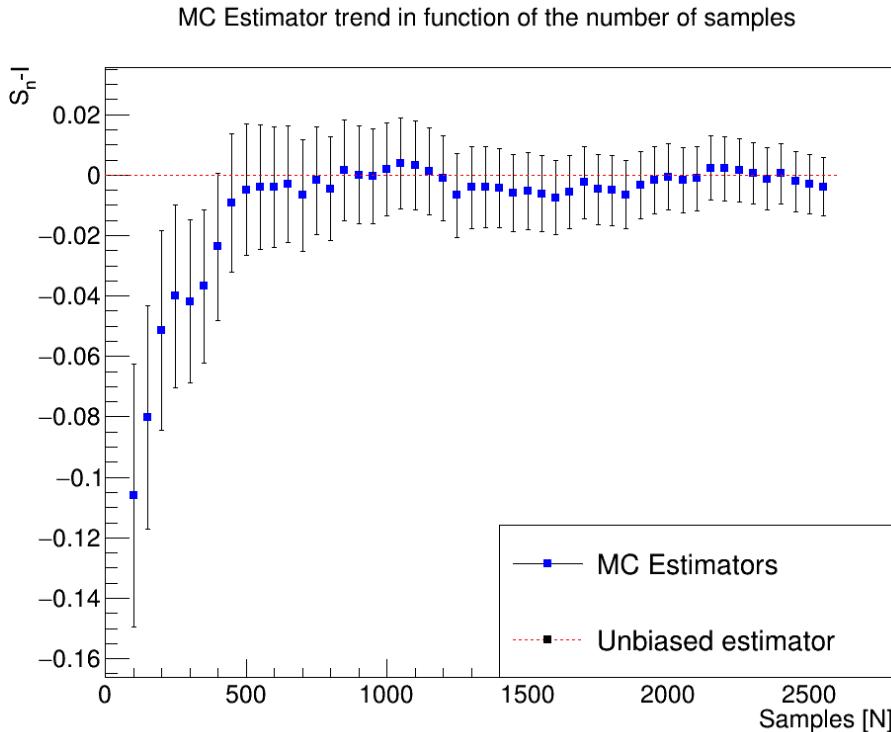


Figura 5.3: Trend del singolo estimatore al variare del numero di campionamenti della funzione da valutare. Viene plottata la differenza con il valore vero  $e - 1$ .

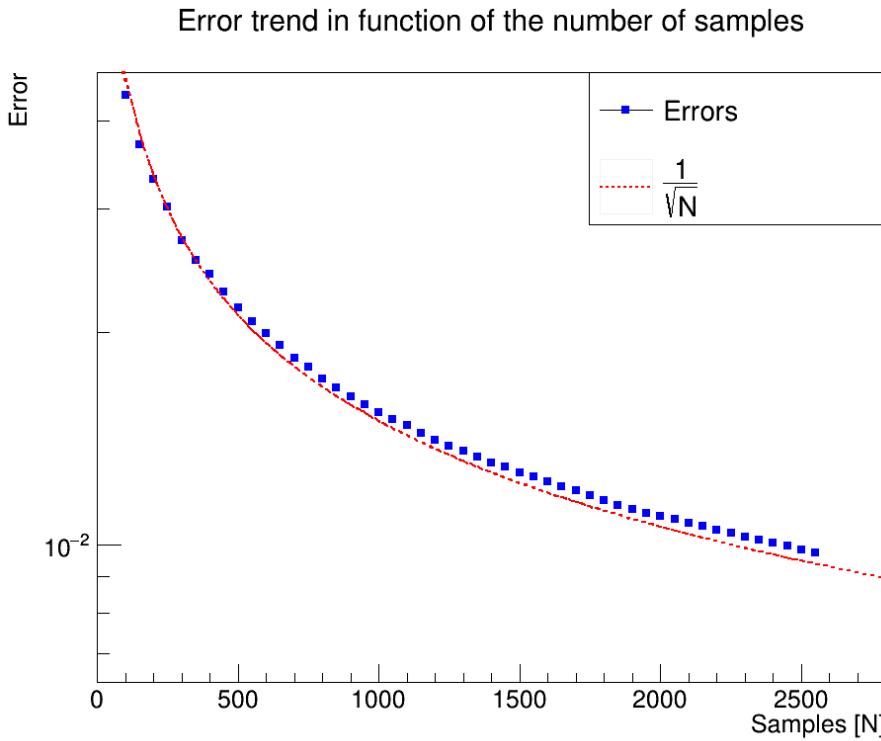


Figura 5.4: Trend dell’errore sul singolo estimatore al variare della grandezza del sample, segue l’inverso della radice del numero di campionamenti. La scala è logaritmica sulle ordinate.

## 5.3 Tecniche di riduzione della varianza

Il metodo Crude Monte Carlo è il metodo più banale e rapido per il calcolo degli integrali. Tecniche di riduzione della varianza possono essere implementate con il vantaggio di rendere la misura più precisa e, a volte, di ridurre la dimensione dei numeri pseudo-random creati, diminuendo così il costo computazionale del calcolo.

### 5.3.1 Stratified Sampling

A volte la funzione da integrare varia differentemente a seconda degli intervalli nel dominio di integrazione. In tale caso può essere utile stratificare il campionamento dei numeri casuali, dividendo il dominio di integrazione in vari sottointervalli e eseguire l’integrazione monte carlo su ognuno di essi.

Più in particolare sia  $X$  variabile casuale. Sia  $\theta = E[X]$  aspettativa della variabile casuale. Nei crude monte carlo l’estimatore del valore d’aspettazione è  $\hat{\theta} = \bar{X}$ .

Si supponga ora che il sampling della variabile casuale  $X$  dipenda da una qualche variabile discreta  $Y$  secondo la probability mass function  $P\{Y|y_i\} = p_i$  per  $i = 1\dots k$  allora possiamo scrivere:

$$E[X] = \sum_{i=1}^k p_i E[X|Y] = \sum_{i=1}^k p_i \left[ \frac{1}{n_i} \sum_{j=1}^{n_i} X_{ij} \right] = \sum_{i=1}^k p_i \bar{X}_i = \hat{\Theta}$$

Quindi  $\Theta$  è stimatore non biassato di  $\theta$ . Gli  $y_i$  suddividono lo spazio di campionamento in  $k$  partizioni del supporto di  $X$  disgiunte dette strati.

Si può dimostrare che la varianza dell'estimatore di  $\theta$  in questo modo può essere significativamente ridotta rispetto a una computazione crude monte carlo:

$$Var[\Theta] = Var[\bar{X}] - \frac{1}{n} Var[E[X|Y]]$$

Se  $S_i^2/(np_i)$  è la varianza campionaria si  $\bar{X}_i$  con  $np_i$  campioni allora:

$$Var[\Theta] = \sum_{i=1}^k p_i^2 Var[\bar{X}_i] = \frac{1}{n} \sum_{i=1}^k p_i S_i^2$$

Nel concreto dividiamo il nostro intervallo di campionamento dei numeri pseudo casuali in  $k$  strati. Se gli strati sono supposti equispaziati allora la probabilità di campionamento per ogni strato sarà  $p_i = \frac{1}{k}$ . Ogni strato è caratterizzato da  $[\frac{i-1}{k}, \frac{i}{k}]$  per  $i = 1 \dots k$  e la procedura per estratte valori in ogni intervallo è la seguente. Si genera un numero causale in  $[0,1]$  e si shifta la generazione rispetto all'intervallo  $i$  in questione:

$$x_{ij} = \frac{i-1}{k} + \frac{1}{k} U[0, 1] \quad j = 1 \dots n_i$$

Quindi per ogni intervallo  $i$  si computa la media e la varianza:

$$\bar{X}_i = \frac{1}{n_i} \sum_{j=0}^{n_i} X_{ij}$$

$$\frac{S_i^2}{np_i} = \frac{1}{n_i} \sum_{j=0}^{n_i} (X_{ij} - \bar{X}_i)^2$$

Una volta computate tutte le medie e le varianze per i singoli intervalli si procede con le formule prima riportate.

Il metodo della classe `Monte_carlo` chiamato `run_stratified_simulation` permette di eseguire tale tecnica di riduzione della varianza solo per probability mass function uniformi. Questo vuol dire che passeremo alla funzione in numero  $k$  di intervalli e ognuno avrà  $p_i = \frac{1}{k}$ . Il numero di elementi per intervallo viene quindi calcolato come  $n_i = \frac{n}{k} = p_i n$ . Si cicla quindi sul numero di esperimenti e vengono istanziati i vettori che conterranno le medie e le varianze computate sui singoli intervalli. Viene quindi fatto un ciclo sugli intervalli, vengono generati dei numeri uniformi nell'intervallo  $[0,1]$  e quindi viene applicato lo shifting per renderli uniformi nell'intervallo  $[\frac{i-1}{k}, \frac{i}{k}]$ . Viene istanziato un oggetto della classe `func` a cui verrà passato quest'ultimo vettore. Il metodo `compute` restituisce quindi le ordinate della funzione valutate nell'intervallo con i numeri generati pseudo-causalmente. Viene quindi fatta la media (che è un montecarlo sull'intervallo) grazie al metodo `mc_integral` e il risultato viene pushato nel vettore delle medie degli intervalli. Viene calcolata la varianza dell'intervallo e anch'essa viene inserita nel rispettivo vettore.

Finiti gli intervalli, non ci resta che applicare le formule totali per ottenere l'estimatore  $\Theta$  e la sua varianza. Si sommano quindi le medie dei vari intervalli e si

divide per il numero di intervalli. Analogamente si fa con le varianze che verranno divise dal quadrato del numero di intervalli.

I risultati ottenuti vengono quindi inseriti nel vettore che contiene i risultati degli esperimenti. Una volta finiti gli esperimenti, quindi il ciclo più esterno, i vettori vengono salvati negli attributi della classe.

Il codice appena descritto è illustrato di seguito:

```
void monte_carlo::run_stratified_simulation(double k){

    //inizializzo seed e vettore finale.
    srand(time(NULL));
    std::vector<double> sim_res;
    std::vector<double> sim_var;

    //Pmf uniforme
    double p = 1./k;
    //Numero di elementi per ogni intervallo
    int elem = int(p*measures);

    //ciclo sul numero di eventi per generare la simulazione.
    for(int i=0; i<events; i++){

        std::vector<double> var;
        std::vector<double> means;

        //ciclo sugli intervalli
        for(int r = 1; r < k+1; r++){
            int s = rand();
            ran evento(elem, rand());
            evento.generate();

            //numeri casuali tra [0,1]
            std::vector<double> myarr = evento.get_arr();

            //numeri casuali nell'intervallo
            std::vector<double> interval_sampling;
            for(int h = 0; h < elem; h++){
                interval_sampling.push_back((r-1)/k + myarr[h]/k);
            }
            func expo(interval_sampling);
            std::vector<double> y = expo.compute(interval_sampling);

            //media nell'intervallo
            double m = expo.mc_integral();
            means.push_back(m);

            //varianza
        }
    }
}
```

```

        double v = 0;
        for(int h = 0; h < elem; h++){
            v += pow(y[h]-m, 2);
        }
        var.push_back(v/elem);
    }

//calcolo medie e varianze globali su tutti gli intervalli
double tot_means = 0;
for(int h = 0; h < means.size(); h++){
    tot_means += means[h];
}
tot_means*=p;

double tot_var = 0;
for(int h = 0; h < var.size(); h++){
    tot_var += var[h];
}
tot_var*=pow(p,2);

sim_res.push_back(tot_means);
sim_var.push_back(tot_var);
std::cout << "Int: " << tot_means << " " << "Var: " << tot_var <<
std::endl;

}

mc_est = sim_res;
mc_var = sim_var;

return;
};


```

Chiamiamo quindi il metodo dal `main` specificando 5 intervalli di stratificazione ovvero  $\{0, 0.2, 0.4, 0.6, 0.8, 1\}$ :

```
mc.run_stratified_simulation(5); //Stratified sampling.
```

I risultati degli esperimenti vengono riportati nelle distribuzioni seguenti:

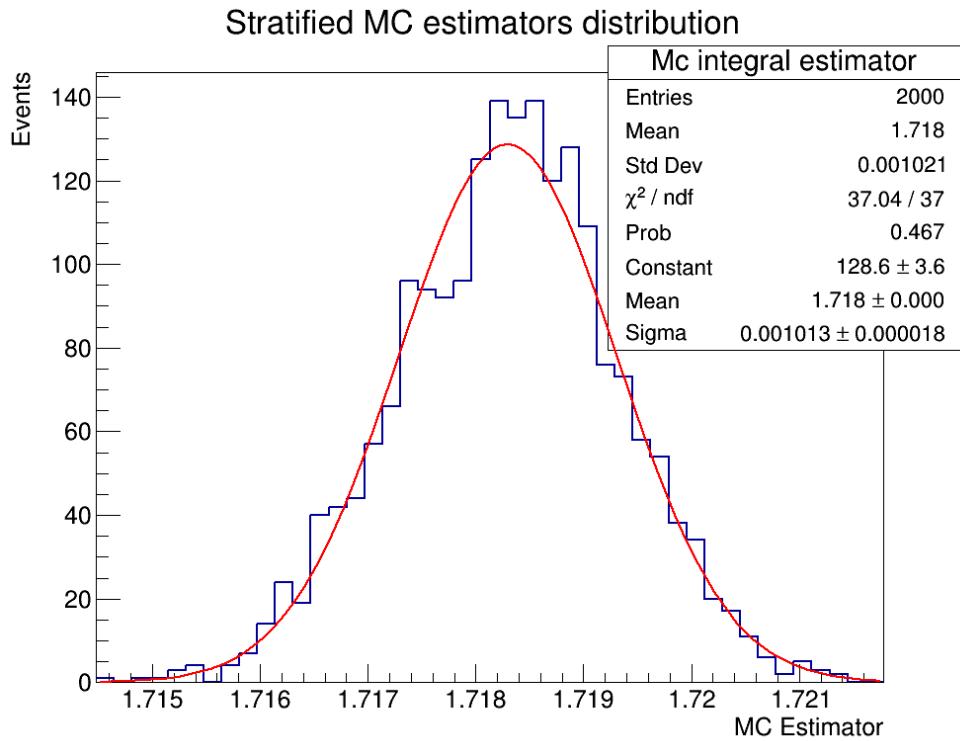


Figura 5.5: Distribuzione dell'estimatore  $\Theta$  dell'integrale definito con metodo di riduzione della varianza stratified sampling a 5 strati.

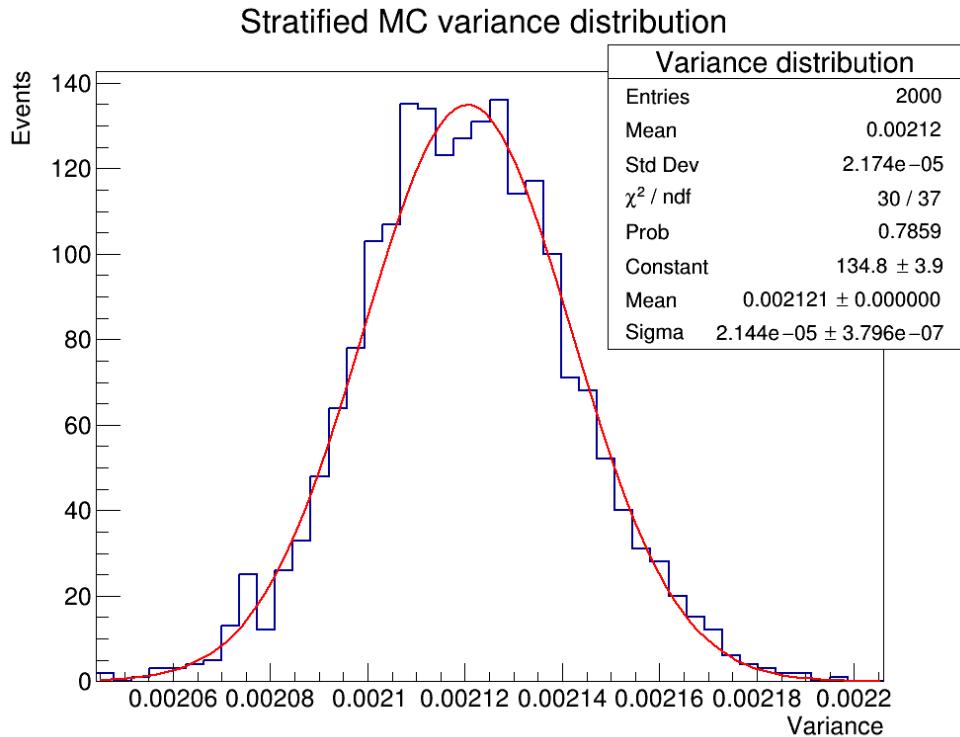


Figura 5.6: Distribuzione della varianza dell'estimatore  $Var[\Theta]$  dell'integrale definito con metodo di riduzione della varianza stratified sampling a 5 strati.

Riportiamo i risultati come medie delle gaussiane corrispondenti:

$$\hat{I}_{SS} = 1.71829913 \pm 0.00001013$$

$$\hat{\sigma}_{SS}^2 = 0.00212058 \pm 0.00000021$$

Può essere utile comprendere come la varianza dell'estimatore e l'estimatore stesso varino all'aumentare delle stratificazioni.

Per condurre una tale analisi imponiamo il numero di esperimenti a uno, così da avere un'unica stima monte carlo e la sua varianza. Si cicla quindi su un numero a piacere di stratificazioni che vengono incrementate e si salvano i risultati:

```
int k = 1;

std::vector<double> all_int;
std::vector<double> all_var;
std::vector<double> all_errors;
std::vector<double> all_k;

for(int m = 0; m < 20; m++){
    std::cout << "Running analysis" << std::endl;
    monte_carlo mc_strat(1, 10000);
    mc_strat.run_stratified_simulation(k);
    std::vector<double> inte = mc_strat.get_est();
    std::vector<double> var = mc_strat.get_var();

    all_int.push_back(inte[0]);
    all_errors.push_back(var[0]/sqrt(10000));
    all_var.push_back(var[0]);
    all_k.push_back(k);

    k++;
}

}
```

Il risultato del trend degli estimatori viene plottato assieme al suo errore in un **TGraphErrors** di ROOT. Il trend delle varianze viene invece plottato in un **TGraph**. I risultati sono riportati di seguito:

Trend of  $\Theta$  stratified estimator varying  $k$

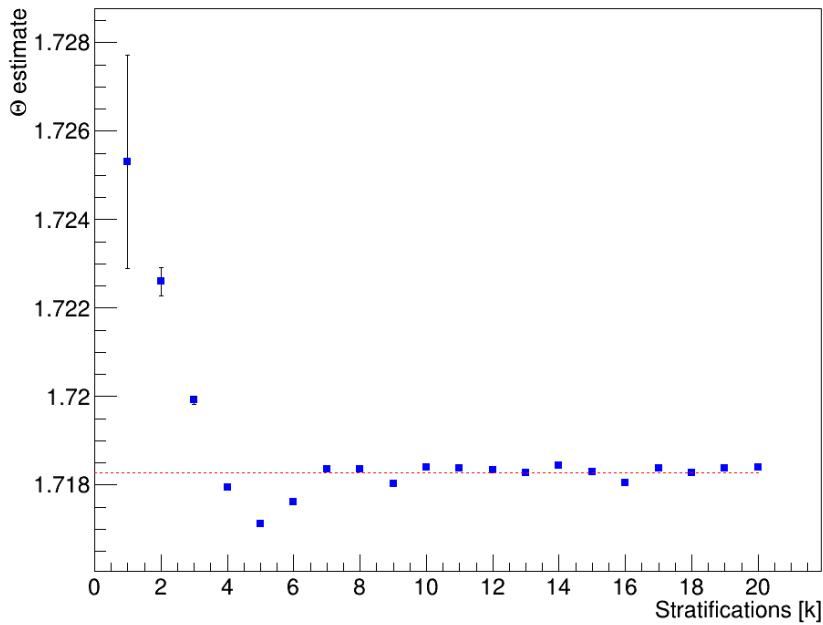


Figura 5.7: Trend dell'estimatore  $\Theta$  per un singolo esperimento monte carlo al variare del numero di stratificazioni  $k$ . La linea orizzontale rossa è il valore vero dell'integrale  $I = e - 1$ .

Trend of the variance of  $\Theta$  stratified estimator

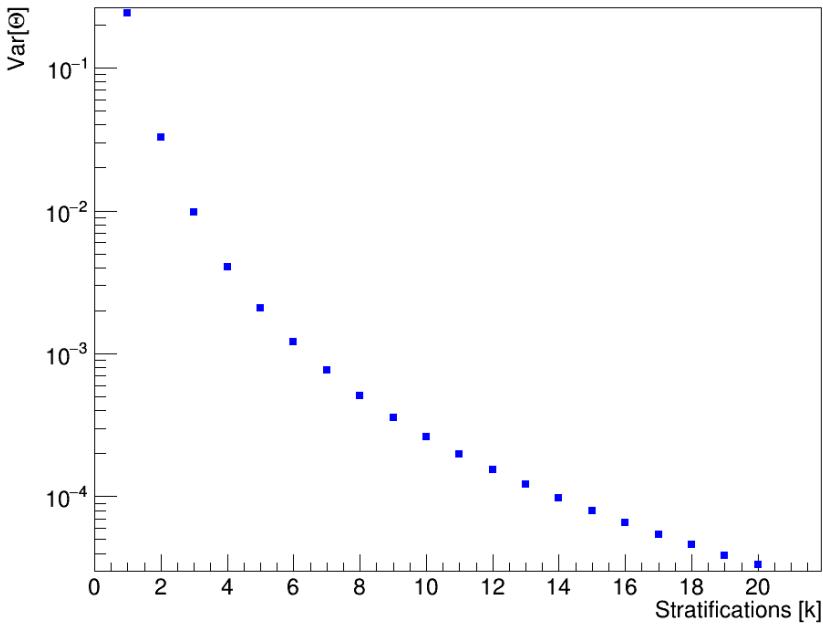


Figura 5.8: Trend della varianza dell'estimatore  $Var[\Theta]$  per un singolo esperimento monte carlo al variare del numero di stratificazioni  $k$ .

Si può notare come il risultato per  $k = 1$  corrisponda al risultato del crude

monte carlo. Il trend della varianza è plottato il scala logaritmica ed è evidente la diminuzione di  $Var[\Theta]$  e in contemporanea una misura più precisa di  $I$

### 5.3.2 Importance sampling

Scegliendo un distribuzione da cui samplare i dati tale che la densità di punti samplati è simile alla shape della funzione integranda allora la varianza dell'integrale è ridotta.

In sostanza la distribuzione uniforme da cui sampliamo i dati viene rimpizatta da una distribuzione detta di importanza:

$$I = \int_0^1 f(x)dx = \int_0^1 \frac{f(x)}{p(x)}p(x)dx = E_p\left[\frac{f(x)}{p(x)}\right]$$

Se la distribuzione  $p(x)$  è normalizzata nell'intervallo di integrazione:  $\int_0^1 p(x)dx = 1$  allora possiamo samplare punti da tale distribuzione e, finalmente:

$$E[I] = S_n = \frac{1}{N} \sum_{i=0}^n \frac{f(x_i)}{p(x_i)} \quad x_i \sim p(x)$$

$$\hat{\sigma}_I^2 = \frac{1}{N} \sum_{i=0}^n \left( \frac{f(x_i)}{p(x_i)} \right)^2 - S_n^2$$

La distribuzione di importanza non deve essere positiva ovunque. Basta che  $p(x) > 0 \quad if \quad f(x) \neq 0$ . Definiamo il dominio  $Q = \{x | p(x) > 0\}$  quindi sia  $D$  il dominio di definizione dell'integrale, dalla precedente sappiamo che se  $x \in D \cap Q^c$  allora  $f(x) = 0$ .

Questa affermazione porta inevitabilmente all'assunzione che non ci sono  $x \in Q | p(x) = 0$ . Perciò possiamo scrivere quanto segue:

$$E_p\left[\frac{f(x)}{p(x)}\right] = I$$

Il rapporto  $f(x)/p(x)$  è chiamato likelihood ratio. Se  $p$  è una distribuzione simile a  $f$  allora il rapporto sarà circa unitario e la varianza sarà ridotta.

Il Sampling dalla distribuzione di importanza deve tuttavia essere facile. E inoltre bisogna prestare attenzione alla scelta della distribuzione di importanza. Ad esempio supponendo che  $p(x)$  tenda a zero per un determinato valore mentre  $f(x)$  tenda ad un valore finito differente allora la varianza aumenterà invece che diminuire. Vedremo un caso di una erronea scelta di distribuzione di importanza.

La prima distribuzione di importanza è la seguente:

$$p(x) = 2.5x^{1.5}$$

Il vantaggio di questa distribuzione è che è normalizzata in  $[0, 1]$ :

$$\int_0^1 2.5x^{1.5}dx = x^{2.5}|_{x=0}^{x=1} = 1$$

E inoltre la sua distribuzione cumulativa è facilmente calcolabile analiticamente:

$$F(x) = \int_0^x 2.5x^{1.5} dx = x^{2.5}$$

Il problema è che  $p(x) \rightarrow 0$  per  $x \rightarrow 0$ . Questo è un problema quando calcoleremo la varianza.

Il plot delle funzioni è riportato nell'immagine seguente dove è chiaro l'andamento:

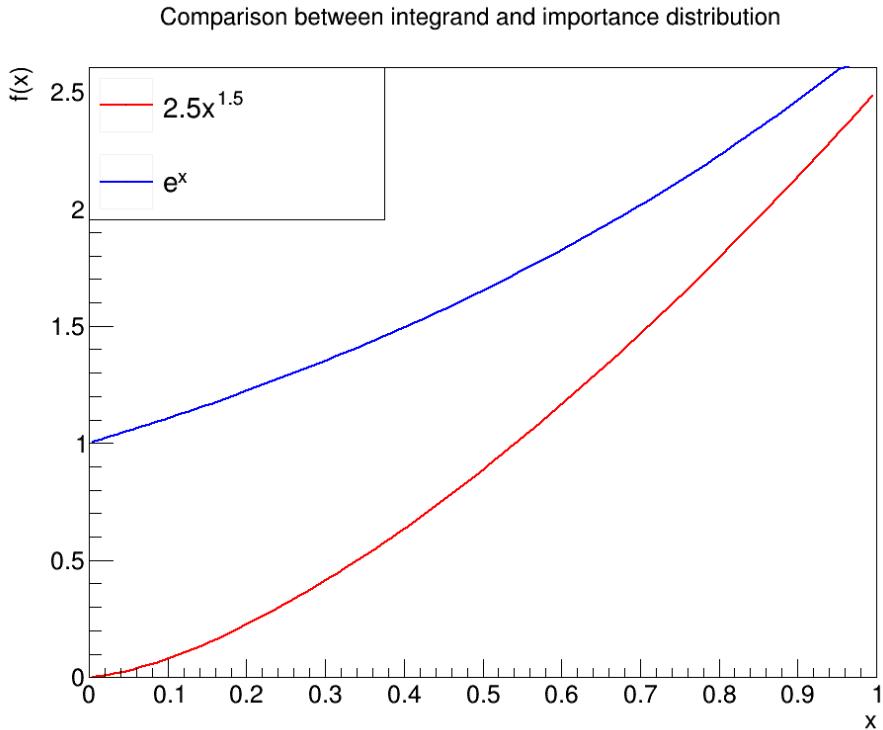


Figura 5.9: Plot della funzione integranda e della funzione  $2.5x^{1.5}$  del primo esempio. Le due funzioni hanno un andamento simile in prossimità di  $x = 1$  tuttavia esse si allontanano per  $x \rightarrow 0$ . Come vedremo il fatto che la funzione di importanza sia identicamente 0 quando  $x = 0$  creerà problemi.

Il sampling della distribuzione  $p(x) = 2.5x^{1.5}$  è eseguito tramite metodo della funzione inversa implementato nella classe `ran`.

Il metodo è chiamato `generate_par` e ha la seguente struttura:

$$y \sim U[0, 1] \quad , \quad x = F^{-1}(y) = y^{\frac{1}{2.5}}$$

```
void ran::generate_par(){

    TRandom3* gen = new TRandom3(seed);

    for(int i = 0; i < size; i++){
        //Numero casuale in [0,1].
        double y = gen->Uniform(0,1);
```

```

    //Metodo funzione inversa
    //secondo  $2.5 \cdot x^{1.5}$ .
    arr.push_back(pow(y, 1./2.5));
}

return; //Void.
}

```

La procedura per la computazione dell'integrale e della varianza è riportato seguendo la funzione `run_importance_par_simu` della classe `monte_carlo`. Il codice è simile al metodo per il crude monte carlo, la differenza importante è che il generatore di numeri pseudo-casuali, ora genererà numeri pseudo-casuali distribuiti seconda la distribuzione di importanza. A questo punto vengono istanziati due oggetti della classe `func`, uno che valuterà l'array appena creato secondo l'esponenziale e l'altro per valutare l'array secondo la distribuzione d'importanza. Il metodo della classe `func` per valutare la distribuzione di importanza è chiamato `compute_par` e ha la seguente struttura:

```

std::vector<double> func::compute_par(std::vector<double> arr){

    int size = arr.size();
    std::vector<double> final_arr;
    for(int i = 0; i < size; i++){
        //Push nel vettore della funzione valutata nell'elemento.
        final_arr.push_back(2.5 * pow(arr[i], 1.5));
    }

    y=final_arr; //Salvo nell'attributo della classe;
    return final_arr;
}

```

A questo punto vengono calcolati l'integrale come somma dei rapporti tra le valutazioni delle singole distribuzioni per ogni istanza dell'array di numeri pseudo-casuali. I risultati per ogni esperimento vengono salvati in un array.

La struttura del metodo `run_importance_par_simu` è la seguente:

```

void monte_carlo::run_importance_par_simu(){

    srand(time(NULL));
    std::vector<double> sim_res;
    std::vector<double> sim_var;

    for(int i=0; i<events; i++){

        //genero numeri pseudo-random secondo
        //la distribuzione di importanza.
        ran evento(measures, rand());
    }
}

```

```

    evento.generate_par();

    std::vector<double> para = evento.get_arr();

//istanzio oggetti funzioni passando array
//delle ascisse
func expo(para);
func parabola(para);

//compuo le ordinate delle due distribuzioni
std::vector<double> y1 = expo.compute(para);
std::vector<double> y2 = parabola.compute_par(para);

//computing Integral and variance:
double S_n = 0;
double Err = 0;
for(int j = 0; j < measures; j++){
    S_n+= y1[j]/y2[j];
    Err+= pow(y1[j]/y2[j], 2);
}

S_n /= measures;
double var = Err/measures - pow(S_n, 2);
sim_var.push_back(var);
sim_res.push_back(S_n);

}

mc_est = sim_res;
mc_var = sim_var;

return;
}

```

Il risultato è catastrofico, in quanto, come detto, il valore della varianza espplode quando campioniamo un  $x \sim 0$  in quanto la valutazione della distribuzione di importanza è prossima a 0. Il valore medio della varianza in questo caso è circa 16 con picchi fino a 6000 mentre il valor medio della stima dell'integrale è di 1.714 . E' stato illustrato il fallimento di questo metodo a scopo di capire a pieno l'operazione di importance sampling e la scelta accurata della distribuzione.

Una correzione banale segue il ragionamento che vogliamo evitare distribuzioni di importanza il cui valore si avvicini a zero.

La scelta è ricaduta sulla seguente distribuzione:

$$p(x) = (x + 1)^{1.5}$$

Il grafico è riportato di seguito analogamente a quanto fatto prima:

Comparison between integrand and importance distribution

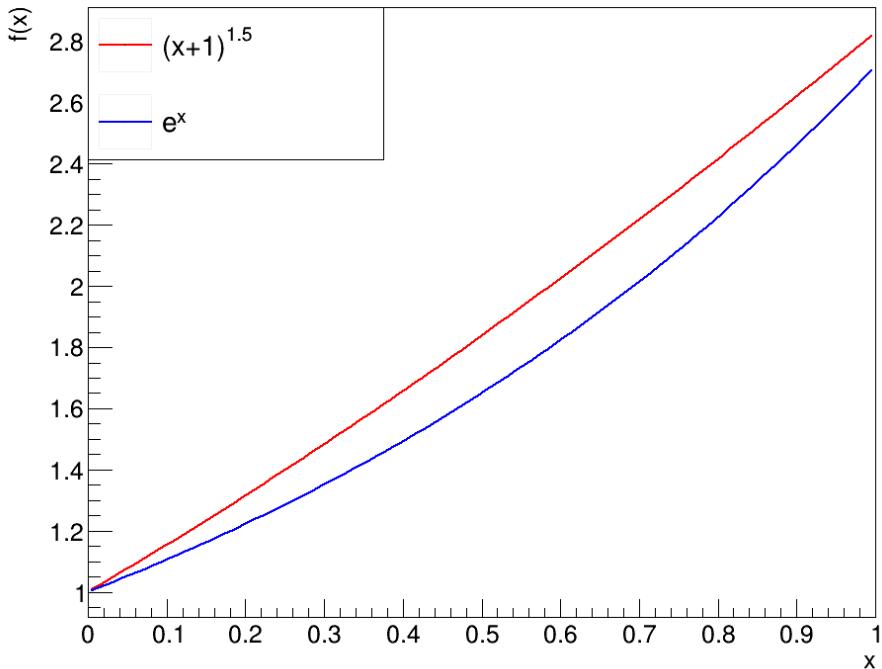


Figura 5.10: Plot della funzione integranda e della funzione  $(x + 1)^{1.5}$  del secondo esempio. Le due funzioni hanno andamento molto simile e, soprattutto, non abbiamo punti nel dominio in cui la distribuzione ha valore 0.

Il vantaggio di utilizzare questa distribuzione è che è molto simile all'integranda. Tuttavia essa non è normalizzata nel dominio di integrazione come la precedente:

$$\int_0^1 (x + 1)^{1.5} dx = \frac{1}{2.5} (x + 1)^{2.5} \Big|_{x=0}^{x=1} = \frac{2^{2.5} - 1}{2.5} \simeq 1.86274\dots$$

Questo non rappresenta un grosso problema se non per la precisione con cui possiamo calcolare l'integrale dalla formula analitica e per eventuali errori di precisione di calcolatore in quanto, per ottenere l'estimatore non biassato, introduciamo una moltiplicazione.

Il codice è in tutto e per tutto simile al precedente, un metodo della classe `ran` genera numeri secondo la distribuzione di importanza:

$$F(x) = \int_0^x (x + 1)^{1.5} dx = \frac{1}{2.5} (x + 1)^{2.5} - \frac{1}{2.5}$$

Si generano uniformemente  $y$  fra 0 e il valore dell'integrale nel dominio. Solo in questo modo avremo delle  $x$  distribuite tra 0 e 1.

$$y \sim [0, \frac{2^{2.5} - 1}{2.5}] \quad , \quad x = (2.5y + 1)^{\frac{1}{2.5}} - 1$$

Il codice del metodo `generate_par2` è riportato di seguito:

```

void ran::generate_par_2(){

    TRandom3* gen = new TRandom3(seed);

    for(int i =0; i < size; i++){
        //Numero casuale in [0,1.8627].
        double y = gen->Uniform(0, (pow(2, 2.5)-1)/2.5;
        //Metodo funzione inversa.
        arr.push_back(pow(2.5*y +1, 1./2.5)-1.);

    }

    return; //Void.
}

```

Un metodo della classe `func` permette di valutare questa seconda distribuzione dai valori estratti pseudo-random:

```

std::vector<double> func::compute_par_2(std::vector<double> arr){
    //importance (x+1)^1.5
    int size = arr.size();
    std::vector<double> final_arr;
    for(int i = 0; i < size; i++){
        //Push nel vettore della funzione valutata nell'elemento.
        final_arr.push_back(pow(arr[i]+1, 1.5));
    }
    y=final_arr; //Salvo nell'attributo della classe;
    return final_arr;
}

```

infine viene calcolato l'estimatore dell'integrale e la varianza attraverso il metodo `run_importanze_par_simu_2` della classe `monte_carlo`:

```

void monte_carlo::run_importance_par_simu_2(){

    srand(time(NULL));
    std::vector<double> sim_res;
    std::vector<double> sim_var;

    for(int i=0; i<events; i++){

        double integral_importance = (pow(2, 2.5)-1)/2.5;
        ran evento(measures, rand());
        evento.generate_par_2();

        std::vector<double> para = evento.get_arr();
    }
}

```

```

func expo(para);
func parabola(para);

std::vector<double> y1 = expo.compute(para);
std::vector<double> y2 = parabola.compute_par_2(para);

//computing inline integral.
double integral = 0;

for(int j = 0; j < measures; j++){
    integral += y1[j]/y2[j];

}

integral /= (measures);

//computing variance
double var = 0;
for(int j = 0 ; j < measures; j++){
    var += pow(y1[j]/y2[j]-integral, 2);
}
var /= measures;
integral*=integral_importance;

sim_res.push_back(integral);
sim_var.push_back(var);

}

mc_var = sim_var;
mc_est = sim_res;

return;
}

```

I risultati sono i seguenti sia per l'estimatore dell'integrale che per la distribuzione delle varianze dai vari esperimenti:

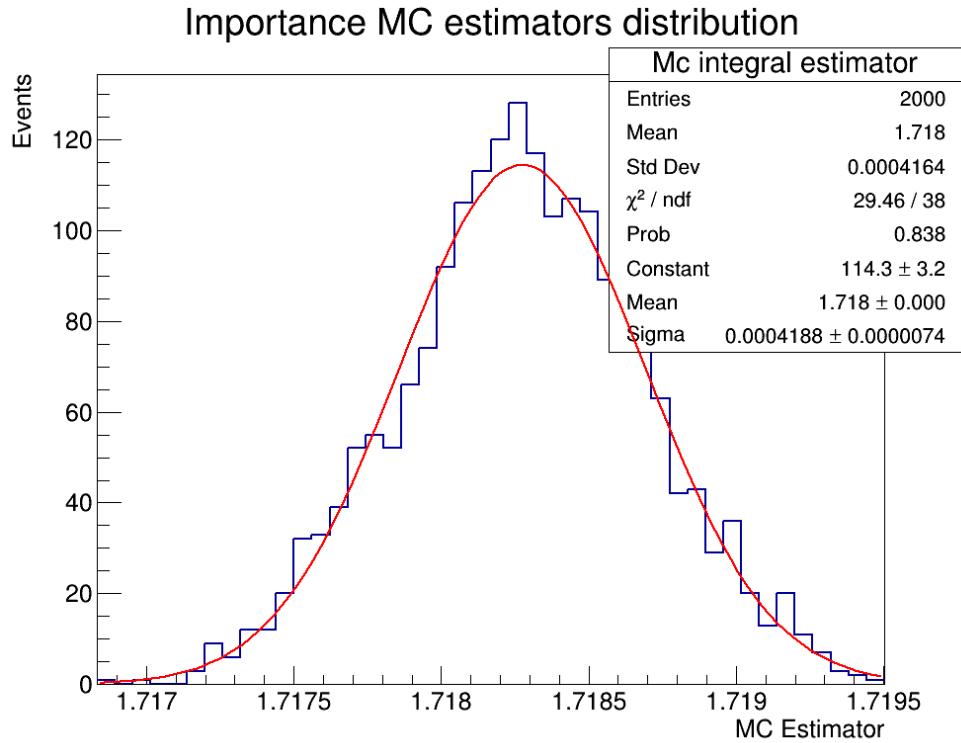


Figura 5.11: Distribuzione degli estimatori dell'integrale per il metodo importance sampling con distribuzione d'importanza  $p(x) = (x + 1)^{1.5}$

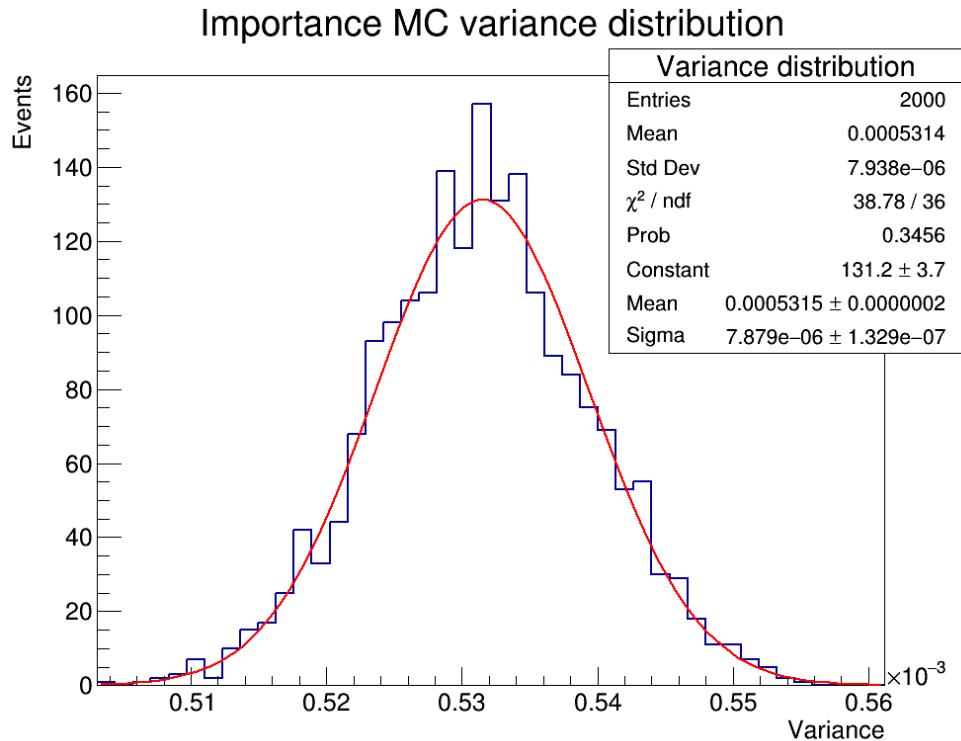


Figura 5.12: Distribuzione degli estimatori delle varianze per il metodo importance sampling con distribuzione d'importanza  $p(x) = (x + 1)^{1.5}$

Possiamo quindi riassumere i risultati:

$$\hat{I}_{IS} = 1.71828458 \pm 0.00000461$$

$$\hat{\sigma}_{IS}^2 = 0.00053130 \pm 0.00000008$$

Vediamo che la varianza dell'estimatore si è ridotta di circa 3 ordini di grandezza rispetto alla varianza effettuata con un crude monte carlo.

### 5.3.3 Antithetic Variates

Il metodo delle variabili antitetiche è un metodo di riduzione della varianza dell'estimatore monte carlo e allo stesso tempo permette di ridurre la dimensione del sample di numeri pseudo-casuali da generare. Si basa sul fatto che una correlazione negativa tra due variabili casuali riduce la varianza totale dell'estimatore.

Per ogni numero generato pseudo-casuale  $X_1 = \{x_1 \dots x_m\}$  si prende la sua antitesi  $X_2 = \{-x_1 \dots -x_m\}$  così da avere un sample di  $2m = N$  numeri.

Siano ora:

$$\hat{\theta}_1 = \frac{f(X_1)}{m}, \quad \hat{\theta}_2 = \frac{f(X_2)}{m}$$

Allora l'estimatore non biassato sarà dato dalla media degli estimatori sui due campioni:

$$\hat{\theta} = \frac{\hat{\theta}_1 + \hat{\theta}_2}{2}$$

La varianza dell'estimatore sarà quindi:

$$\hat{Var}[\theta] = \frac{Var[\hat{f}(X_1)] + Var[\hat{f}(X_2)] + 2Cov[\hat{f}(X_1), \hat{f}(X_2)]}{4}$$

La varianza dell'estimatore sarà quindi ridotta se la covarianza è negativa.

Ad esempio sia  $X_1 = U[0, 1]$  un sample di grandezza  $m$  di variabili casuali distribuite uniformemente nell'intervallo  $[0, 1]$ . Si prenda ora  $X_2 = 1 - X_1$ , anche questo sarà un sample di variabili casuali distribuite uniformemente nell'intervallo  $[0, 1]$  ma in questo modo:

$$Cov[X_1, X_2] = -1$$

La covarianza dei due sample è massima.

In generale la covarianza non sarà mai al suo valore limite ma ci basta che sia negativa per ottenere una riduzione della varianza.

Nel nostro caso lo schema di computazione è il seguente:

$$X_1 = U[0, 1], \quad X_2 = 1 - X_1$$

$$y_1 = e^{X_1}, \quad y_2 = e^{X_2}$$

$$S_n = \frac{1}{m} \sum_{i=0}^m \frac{y_1(i) + y_2(i)}{2}$$

$$Var[S_n] = \frac{1}{m} \sum_{i=0}^m \left( \frac{y_1(i) + y_2(i)}{2} \right)^2 - S_n^2$$

Il codice è concettualmente analogo al crude monte carlo con l'unica differenza nel computare ogni operazione due volte, una per l'insieme generato uniformemente di grandezza  $m$  e una per l'insieme ad esso antitetico di egual grandezza così da costruire un sample totale della grandezza desiderata.  
il metodo `run_antithetic_simu` è riportato come segue:

```
void monte_carlo::run_antithetic_simu(){

    srand(time(NULL));
    std::vector<double> sim_res;
    std::vector<double> sim_var;

    for(int i = 0; i < events; i++){

        int anti_meas = int(measures/2);
        ran evento(anti_meas, rand());
        evento.generate();
        std::vector<double> myarr = evento.get_arr();
        std::vector<double> anti = myarr;
        std::for_each(anti.begin(), anti.end(), [] (double& d) { d=1.0-d;});

        func expo1(myarr);
        func expo2(anti);
        std::vector<double> y1 = expo1.compute(myarr);
        std::vector<double> y2 = expo2.compute(anti);

        //computo integrale e varianza e appendo alla lista finale

        double S_n = 0; //integrale
        double Error = 0; //varianza
        for(int j = 0; j < anti_meas; j++){
            S_n += (y1[j]+y2[j])/2;
            Error += pow((y1[j]+y2[j])/2,2);
        }

        S_n = S_n/anti_meas;
        double variance = Error/anti_meas -pow(S_n, 2);

        sim_res.push_back(S_n);
        sim_var.push_back(variance);

    }

    mc_est = sim_res;
    mc_var = sim_var;

    return;
}
```

}

Il plot delle distribuzioni degli estimatori e delle varianze nei vari esperimenti è riportato di seguito:

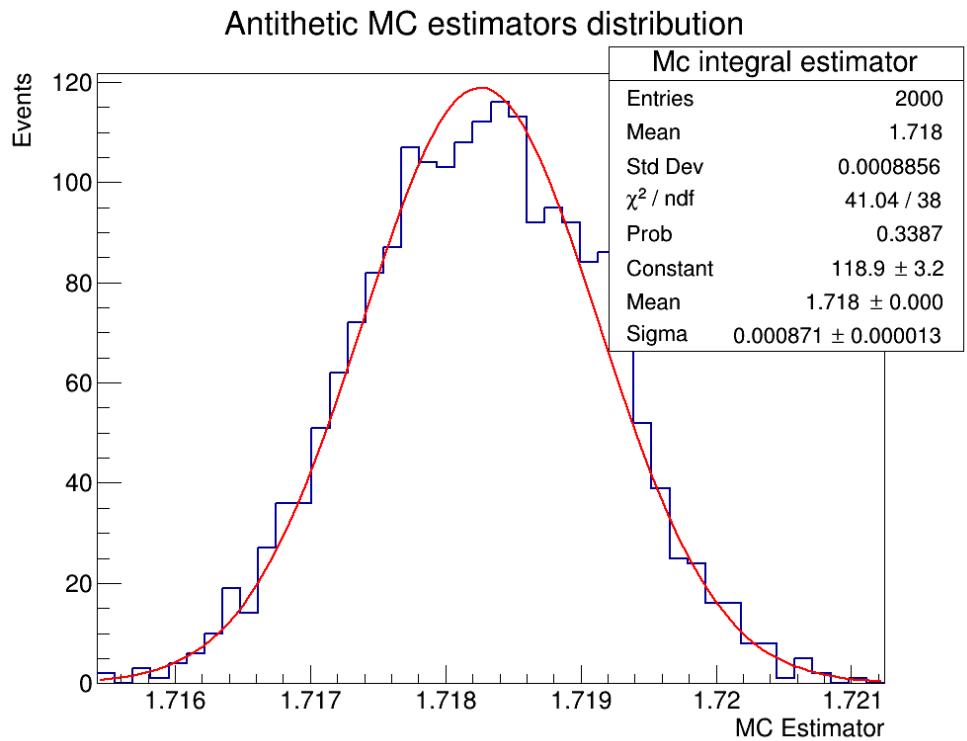


Figura 5.13: Distribuzione degli estimatori dell'integrale per il metodo antithetic variates.

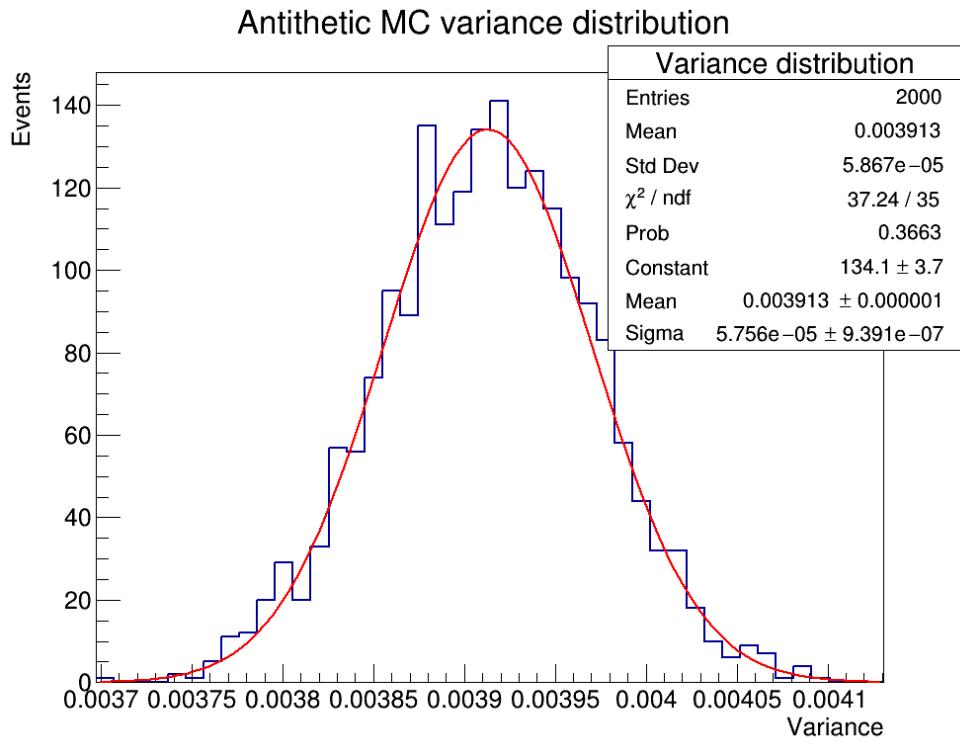


Figura 5.14: Distribuzione degli estimatori delle varianze per il metodo antithetic variates.

Possiamo quindi riportare i risultati:

$$\hat{I}_{AV} = 1.71826148 \pm 0.00000879$$

$$\hat{\sigma}_{AV}^2 = 0.00391309 \pm 0.00000058$$

## 5.4 Comparazione finale

Come risultato finale viene plottato il valore dell'estimatore media su 2000 esperimenti e l'associato errore come computato dalla media della gaussiana delle varianze diviso per la radice del numero di esperimenti:

### MC estimators for different variance techniques

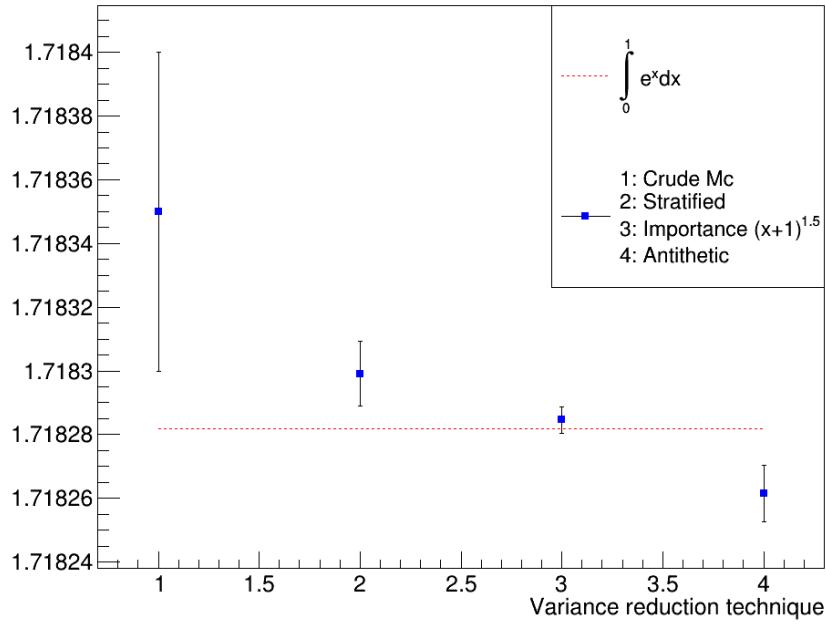


Figura 5.15: Comparazione degli estimatori medi dell'integrale e della varianza dell'integrale.

Chiaramente possiamo ridurre l'errore e quindi aumentare la precisione nella computazione dell'integrale, aumentando il numero di misure che, per questo esempio era fisso a 10000. D'altro canto possiamo aumentare a nostro piacimento il numero di intervalli nello stratified sampling quindi i risultati sono specifici al nostro esempio, non possiamo giudicare un metodo di riduzione della varianza migliore o peggiore.

Ciò che possiamo osservare in ogni caso è che ognuno dei tre metodi di riduzione della varianza porta a stime più precise dell'integrale e a una riduzione della varianza dello stesso.

## 5.5 Analisi con generatore xorshiro

Abbiamo prima citato che il generatore pseudo-casuale `TRandom3` ha dei problemi conosciuti in letteratura che esulano dallo scopo di questo scritto.

Siamo stati insospettiti dalla inaccuratezza del processo di generazione dal grafico dell'errore sull'estimatore integrale in figura (5.4). Vediamo che l'andamento è circa simile tuttavia significativamente non uguale a quello teorico.

Abbiamo imputato questo comportamento al generatore di ROOT. Si è quindi rieseguita l'analisi utilizzando l'algoritmo `xorshiro128p` che sappiamo essere affidabile. Per fare ciò linkiamo la libreria `xorshiro` e `prng` create per gli Esercizi 1 e 2. Sostituiamo ogni generatore `TRandom3` con uno `xorshiro` nel modo seguente:

```
//TRandom3* gen = new TRandom3();
xorshiro gen;
```

Utilizzeremo poi il metodo `rand(min, max)` dell'oggetto `xorshiro` per generare i numeri casuali.

Vengono ripetuti i metodi in modo identico a prima, riportiamo solo i risultati nell'ordine precedente:

### 5.5.1 Crude MC xorshiro

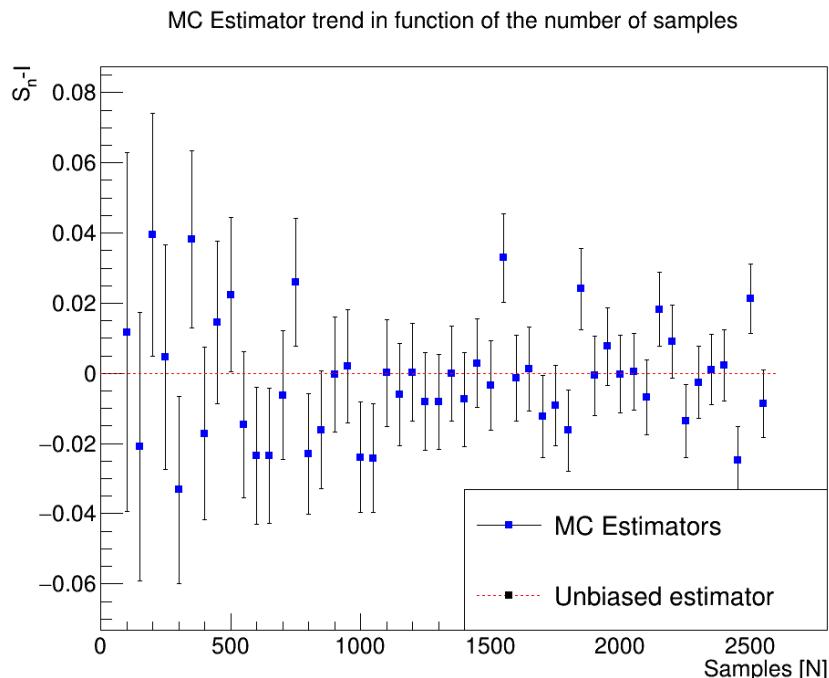


Figura 5.16: Trend corretto dal generatore `xorshiro128p` del singolo estimatore al variare del numero di campionamenti della funzione da valutare. Viene plottata la differenza con il valore vero  $e - 1$ .

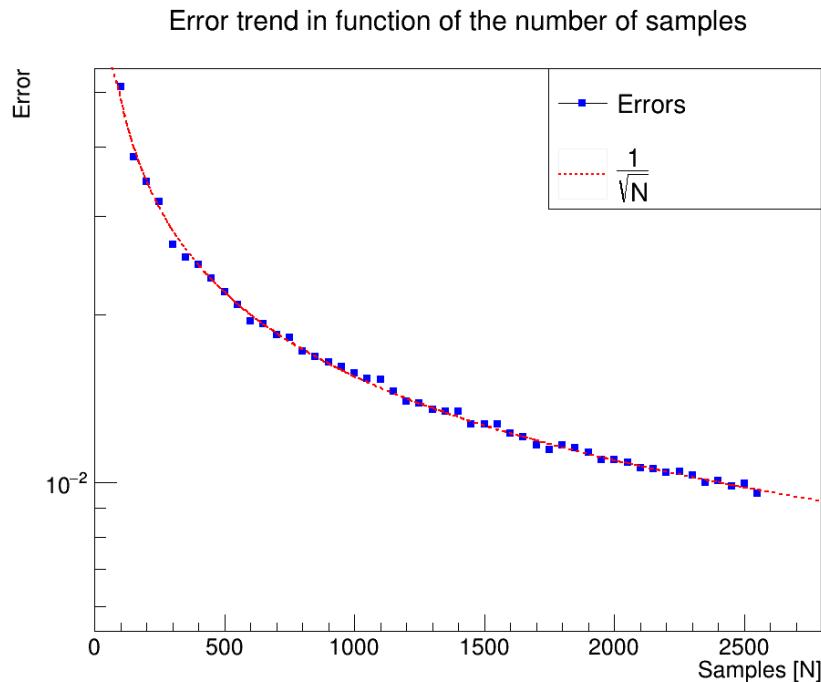


Figura 5.17: Trend dell'errore sul singolo estimatore ottenuto tramite la modifica dell'algoritmo di generazione xorshiro128p, al variare della grandezza del sample, segue l'inverso della radice del numero di campionamenti

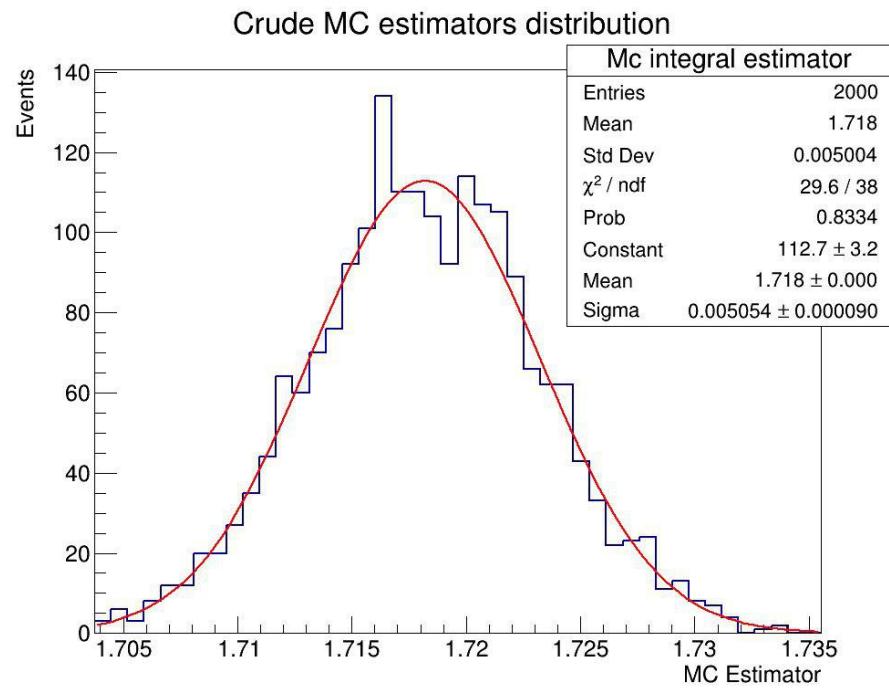


Figura 5.18: Istogramma di 2000 computazioni dell'integrale ognuna eseguita con un sampling di 10000 punti attraverso l'algoritmo xorshiro.

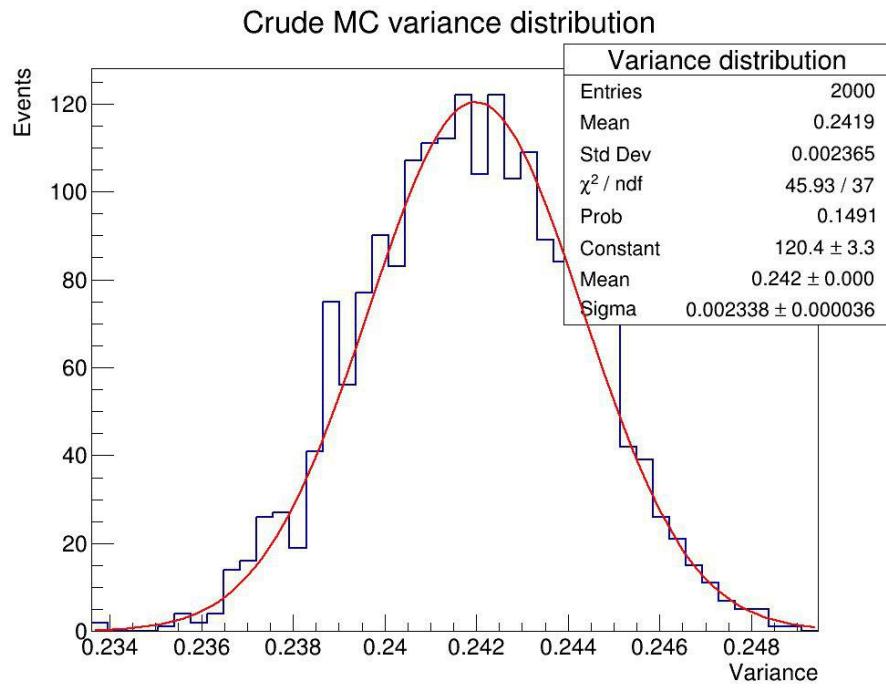


Figura 5.19: Istogramma di 2000 computazioni della varianza dell'estimatore integrale ognuna eseguita con un sampling di 10000 punti attraverso l'algoritmo xorshiro.

$$\hat{I}_{\text{crude\_xor}} = 1.71820764 \pm 0.00011599$$

$$\hat{\sigma}_{\text{crude\_xor}}^2 = 0.24198802 \pm 0.00002338$$

### 5.5.2 Stratified Sampling xorshiro

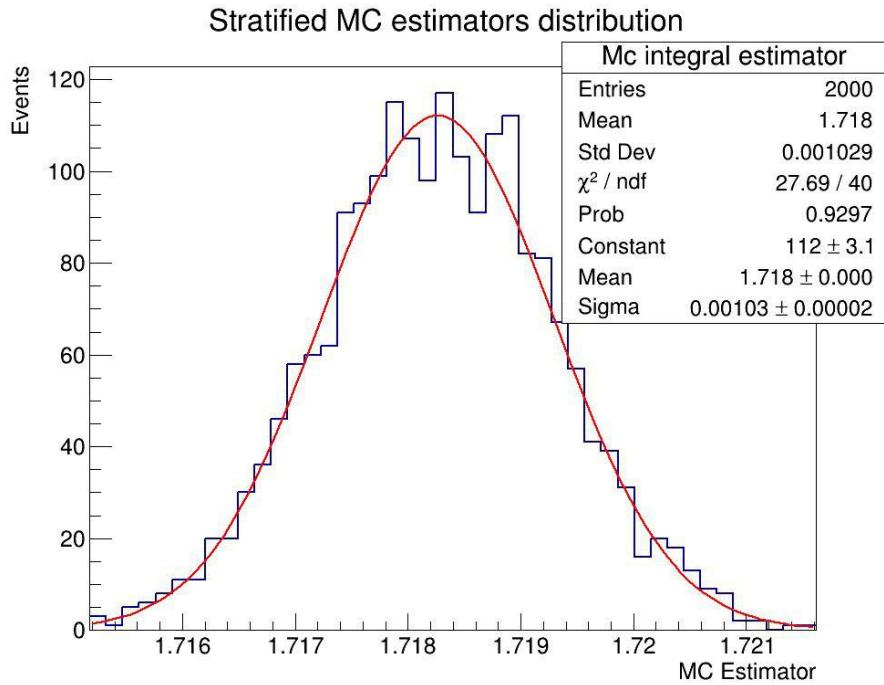


Figura 5.20: Distribuzione dell'estimatore  $\Theta$  dell'integrale definito con metodo diriduzione della varianza stratified sampling a 5 strati. Algoritmo xorshiro.

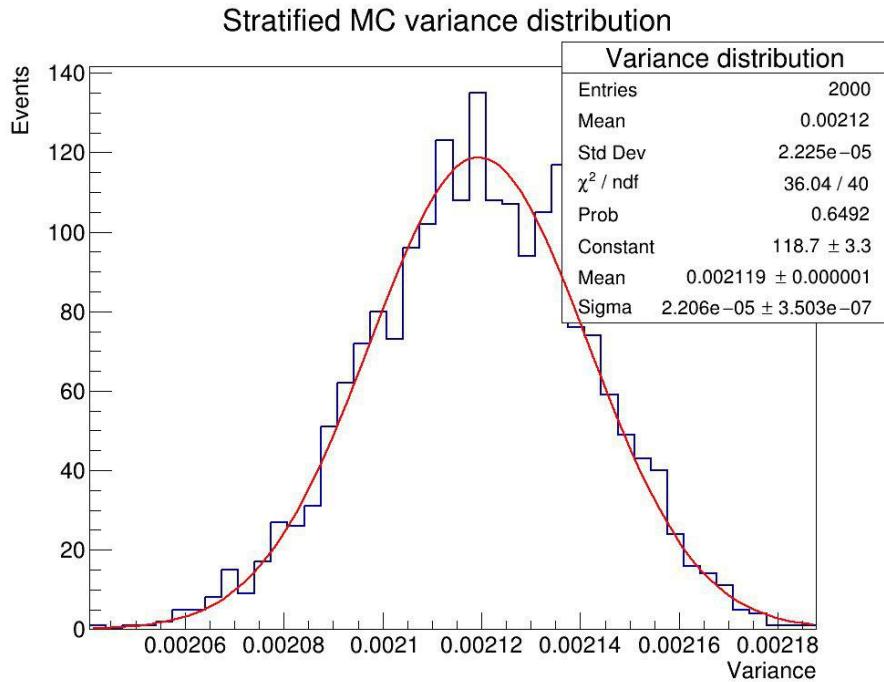


Figura 5.21: Distribuzione dell'estimatore  $\Theta$  dell'integrale definito con metodo diriduzione della varianza stratified sampling a 5 strati. Algoritmo xorshiro.

Trend of  $\Theta$  stratified estimator varying  $k$

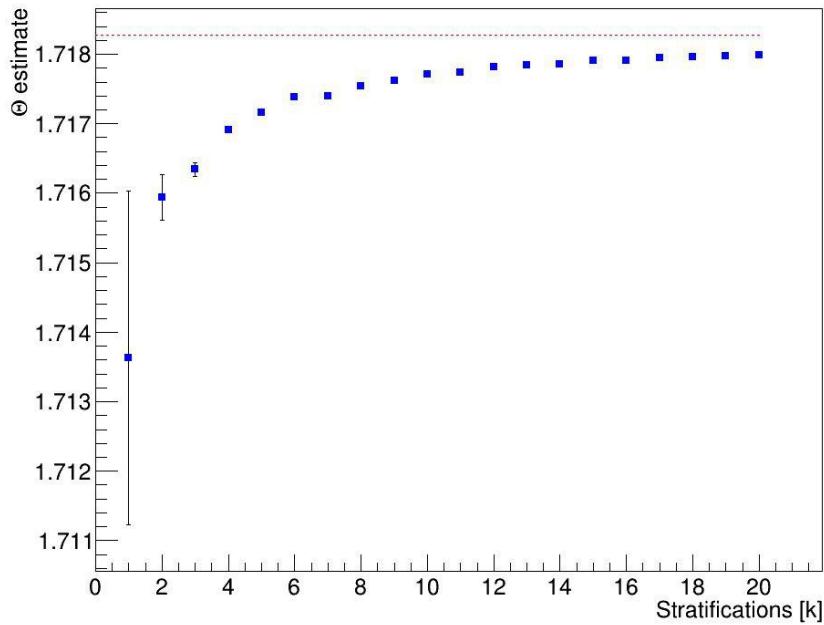


Figura 5.22: Trend dell'estimatore  $\Theta$  per un singolo esperimento monte carlo con generatore xorshiro al variare del numero di stratificazioni  $k$ . La linea orizzontale rossa è il valore vero dell'integrale  $I = e - 1$ .

Trend of the variance of  $\Theta$  stratified estimator

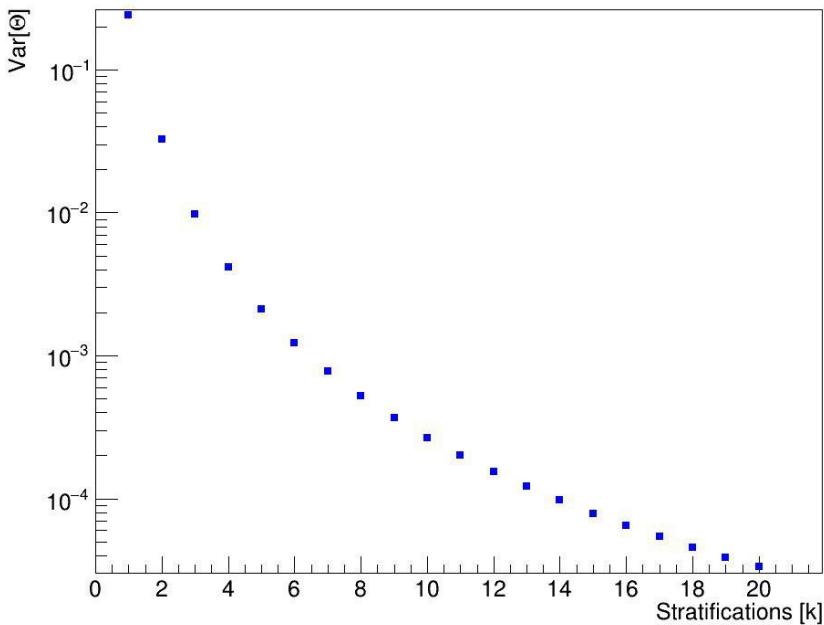


Figura 5.23: Trend della varianza dell'estimatore  $Var[\Theta]$  per un singolo esperimento monte carlo con generatore xorshiro al variare del numero di stratificazioni  $k$ .

$$\hat{I}_{strat\_xor} = 1.71826621 \pm 0.00001030$$

$$\hat{\sigma}_{strat\_xor}^2 = 0.00211949 \pm 0.00000022$$

### 5.5.3 Importance sampling xorshiro

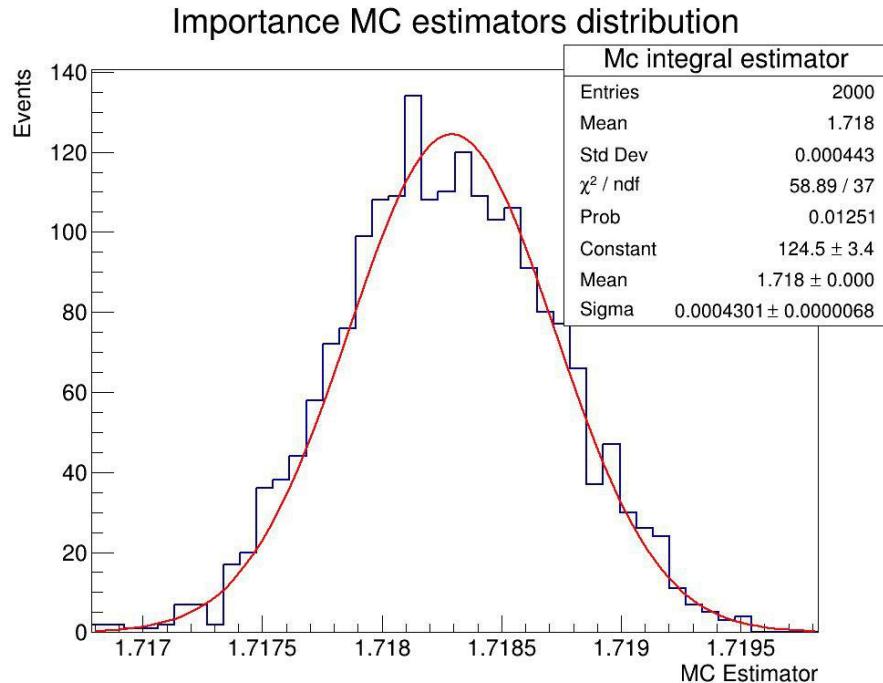


Figura 5.24: Distribuzione degli estimatori dell'integrale per il metodo importance sampling a generatore xorshiro con distribuzione d'importanza  $p(x) = (x + 1)^{1.5}$

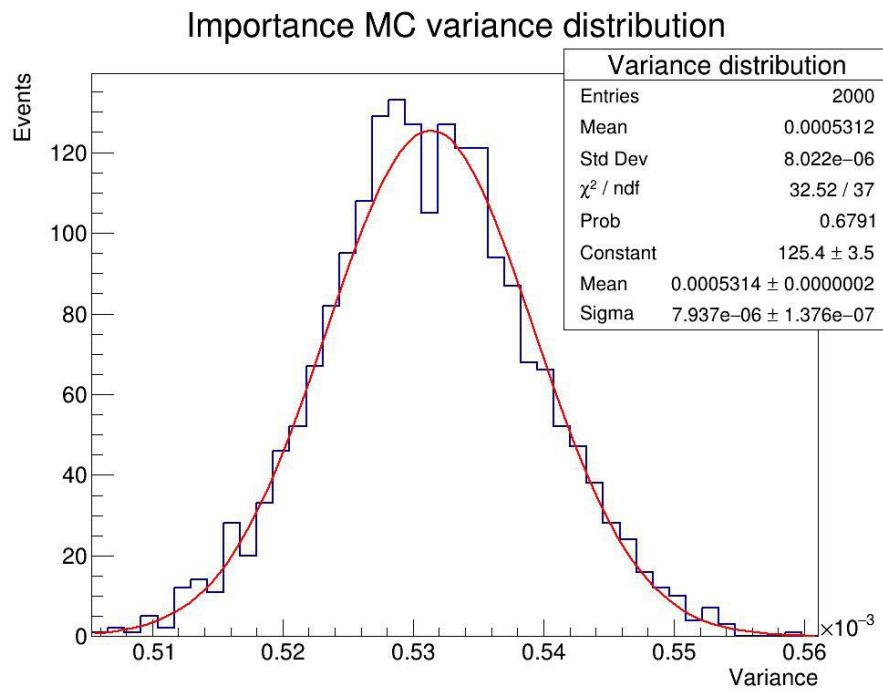


Figura 5.25: Distribuzione degli estimatori delle varianze dell'integrale per il metodo importance sampling a generatore `xorshiro` con distribuzione d'importanza  $p(x) = (x + 1)^{1.5}$

$$\hat{I}_{imp\_xor} = 1.71829322 \pm 0.00000430$$

$$\hat{\sigma}_{imp\_xor}^2 = 0.00053135 \pm 0.00000008$$

### 5.5.4 Antithetic Variates xorshiro

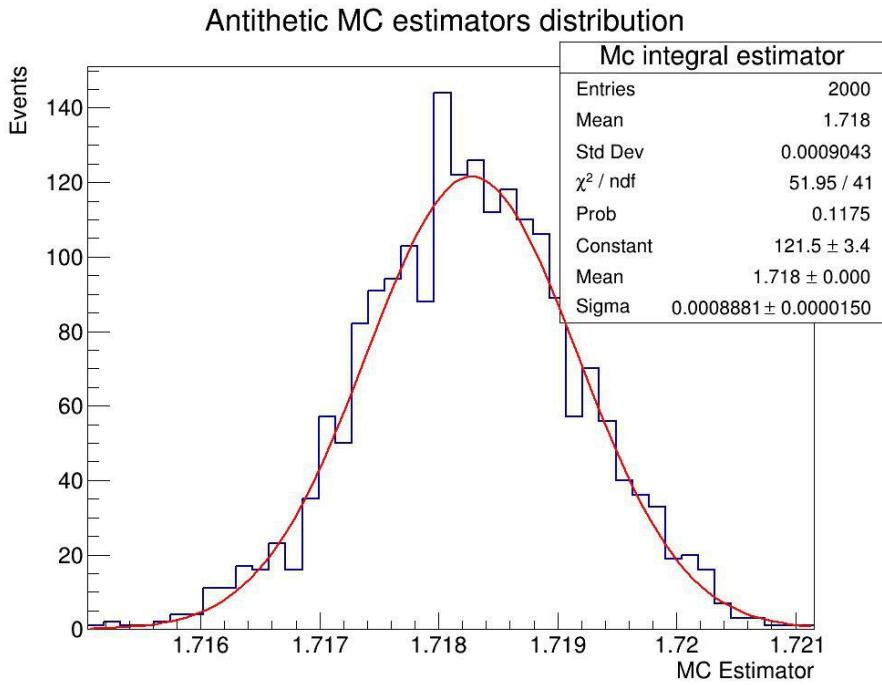


Figura 5.26: Distribuzione degli estimatori dell'integrale per il metodo antithetic variates usando generatore **xorshiro**

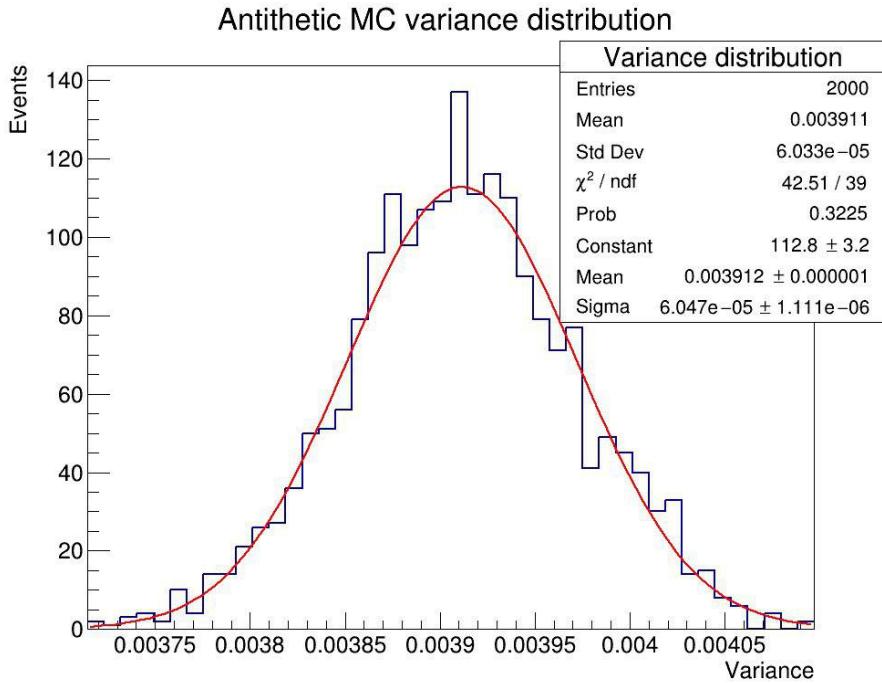


Figura 5.27: Distribuzione della varianza degli estimatori dell'integrale per il metodo antithetic variates usando generatore **xorshiro**

$$\hat{I}_{imp\_xor} = 1.71828255 \pm 0.00000888$$

$$\hat{\sigma}_{imp\_xor}^2 = 0.00391174 \pm 0.00000060$$

## 5.6 Risultati xorshiro

Il plot dei risultati analogo al precedente è riportato di seguito:

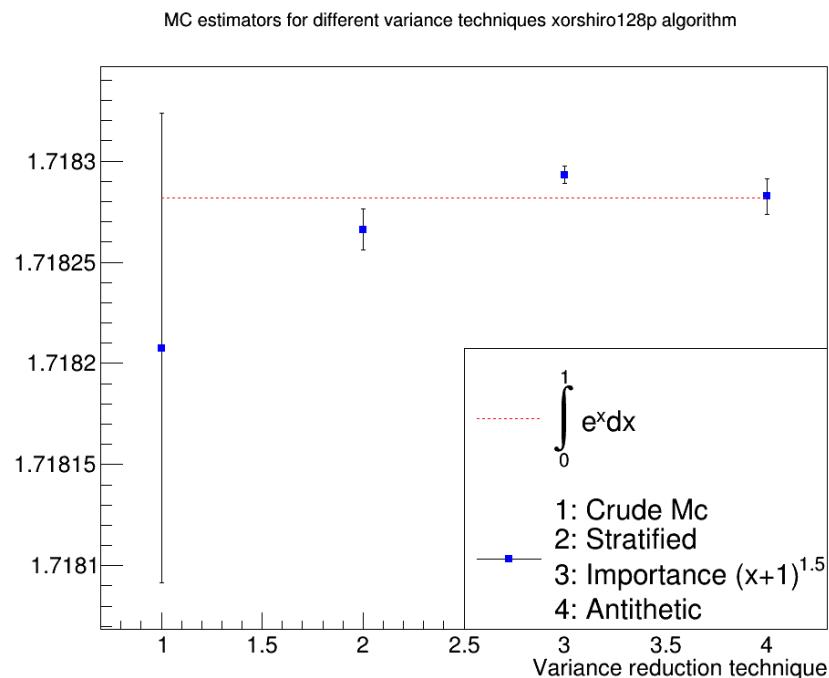


Figura 5.28: Comparazione degli estimatori media dell'integrale e della varianza dell'integrale usando il generatore xorshiro128p

# Capitolo 6

## Esercizio 6

L'esercizio 6 propone di simulare la diffrazione da un'apertura circolare.

Supponendo una fenditura circolare uniformemente illuminata, il pattern risultante su uno schermo posto a distanza  $d$  dal centro della fenditura presenta una regione centrale chiamata Airy disk molto illuminata, all'esterno sono presenti dischi concentrici luminosi separati da regioni di interferenza distruttiva. Il pattern così creato è detto Airy pattern in onore di George Biddell Airy, il primo che è riuscito a descrivere formalmente il fenomeno.

Matematicamente, possiamo descrivere l'intensità come funzione dell'angolo tra la congiungente del centro della fenditura con un punto generico sullo schermo e la normale al centro della fenditura in direzione dello schermo stesso. I parametri geometrici dell'apparato entrano in gioco sotto forma di costanti nella seguente espressione:

$$I(\theta) = I_0 = \left( \frac{2J_1(k \sin(\theta))}{k \sin(\theta)} \right)^2 = I_0 \left( \frac{2J_1(x)}{x} \right)^2 \quad (6.1)$$

Dove  $I_0$  indica l'intensità massima al centro del disco di Airy,  $a$  è il raggio della fenditura,  $k = \frac{2\pi}{\lambda}$  è il numero d'onda della radiazione incidente di lunghezza d'onda  $\lambda$ ,  $\theta$  è l'angolo d'osservazione,  $J_1$  è la funzione di Bessel del primo tipo di primo ordine.

In particolare, le funzioni di Bessel del primo tipo sono soluzioni dell'equazione differenziale di Bessel:

$$x^2 \frac{d^2 y}{dx^2} + x \frac{dy}{dx} + (x^2 + n^2)y = 0$$

Dove  $n$  è detto ordine della funzione.

Non è possibile definire le funzioni di Bessel in forma chiusa ma sono attraverso approssimazioni in particolare, uno dei tanti modi per definire le funzioni di Bessel è la seguente:

$$J_n(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!(m+n)!} \left( \frac{x}{2} \right)^{2m+n}$$

Per il nostro caso specifico  $n = 1$  osserviamo che:

$$J_1(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!(m+1)!} \left( \frac{x}{2} \right)^{2m+1}$$

Possiamo plottare la funzione valutandola in punti linearmente equispaziati grazie alla libreria **GSL** (GNU Scientific Library) . La libreria computa i valori della funzione grazie alle regole di ricorsione delle funzioni di Bessel quindi, come sottolineato nelle specifiche, i valori computati potrebbero differire dai valori esatti.

### $J_1(x)$ Bessel function of first kind first order

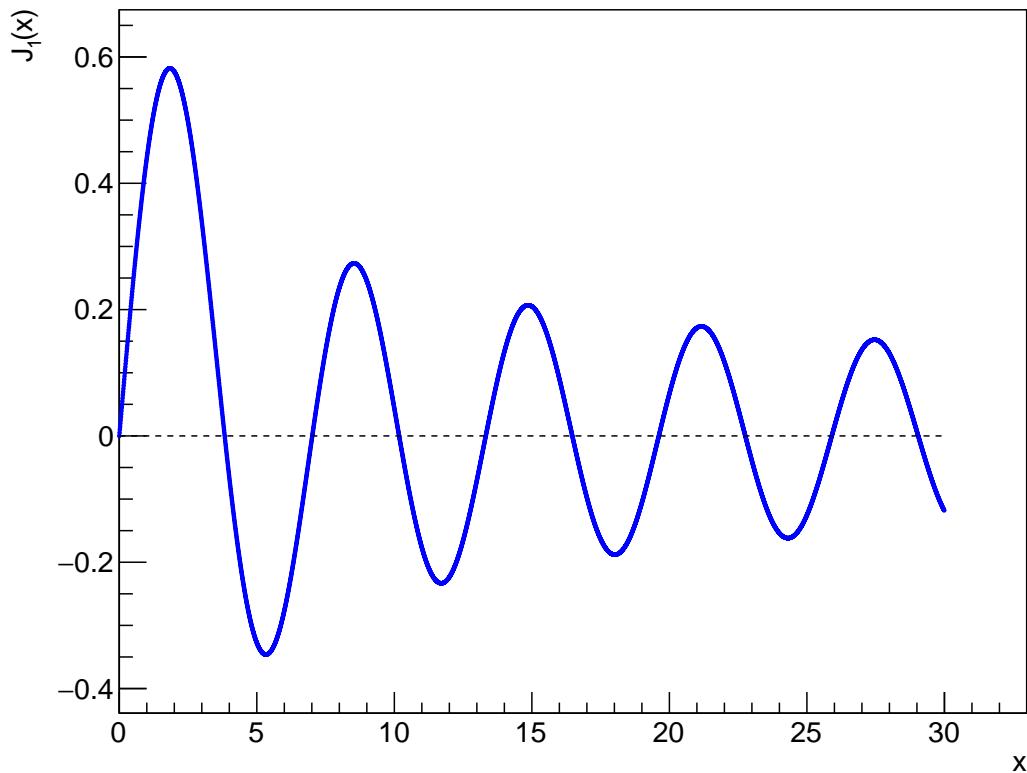


Figura 6.1: Plot dei valori computati da **GSL** per la funzione di bessel  $J_1(x)$  per  $x$  equispaziati di 0.01 tra [0,11].

La funzione di Airy per  $I(\theta)$  può essere plottata nello stesso modo, fissando parametri arbitrari per la configurazione del sistema:

## Theoretical $I(\theta)$ Airy function

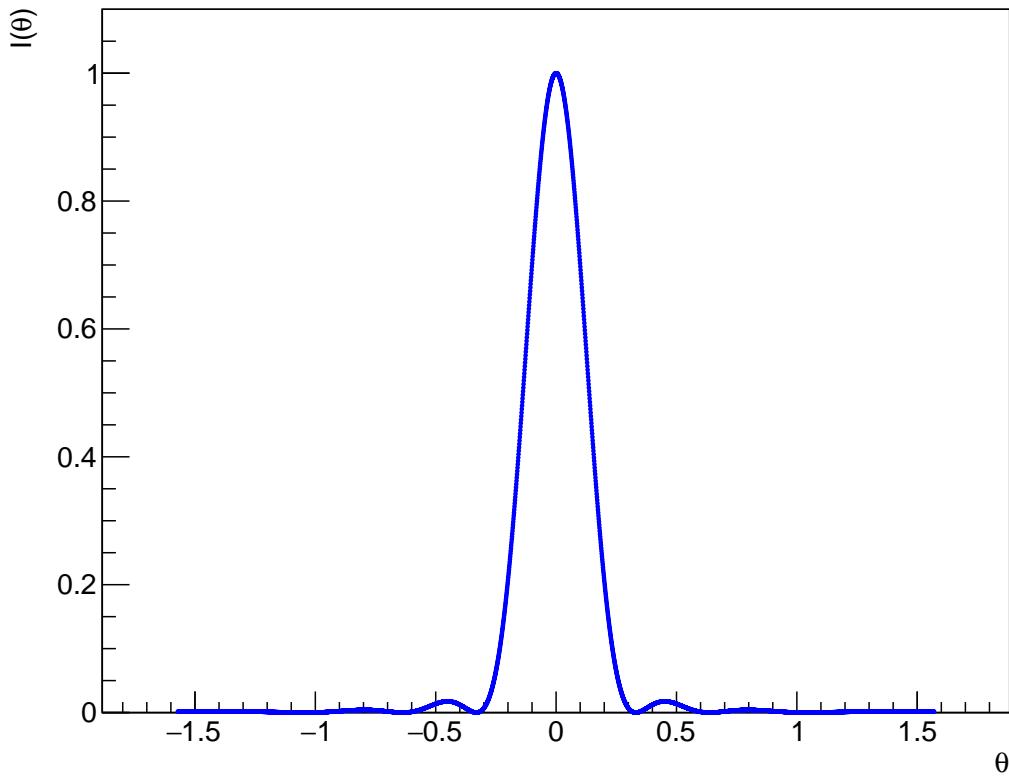


Figura 6.2:  $I(\theta)$  teorica fissando i parametri dell'esperimento e  $I_0 = 1$

### 6.1 Generazione $I(\theta)$ Monte Carlo

L'esercizio chiede quindi di simulare la diffrazione circolare generando 50000 eventi monte carlo distribuiti come la legge di Airy. (1.1)

Per la task viene costruita una libreria `simulation` con la seguente struttura:

```
#ifndef SIMULATION_
#define SIMULATION_

#include "prng.h"
#include "xorshiro.h"
#include "xorshiroGen.h"
#include <cmath>
#include <iostream>

class simulation{
public:
    simulation(int N, double sm_fact_n, double lambda_n, double diameter_n);
    ~simulation();
    double diffraction_fun(double theta);
    double* diffraction_tc(double xmin, double xmax, double ymin, double ymax);
```

```

        double* diffraction_smeared(double xmin, double xmax, int bins);

private:
    xorshiro uniform;
    xorshiroGen dist;
    int N_ev;
    double sm_fact;
    double lambda;
    double diameter;
    double* clean;
    double* smeared;
};

#endif //SIMULATION

```

La libreria si appoggia sulle librerie prng, xorshiro e xorshiroGen per la generazione di numeri pseudo-causali (`xorshiro`) e per il sampling di numeri pseudo-causali distribuiti secondo distribuzioni predefinite (`xorshiroGen`) come illustrato nell'esercizio 1.

Gli attributi privati della classe comprendono le seguenti variabili:

- **uniform**: istanza della classe xorshiro, viene usata per generare numeri pseudo-causali distribuiti uniformemente.
- **dist**: istanza della classe xorshiroGen, viene usata per generare numeri pseudo-causali distribuiti come una normale con parametri  $\mu, \sigma$ .
- **N\_ev**: Numero di eventi da generare con metodo monte carlo.
- **sm\_fact**: Fattore di smearing.
- **lambda**: Lunghezza d'onda della radiazione incidente.
- **diameter**: Raggio della fenditura circolare.
- **clean**: Puntatore al vettore contenente i valori generati dalla simulazione.
- **smeared**: Puntatore al vettore contenente i valori generati dallo smearing gaussiano.

I metodi definiti come public implementano le seguenti funzioni:

- **simulation(int N, double sm\_fact\_n, double lambda\_n, double diameter\_n)**: Costruttore dell'oggetto `simulation`, vengono passate le costanti della simulazione ovvero il numero di eventi da generare, il fattore di smearing, la lunghezza d'onda della radiazione incidente e il raggio dell'apertura circolare.
- **simulation()**: Distruttore.

- **diffraction\_fun(double theta)**: Funzione (1.1) per diffrazione circolare in funzione della variabile indipendente  $\theta$ .
- **diffraction\_tc()** Genera vettore di valori distribuiti secondo la legge di Airy per diffrazione da apertura circolare.
- **diffraction\_smeared()**: Applicazione dello smearing ai dati generati dalla funzione `diffraction_tc`.

Viene settato l'intensità a un valore fisso  $I_0 = 1$  in quanto esso è definito da costanti del sistema:

$$I_0 = \frac{P_0 A}{\lambda^2 R^2}$$

dove  $P_0$  è la potenza totale incidente sull'apertura,  $A = \pi a^2$  è l'area della fenditura,  $R$  è la distanza dall'apertura.

La funzione `diffraction_fun` valuta la funzione (1.1)  $I(\theta)$  per un certo  $\theta$  passato come input alla funzione nel modo seguente:

```
double simulation::diffraction_fun(double theta){

    //numero d'onda
    double k = 2*M_PI/lambda;
    //x
    double alpha = k*diameter*sin(theta);
    //J(x)
    double J_x = 2*gsl_sf_bessel_J1(alpha);
    double I_0 = 1.;

    //return I_0(J(x)/x)^{2}
    return I_0*((J_x)/alpha)*((J_x)/alpha);

}
```

La distribuzione degli eventi secondo la relazione di Airy è quindi eseguita dal metodo `diffraction_tc` nel modo seguente. Vengono passati quattro parametri per il range di generazione dei valori. Vengono generati  $x \sim U[x_{min}, x_{max}]$  e  $y \sim U[y_{min}, y_{max}]$  attraverso l'attributo `uniform`, istanza della classe `xorshiro` che utilizza l'algoritmo `xorshiro128p` per la generazione pseudo-casuale e viene eseguito un accept-reject ovvero se il valore di  $y$  è minore della funzione di Airy valutata nel punto  $x$  allora si tiene il valore  $x$  generato. Viene eseguita questa operazione finché non si sono generati  $N_{ev}$  numeri pseudo-casuali distribuiti come  $I(\theta)$ :

```
double* simulation::diffraction_tc(double xmin,double xmax,
                                    double ymin,double ymax){

    clean = new double[N_ev];
    int i = 0;
```

```

    while(i < N_ev){
        double x = uniform.rand(xmin,xmax);
        double y = uniform.rand(ymin,ymax);

        if(y < diffraction_fun(x)){
            clean[i] = x;
            i++;
        }
    }
    return clean;
}

```

Nel main istanziamo i parametri per la generazione pseudo-casuali e dei parametri della simulazione nel modo seguente. In particolare, supponiamo una radiazione monocromatica incidente nel range del visibile (rosso):

- Eventi: 50000;
- Raggio:  $0.75 \times 10^{-6}$ ;
- Lunghezza d'onda: 400 nm;
- Smearing: 0.05;
- Intervallo per generazione:  $\theta \sim U[-\frac{\pi}{2}, \frac{\pi}{2}]$

```

int main(){
    //Simulation parameters
    int N = 50000;
    double diameter = 0.75e-6;
    double lambda = 400e-9;
    double k = (2*M_PI)/lambda;
    double c = 0.05;

    //Histo parameter
    double xmax = M_PI/2;
    double xmin = -M_PI/2;
    double ymax = 20;
    double ymin = 0;

    //sigma_smearing = c*delta_theta (bin width)
    double std = c * (xmax - xmin) / bins;
}

```

Viene quindi istanziato l'oggetto della classe `simulation` passando il numero di eventi, parametro per lo smearing, lunghezza d'onda e il raggio della fenditura:

```
simulation experiment(N,c,lambda,diameter);
```

Creiamo la distribuzione simulata con metodo accept-reject monte carlo:

```
double* clean = experiment.diffraction_tc(xmin,xmax,ymin,ymax);
```

Viene restituito il puntatore al primo valore dell'array di valori generati. Possiamo procedere al filling di un'istogramma per visualizzare i dati generati. Viene scalato l'istogramma per l'entrata massima in quanto sappiamo che a priori abbiamo fissato l'intensità al centro del disco a 1, l'operazione è possibile grazie al metodo `scale`. Per un  $I_0$  generico, chiamata  $m$  l'altezza massima dell'istogramma dovremo dividere ogni entrata per  $I_0/m$ . Il binnaggio dell'histogramma segue  $\sqrt{N}$ :

```
TH1D* h = new TH1D("Clean I(#theta)","Clean I(#theta)",bins,-k*diameter,k*diameter);
h->SetTitle("I(#theta) Airy disk pattern");
h->GetXaxis()->SetTitle("kasin(#theta)");
h->GetYaxis()->SetTitle("Counts");
h->SetLineWidth(2);

for(int i = 0; i < N; i++){
    double x = clean[i];
    h->Fill(k*diameter*sin(clean[i]));
}

double m = h->GetMaximum();
h->Scale(1./m);
```

Il risultato viene plottato su un TCanvas:

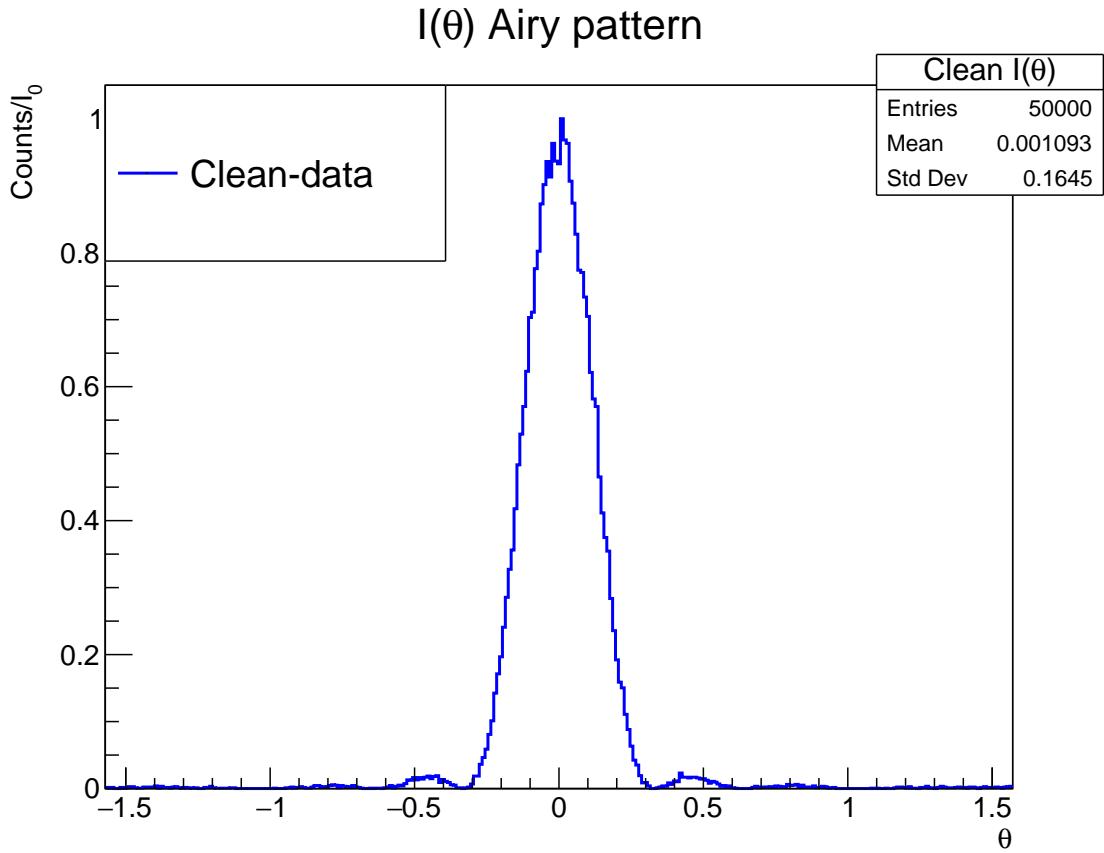


Figura 6.3: Istogramma degli eventi generati con metodo accept-reject monte carlo. Notiamo come la distribuzione sia simmetrica rispetto a 0. Sull’asse delle ascisse è stato plottato  $\theta$  mentre sull’asse delle y viene plottata la distribuzione dei dati generati che tende per  $N \rightarrow \infty$  a (indichiamo con  $x = \text{kasin}(\theta)$ )  $y = I_0\left(\frac{2J_1(x)}{x}\right)^2$

## 6.2 Smearing gaussiano

Viene effettuato quindi uno smearing dei dati per simulare l’efficienza finita del detector. Grazie alla funzione `diffraction_smeared` possiamo applicare un rumore gaussiano centrato in zero e larghezza  $sm\_fact$  ad ogni punto del dataset generato in precedenza:

```
double* simulation::diffraction_smeared(){

    smeared = new double[N_ev];

    for(long int i = 0; i < N_ev; i++){

        double x = clean[i];
        smeared[i] = x + dist.gauss_bm(0,sm_fact);

    }
}
```

```

    return smeared;
}

```

Il metodo fa uso del metodo `gauss_bm` della classe `xorshiroGen` che genera valori distribuiti normalmente secondo metodo della funzione inversa:

```

double xorshiroGen::gauss_bm(double mu, double sigma){
    // box-muller algorithm
    double u1 = uniform.rand(0,1);
    double u2 = uniform.rand(0,1);
    double z0 = sqrt( -2.0 * log(u1) ) * cos (2*M_PI*u2);
    // double z1: discarded
    return z0 * sigma + mu ;
}

```

Nel main viene chiamato il metodo che restituisce un puntatore al primo elemento dell'array dei valori dopo l'operazione:

```
double* smeared = experiment.diffraction_smeared();
```

Attraverso il metodo `set_smear()` possiamo cambiare il valore della costante di smear  $c$  per testare lo smearing delle distribuzioni al variare del parametro:

```
void set_smear(double sm){ sm_fact = sm; } ;
```

Il parametro di smearing, ovvero la deviazione standard della gaussiana da cui estrarre un numero pseudo-casuale distribuito normalmente, è definito come:

$$\sigma = c \frac{x_{min} - x_{max}}{bin_c} \in (0, 0.04)$$

Ovvero come moltiplicazione di una variabile ( $0 < c < 1$ ) e la larghezza dei bin dell'istogramma pulito  $\Delta\theta$ .

Nel main si fissa il numero dei bin per tutti gli istogrammi a  $bin_c = 80$ , si chiama quindi la funzione `diffraction_smeared` e si filla un istogramma con i valori del vettore in output. Si cambia la costante  $c$  e si ripete l'operazione. Gli istogrammi vengono scalati per una costante tale per cui l'area dell'istogramma pulito approssimi l'area della distribuzione di Airy.

```

double* clean = experiment.diffraction_tc(xmin,xmax,ymin,ymax);

TH1D* h = new TH1D("Clean I(#theta)","Clean I(#theta)",bins,-M_PI/2,M_PI/2);
h->SetTitle("I(#theta) Airy pattern");
h->GetXaxis()->SetTitle("#theta");
h->GetYaxis()->SetTitle("Counts/I_{0}");
h->SetLineWidth(1);
h->SetLineColor(kBlack);

for(int i = 0; i < N; i++){

```

```

    h->Fill(clean[i]);

}

double m = h->GetMaximum();
h->Scale(1./m);

double* smeared = experiment.diffraction_smeared();

TH1D* h_sm = new TH1D("histo_sm","histo_sm",bins,-M_PI/2,M_PI/2);
h_sm->SetLineColor(kRed);
for(int i = 0; i < N; i++){
    h_sm->Fill(smeared[i]);
}
h_sm->Scale(1./m);

experiment.set_smear(0.5*(xmax - xmin)/bins);
double* smeared2 = experiment.diffraction_smeared();

TH1D* h_sm2 = new TH1D("histo_sm2","histo_sm2",bins,-M_PI/2,M_PI/2);
h_sm2->SetLineColor(kBlue);
for(int i = 0; i < N; i++){
    h_sm2->Fill(smeared2[i]);
}
h_sm2->Scale(1./m);

experiment.set_smear(0.7*(xmax - xmin) / bins);
double* smeared3 = experiment.diffraction_smeared();

TH1D* h_sm3 = new TH1D("histo_sm3","histo_sm3",bins,-M_PI/2,M_PI/2);
h_sm3->SetLineColor(kOrange);

for(int i = 0; i < N; i++){
    h_sm3->Fill(smeared3[i]);
}

h_sm3->Scale(1./m);

experiment.set_smear(0.95*(xmax - xmin) / bins);
double* smeared4 = experiment.diffraction_smeared();

TH1D* h_sm4 = new TH1D("histo_sm4","histo_sm4",bins,-M_PI/2,M_PI/2);
h_sm4->SetLineColor(kGreen);
for(int i = 0; i < N; i++){
    h_sm4->Fill(smeared4[i]);
}

```

```
h_sm4->Scale(1./m);
```

Il risultato per valori ci c  $c = \{0.3, 0.5, 0.7, 0.95\}$  è il seguente:

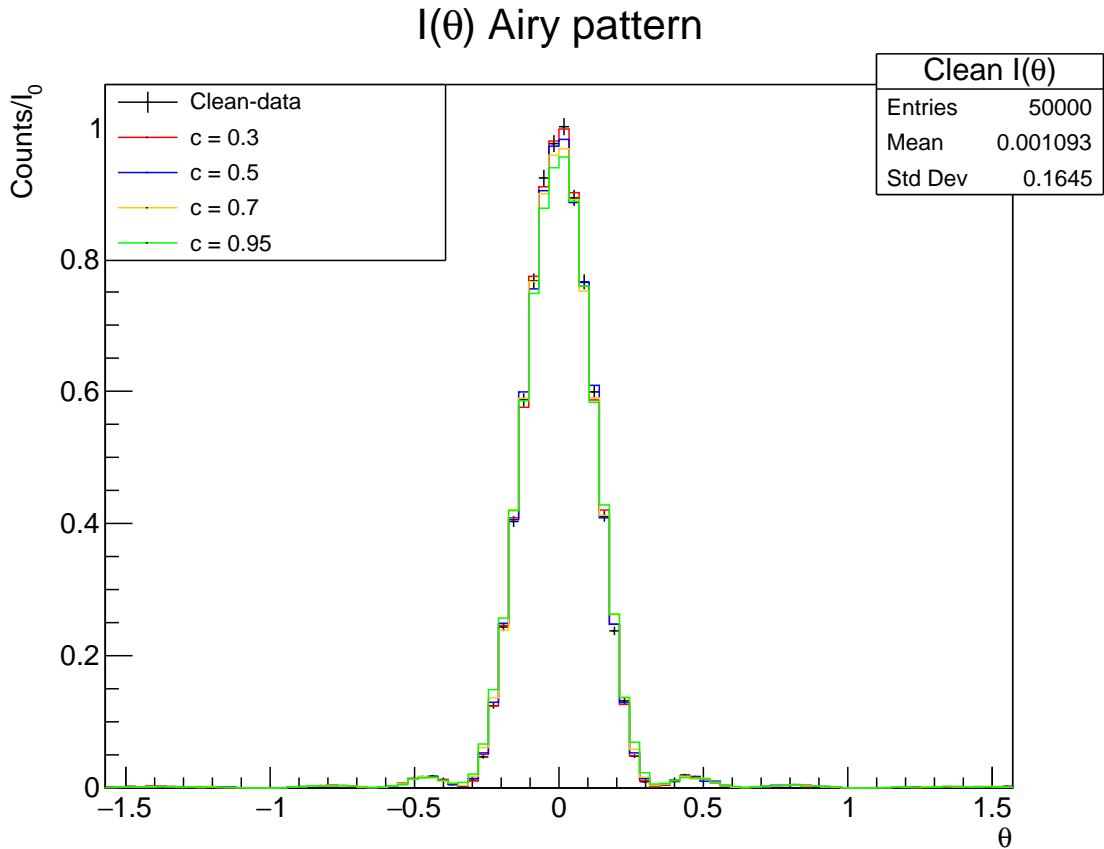


Figura 6.4: Istogrammi con diversi parametri di smearing rispetto alla distribuzione di Airy. I dati originali sono indicati dai punti con errore di colore nero.

Notiamo che, applicando lo smearing, tendiamo a perdere i picchi secondari e inoltre l'intensità centrale massima  $I_0$  è ridotta rispetto a quella originale. L'effetto è tuttavia poco marcato in quanto stiamo applicando uno smearing piccolo. Per valori di  $c$  superiori ad 1 le distribuzioni si appiattiscono fino a perdere totalmente l'informazione che contengono.

### 6.3 Unfolding della distribuzione sperimentale

Tecniche di unfolding dei dati sperimentali sono state sviluppate per approssimare, con un certo livello di confidenza, la pdf di una variabile casuale  $y$  la misura della quale porta a una distribuzione distorta a causa dei vari effetti sperimentali come eventuali bias, efficienza nella misura, risoluzione del rivelatore e effetti dovuti al fondo.

Come esempio, si può pensare lo smearing gaussiano come una simulazione della

risoluzione finita nella misura dello spettro di diffrazione.

Il procedimento adottato nel caso di misure affette dall'apparato sperimentale è di foldare la teoria con la risposta sperimentale. Si calcola una matrice di correzione e si ricava l'approssimazione della distribuzione originaria delle misure:

$$\nu_i = \sum_{j=1}^M R_{ij} \mu_j + \beta_i \quad (6.2)$$

Dove con  $\nu$  indichiamo la misura,  $\mu$  il suo valore vero,  $\beta$  un eventuale bias e  $R_{ij}$  l'elemento della matrice di correzione che ci permette di passare dal valore vero al valore misurato e viceversa.

L'elemento di matrice esprime la probabilità di misurare  $n$  eventi nel bin  $i$ -esimo data il valore vero d'aspettazione per le entrate del bin.

Per trovare un estimatore del valore vero  $\mu$  ci servono o una *pdf* teorica oppure una matrice di covarianza.

Gli ingredienti che sono necessari affinché l'unfolding funzioni sono i seguenti:

- L'istogramma dei valori veri  $\mu$  generati da un modello teorico.
- I valori d'aspettazione per l'istogramma misurato  $\nu$  e il numero di entries effettivamente osservato  $n$ .
- $R_{ij}$ , elemento di matrice  $P(\text{osservazionibini}|\text{valorevero}binj)$ .
- Efficienze dei fondi attesi:  $\epsilon_j = \sum_{i=0}^N R_{ij}$

Il risultato è quindi ottenuto invertendo l'equazione (6.2):

$$\hat{\mu} = \mathbf{R}^{-1}(\bar{n} - \bar{\beta}) \quad (6.3)$$

A causa della non conoscenza del valore vero delle entrate nei vari bin, siamo costretti ad usare il valore misurato  $\bar{n}$  il che può portare a risultati catastrofici.

Si dimostra il caso che lo stimatore di  $\mu$  così costruito è non biassato:

$$E[\hat{\mu}] = R^{-1}(E[n] - \beta) = \mu$$

$$\text{cov}[\mu_i, \mu_j] = \sum_{k=1}^N (R^{-1})_{ik} (R^{-1})_{jk} \nu_k$$

La varianza è molto grande anche se è la minima possibile per un tale estimatore. Una procedura utilizzata è quella del trade-off bias-variance ovvero ammettiamo di introdurre un po' di bias nell'estimatore al fine di ridurne la varianza attraverso tecniche di regolarizzazione.

Supponiamo infatti  $M = N$ , per stimare  $\mu$  possiamo utilizzare:

$$\hat{\mu}_i = C_i(n_i - \beta_i)$$

Dove  $C_i$  è un parametro moltiplicativo ricavato da generazioni montecarlo:

$$C_i = \frac{\hat{\mu}_i^{MC}}{\hat{\nu}_i^{MC}}$$

Dove  $\mu_i^{MC}$  sono ricavati simulando la distribuzione ideale attraverso tecniche monte carlo mentre  $\nu_i^{MC}$  sono ricavati dalla simulazione del detector ai dati generati in precedenza secondo il modello teorico. Se la statistica è sufficientemente grande allora gli errori sui  $C_i$  si possono supporre relativamente trascurabili. La regolarizzazione degli estimatori  $\hat{\mu}$  produce degli andamenti smooth degli estimatori. La soluzione per  $\mu$  può essere ottenuta supponendo una statistica Poissoniana per le entrate dei bin e massimizzando il logaritmo delle likelihood:

$$\log L(\mu) = \sum_{i=1} N \log P(n_i; \nu_i)$$

Dove:

$$P(n_i; \nu_i) = \frac{\nu_i^{n_i}}{n_i!} e^{-\nu_i}$$

è Poissoniana nel bin  $i$ . Possiamo cercare un compromesso, cercando una soluzione con un andamento più smooth al prezzo di un bias non zero ed una varianza eventualmente maggiorata:

$$\log L(\bar{\mu}) - (\log L_{max} - (L))$$

Questa formula definisce il trade off tra bias e varianza ottenuta nell'istogramma unfoldata.

Descriviamo la smoothness della soluzione  $\mu$  attraverso una funzione di smoothness  $S(\mu)$  chiamata funzione di regolarizzazione. L'idea è di scegliere la soluzione  $\mu$  che massimizza la smoothness ovvero bisogna massimizzare:

$$\alpha[\log L(\bar{\mu}) - (\log L_{max} - \Delta \log L)] + S(\bar{\mu})$$

Rispetto sia  $\mu$  che ad  $\alpha$ . Fissato  $\alpha$  si massimizza:

$$\Phi(\bar{\mu}) = \alpha \log L(\bar{\mu}) + S(\bar{\mu})$$

Ci sono vari termini di regolarizzazione:

- Tikhonov minimum second derivative:

$$C(\bar{\mu}) = - \int [f_y''(y)]^2 dy \simeq \sum_{i=1}^{M-2} [-\bar{\mu}_i + 2\bar{\mu}_{i+1} - \bar{\mu}_{i+2}]^2$$

- Minimum Variance:

$$C(\bar{\mu}) = -Var[\bar{\mu}] = ||C(\bar{\mu})||^2 = - \sum_i \bar{\mu}_i^2$$

- Maximum entropy (MaxEnt):

$$C(\bar{\mu}) = - \sum_i p_i \ln p_i = - \sum_i \frac{\mu_i}{\mu_T} \ln \frac{\mu_i}{\mu_T}$$

- Cross-entropy:

$$C(\bar{\mu}) = - \sum_i p_i \ln \frac{p_i}{q_i} = - \sum_i \frac{\mu_i}{\mu_T} \ln \frac{\mu_i}{\mu_T q_i}$$

dove  $\bar{q} = (q_1 \dots q_n)$  è la shape a priori più probabile per la distribuzione vera di  $\mu_i$

L'unfolding si basa quindi sul principio iterativo (1930):

$$N_{ij}(th) = NP_{ij}(obs) = N \sum_{i'j'} P_{i'j'}(true) P_\nu(obs_{ij}|true_{i'j'})$$

Scriviamo questa equazione considerando la presenza dell'operatore R:

$$n = R \times \mu$$

$$\mu_{k+1} = \mu_k + \beta[n - R \times \mu_k]$$

Il metodo è basato sulla nota equazione:

$$\sum_{i=0}^k q^n = \frac{1 - q^{k+1}}{1 - q}$$

Per  $k \rightarrow \infty$  la serie converge se  $|q| < 1$ . Applicando iterativamente l'ultima formula otteniamo che:

$$\mu_{k+1} = \sum_{i=0}^k \beta(1 - \beta R)^i n = \frac{I - (I - \beta R)^{k+1}}{\beta R} \beta n \rightarrow R^{-1} n = \mu \quad , \quad \text{for } k \rightarrow \infty$$

Se e solo se  $|I - \beta R| < 1$ .

Consideriamo il caso con fluttuazioni statistiche:

$$\mu_i \rightarrow \nu_i \rightarrow n_i$$

$$n = R\mu + r$$

Per avere soluzioni regolari, e per implementare un metodo robusto, dobbiamo cercare una soluzione iterativa che massimizzi il  $\chi^2$ :

$$\chi^2 = \|R \times \mu - n\|^2 = \frac{1}{2} \sum_{ik} \left( \sum_{mn} R_{i-m,k-n} \mu_{mn} - n_{ik} \right)^2$$

Possiamo ricavare che il minimo del chi quadro rispetto ai valori veri  $\mu_{ik}$  da la seguente formula:

$$\frac{\partial \chi^2}{\partial \mu} = R \times [R \times \mu - n] = 0$$

Supponiamo di aggiungere ora un parametro di regularizzazione:

$$\mu_{k+1} = \mu_k + \beta_k R \times [n - R \times \mu_k]$$

Il metodo converge se e solo se:

$$||I - \times R|| < 1$$

Il metodo aggiunge quindi un parametro di regolarizzazione al chi quadro nel modo seguente:

$$\chi^2 = ||R \times \mu - n||^2 + \alpha ||C \times \mu||^2$$

La soluzione iterativa diventa:

$$\mu_{k+1} = \mu_k + \beta_k [R \times n - (R \times R + \alpha C \times C) \mu_k]$$

Vari modi per definire il parametro di regolarizzazione. La funzione oggettiva da minimizzare è:

$$-F(\mu) = -2\ln L(n|\mu) - (\mu) + \lambda(n_T - \sum_i \nu_i)$$

L'approccio Bayesiano impone come parametro

$$\alpha = \frac{1}{\mu_T}$$

Che in genere introduce troppo smoothing.

### 6.3.1 Bayes Approach

Usiamo il pacchetto *RooUnfold* che implementa diversi algoritmi per l'unfolding di distribuzioni sperimentali.

Il primo passo è generare la matrice di risposta. Per farlo il pacchetto richiede che sia creato un oggetto *RooUnfoldResponse* al quale vanno passate le entrate dei bin degli histogrammi creati da simulazioni monte carlo, il primo per la simulazione della misura mentre il secondo della distribuzione teorica. Bisogna stare attenti in quanto i vettori sono intesi come dati di train. Bisognerà generare un secondo set di misurazioni per testare l'unfolding. Se testassimo il metodo sugli stessi dati su cui è stato allenato allora otterremo un risultato che, ovviamente, è perfetto.

Per il dataset con smearing utilizziamo quello con parametro  $c = 0.95$ .

```
RooUnfoldResponse response(bins,xmin,xmax); //Unfolding

TH1D* Intensity_measured = new TH1D("Intensity Measured",
"Intensity Measured", bins, xmin, xmax);

experiment.set_smear(0.95*(xmax - xmin)/bins);
double* smeared_to_response = experiment.diffraction_smeared();
double* smeared_to_unfold = experiment.diffraction_smeared();

for(int i = 0; i < N; i++){
    response.Fill(smeared_to_response[i]);
    unfold.Fill(smeared_to_unfold[i]);
}
```

```

    Intensity_measured->Fill(smeared_to_unfold[i]);
    response.Fill(smeared_to_response[i],clean[i]);
}

Intensity_measured->Scale(1./m);

```

Il primo metodo di unfolding implementa la regolarizzazione Bayes o di D'Agostini. Viene costruito un oggetto *RooUnfoldBayes* al quale viene passato l'indirizzo della matrice di risposta e un nuovo sample di misure *smeared\_to\_unfold* sotto forma di istogramma con stesso binnaggio degli altri. Il terzo parametro è un termine intero di regolarizzazione che viene posto a 4:

```
RooUnfoldBayes unfold(&response, Intensity_measured, 4);
```

Si procede quindi con l'unfolding attraverso il metodo *HReco* che ricostruisce l'istogramma teorico:

```

std::cout<< "----->UNFOLDING<-----" << std::endl;
std::cout<< "----->Bayes<-----" << std::endl;
TH1D* h_unfold =(TH1D*)unfold.Hreco();
h_unfold->SetLineWidth(2);
h_unfold->SetMarkerSize(2);

```

Il risultato viene quindi plottato su un TCanvas. Vengono plottate le distribuzioni originarie ovvero la distribuzione teorica e quella sperimentale. La ricostruzione è plottata con dei TMarker verdi con le relative barre d'errore.

In un grafico inferiore plottiamo il risultato del metodo *Divide* che effettua una divisione bin-per-bin dell'istogramma teorico e quello ricostruito. Ci aspettiamo che il rapporto tenda a uno. Gli errori della computazione, che dipendono principalmente dalla poca statistica presente nelle code delle distribuzioni, sono plottati come aree con FillStyle a x.

La funzione per il secondo grafico è la seguente:

```

TH1D* createRatio(TH1D* h1, TH1D* h2, const char* label="ratio"){
    TH1D* h3 = (TH1D*) h1->Clone("h3");
    h3->SetMarkerStyle(21);
    h3->SetMarkerSize(1);
    h3->SetMarkerColor(kBlack);
    h3->SetLineColor(kBlack);
    h3->SetTitle("");
    // Set up plot for markers and errors
    h3->Sumw2();
    h3->SetStats(0);
    h3->Divide(h2);
    //Fix automatically the ratio range
    h3->SetMaximum(h3->GetMaximum() + 0.05);
    h3->SetMinimum(h3->GetMinimum() - 0.05);

    // Adjust y-axis settings

```

```

    h3->GetYaxis()->SetTitle(label);
    return h3;
}

```

Il plot principale viene generato nel modo seguente:

```

TCanvas* c_bayes = new TCanvas("c_bayes", "c_bayes", 1000, 1000, 1000, 1000);
// Upper histogram plot is pad1
TPad* pad1 = new TPad("pad1", "pad1", 0, 0.28, 1, 1);
pad1->SetBottomMargin(0); // joins upper and lower plot
pad1->SetRightMargin(0.04);
pad1->SetLeftMargin(0.13);
pad1->SetTickx(1);
pad1->SetTicky(1);
pad1->Draw();
// Lower ratio plot is pad2
c_bayes->cd(); // returns to main canvas before defining pad2
TPad* pad2 = new TPad("pad2", "pad2", 0, 0.0, 1, 0.28);
pad2->SetTopMargin(0); // joins upper and lower plot
pad2->SetRightMargin(0.04);
pad2->SetLeftMargin(0.13);
pad2->SetBottomMargin(0.13);
pad2->SetGridx();
pad2->SetGridy();
pad2->SetTickx(1);
pad2->SetTicky(1);
pad2->Draw();

c_bayes->Update();
pad1->cd();

gStyle->SetOptStat(0);
h->Draw("hist");
h->SetTitle("Unfolding result using Bayesian approach");
h->GetYaxis()->SetTitleSize(.04);
h->GetXaxis()->SetTitleSize(.04);
Intensity_measured->Draw("hist same");
h_unfold->Draw("SAME");
leg1->Draw();

TH1D* rateo = createRatio(h, h_unfold);

pad2->cd();
rateo->Draw("hist");
TH1F* hratioerror =(TH1F*) rateo->DrawCopy("E2 same");
hratioerror->SetFillStyle(3013);
hratioerror->SetFillColor(13);
hratioerror->SetMarkerStyle(1);

```

```
c_bayes->cd();
c_bayes->Update();
c_bayes->Draw();
```

Il risultato è il seguente:

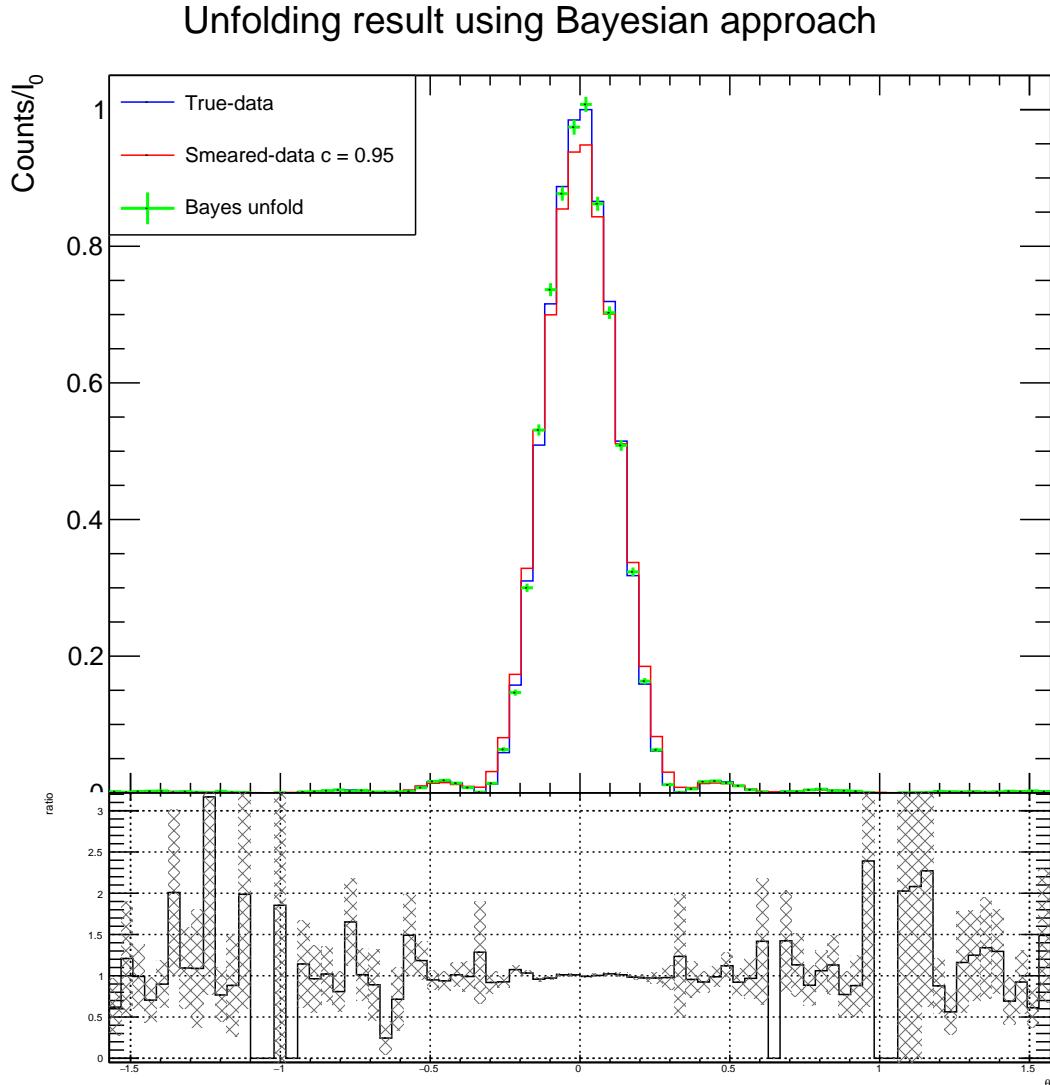


Figura 6.5: Unfolding della distribuzione sperimentale supponendo uno smearing gaussiano  $c = 0.95$  per il pattern di Airy la cui distribuzione teorica è rappresentata in blu. L'istogramma ricostruito grazie alla matrice di risposta, con metodo di regolarizzazione Bayesiano/D'Agostini è rappresentato con delle cross verdi.

Possiamo inoltre riprodurre un plot della matrice di correlazione bin-per-bin del metodo definita come  $P(\text{ossbini}|\text{aspettativabinj})$ :

```
TCanvas* c_cov_bayes = new TCanvas("c_c_b", "c_c_b", 1000, 1000, 1000, 1000);
unfold.Ereco().Draw("colz");
c_cov_bayes->Draw();
```

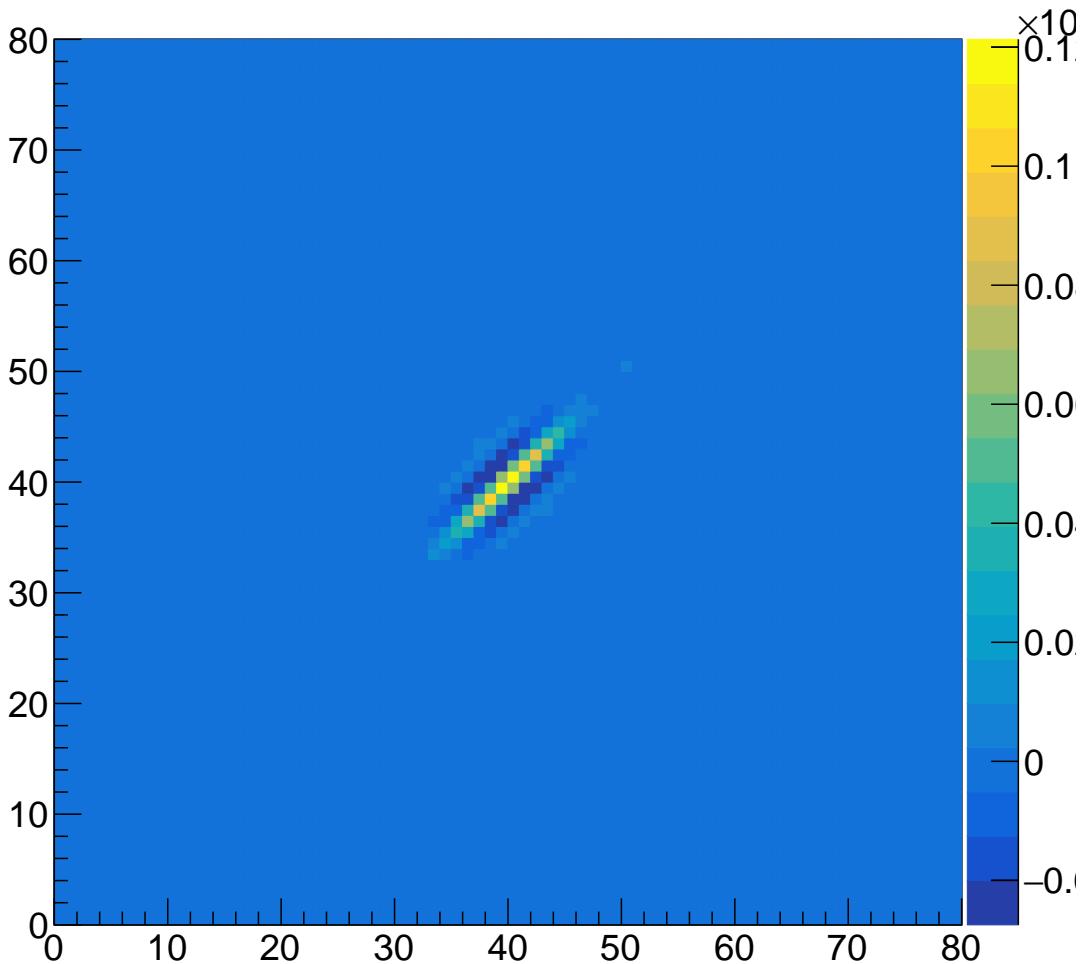


Figura 6.6: Matrice di covarianza bin per bin con metodo Bayes/D'Agostini

### 6.3.2 Bin-by-Bin Unfolding

Il metodo Bin-by-Bin per la regolarizzazione suppone un parametro di correzione:

$$C_i = \frac{\mu_i}{\nu_i}$$

Dove  $\mu$  e  $\nu$  sono i valori d'aspettazione per le entrate del bin  $i$ -esimo per, rispettivamente, il monte carlo della distribuzione vera e della distribuzione misurata affetta da effetti strumentali.

Il metodo è stato criticato per non tener conto a pieno della correlazione tra i bin. Implementiamo, in modo analogo al precedente, un oggetto *RooUnfoldBinByBin* il quale non richiede parametri aggiuntivi oltre alla matrice di risposta e all'istogramma degli eventi misurati.

Viene ripetuta la procedura di plotting prima riportata:

## Unfolding result using Bin-by-Bin approach

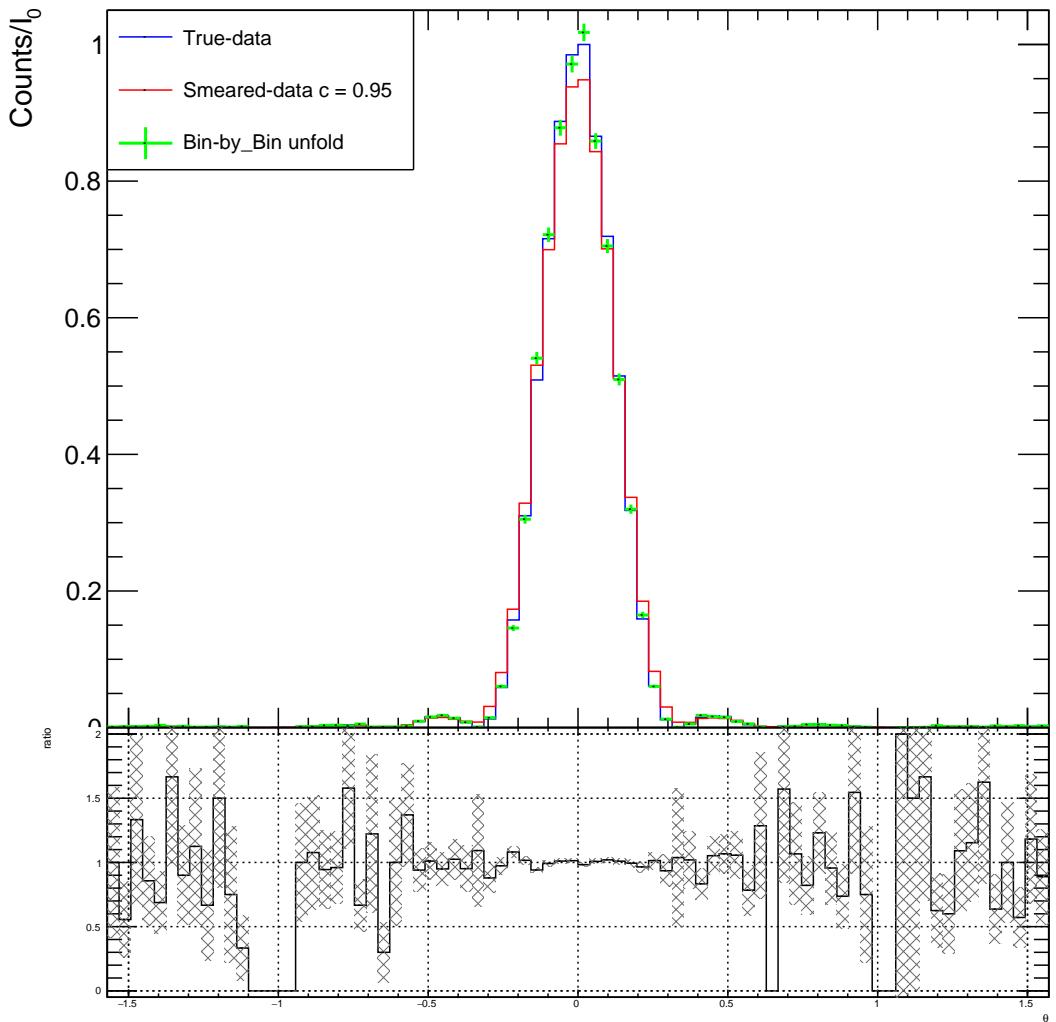


Figura 6.7: Unfolding della distribuzione sperimentale supponendo uno smearing gaussiano  $c = 0.95$  per il pattern di Airy la cui distribuzione teorica è rappresentata in blu. L'istogramma ricostruito grazie alla matrice di risposta, con metodo di regolarizzazione Bin-by-Bin è rappresentato con delle cross verdi.

La matrice di correlazione è invece la seguente:

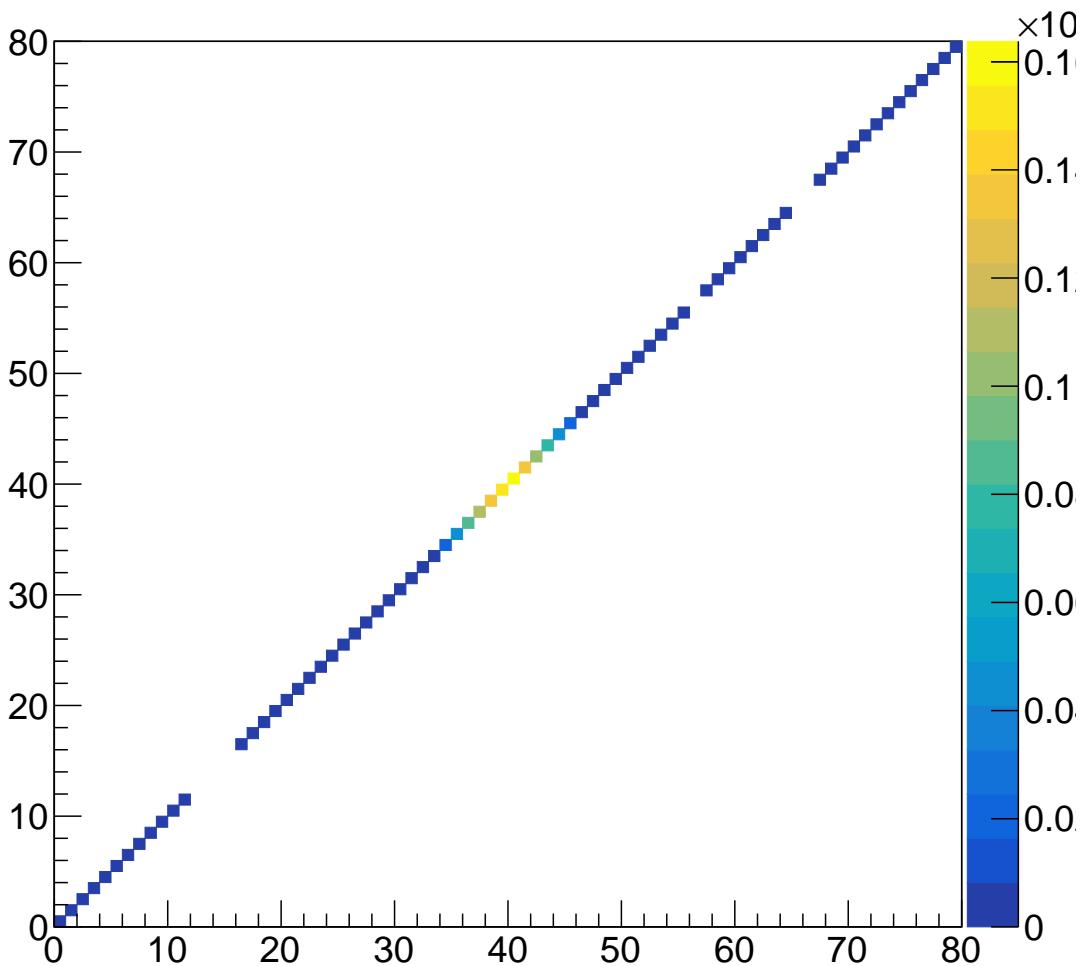


Figura 6.8: Matrice di covarianza bin per bin con metodo Bin-by-Bin

### 6.3.3 SVD unfolding

Il metodo SVD fa uso del metodo della Single Value Decomposition della matrice di risposta per implementare il metodo d'unfolding. Il metodo è caratterizzato da un parametro  $k$  di regolarizzazione positivo e intero:

```
RooUnfoldSvd unfold_svd(&response, Intensity_measured, 20);
```

Il parametro di regolarizzazione è stato scelto per ottimizzare il problema da un punto di vista visivo. Ci sono metodi matematici per stimare il valore ottimale di  $k$  come nella seguente referenza [HK95]

Plottiamo i risultati come prima:

## Unfolding result using SVD approach

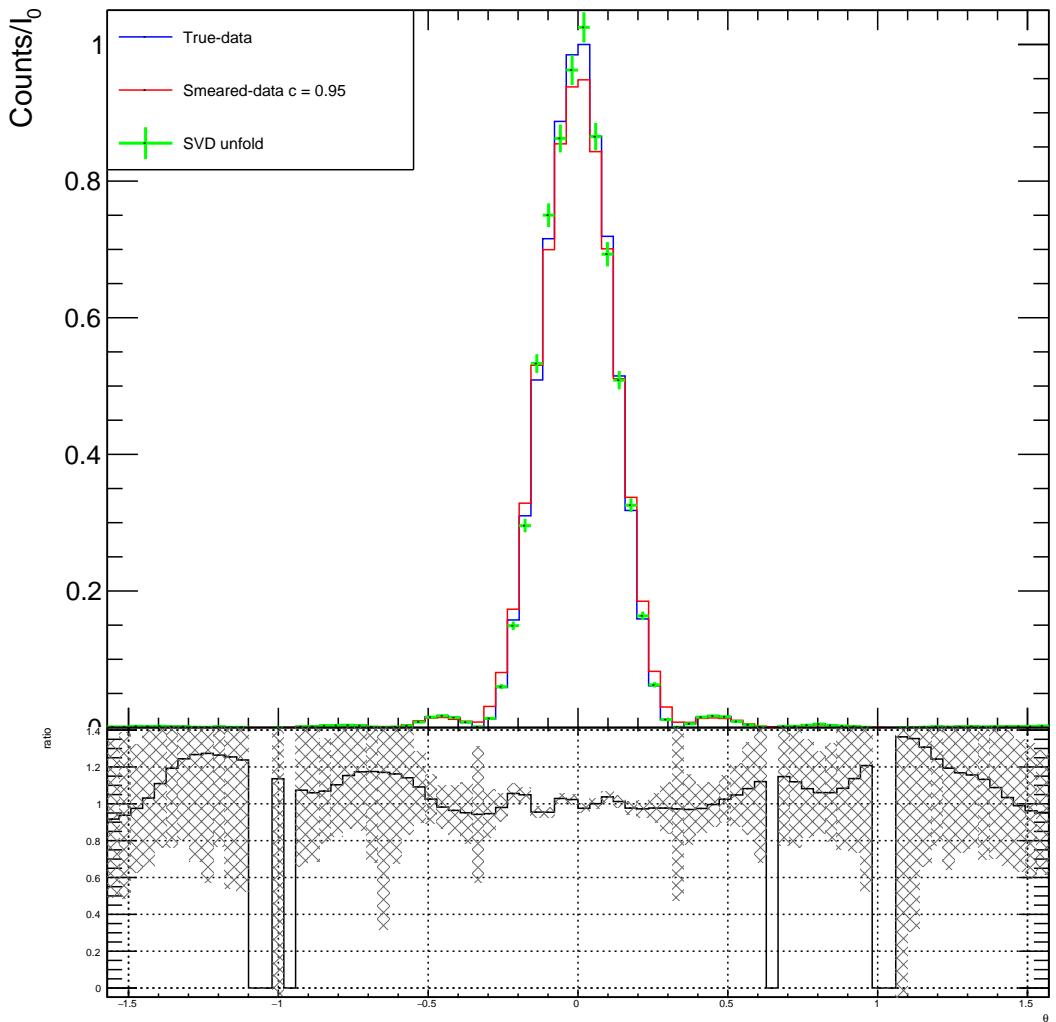


Figura 6.9: Unfolding della distribuzione sperimentale supponendo uno smearing gaussiano  $c = 0.95$  per il pattern di Airy la cui distribuzione teorica è rappresentata in blu. L'istogramma ricostruito grazie alla matrice di risposta, con metodo di regolarizzazione SVD  $k = 20$  è rappresentato con delle cross verdi.

La matrice di correlazione è invece la seguente:

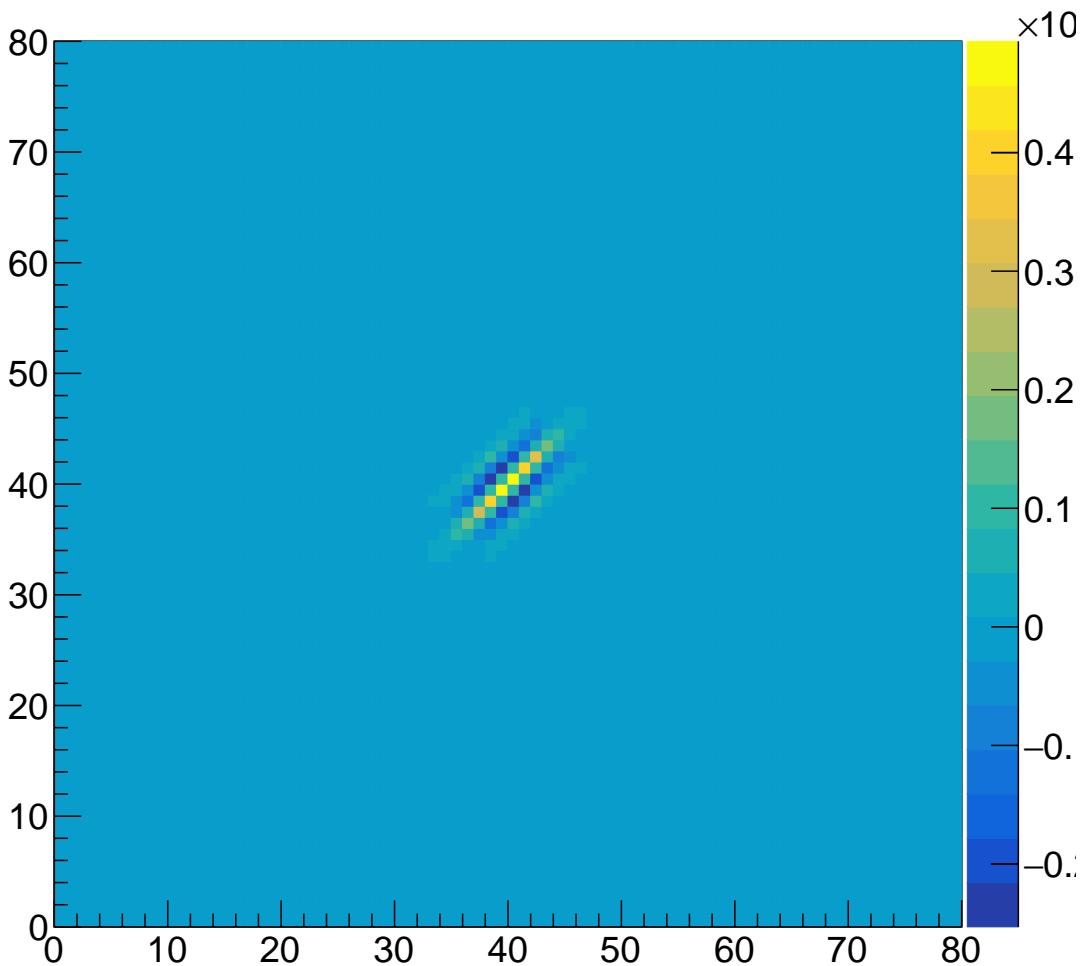


Figura 6.10: Matrice di covarianza bin per bin con metodo SVD

Da un punto di vista qualitativo possiamo affermare che l'unfolding con regolarizzazione SVD sia quello che meglio ricostruisce la distribuzione teorica come confermato dal ratio plot. Il rapporto in questo caso è costante a circa uno per ogni bin nell'area centrale della distribuzione e subisce, relativamente alla scala, oscillazioni minori. L'andamento si presenta più stabile anche per i bin a bassa statistica, relativamente agli altri algoritmi.

Come contro, il grafico di covarianza per i bin è una diagonale al posto di riempire tutto lo spazio bidimensionale. La correlazione è quindi poco sensibile ai bin esterni e dipende solo dal bin stesso, ponendo maggiore enfasi sui bin centrali ad alta statistica. Questo potrebbe portare a problemi come è stato, storicamente, imputato.

In linea di massima gli algoritmi ricostruiscono la distribuzione teorica in modo soddisfacente sotto la sola ipotesi di efficienza di risoluzione finita del detector con uno smearing in valore assoluto minore della larghezza dei bin dell'istogramma.

## 6.4 Comportamento ad alte $\sigma$

Per testare le possibili applicazioni dell'unfolding ad istogrammi con smearing dei dati sperimentali, supponiamo che la variabile  $\theta$  sia misurata con una risoluzione molto bassa, in particolare supponiamo che lo smearing gaussiano abbia una deviazione standard di  $\sigma = \{0.1, 0.3, 0.5\}$ . Da notare che essendo il range della variabile  $\theta \in [-\frac{\pi}{2}, \frac{\pi}{2}]$  una risoluzione di 0.3 rappresenta una situazione irrealistica, l'esempio è riportato esclusivamente a scopo illustrativo dell'algoritmo.

Generiamo i dati usando una dummy variabile *experiment\_dummy*. In questo modo supponiamo che la matrice di risposta sia costruita su un esperimento mentre l'unfolding sia eseguito su misure indipendenti per testare la consistenza dell'algoritmo. Generiamo quindi 3 esperimenti con smearing differenti, costruiamo le matrici di risposta e unfoldiamo la teoria:

```
RooUnfoldResponse response1(bins,xmin,xmax); //Unfolding
RooUnfoldResponse response2(bins,xmin,xmax); //Unfolding
RooUnfoldResponse response3(bins,xmin,xmax); //Unfolding

TH1D* Intensity_measured1 = new TH1D("Intensity Measured1",
"Intensity Measured1", bins, xmin, xmax);
TH1D* Intensity_measured2 = new TH1D("Intensity Measured2",
"Intensity Measured2", bins, xmin, xmax);
TH1D* Intensity_measured3 = new TH1D("Intensity Measured3",
"Intensity Measured3", bins, xmin, xmax);

double* clean_dummy = experiment_dummy.diffraction_tc(xmin,xmax,ymin,ymax);

//-----0.1-----
experiment.set_smear(0.1);
experiment_dummy.set_smear(0.1);
double* smeared_to_response1 = experiment.diffraction_smeared();
double* smeared_to_unfold = experiment_dummy.diffraction_smeared();

for(int i = 0; i < N; i++){
    Intensity_measured1->Fill(smeared_to_unfold[i]);
    response1.Fill(smeared_to_response1[i],clean[i]);
}

Intensity_measured1->Scale(1./m);

std::cout<< "Bayes" << std::endl;
RooUnfoldBayes unfold1(&response1, Intensity_measured1, 4);

//-----0.3-----
experiment.set_smear(0.3);
```

```

experiment_dummy.set_smear(0.3);
double* smeared_to_response2 = experiment.diffraction_smeared();
double* smeared_to_unfold2 = experiment_dummy.diffraction_smeared();

for(int i = 0; i < N; i++){
    Intensity_measured2->Fill(smeared_to_unfold2[i]);
    response2.Fill(smeared_to_response2[i], clean[i]);
}

Intensity_measured2->Scale(1./m);

std::cout<< "Bayes" << std::endl;
RooUnfoldBayes unfold2(&response2, Intensity_measured2, 4);

//-----0.5-----
experiment.set_smear(0.5);
experiment_dummy.set_smear(0.5);
double* smeared_to_response3 = experiment.diffraction_smeared();
double* smeared_to_unfold3 = experiment_dummy.diffraction_smeared();

for(int i = 0; i < N; i++){
    Intensity_measured3->Fill(smeared_to_unfold3[i]);
    response3.Fill(smeared_to_response3[i], clean[i]);
}

Intensity_measured3->Scale(1./m);

std::cout<< "Bayes" << std::endl;
RooUnfoldBayes unfold3(&response3, Intensity_measured3, 4);

```

Plottiamo quindi le distribuzioni con smearing rispetto alla distribuzione teorica:

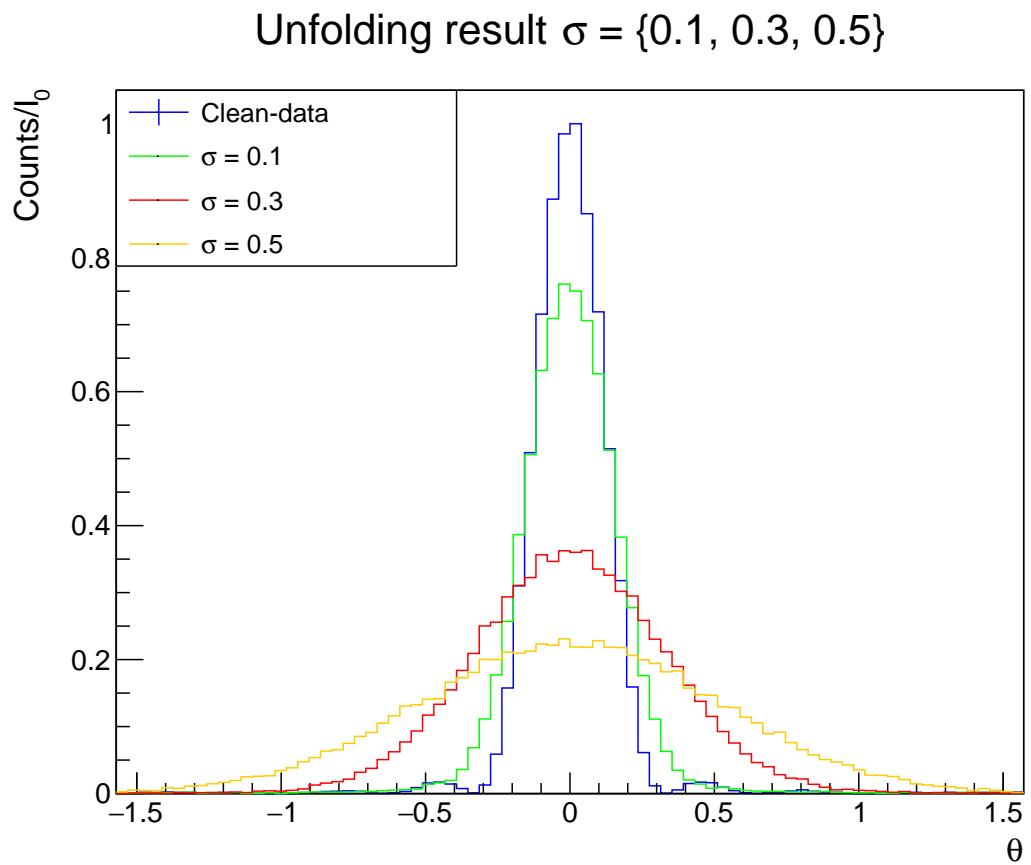


Figura 6.11: Smearing ad alte  $\sigma$ . Notiamo che perdiamo qualsiasi informazione sui picchi secondari. L'effetto di diffrazione è scomparso e ci porterebbe erroneamente a pensare ad un comportamento diverso della luce quando passa attraverso una fenditura circolare.

L'unfolding è quindi eseguito in modo identico al precedente utilizzando solamente il metodo Bayes per la definizione del parametro di regolarizzazione. Il risultato è plottato nella seguente immagine:

### Unfolding result $\sigma = \{0.1, 0.3, 0.5\}$

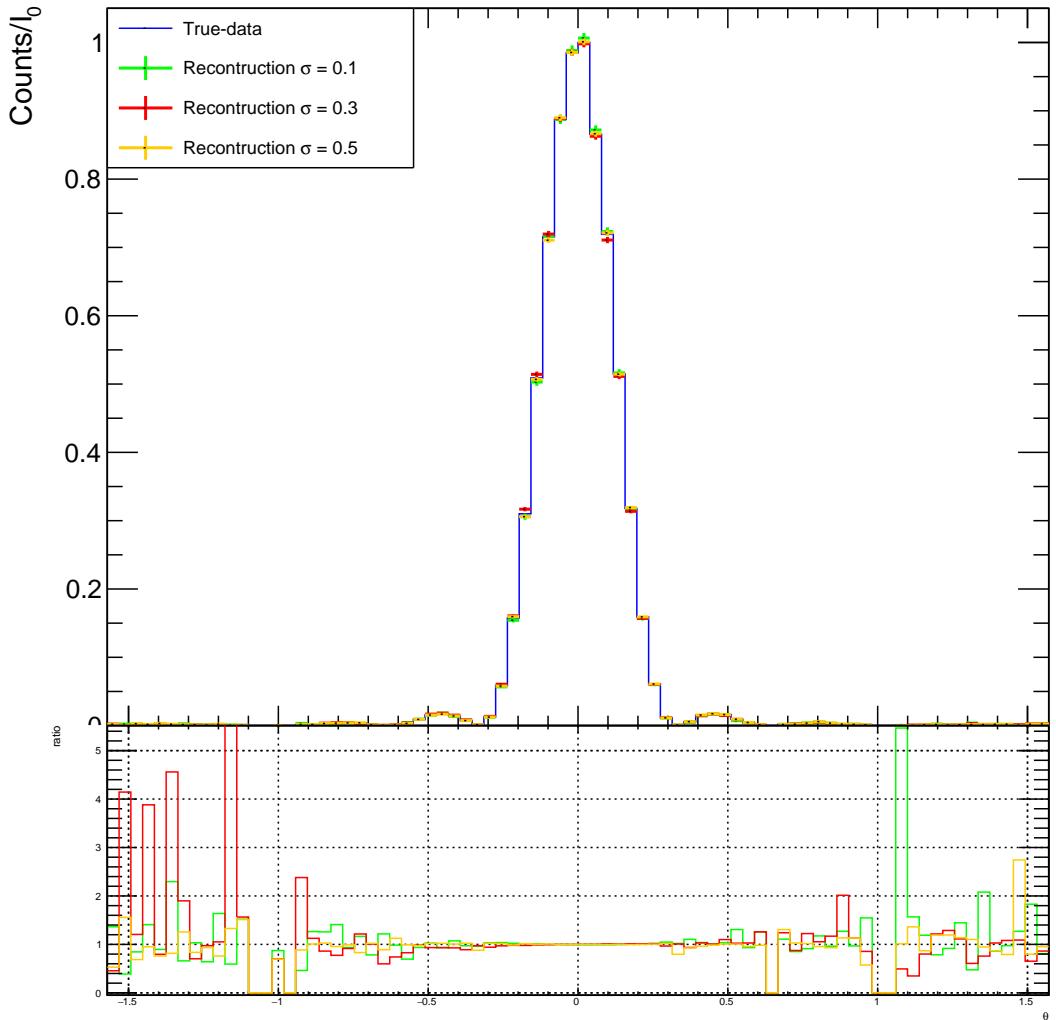


Figura 6.12: Unfolding per distribuzioni con smearing ad alto  $\sigma$ . Notiamo come la ricostruzione è quasi perfetta nei bin più popolati anche se gli esperimenti per definizione della matrice di covarianza e per l'unfolding stesso sono indipendenti.

Si nota come l'algoritmo ricostruisca quasi perfettamente la distribuzione teorica per i bin più popolati. Bin con poca statistica tuttavia sono ricostruiti in modo peggiore probabilmente a causa della perdita di informazione durante il binnaggio che incide, in modo maggiore, sui bin a bassa statistica.

# Capitolo 7

## Esercizio 7

Esercizio su classificazione con tecniche di analisi multivariata.

Viene proposto di generare due sample di dati gaussiani bi dimensionali, uno rappresentate il segnale e uno il fondo nel modo seguente:

$$pdf(\bar{x}) = \mathcal{N}(\mu, \Sigma) = \frac{1}{2\pi\sqrt{|\Sigma|}} e^{-\frac{1}{2}(\bar{x}-\bar{\mu})^T \Sigma^{-1} (\bar{x}-\bar{\mu})}$$

Dove  $\Sigma$  rappresenta la matrice di covarianza e  $|\Sigma|$  il suo determinante. Detto  $\rho = Cov[X, Y]/\sigma_X\sigma_Y$  il coefficiente di correlazione lineare di Pearson, possiamo scrivere la seguente espressione:

$$|\Sigma| = \det \begin{bmatrix} \sigma_X^2 & \rho\sigma_X\sigma_Y \\ \rho\sigma_X\sigma_Y & \sigma_Y^2 \end{bmatrix} = \sigma_X^2\sigma_Y^2(1 - \rho^2)$$

$\Sigma^{-1}$  è l'inversa della matrice di covarianza.

Dai dati dell'esercizio sappiamo che la pdf di segnale è caratterizzata dai seguenti parametri:

$$\mu_S = (0, 0) , \quad \sigma_{SX} = \sigma_{SY} = 0.3 , \quad \rho_S = 0.5$$

Mentre la pdf del fondo è caratterizzata dai parametri:

$$\mu_B = (4, 4) , \quad \sigma_{BX} = \sigma_{BY} = 1 , \quad \rho_B = 0.4$$

In entrambi i casi la matrice di covarianza è simmetrica e reale. Inoltre essa è definita positiva, il che ci assicura che ammette l'inversa  $\Sigma^{-1}$  con espressione:

$$\Sigma^{-1} = \frac{1}{|\Sigma|} \begin{bmatrix} \sigma_Y^2 & -\rho\sigma_X\sigma_Y \\ -\rho\sigma_X\sigma_Y & \sigma_X^2 \end{bmatrix}$$

Vengono quindi generati dei random sample con le distribuzioni prima descritte grazie al metodo accept or reject discusso in quanto segue.

Verrà usato, come generatore di numeri pseudo-casuali, il generatore xorshiro128plus, descritto ampiamente nell' Esercizio 1.

Al fine di generare numeri casuali, viene istanziato il generatore come oggetto della classe `xorshiro`. Viene quindi definita una funzione `rand_uni` con input gli estremi dell'intervallo in cui si vogliono generare i numeri pseudo-casuali. All'interno della funzione si genera quindi un numero pseudo-casuale grazie al metodo `rand` della classe `xorshiro`.

```

//genero numeri casuali con algoritmo xorshiro128p
xorshiro random_number; //Nota istanza da fare fuori dalla funzione!

double rand_uni(double min = 0., double max = 1.){

    return random_number.rand(min,max);

}

```

Viene quindi definita una **struct** per contenere il risultato del sampling gaussiano avente due coordinate  $x,y$

```

//struct punti x e y
struct xy{
    double x;
    double y;
};

```

Viene inoltre definita la funzione gaussiana bidimensionale, che restituisce il valore della pdf rispetto al vettore in input  $[x, y]$  e ai parametri  $\bar{\mu}, \sigma_x, \sigma_y, \rho$ :

```

//funzione gaussiana 2d
double gaussian2d(double x, double y, double mux,double muy,
                  double sx, double sy,double rho){

    double t_rho = 1 - rho*rho;
    double norm = 1./(2*M_PI * sx*sy*sqrt(t_rho));
    double t_mux = (x - mux)/sx;
    double t_muy = (y - muy)/sy;
    double exponent = (-1./(2*t_rho))*(t_mux*t_mux + t_muy*t_muy -
                                         2*rho*(t_mux*t_muy));

    return norm*exp(exponent);

}

```

Il metodo di sampling accept or reject viene implementato quindi nella funzione **gauss\_tc** la quale ritorna una struct *xy* contenente le coordinate dei punti. Tale funzione genera una tripletta di numeri casuali  $x,y,z$  grazie al generatore **xorshiro128plus** prima definito. Viene quindi valutata la *pdf* gaussiana nei punti  $x, y$ . Se il risultato della valutazione è maggiore di  $z$  si continua a generare la tripletta, altrimenti i punti  $x, y$  vengono salvati negli attributi della **struct** e vengono restituiti, terminando quindi la funzione:

```

//genero coppie di numeri che seguono distribuzione gauss2d
//con try and catch. Nota ritorno una struct.
xy gauss_tc(double mux,double muy, double sx, double sy,double rho,
            double xmin, double xmax, double ymin, double ymax){

```

```

//istanzia contenitore per le coordinate
xy rndxy;

//ciclo finchè rndGauss < rndz
while(1){

    //genero x,y pseudo-casuali
    rndxy.x = rand_uni(xmin,xmax);
    rndxy.y = rand_uni(ymin,ymax);

    //genero z casuale tra [0,1]
    double rndz = rand_uni();

    //valuto gaussiana in x,y
    double rndGauss = gaussian2d(rndxy.x,rndxy.y,mux,muy,sx,sy,rho);

    //accept or reject
    if(rndGauss > rndz){

        //restituisce struct punti x,y
        return rndxy;
    }

}

}

```

Nel main vengono quindi istanziati i parametri per le distribuzioni di segnale e fondo e il numero di sample da generare, posto a 10000 sia per il fondo che per il segnale:

```

int main(int argc, char** argv){

    //Signal gaussian params
    double mux_sig = 0.;
    double muy_sig = 0.;
    double sx_sig = 0.3;
    double sy_sig = 0.3;
    double rho_sig = 0.5;

    //Background gaussian params
    double mux_bkg = 4.;
    double muy_bkg = 4.;
    double sx_bkg = 1;
    double sy_bkg = 1;
    double rho_bkg = 0.4;
}

```

```

int N_sig = 10000;
int N_bkg = 10000;

```

Vengono creati due oggetti della classe `TTree` di ROOT che altro non sono che dataset organizzati in branches e leaves, uno per il segnale e uno per il fondo. Questa operazione è necessaria nel caso si vogliano passare i dati agli algoritmi di analisi multivariata della classe `TMVA` di ROOT.

```

double xentry_sig;
double yentry_sig;
double xentry_bkg;
double yentry_bkg;

//TMVA Legge dati dai tree, ne creo uno con dentro le variabili
//x e y per segnale e fondo.
TTree* tree_sig = new TTree("Signal","Signal");
tree_sig->Branch("x", &xentry_sig);
tree_sig->Branch("y", &yentry_sig);

TTree* tree_bkg = new TTree("Background","Background");
tree_bkg->Branch("x",&xentry_bkg);
tree_bkg->Branch("y",&yentry_bkg);

//istanzio un TFile per l'output della generazione
TFile* outfile = new TFile("dataset.root", "RECREATE");

```

Si cicla quindi sul numero degli eventi e si generano i dati distribuiti come gaussiane con parametri per il fondo e per il segnale.

```

xy xy_sig; //Struct per il segnale
xy xy_bkg; //Struct per il fondo
for(int i = 0; i < N_sig; i++){

    //accept or reject, restituisce struct di punti
    //x e y distribuiti secondo il segnale
    xy_sig = gauss_tc(mu_x_sig,mu_y_sig,sigma_x_sig,sigma_y_sig,
                      rho_xy,xmin,xmax,ymin,ymax);

    //assegniamo all'indirizzo delle variabili
    //del TTree i valori generati.
    xentry_sig = xy_sig.x;
    yentry_sig = xy_sig.y;

    //Riempiamo il tree
    tree_sig->Fill();
}

//analogo per il background

```

```

for(int i = 0; i < N_bkg; i++){
    xy_bkg = gauss_tc(mux_bkg,muy_bkg,sx_bkg,sy_bkg,
                       rho_bkg,xmin,xmax,ymin,ymax);

    xentry_bkg = xy_bkg.x;
    yentry_bkg = xy_bkg.y;

    tree_bkg->Fill();
}

```

Il risultato della generazione viene salvato oltretutto il un oggetto TH2F e viene quindi plottato su un canvas.

Il risultato della generazione è il seguente:

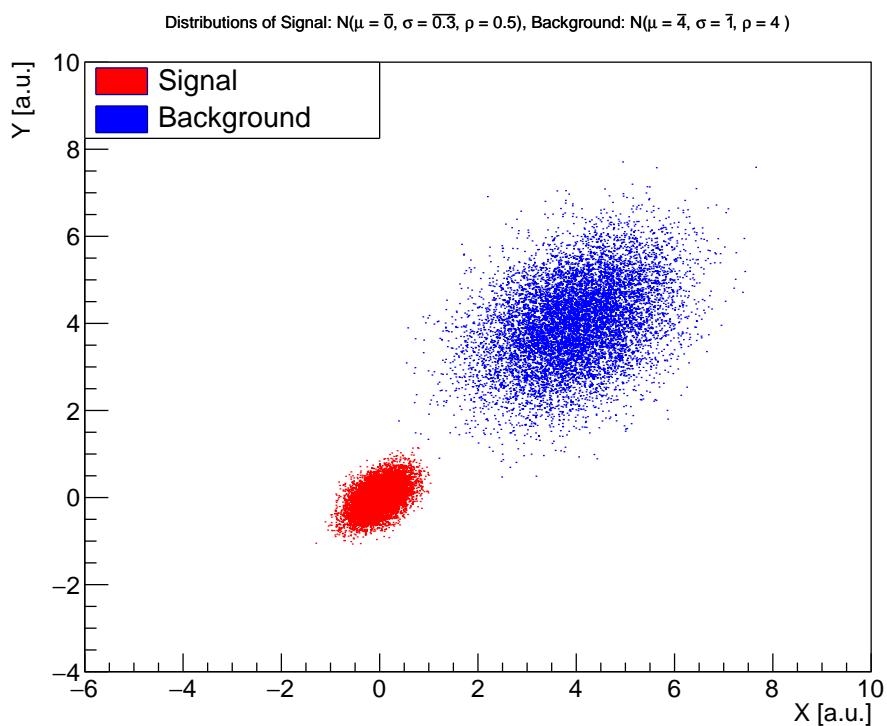


Figura 7.1: Il dataset generato, con distinzione in colore per segnale  $\bar{x}_{sig} \sim \mathcal{N}(\bar{\mu} = [0, 0], \bar{\sigma} = [0.3, 0.3], \rho = 0.5)$  (rosso) e fondo  $\bar{x}_{bkg} \sim \mathcal{N}(\bar{\mu} = [4, 4], \bar{\sigma} = [1, 1], \rho = 0.4)$  (blu)

## 7.1 Modelli Deterministici

I metodi deterministici si propongono di risolvere problemi di classificazione sfruttando equazioni risolvibili in forma analitica e quindi direttamente applicabili ai dati. Requisito importante e limitante di tali modelli è che le distribuzioni dei dati devono essere note a priori. Tra questi modelli troviamo il discriminante lineare e

il discriminante quadratico.

Considerato un sample di osservazioni  $\bar{x}$  con un informazione di classe  $y$  conosciuta a priori. Il sample è chiamato in genere training set. Il problema di classificazione è quindi quello di trovare un predittore ottimale della classe  $y$  date delle osservazioni  $\bar{x}$ .

I discriminanti lineare e quadratico affrontano il problema nel caso in cui le distribuzioni di densità di probabilità dei sample  $f(\bar{x}|y=0), f(\bar{x}|y=1)$  siano gaussiane con parametri  $(\mu_0, \Sigma_0), (\mu_1, \Sigma_1)$ . Per lemma di Neyman-Pearson, la miglior statistica di test, date due ipotesi semplici  $H_0, H_1$  è il rapporto delle likelihood:

$$t(\bar{x}) = \frac{f(\bar{x}|H_0)}{f(\bar{x}|H_1)} \geq c$$

Ai fini computazionali, in genere si usa il logaritmo della statistica. Essendo il logaritmo funzione monotona, le qualità della statistica rimangono invariate dalla trasformazione:

$$\lambda(\bar{x}) = \log t(\bar{x}) = \log f(\bar{x}|H_0) - \log f(\bar{x}|H_1) \geq T$$

Se le ipotesi  $H_0, H_1$  sono ipotesi semplici e le osservazioni  $x_i$  sono indipendenti allora la likelihood è la produttoria delle *pdf* delle variabili causali  $x_i$ .

Considerando il fatto che le ipotesi semplici  $H_0, H_1$  sono gaussiane possiamo scrivere:

$$\begin{aligned} \lambda(\bar{x}) &= -\frac{1}{2}(\bar{x} - \bar{\mu}_0)^T \Sigma_0^{-1}(\bar{x} - \bar{\mu}_0) - \log(2\pi\sqrt{|\Sigma_0|}) + \\ &\quad + \frac{1}{2}(\bar{x} - \bar{\mu}_1)^T \Sigma_1^{-1}(\bar{x} - \bar{\mu}_1) + \log(2\pi\sqrt{|\Sigma_1|}) \geq T \end{aligned} \tag{7.1}$$

Senza ulteriori assunzioni questa è la formula per il discriminante quadratico. Un caso particolare è il discriminante lineare di Fischer che agisce sotto l'ulteriore ipotesi di omoschedasticità ovvero  $\Sigma_0 = \Sigma_1 = \Sigma$ .

In tale caso sono possibili semplificazioni:

$$\lambda(\bar{x}) = 2\bar{\mu}_0^T \Sigma^{-1} \bar{x} - 2\bar{\mu}_1^T \Sigma^{-1} \bar{x} + \bar{\mu}_1^T \Sigma^{-1} \bar{\mu}_1 - \bar{\mu}_0^T \Sigma^{-1} \bar{\mu}_0 \geq 2T$$

$$(\bar{\mu}_0 - \bar{\mu}_1)^T \Sigma^{-1} \bar{x} \geq \frac{1}{2}(2T - \bar{\mu}_1^T \Sigma \bar{\mu}_1 + \bar{\mu}_0^T \Sigma \bar{\mu}_0)$$

Da notare che la LHS è uno scalare possiamo scrivere:

$$\lambda(\bar{x}) = \bar{w}^T \bar{x} \geq \hat{T}$$

$$\bar{w} = \Sigma^{-1}(\bar{\mu}_0 - \bar{\mu}_1)$$

$$\hat{T} = \frac{1}{2}(2T - \bar{\mu}_1^T \Sigma \bar{\mu}_1 + \bar{\mu}_0^T \Sigma \bar{\mu}_0)$$

Quindi nel caso di omoschedasticità e di ipotesi gaussiane, il discriminante ottimale è una combinazione lineare delle osservabili  $\bar{x}_i$

Questo non è il nostro caso in quanto  $\sigma_{XS} = \sigma_{YS} \neq \sigma_{XB} = \sigma_{YB}$  quindi il discriminante non è ottimale per lemma di Neyman Pearson.

I discriminanti lineare e quadratico sono classificatori naive in quanto la regione di separazione può essere calcolata in forma chiusa, non dipende da parametri liberi. Questa semplicità tuttavia porta a modelli molto rigidi. In particolare il discriminatore lineare è il modello di classificazione più rigido in assoluto, quello quadratico è leggermente più sensibile.

Per quanto riguarda il nostro dataset, il discriminante lineare di Fischer non può essere applicato. Tuttavia vedremo un esempio di classificatore che agisce con allo stesso modo, con l'unica differenza che  $\bar{w}$  non è dato a priori (dalle matrici di covarianza e dalle medie delle gaussiane) ma sarà appreso dal modello in maniera da massimizzare il rapporto tra le likelihood per quanto possibile.

## 7.2 Discriminante quadratico

Per trovare il confine di separazione ottimale nel caso di distribuzioni normali con medie e matrici di covarianza differenti supponiamo genericamente:

$$\bar{\mu}_0 < \bar{\mu}_1 , \quad \Sigma_0 \neq \Sigma_1$$

Per un punto  $\bar{x}$  generico, avremo una probabilità non nulla di commettere un errore nella predizione della classe di appartenenza denominata  $y_0$  oppure  $y_1$ . Ad un punto  $\bar{x}^*$  questa probabilità è uguale per le due classi perciò  $\bar{x}^*$  è sul confine di separazione:

$$\bar{\mu}_0 < \bar{x}^* < \bar{\mu}_1$$

. La probabilità di errore è quindi:

$$P(error) = P(\bar{x} > \bar{x}^*, \bar{x} \in y_0) + P(\bar{x} < \bar{x}^*, \bar{x} \in y_1)$$

Dal momento che:

$$P(A, B) = P(A|B)P(B)$$

Possiamo affermare:

$$P(error) = P(\bar{x} > \bar{x}^* | \bar{x} \in y_0)P(\bar{x} \in y_0) + P(\bar{x} < \bar{x}^* | \bar{x} \in y_1)P(\bar{x} \in y_1)$$

Al fine di ottenere un'equazione ottima per il confine decisionale, vogliamo che l'errore sia minimo:

$$\frac{\partial P(error)}{\partial \bar{x}^*} = 0 , \quad \bar{x}^* = minimum$$

Dalla definizione di distribuzione cumulativa di probabilità (CDF) possiamo esprimere la probabilità di commettere un errore come:

$$P(\bar{x} < c | \bar{x} \in y_0) = F_0(c) \rightarrow P(\bar{x} > \bar{x}^* | \bar{x} \in y_0) = 1 - F_0(\bar{x}^*)$$

$$P(\bar{x} < \bar{x}^* | \bar{x} \in y_1) = F_1(\bar{x}^*)$$

In accordo con la definizione di *pdf* scriviamo quindi la prior come:

$$P(\bar{x} \in y_0) = f_0(\bar{x}) = \pi_0$$

$$P(\bar{x} \in y_1) = f_1(\bar{x}) = \pi_1$$

Essendo per definizione:

$$f(x) = \frac{\partial F(x)}{\partial x}$$

Possiamo minimizzare quindi la probabilità di errore:

$$\begin{aligned} \frac{\partial P(error)}{\partial \bar{x}^*} &= -f_0(\bar{x}^*)\pi_0 + f_1(\bar{x}^*)\pi_1 = 0 \\ f_0(\bar{x}^*)\pi_0 &= f_1(\bar{x}^*)\pi_1 \end{aligned}$$

Sotto assunzione di prior uguale per le due classi allora il confine è totalmente identificato dall'equazione:

$$f_0(\bar{x}^*) = f_1(\bar{x}^*) \quad (7.2)$$

Nel caso di distribuzioni normali per le due classi  $y_0, y_1$  l'equazione del confine è quadratica nel caso di non omoschedasticità.

Prendendo il logaritmo dell'equazione (1.2) allora otterremo esattamente la formula per il calcolo del rapporto tra le likelihood dell'equazione (1.1) che riportiamo di seguito in forma più compatta:

$$\bar{x}^T(\Sigma_1^{-1} - \Sigma_0^{-1})\bar{x} + 2(\Sigma_0^{-1}\bar{\mu}_0 - \Sigma_1^{-1}\bar{\mu}_1)^T\bar{x} + (\bar{\mu}_1^T\Sigma_1^{-1}\bar{\mu}_1 - \bar{\mu}_0^T\Sigma_0^{-1}\bar{\mu}_0) + \ln\left(\frac{|\Sigma_1|}{|\Sigma_0|}\right) = 0 \quad (7.3)$$

Nel nostro caso bi dimensionale, la superficie sarà una conica, più precisamente un ellisse esprimibile tramite un polinomio:

$$ax^2 + 2bxy + cy^2 + 2dx + 2fy + g = 0$$

Nel nostro caso, l'ellisse ha equazione:

$$-13.62x^2 + 13.858xy - 13.62y^2 - 5.714x - 5.714y + 27.7986 = 0$$

Il codice per ricavare i parametri dell'ellisse riporta semplicemente il calcolo generico della formula (1.1). Per il calcolo dei vertici, dei semi-assi e dell'angolo di inclinazioni sono state usate le seguenti formule:

$$\begin{aligned} x_0 &= \frac{cd - bf}{b^2 - ac} \quad , \quad y_0 = \frac{af - bd}{b^2 - ac} \\ r_{maj} &= \sqrt{\frac{2(af^2 + cd^2 + gb^2 - 2bdf - acg)}{(b^2 - ac)[\sqrt{(a - c)^2 + 4b^2} - (a + c)]}} \\ r_{min} &= \sqrt{\frac{2(af^2 + cd^2 + gb^2 - 2bdf - acg)}{(b^2 - ac)[- \sqrt{(a - c)^2 + 4b^2} - (a + c)]}} \\ \tan(2\theta) &= \frac{b}{a - c} \end{aligned}$$

Viene definita una struct `ellipse` che conterrà le coordinate del centro, il valore delle lunghezze dei semiassi e l'angolo di inclinazione. Si noti che se  $a = c$  allora l'angolo di inclinazione dell'ellisse è  $\frac{\pi}{4}$ . La funzione `quadratic_classifier` esegue le computazioni con i parametri delle gaussiane segnale e fondo e restituisce la struct dell'ellisse:

```

struct ellipse{

    double x0;
    double y0;
    double theta;
    double r1;
    double r2;
};

ellipse quadratic_discriminator(double mux_s,double muy_s, double sx_s,
double sy_s,double rho_s, double mux_b,double muy_b, double sx_b,
double sy_b,double rho_b){

    //determinanti matrici covarianza
    //segnale e fondo
    double det_s = pow(sx_s,2)*pow(sy_s,2)-
                  -pow(rho_s,2)*pow(sx_s,2)*pow(sy_s,2);
    double det_b = pow(sx_b,2)*pow(sy_b,2)-
                  -pow(rho_b,2)*pow(sx_b,2)*pow(sy_b,2);

    // x^2 coeff
    double a = pow(sy_b,2)/det_b -pow(sy_s, 2)/det_s;
    // y^2 coeff
    double c = pow(sx_b,2)/det_b -pow(sx_s, 2)/det_s;
    // 2*xy coeff
    double b = 2*(-(rho_b*sx_b*sy_b)/det_b + (rho_s*sx_s*sy_s)/det_s);
    // 2*x coeff
    double d = ((2/det_s)*(mux_s*sy_s*sy_s-rho_s*sx_s*sy_s*muy_s))-
                  ((2/det_b)*(mux_b*sy_b*sy_b-rho_b*sx_b*sy_b*muy_b));
    // 2*y coeff
    double f = ((2/det_s)*(muy_s*sy_s*sy_s-rho_s*sx_s*sy_s*mux_s))-
                  -((2/det_b)*(muy_b*sy_b*sy_b-rho_b*sx_b*sy_b*mux_b));
    // costante
    double g = (1/det_b)*mux_b*mux_b*sy_b*sy_b -
                  -(1/det_s)*mux_s*mux_s*sy_s*sy_s+
                  +(2/det_b)*(rho_b*sx_b*sy_b)*muy_b*mux_b-
                  -(2/det_s)*(rho_s*sx_s*sy_s)*muy_s*mux_s +
                  +(1/det_b)*sx_b*sx_b*muy_b*muy_b -
                  -(1/det_s)*sx_s*sx_s*muy_s*muy_s +log(det_b/det_s);

    // troviamo i parametri veri
b /= 2;
d /= 2;
f /= 2;
}

```

```

double x0 = (c*d-b*f)/(b*b-a*c);
double y0 = (a*f-b*d)/(b*b-a*c);

double r_one = sqrt((2*(a*f*f+c*d*g*b*b-2*b*d*f-a*c*g))/(
    ((b*b-a*c)*(sqrt(pow(a-c,2)+4*b*b)-(a+c)))); 
double r_two = sqrt((2*(a*f*f+c*d*g*b*b-2*b*d*f-a*c*g))/(
    ((b*b-a*c)*(-sqrt(pow(a-c,2)+4*b*b)-(a+c))));

double theta;

if (a == c){
    theta = 45. ;
}

ellipse result;
result.x0 = x0;
result.y0 = y0;
result.theta = theta;
result.r1 = r_one;
result.r2 = r_two;

return result;
}

```

Viene plottata quindi l'ellisse di attraverso la classe `TEllipse` di ROOT:

```

ellipse disc = quadratic_discriminator(mux_sig, muy_sig, sx_sig, sy_sig,
                                         rho_sig, mux_bkg, muy_bkg, sx_bkg, sy_bkg, rho_bkg);
TEllipse* el = new TEllipse(disc.x0, disc.y0, disc.r2, disc.r1,
                            0, 360, disc.theta);
el->SetFillStyle(0);

```

Il risultato viene plottato quindi assieme ai dati su un canvas.  
Va sottolineato che la regola di classificazione sarà quindi la seguente:

$$ax^2 + 2bxy + cy^2 + 2dx + 2fy + g < 0 \rightarrow \text{background}$$

$$ax^2 + 2bxy + cy^2 + 2dx + 2fy + g \geq 0 \rightarrow \text{signal}$$

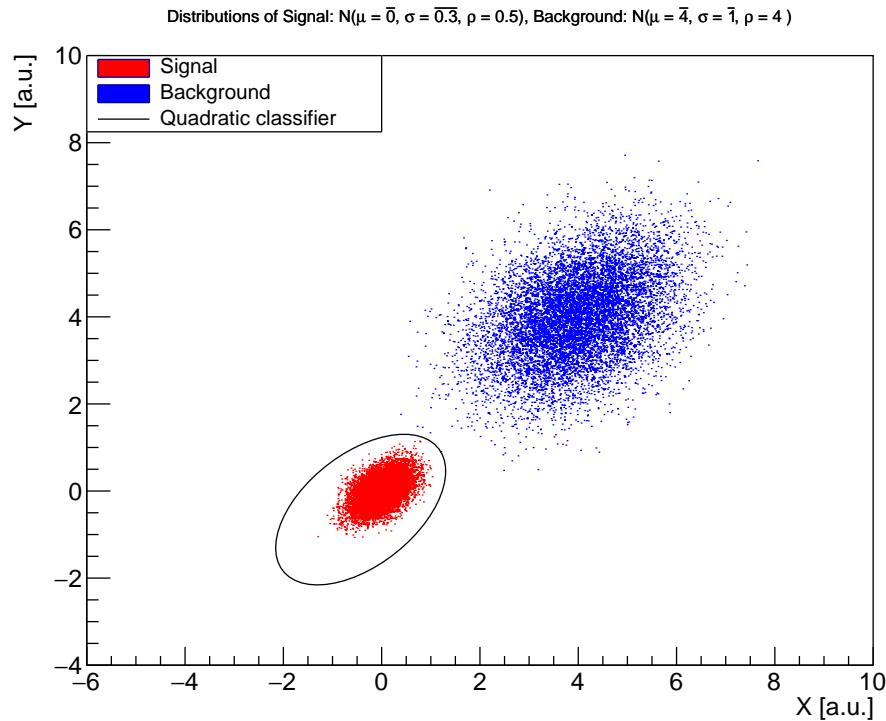


Figura 7.2: Plot del dataset segnale e fondo con distribuzione gaussiana assieme al decision boundary del classificatore quadratico. I punti interni alla circonferenza ( $>0$ ) saranno classificati come segnale mentre quelli esterni ( $<0$ ) saranno classificati come fondo.

Analizziamo quindi i risultati ottenuti con questo modello. La metrica che useremo, essendo questa una classificazione binaria, sarà l'accuracy:

$$acc = \frac{TP + TN}{P + N}$$

Ovvero la somma delle istanze di segnale e di fondo che l'algoritmo ha predetto correttamente ( $TP+TN$ ) diviso la somma delle istanze totali nel dataset. In pratica è la frazione delle istanze che l'algoritmo ha predetto correttamente su tutto il dataset.

L'errore commesso dall'algoritmo è misurato con il rate di errore che altro non è che la frazione del totale delle istanze classificate erroneamente:

$$E = 1 - acc = \frac{P + N - TP - TN}{P + N}$$

Costruiamo una funzione che valuta ogni istanza creata secondo l'equazione dell'ellisse. La funzione restituisce il valore uno o zero a seconda che la funzione sia valutata negativa o positiva:

```
int disc_evaluator(ellipse params, double x, double y){

    if(params.a*x*x + 2*params.b*x*y +params.c*y*y+ 2*params.d*x+
    2*params.f*y +params.g >= 0){
```

```

        return 0;
    }
    else{
        return 1;
    }
}

```

Al momento di generazione dei dati nel main verrà chiamata la funzione e verrà incrementato, sia per segnale che per il fondo il valore dei  $TP, TN$ :

```

ellipse disc = quadratic_discriminator(mux_sig, muy_sig, sx_sig,
                                         sy_sig, rho_sig, mux_bkg, muy_bkg, sx_bkg, sy_bkg, rho_bkg);

int signal_true = 0;
int signal_neg = 0;
int bkg_true = 0;
int bkg_neg = 0;

for(int i = 0; i < N_sig; i++){

    xy_sig = gauss_tc(mux_sig,muy_sig,sx_sig,sy_sig,
                       rho_sig,xmin,xmax,ymin,ymax);

    xentry_sig = xy_sig.x;
    yentry_sig = xy_sig.y;

    if(disc_evaluator(disc, xy_sig.x, xy_sig.y) == 0){
        signal_true += 1;
    }
    else{
        signal_neg += 1;
    }

    gauss_sig->Fill(xy_sig.x,xy_sig.y);
    tree_sig->Fill();
}

std::cout<< "True positive: " << signal_true << " " <<
          "Accuracy sig: " << signal_true/N_sig << std::endl;
std::cout<< "False negative: " << signal_neg << " " <<
          "Test Error Rate sig: " << signal_neg/N_sig << std::endl;

for(int i = 0; i < N_bkg; i++){

    xy_bkg = gauss_tc(mux_bkg,muy_bkg,sx_bkg,sy_bkg,
                       rho_bkg,xmin,xmax,ymin,ymax);
}

```

```

xentry_bkg = xy_bkg.x;
yentry_bkg = xy_bkg.y;

if(disc_evaluator(disc, xy_bkg.x, xy_bkg.y) == 1){
    bkg_true += 1;
}
else{
    bkg_neg += 1;
}

gauss_bkg->Fill(xy_bkg.x,xy_bkg.y);
tree_bkg->Fill();
}
std::cout << "True negative: " << signal_true << " " <<
    "Accuracy bkg: " << signal_true/N_sig << std::endl;
std::cout << "False positive: " << signal_neg << " " <<
    "Test Error Rate bkg : " << signal_neg/N_sig << std::endl;

std::cout << " Total accuracy in classifying sig+bkg: " <<
    (signal_true+bkg_true)/(N_sig+N_bkg) << std::endl;
std::cout << " Total error rate in classifying sig+bkg: " <<
    1 - (signal_true+bkg_true)/(N_sig+N_bkg) << std::endl;

```

Il risultato da terminale è quindi il seguente:

```

True positive: 10000 Accuracy sig: 1
False negative: 0 Test Error Rate sig: 0

```

```

True negative: 10000 Accuracy bkg: 1
False positive: 0 Test Error Rate bkg : 0

```

```

Total accuracy in classifying sig+bkg: 1
Total error rate in classifying sig+bkg: 0

```

Due metodi di visualizzazione dei risultati di un classificatore binario sono la confusion matrix e la roc curve.

La prima plotta in una matrice 2x2 i seguenti valori:  $TP, TN, FP, FN$ .

La roc curve plotta il  $TP$  in funzione del  $FP$  a varie soglie di classificazione. L'area sottesa alla curva è l'accuracy del modello.

Tali metodi di visualizzazione sono tuttavia npn necessari in quanto il classificatore è ideale in questa situazione. Il modello predice perfettamente le classi di segnale e background sotto le assunzioni fatte. Purtroppo nei casi realistici le *pdf* delle variabili spesso non sono conosciute e i metodi deteministici falliscono a causa della loro rigidità. Metodi approssimativi per stimare la statistica di test ottimale esistono e saranno spiegati successivamente.

## 7.3 Percettrone

Abbiamo visto che il discriminante lineare di Fischer è la migliore statistica di test. Il problema non soddisfa tuttavia l'assunzione di omoschedasticità del discriminante di Fischer che, in questo caso, perde le sue qualità.

Possiamo comunque trovare un approssimazione attraverso metodi non analitici del likelihood ratio:

$$t(\bar{x}) = \frac{f(\bar{x}|H_0)}{f(\bar{x}|H_1)}$$

Il percettrone fu uno dei primi tentativi di emulare il funzionamento dei neuroni umani, dando vita a una disciplina che oggi vede un periodo di massimo splendo, il machine learning.

Con machine learning intendiamo tutti gli studi scientifici di modelli statistici e probabilistici che i compilatori possono utilizzare per apprendere task specifiche senza essere specificamente programmati per farlo, affidandosi all'analisi dei pattern e all'inferenza statistica.

Il perceptrone ricade nella categoria dei modelli lineari. Esso utilizza una funzione non-lineare delle variabili in ingresso per produrre l'output.

Particolare da notare è che la forma funzionale del discriminatore è del tutto uguale a quella del discriminante lineare di Fischer ma con una differenza, mentre il discriminante di Fischer è esprimibile in forma chiusa dai parametri delle pdf dei dati di segnale e fondo, il percettrone ha invece dei parametri adattivi, che verranno raffinati in modo da approssimare il più possibile il rapporto tra le likelihood.

$$y(\bar{x}, \bar{w}) = f\left(\sum_{j=0}^M w_j \phi(\bar{x}_j)\right)$$

Dove  $\bar{x}$  rappresenta il vettore delle variabili di input,  $y$  rappresenta l'output o la predizione,  $\bar{w}$  è il vettore dei parametri adattivi del modello mentre  $\phi(\cdot)$  è la funzione non lineare dei parametri di input fissata e non adattiva. La funzione  $f(\cdot)$  è invece chiamata funzione di attivazione e, nel caso specifico del perceptrone essa assume la forma di un gradino di Heavyside:

$$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x < 0 \end{cases}$$

L'algoritmo di apprendimento dei parametri adattivi  $\bar{w}$  si basa sulla minimizzazione di un'appropriata funzione di errore o, in gergo, loss.

Supponendo di voler trovare un set di parametri tali che:

$$\bar{w}^T \phi(\bar{x}) > 0 \text{ if } \bar{x} \in y_0$$

$$\bar{w}^T \phi(\bar{x}) < 0 \text{ if } \bar{x} \in y_1$$

E supponendo di decidere dei target arbitrari  $t \in \{1, -1\}$  relativi alla classe  $y_0, y_1$  allora è ovvio che, per ogni istanza  $\bar{x}$  appartenente a  $y_0$  o  $y_1$ , vale la seguente disequazione  $\bar{w}^T \phi(\bar{x}) t > 0$ . Perciò il percettrone associa 0 a ogni pattern correttamente

classificato mentre, per ogni istanza classificata erroneamente, cerca di adattare i parametri in modo da minimizzare l'errore:

$$E_P = - \sum_{n \in M} \bar{w}^T \phi(\bar{x}_n) t_n$$

La funzione di costo è quindi lineare nella regione di misclassificazione mentre è identicamente zero nella regione di classificazione corretta.

Il metodo di aggiornamento dei parametri è chiamato Stochastic Gradient Descent (SGD) che nel caso semplice del percepitrone assume la seguente forma.

Sia  $\bar{w}^\tau$  il vettore dei parametri all'iterazione  $\tau$  e sia  $\bar{x}_n$  un'istanza di train presentata al percepitrone :

$$\bar{w}^{\tau+1} = \bar{w}^\tau - \eta \nabla E_P(\bar{w}) = \bar{w}^\tau + \eta \phi(\bar{x}_n) t_n$$

$\eta$  è detto learning rate del modello ed è un parametro fissato a piacere.

Il teorema del percepitrone ci assicura che tale algoritmo convergerà nel caso in cui i dati siano geometricamente linearmente separabili. Nel caso opposto non è garantita la convergenza (problema XOR).

Creiamo una classe `p_clf` con cui creare il classificatore. Creiamo inoltre una struct `line_bound` per contenere il confine decisionale lineare del modello.

```
#ifndef PERCEPTRONE_H
#define PERCEPTRONE_H
#include <vector>

struct line_bound{

    double m;
    double q;
};

class p_clf{
private:

    int dimension;
    int size;
    std::vector<double> weights;

public:

    p_clf(int d, int s){ size = s; dimension = d ; } ;
    ~p_clf(){};
    std::vector<double> get_w(){ return weights ; };
    void set_weights();
    int predict(double x, double y);
    void train(double x , double y, int label, double eta);
    line_bound get_boundary();
}
```

```

};

#endif // PERCETTRONE_H

```

Gli attributi privati prevedono le seguenti variabili:

- **dimension**: dimensione del dataset. Nel nostro caso siamo in 2 dimensioni.
- **size**: numero di istanze nel dataset di train.
- **weights**: parametri adattivi del modello percettrone.

I metodi della classe sono minimali e comprendono:

- **p\_clf(int d, int s)**: costruttore dell'oggetto. Vengono passate informazioni dimensionali sul dataset di train.
- **~p\_clf()**: distruttore dell'oggetto.
- **get\_w()**: restituisce il vettore dei parametri adattivi.
- **set\_weights()**: Necessario istanziare il vettore dei pesi prima di effettuare il train. La funzione istanzia i pesi random in [0,1], metodi più sofisticati (che portano a una convergenza più rapida dell'algoritmo) possono essere implementati.
- **predict(double x, double y)**: inferenza sull'istanza (x,y) passata in input.
- **train(double x, double y, int label, double eta)**: Viene presentata al modello un'istanza (x,y) del dataset. La funzione train aggiorna il vettore dei parametri in base all'algoritmo di apprendimento.
- **get\_boundary()**: restituisce la struct contenente i coefficienti del confine di separazione lineare.

Vengono illustrate velocemente i metodi della classe. Da notare che il vettore dei pesi ha dimensione maggiore di un'unità rispetto alla dimensione del train. Il valore in eccesso è chiamato **bias**:  $w_0$ :

```

void p_clf::set_weights(){

    //inizializza random il vettore dei pesi
    TRandom3* gen = new TRandom3();
    for(int i = 0; i < dimension+1; i++){
        weights.push_back(gen->Uniform(0,1));
    }

    return;
}

```

La predizione viene quindi fatta secondo la combinazione lineare dei parametri e dei dati in ingresso:

$$y_p = f(\bar{w}^T \bar{x} + bias)$$

Dove  $f$  è la funzione di heavyside, sarà uno se l'argomento della funzione è positivo mentre restituirà zero se l'argomento è negativo. L'output è quindi la predizione della classe dell'istanza:

```
int p_clf::predict(double x, double y){

    double activation = weights[0];
    double sum = x*weights[1] + y*weights[2] + activation;

    if(sum >= 0){
        return 1;
    }
    else{
        return 0;
    }

}
```

La funzione di train implementa semplicemente l'algoritmo di aggiornamento dei parametri adattivi per ogni istanza  $(x,y)$  che viene presentata all'algoritmo:

```
void p_clf::train(double x , double y, int label, double eta){

    int pred = predict(x,y);
    double error = label-pred;
    weights[0] = weights[0] + eta*error;
    weights[1] = weights[1] + eta*error*x;
    weights[2] = weights[2] + eta*error*y;

    return;
}
```

I coefficienti lineari sono invece calcolati a partire dai parametri adattivi come:

$$y = mx + q \quad , \quad m = -\frac{w_1}{w_2} \quad , \quad q = -\frac{w_0}{w_2}$$

Nel main leggiamo il dataset creato in precedenza e salvato in un .root:

```
double x,y;

TFile* dataset = new TFile("./dataset.root");
TTree* signal = (TTree*) dataset->Get("Signal");
signal->SetBranchAddress("x", &x);
signal->SetBranchAddress("y", &y);
TTree* background = (TTree*) dataset->Get("Background");
```

```

background->SetBranchAddress("x", &x);
background->SetBranchAddress("y", &y);

int N_sig = signal->GetEntries();
int N_bkg = background->GetEntries();

```

Istanziamo quindi l'oggetto perceptron e chiamamo la routine per generare dei pesi random:

```

p_clf perceptron(2, N_sig+N_bkg);
perceptron.set_weights();
std::vector<double> w = perceptron.get_w();

```

L'output da terminale sarà quindi:

```
0.999742 0.16291 0.282618
```

Cicliamo quindi su ogni istanza contenuta nel dataset di segnale e di fondo e chiamamo la routine per il train dei parametri:

```

double eta = 0.01;

for(int i = 0; i < N_sig; i++){
    signal->GetEntry(i);
    perceptron.train(x, y, 0, eta);

    background->GetEntry(i);
    perceptron.train(x, y, 1, eta);

}

w = perceptron.get_w();
std::cout << w[0] << " " << w[1] << " " << w[2] << std::endl;

```

Ricontrolliamo lo stato dei parametri dopo la procedura di train e, da terminale, osserviamo:

```
-0.190258 0.065953 0.134029
```

Una volta allenato il modello possiamo predirre sugli stessi dati di train per osservare come il modello classifica il segnale e fondo computando l'accuracy. Bisogna prestare attenzione. In genere testare il modello sui dati di train è una procedura da evitare nel caso generale. In questo caso siamo giustificati dal fatto che conosciamo a proprio le distribuzioni da cui provengono i dati quindi nuove istanze saranno ancora distribuite gaussianamente e non abbiamo incertezze. Questo tuttavia raramente è un caso realistico.

```

int t_p = 0;
int t_n = 0;
int f_p = 0;

```

```

int f_n = 0;

for(int i = 0; i < N_sig; i++){

    signal->GetEntry(i);
    int p_sig = perceptron.predict(x, y);
    if (p_sig == 0){
        t_p += 1;
    }
    else{
        f_n += 1;
    }

    background->GetEntry(i);
    int p_bkg = perceptron.predict(x, y);
    if (p_bkg == 1){
        t_n += 1;
    }
    else{
        f_p += 1;
    }

}

std::cout << "TP: " << t_p << std::endl;
std::cout << "FN: " << f_n << std::endl;
std::cout << "TN: " << t_n << std::endl;
std::cout << "FP: " << f_p << std::endl;
std::cout << "Accuracy: " << (double) (t_p+t_n)/(t_p+t_n+f_n+f_p)
             << std::endl;
std::cout << "Error rate: " << (double) 1- (t_p+t_n)/(t_p+t_n+f_n+f_p)
             << std::endl;

```

L'output a terminale è il seguente:

```

TP: 9998
FN: 2
TN: 10000
FP: 0
Accuracy: 0.9999
Error rate: 0.0001

```

Il modello quindi è estremamente preciso tuttavia non è perfetto in quanto i dati non sono totalmente linearmente separabili. In questo caso sappiamo a prescindere che il percepitrone non è un classificatore perfetto.

Plottiamo il confine decisionale del modello assieme ai dati:

```
line_bound line = perceptron.get_boundary();
```

```
TF1* bound = new TF1("Decision boundary", "[0]*x + [1]", xmin, xmax);
bound->SetParameters(line.m, line.q);
```

Il risultato è il seguente:

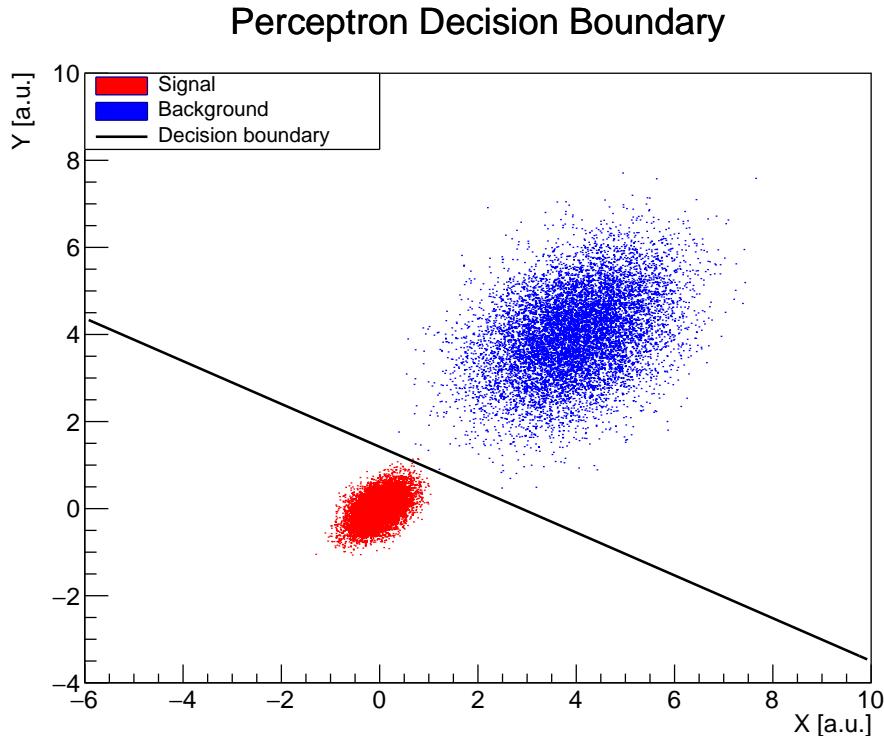


Figura 7.3: Plot del confine decisionale del perceptrone assieme ai dataset di segnale e di fondo. Osserviamo che visivamente i dati potevano essere supposti linearmente separabili e quindi è giustificato l'uso di un classificatore lineare anche se non è garantita la perfezione delle predizioni.

## 7.4 Support vector machine

Abbiamo visto che il classificatore lineare percettrone porta a buoni risultati di classificazione pur essendo un algoritmo semplice e estremamente rigido.

Tuttavia, l'errore nella classificazione è inevitabile in quanto i dati sono intrinsecamente linearmente non separabili.

Un modo per estendere il concetto di classificatore lineare può essere l'approccio delle Support Vector Machine (SVM).

Nel caso di dati linearmente separabili, l'obiettivo del SVM è di trovare l'iperpiano ottimo di divisione delle classi. Siano  $A, B$  due insiemi disgiunti nello spazio  $n$ -dimensionale  $\mathbb{R}^n$  e che siano linearmente separabili ovvero che esista un iperpiano  $H = \{x \in \mathbb{R}^n | w^T x + b = 0\}$  per cui tutti i punti  $x \in A$  appartengano ad un semispazio e tutti i punti  $x \in B$  appartengano al semispazio complementare. Esistono quindi un vettore di parametri reali  $w$  e una costante reale  $b$  tali per cui:

$$w^T \bar{x} + b \geq 1 \quad , \quad \forall \bar{x} \in A$$

$$\bar{w}^T \bar{x} + b \leq -1 \quad , \quad \forall \bar{x} \in B$$

La formula è identica al caso del percettrone. Tuttavia è dimostrabile che, nel caso di dati linearmente separabili, i possibili iperpiani tali che soddisfano le equazioni precedenti sono infiniti. Le SVM discriminano quindi il piano ottimo di separazione attraverso il concetto di margine ovvero la massima distanza  $\rho$  tra i punti di  $AUB$  e l'iperpiano  $H$ :

$$\rho = \min_{\bar{x} \in AUB} \left\{ \frac{|\bar{w}^T + b|}{\|\bar{w}\|} \right\}$$

L'iperpiano ottimo  $H(\bar{w}^*, b^*)$  è l'iperpiano che massimizza il margine  $\rho$ . In tal caso si può dimostrare che l'iperpiano ottimo è unico. L'interesse nella ricerca dell'iperpiano ottimo è giustificato dalla minimizzazione del rischio empirico  $R$  definito come:

$$R(\bar{w}, b) \leq R_{emp}(\bar{x}, b) + \sqrt{\frac{h(\log(\frac{2l}{f}) + 1) - \log(\frac{\eta}{4})}{l}}$$

$$R_{emp}(\alpha) = \frac{1}{2l} \sum_{i=0}^l |y^i - f(\bar{x}^i, \alpha)|$$

Dove  $l$  è il numero di istanze del dataset,  $h$  è la VC dimension (massimo numero di punti che possono essere frammentati da una funzione  $f(\alpha)$ ),  $\eta$  è un valore tale che  $0 \leq \eta \leq 1$  e  $1 - \eta$  è la probabilità con cui possiamo assicurare la diseguaglianza del rischio empirico.

Per definizione, ogni iperpiano  $H$  di separazione ha rischio empirico nullo  $R_{emp} = 0$  se i dati sono linearmente separabili. L'iperpiano ottimo massimizza il margine  $\rho$  e di conseguenza si dimostra che  $h$  è minimo. Essendo la RHS della disequazione per  $R(\bar{w}, b)$  monotona crescente in  $h$  allora possiamo affermare che l'upper bound del rischio effettivo per l'iperpiano ottimo  $H$  è minore di qualsiasi altro iperpiano di separazione.

Il problema di non linearità della separazione è affrontato dalle SVM proiettando i dati  $\bar{x} \in \mathbb{R}^n$  in uno spazio  $\mathbb{R}^m$  con  $m > n$ , attraverso una funzione detta kernel, ove i dati sono separabili da un'iperpiano ottimo. Sia  $X$  sottoinsieme di  $\mathbb{R}^n$ :

$$k : X \times X \rightarrow \mathbb{R}$$

è un kernel se soddisfa la seguente proprietà:

$$k(x, y) = \langle \phi(x), \phi(y) \rangle \quad \forall x, y \in X$$

Dove  $\phi$  è un'applicazione lineare da  $X \rightarrow H$  con  $H$  spazio euclideo cioè uno spazio lineare con un prodotto scalare fissato.

Siano  $\bar{x}_i$   $i = 1 \dots l$  i vettori di training e  $y_i \in \{-1, 1\}$  le corrispondenti labels che individuano la classe di appartenenza del vettore di train. Si consideri l'applicazione

$$\phi \rightarrow H$$

Con  $H$  spazio euclideo a dimensione maggiore di  $n$ . Lo spazio  $H$  viene denominato spazio delle features.

Si considerino i vettori trasformati  $\phi(\bar{x}_i)$  dei vettori di training  $\bar{x}_i$  e il problema della determinazione dell'iperpiano ottimo nel feature space  $H$  in cui vivono i vettori trasformati.

La trattazione è analoga al modello per classificazione lineare ma nello spazio delle feature  $\bar{x}_i \rightarrow \phi(\bar{x}_i)$ . Da notare che la classificazione è lineare nello spazio  $H$  e non lineare nello spazio  $X$ .

Il problema di ottimizzazione è quadratico di tipo convesso e ammette minimo unico. Senza approfondire il formalismo matematico, il problema di ottimizzazione si riassume nelle seguenti equazioni (riportata la formulazione duale del problema):

$$\min \Gamma(\lambda) = \frac{1}{2} \sum_{i=1}^l \sum_{j=1}^l y^i y^j \phi(\bar{x}^i)^T \phi(\bar{x}^j) \lambda_i \lambda_j - \sum_{i=1}^l \lambda_i$$

tale che:

$$\begin{aligned} \sum_{i=1}^l \lambda_i y^i &= 0 \\ 0 \leq \lambda_i &\leq C \quad i = 1 \dots l \end{aligned}$$

Dove  $\lambda$  denota i moltiplicatori di Lagrange e minimizzare la funzione  $\Gamma(\lambda)$  rispetto ai moltiplicatori equivale a massimizzare la Lagrangiana del sistema. Il problema ammette almeno una soluzione ottima  $\lambda^*$ . Ne segue che il vettore dei parametri ottimi  $w^*$  può essere determinato come:

$$\bar{w}^* = \sum_{i=1}^l \lambda^* y^i \phi(\bar{x}^i)$$

Noto  $\bar{w}^*$  e considerato un qualsiasi moltiplicatore  $0 < \lambda_i^* < C$ , lo scalare  $b^*$  può essere determinato utilizzando la seguente:

$$y^i \left( \sum_{j=1}^l \lambda_j^* y^j \phi(\bar{x}_j)^T \phi(\bar{x}_i) + b^* \right) - 1 = y^j \left( \sum_{j=1}^l \lambda_j^* y^j k(\bar{x}_j, \bar{x}_i) + b^* \right) - 1 = 0$$

La funzione di decisione assume quindi la forma:

$$f(x) = \operatorname{sgn}((\bar{w}^*)^T \phi(\bar{x}) + b^*) = \operatorname{sgn}\left(\sum_{j=1}^l \lambda_j^* y^j k(\bar{x}_j, \bar{x}) + b^*\right)$$

Una trattazione approfondita e formale può essere trovata alla seguente pagina:  
[http://www.dis.uniroma1.it/or/gestionale/svm/svm\\_sciandrone.pdf](http://www.dis.uniroma1.it/or/gestionale/svm/svm_sciandrone.pdf)

Applicheremo al nostro caso un kernel polinomiale inomogeneo con  $d = 2$ :

$$K(x, x') = (1 + \bar{x}^T \bar{x}')^d = (1 + \bar{x}^T \bar{x}')^2$$

La cui mappa nel future space sarà:

$$\phi(\bar{x}) = [1, \sqrt{2}x, \sqrt{2}y, x^2, \sqrt{2}xy, y^2]$$

La scelta della polinomiale è giustificata dal seguente fattore fondamentale. Il confine decisionale del modello sarà in questo caso:

$$(\bar{w}^*)^T \phi(\bar{x}) + b^* = w_0 + b^* + w_1\sqrt{2}x + w_2\sqrt{2}y + w_3x^2 + w_4\sqrt{2}xy + w_5*y^2 = 0$$

Abbiamo ritrovato in questo modo che il dominio decisionale è separato dall'equazione di una conica proprio come nel caso ottimale del classificatore quadratico. Ci aspettiamo che i parametri si modellino sui parametri ottenuti in forma chiusa dal primo classificatore per generare un ellisse.

Ci limitiamo ad illustrare le componenti principali dell'algoritmo di apprendimento. La loss utilizzata è la Hinge loss (in documentazione si può trovare una trattazione rigorosa per l'origine di questa funzione):

$$h_{loss} = \max(0, 1 - y_i f(\phi(\bar{x}_i)))$$

La regolarizzazione sulla loss è data dal modulo del vettore dei parametri. Il problema di minimizzazione si riduce quindi al seguente:

$$\min_{\bar{w}} C(\bar{w}) = \frac{\lambda}{2} \|\bar{w}\|^2 + \frac{1}{N} \sum_i^N \max(0, 1 - y_i f(\phi(\bar{x}_i))) , \quad \lambda = \frac{2}{NC}$$

Dal momento che la hinge loss non è differenziabile, useremo la tecnica del sub-gradient descend per l'ottimizzazione dei pesi il che porta all'algoritmo:

$$w^{\tau+1} = w^\tau - \eta(\lambda w^\tau - y_i x_i) \text{ if } y_i f(\phi(x_i)) < 1$$

$$w^{\tau+1} = w^\tau - \eta \lambda w^\tau \text{ otherwise}$$

Riportiamo senza commenti il codice composto dall'header e dalla classe SVM implementata a mano. La struttura della classe è la seguente:

```
#ifndef SVM_H
#define SVM_H
#include <vector>

class svm_clf{
private:
    double reg;
    int size;
    int dimension;
    int f_space_dim = 6;
    std::vector<double> weight;

public:
    svm_clf(int s, int d, int r){ size = s; dimension = d; reg = r; } ;
    ~svm_clf(){};
    int predict(double x, double y);
    double predict_train(double x, double y);
};
```

```

        double kernel(double x, double y);
        void set_weights();
        void train(double x, double y, int label, double eta);
        std::vector<double> get_weight(){ return weight ;} ;
        std::vector<double> mapping(double x, double y);

    };

#endif //SVM_H

```

I metodi utilizzati per allenare e predire con la svm sono i seguenti:

```

#include "TRandom3.h"
#include "svm.h"
#include <vector>
#include <iostream>
#include <numeric>

void svm_clf::set_weights(){

    //inizializza random il vettore dei pesi
    TRandom3* gen = new TRandom3();
    for(int i = 0; i < f_space_dim; i++){
        //meglio settare pesi randomici
        //grandi e iterare più volte su tutto
        //il dataset.
        weight.push_back(gen->Uniform(-20,20));
    }

    return;
}

std::vector<double> svm_clf::mapping(double x, double y){

    std::vector<double> pol_map;
    pol_map.push_back(1);
    pol_map.push_back(sqrt(2)*x);
    pol_map.push_back(sqrt(2)*y);
    pol_map.push_back(x*x);
    pol_map.push_back(sqrt(2)*x*y);
    pol_map.push_back(y*y);

    return pol_map;
}

double svm_clf::predict_train(double x, double y){

```

```

    double r1 = 0;

    std::vector<double> phi_x = mapping(x, y);
    for(int i = 0; i < f_space_dim; i++){
        r1+= weight[i]*phi_x[i];
    }
    return r1;
}

int svm_clf::predict(double x, double y){

    std::vector<double> phi_x = mapping(x, y);
    double r1 = std::inner_product(weight.begin(),
                                   weight.end(), phi_x.begin(), 0);
    if(r1 > 0) return 1;
    else return -1;
}

void svm_clf::train(double x, double y, int label, double eta){

    double p = predict_train(x,y);

    std::vector<double> phi = mapping(x,y);
    if(label*p < 1 ){
        for(int i = 0; i < f_space_dim; i++){
            weight[i] = weight[i] - eta*((2*weight[i])/(reg*size)
                                         - label*phi[i]);
        }
    }
    else{
        for(int i = 0; i < f_space_dim; i++){
            weight[i] = weight[i]-eta*(2/(reg*size))*weight[i];
        }
    }
}

```

Nel main costruiamo l'oggetto SVM. Introduciamo un parametro di regolarizzazione molto alto perché si realizzi la condizione di hard margin classification. Definiamo quindi il learning rate e iteriamo più volte su tutto il dataset. Questo viene fatto in quanto l'SVM deve classificare correttamente tutti i punti del dataset, iterando una sola volta rischiamo che l'algoritmo fallisca. Il seguente metodo di train è chiamato train ad epoch. Ogni epoch è un il processo di aggiornamento

di pesi per ogni istanza del dataset.

Il parametro learning rate viene diviso per il numero d'iterazione, in questo modo creiamo un learning rate variabile che è più sensibile alla ricerca globale (algoritmo Pegaso:  $\eta = \frac{1}{\lambda\tau}$  )

```
//-----
//-----SVM-----
//-----

//oggetto sum_clf. Il terzo parametro è
//la regolarizzazione, più è alta più il
//classificatore è detto hard-margin (tollerà poco).
svm_clf svm(N_sig+N_bkg, 2, 10000000);
svm.set_weights();
std::vector<double> w = svm.get_weight();

std::cout << w[0] << " " << w[1] << " " << w[2] <<
" " << w[3] << " " << w[4] << " " << w[5] << std::endl;
double eta = 0.001;

for(int j = 0; j < 5000; j++){
    for(int i = 0; i < N_sig; i++){
        signal->GetEntry(i);
        svm.train(x, y, 1, eta/i);

        background->GetEntry(i);
        svm.train(x, y, -1, eta/i+1);

    }
}

w = svm.get_weight();

std::cout << w[0] << " " << w[1] << " " << w[2]
<< " " << w[3] << " " << w[4] << " " << w[5] << std::endl;
```

Passiamo quindi ad analizzare i risultati:

```
Initial weights: 14.9923 -10.1127 -6.52147 13.416 -8.0503 -0.450792
Final weights: 21.1562 -7.20064 -1.81418 2.87424 -8.06866 -1.42631
```

```
TP: 10000
FN: 0
TN: 10000
FP: 0
Accuracy: 1
Error rate: 0
```

Come ci aspettavamo i risultati della SVM sono quelli di un classificatore ideale. Tuttavia il valore dei parametri ottimizzati differisce particolarmente da quelli per il quadratic classifier. Infatti il decision boundary non è un ellisse ma un'iperbole.

In effetti il classificatore così implementato è estremamente sensibile alla scelta iniziale dei parametri adattivi del modello. E' probabile che l'algoritmo si fermi in un minimo locale. In effetti notiamo che l'accuracy in questo modo è perfetta e il modello non ha più chance di avanzare.

Tuttavia se proviamo a settare il valore iniziale dei parametri a quello che conosciamo teoricamente:

```
void svm_clf::set_weights(){

    //inizializza random il vettore dei pesi
    TRandom3* gen = new TRandom3();
    std::vector<double> tr_w {27.7986, -5.714, -5.714, -13.62, +13.85, -13.62};
    for(int i = 0; i < f_space_dim; i++){
        //meglio settare pesi randomici
        //grandi e iterare più volte su tutto
        //il dataset.
        //weight.push_back(gen->Uniform(-20,20));
        weight.push_back(tr_w[i]);
    }

    return;
}
```

Allora il modello, come ci aspettiamo, non subisce grossi cambiamenti dei parametri in quanto essi sono già ottimizzati:

```
Initial weights: 27.7986 -5.714 -5.714 -13.62 13.85 -13.62
Final weights: 27.7985 -5.71399 -5.71399 -13.62 13.85 -13.62
TP: 10000
FN: 0
TN: 10000
FP: 0
Accuracy: 1
Error rate: 0
```

E' comunque possibile implementare funzioni di kernel differenti tra le quali le più importanti sono:

- RBF kernel: kernel a radial basis function ad esempio kernel gaussiano:  
 $k(x, x') = \exp\left(-\frac{\|x-x'\|^2}{2\sigma^2}\right)$
- Linear kernel:  $k(x, x') = x^T x'$
- Polynomial homogeneous kernel:  $k(x, x') = (x^T x')^d$

Un modo per aiutare l'algoritmo alla convergenza ottima potrebbe essere implementare un momento ovvero, se l'accuracy del modello differisce dall'unità per  $n$  epochs, una funzione aggiunge un valore random ai pesi così da spostare la convergenza verso un possibile altro minimo ad esempio Nesterov Momentum formula. Come esempio si è implementato un momento semplice che aggiunge ad ogni parametro un numero pseudo-casuale se l'accuracy del modello sui dati di train non decresce per 10 epochs:

```
int counter = 0;
while(accuracy != 1){

    if(counter == 10){
        svm.momentum();
        counter = 0;
        std::cout << "Momentum" << std::endl;
    }
    for(int j = 0; j < 10; j++){
        for(int i = 0; i < N_sig; i++){
            .
            .
            .
        }
    }
}
```

La funzione `momentum()` è la seguente:

```
TRandom3* gen = new TRandom3();

void svm_clf::momentum(){

    for(int i = 0; i < f_space_dim; i++){
        weight[i] += gen->Uniform(-20, 20);
    }
    return;
}
```

Stampiamo quindi ad ogni epoch il valore dei parametri, i punti misclassificati e l'accuracy. Il risultato è il seguente:

```
.
.
.

67.8884 -19.3976 -9.69751 6.11301 -22.8362 0.382125
0.438515 1.79919
0.99995
67.7884 -19.4597 -9.95195 6.09378 -22.9478 0.0584151
0.438515 1.79919
0.99995
67.6884 -19.5217 -10.2064 6.07455 -23.0594 -0.265294
0.438515 1.79919
0.99995
```

```

67.5884 -19.5837 -10.4608 6.05532 -23.1709 -0.589004
0.438515 1.79919
0.99995
67.4884 -19.6457 -10.7153 6.03609 -23.2825 -0.912714
1

```

Si può osservare che i parametri sono ancora una volta diversi ma l'accuracy è ancora perfetta. Il confine decisionale non è unico.

Il plot del decision boundary è fatto in python in quanto c++ non supporta una facile implementazione di plotting di funzioni coniche. I dati sono importati grazie alla libreria `uproot` dal `.root` creato all'inizio dell'esercizio. Il risultato è il seguente:

```

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

#Plotting data and decision boundary SVM

f = uproot.open("/Users/boldrinocoder/uni/Analisi-Statistica
                 /code/es7/dataset.root")
h = f.get("Signal")
b = f.get("Background")

x_s = h.array("x")
y_s = h.array("y")

x_b = b.array("x")
y_b = b.array("y")

x = np.linspace(-3, 10, 400)
y = np.linspace(-3, 10, 400)
x, y = np.meshgrid(x, y)
plt.figure(figsize=(13,8))
plt.title("SVM Classifier decision boundary")
plt.xlabel("X [a.u.]")
plt.ylabel("Y [a.u.]")
plt.scatter(x_s,y_s, c = 'red', s=0.3, label="Signal")
plt.scatter(x_b,y_b, c = 'blue', s = 0.2, label = "Background")
plt.contour(x, y, 67.4884 -np.sqrt(2)*19.6457*x -np.sqrt(2)*10.7153*y +
6.03609*x*x -np.sqrt(2)*23.2825*x*y -0.912714*y*y, [0], colors='k',
label="SVM Decision boundary")
plt.legend(loc = 'upper left')
plt.savefig("./SVM.pdf")

```

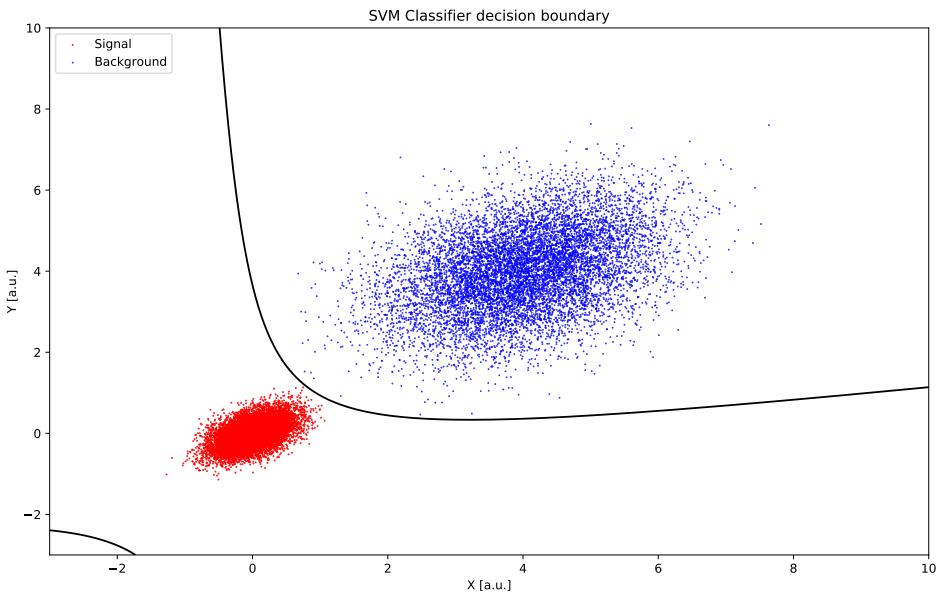


Figura 7.4: Plot del dataset segnale e fondo con distribuzione gaussiana assieme al decision boundary del classificatore SVM con momento sui parametri. Questo classificatore è ottimale in quanto ha accuracy=1 tuttavia si discosta dal classificatore quadratico. Questo perché, a differenza del classificatore quadratico, l'SVM non ha idea della distribuzione dei dati.

Possiamo utilizzare librerie che implementano l'algoritmo in modo più formale e preciso ad esempio utilizzando il pacchetto TMVA di ROOT nel modo seguente (per lo script completo consultare il capitolo 7.5 sulle reti neurali, basta aggiungere le seguenti righe al codice):

```
...
    factory->BookMethod
    (
        dataloader,
        TMVA::Types::kSVM,
        "SVM"
    );
...

```

Il risultato è identico al precedente:

```
: Evaluation results ranked by best signal efficiency and purity (area)
:
: DataSet      MVA
: Name:        Method:      ROC-integ
: NN_classification SVM      : 1.000
:
:
: Testing efficiency compared to training efficiency (overtraining check)
:
: DataSet      MVA          Signal efficiency: from test sample (from training sample)
: Name:        Method:      @B=0.01      @B=0.10      @B=0.30
:
: NN_classification   SVM      : 1.000 (1.000)  1.000 (1.000)  1.000 (1.000)
:
```

Possiamo plottere la distribuzione degli output come predizioni  $\hat{y} \in [0, 1]$  dove 0 indica segnale e 1 indica fondo:

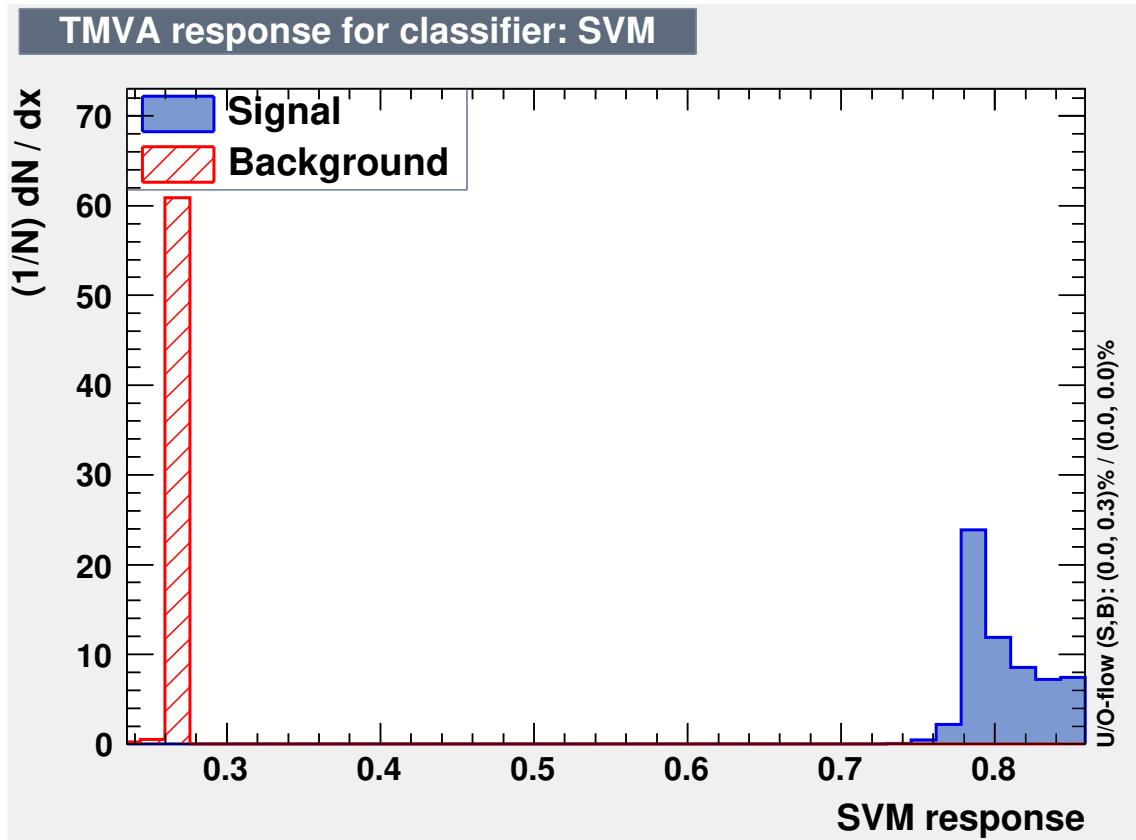


Figura 7.5: Distribuzione degli output del classificatore SVM.

I metodi implementati dalle varie librerie sono molto performanti e rapide, tuttavia l'applicazione è molto cieca, non sappiamo controllare in genere al 100% l'algoritmo a dispetto di un algoritmo implementato a mano come i precedenti.

## 7.5 ANN

Anche se l'utilizzo delle reti neurali per questo tipo di problema è, probabilmente, eccessivo, è comunque utile osservare il comportamento di diversi classificatori nella risoluzione della medesima classificazione.

Le reti neurali sono la logica estensione del modello del perceptrone illustrato qualche capitolo fa. In termini semplici una rete neurale è una composizione di neuroni artificiali, organizzati in livelli chiamati layers che si suddividono in input layer, hidden layer(s) e output layer. Il numero di neuroni per livello ed il numero di hidden layers sono parametri arbitrari che, assieme ad altri formano un set chiamato hyperparameters.

Il modello del neurone è tuttavia differente (se ben simile) da quello del perceptrone. L'unica differenza è la funzione di attivazione che passa da funzione a gradino a

funzione logistica

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

che è simile alla precedente ma ha l'enorme vantaggio di essere differenziabile. Il modello probabilistico è chiamato regressione logistica e agisce nel modo seguente:

$$P(y = 1 | \bar{x}) = \sigma(\bar{w}^T \bar{x} + b)$$

Si ignora qualsiasi assunzione sul modello e si cerca solamente di massimizzare la likelihood dei dati ottimizzando i parametri adattivi:

$$\begin{aligned} argmax_{\bar{w}, b} P(\bar{d} | \bar{w}, b) &= \\ argmax_{\bar{w}, b} \prod_{\bar{x}_i, y_i \in \bar{d}} P(Y = y_i | \bar{x}_i, \bar{w}, b) &= \\ argmax_{\bar{w}, b} \prod_{\bar{x}_i, y_i \in \bar{d}} \sigma(\bar{w}^T \bar{x}_i + b)^{y_i} (1 - \sigma(\bar{w}^T \bar{x}_i + b))^{1-y_i} &= \\ argmin_{\bar{w}, b} \prod_{\bar{x}_i, y_i \in \bar{d}} -y_i \log \sigma(\bar{w}^T \bar{x}_i + b) - (1 - y_i) \log(1 - \sigma(\bar{w}^T \bar{x}_i + b)) &= \\ argmin_{\bar{w}, b} \mathcal{L}(\bar{w}, b) &= argmin_{\bar{w}, b} \sum_i l(y_i, \hat{y}(\bar{x}_i, \bar{w}, b)) \end{aligned}$$

Questa loss è un'istanza della cross-entropy:

$$H(p, q) = E_p[-\log(q)]$$

per  $p = Y | \bar{x}_i$  e  $q = \hat{Y} | \bar{x}_i$

La minimizzazione della loss avviene grazie allo Stochastic Gradient Descent per l'apprendimento dei parametri adattivi ottiale. Sia  $\theta$  uno qualsiasi dei parametri adattivi l'algoritmo iterativo di apprendimento è dato da:

$$\theta_{t+1} = \theta_t - \gamma \nabla l(y_{i(t+1)}, f(\bar{x}_{i(t+1)}; \theta_t))$$

E' possibile dimostrare che quanto decomponiamo l'errore in temini di approssimazione, inferenza e ottimizzazione, gli algoritmi stocastici rendono la migliore performance di generalizzazione, al costo di essere i peggiori algoritmi di ottimizzazione:

$$E[R(\tilde{f}_*) - R(f_B)] = \epsilon_{app} + \epsilon_{inf} + \epsilon_{opt}$$

Possiamo quindi comporre i livelli della rete neurale come composizione in serie dei neuroni artificiali. Indichiamo  $\bar{h}_i$  l'output del livello  $i$ -esimo:

$$\bar{h}_0 = \bar{x}$$

$$\bar{h}_1 = \sigma(\bar{W}_1^T \bar{h}_0 + \bar{b}_1)$$

...

$$\bar{h}_L = \sigma(\bar{W}_L^T \bar{h}_{L-1} + \bar{b}_L)$$

$$f(\bar{x}, \theta) = \hat{y} = \bar{h}_L$$

Dove  $\theta$  denota i parametri del modello  $\{\bar{W}_k, \bar{b}_k, \dots | k = 1, \dots, L\}$ . Questo è il modello per le reti neurali artificiali o multi-layer perceptron nella configurazione nota come FeedForward Neural Network. Esistono altre configurazioni che differenziano dal tipo di connessione possibile tra i neuroni stessi. L'architettura feedforward ammette connessioni complete tra un layer e il successivo ma nessuna connessione tra neuroni dello stesso livello.

Per la classificazione binaria di interesse al problema la dimensione  $q$  dell'output layer  $L$  è impostata a  $q = 1$  che risulta in un singolo output  $h_L \in [0, 1]$  che modella la probabilità di appartenenza dell'istanza a una classe:

$$P(Y = 1 | \bar{x})$$

Consideriamo ora il caso di un MLP a 2 livelli

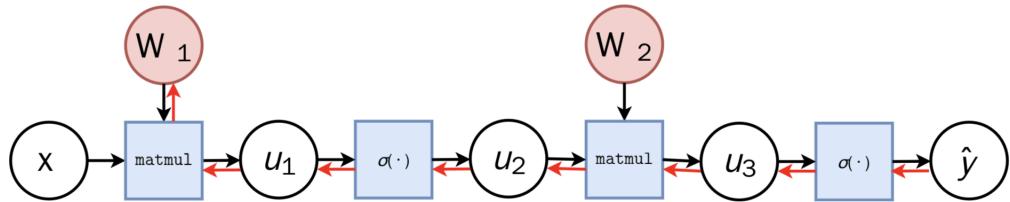


Figura 7.6: 2 layers MLP.

e la seguente funzione di loss:

$$f(\bar{x}, \bar{w}_1, \bar{w}_2) = \sigma(\bar{w}_2^T \sigma(\bar{w}_1^T \bar{x}))$$

$$l(y, \hat{y}, \bar{w}_1, \bar{w}_2) = cross_{ent}(y, \hat{y}) + \lambda(||\bar{w}_1||_2 + ||\bar{w}_2||_2)$$

Per  $\bar{x} \in R^n$ ,  $y \in R$ ,  $\bar{w}_1 \in R^{n \times m}$  ed  $\bar{w}_2 \in R^m$

Attraverso un *forward pass*, i valori di  $u_1, u_2, u_3$  e  $\hat{y}$  sono computati attraversando il grafico dagli input  $\bar{x}$  agli output, dati i pesi  $\bar{w}_1, \bar{w}_2$ . L'aggiornamento dei pesi viene invece fatto grazie ad un *backward pass*. Dalla chain rule abbiamo:

$$\begin{aligned} \frac{d\hat{y}}{d\bar{w}_1} &= \frac{\partial \hat{y}}{\partial u_3} \frac{\partial u_3}{\partial u_2} \frac{\partial u_2}{\partial u_1} \frac{\partial u_1}{\partial \bar{w}_1} = \\ &= \frac{\partial \sigma(u_3)}{\partial u_3} \frac{\partial \bar{w}_2^T u_2}{\partial u_2} \frac{\partial \sigma(u_1)}{\partial u_1} \frac{\partial \bar{w}_1^T u_1}{\partial \bar{w}_1} \end{aligned}$$

Questo algoritmo è il backpropagation. Dato che la differenziazione è un operatore lineare, algoritmi di autodifferenziazione sono spesso implementati in termini di operazioni tensoriali.

Si può notare un primo problema nella scelta della funzione d'attivazione.  $\sigma$  funzione sigmoide presenta quello che in gergo è chiamato vanishing gradient problem, un problema che ha limitato le reti neurali artificiali fino a circa il 2011. Infatti,

gradienti piccoli possono rallentare e al massimo stoppare l'algoritmo di Stochastic Gradient Descent. Come effetto, la capacità di apprendimento è limitata. La funzione sigmoide come la tangente iperbolica presentano questo problema mentre altri tipi di funzioni d'attivazione ad esempio ReLU, leaky ReLU, ELU non lo hanno. E' sempre meglio usare una funzione d'attivazione non saturante per allenare i propri modelli, tuttavia, come vedremo, i metodi di TMVA non supportano queste nuove funzioni.

L'implementazione dell'ANN è stata fatta grazie al toolkit TMVA di ROOT. E' stato usato un sample di 8000 eventi sia di segnale che di fondo per il training e di 2000 eventi segnale e fondo per il set di test. L'architettura è composta da un singolo hidden layer a 5 neuroni con funzione d'attivazione tangente iperbolica. Non è infatti necessario aggiungere ulteriori layers, il rischio di tale operazione è che vengano inseriti troppi parametri liberi nel modello che causerebbero inevitabilmente un overfitting del modello ai dati di train, situazione che vogliamo evitare in ogni caso.

I dati di train vengono gaussianizzati prima di iniziare l'addestramento. Infatti le reti neurali artificiali sono molto sensibili ad outliers o comunque a scale diverse degli attributi. E' quindi buona pratica standardizzare i dati, la quale è un'operazione che non comporta alcuna perdita d'informazione.

Il training method è stato impostato sulla backpropagation (potevamo optare per BFGS alternativamente).

L'architettura del network è quindi la seguente:

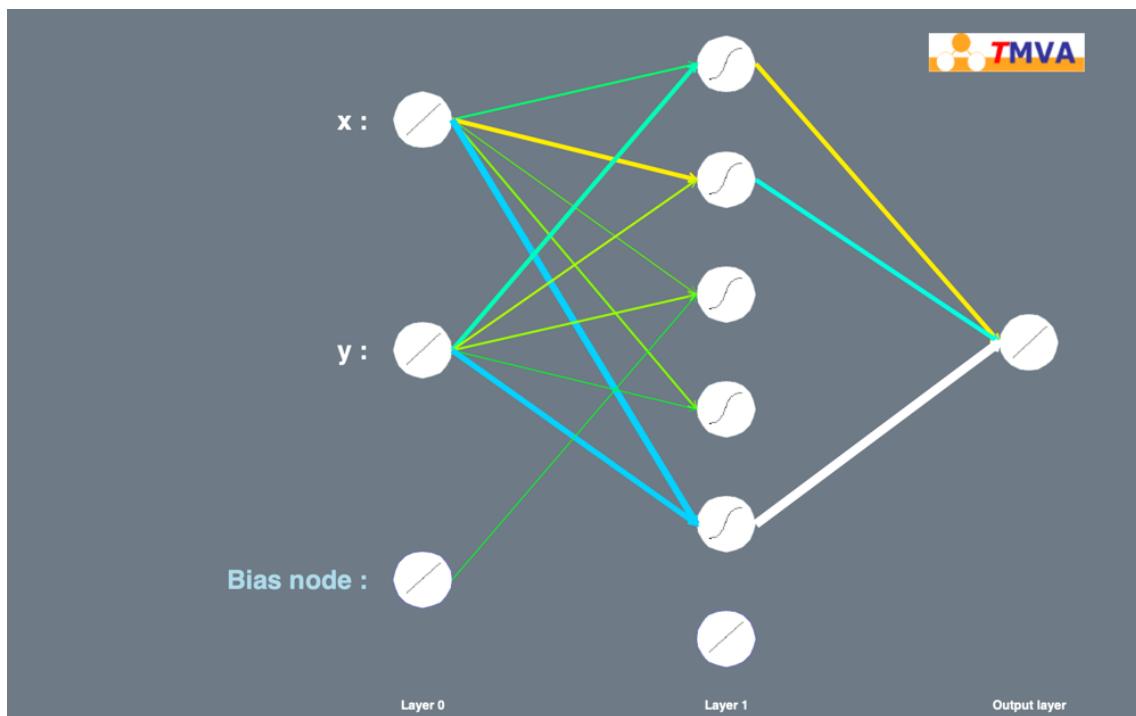


Figura 7.7: Architettura usata per risolvere la classificazione.

Ci limitiamo a riportare senza commenti il codice per il train del MLP con TMVA:

```

int TMVATrain(){

    TMVA::Tools::Instance();

    //TTree reading

    TFile * dataset = TFile::Open("dataset.root");

    TTree* signal = (TTree*) dataset ->Get("Signal");
    TTree* background = (TTree*) dataset ->Get("Background");

    TString outputfileName = ("TMVATrain.root");
    TFile* outputFile = new TFile(outputfileName, "RECREATE");

    //Factory definition

    TMVA::Factory* factory = new TMVA::Factory
    (
        "TMVAClassification",
        outputFile,
        "!V:!Silent:Color:DrawProgressBar:Transformations
         =I;P;G:AnalysisType=Classification"
    );

    //Adding input variables

    TMVA::DataLoader * dataloader = new TMVA::
    DataLoader("NN_classification");

    dataloader->AddVariable("x", 'F');
    dataloader->AddVariable("y", 'F');

    dataloader->AddSignalTree(signal, 1.); //tree, weights
    dataloader->AddBackgroundTree(background, 1.);

    TString sample_train = "8000";
    TCut cut_sig = "";
    TCut cut_bkg = cut_sig;

    dataloader->PrepareTrainingAndTestTree
    (
        cut_sig,
        cut_bkg,
        "nTrain_signal=8000:nTrain_Background=8000:
         SplitMode=Random:NormMode=EqualNumEvents:!V"
    );
}

```

```

);

factory->BookMethod
(
    dataloader,
    TMVA::Types::kMLP,
    "MLP",
    "!H:!V:NeuronType=tanh:VarTransform=Gauss:NCycles=200:
    HiddenLayers=5:TestRate=15:TrainingMethod=BP:SamplingTesting=True:
    ConvergenceImprove=1e-5:ConvergenceTests=75"
);

factory->TrainAllMethods();
factory->TestAllMethods();
factory->EvaluateAllMethods();

factory->Write();
//ROC->Write();
dataloader->Write();

outputfile->Close () ;

delete factory ;
delete dataloader ;
delete signal;
delete background ;
delete outputfile ;

if (!gROOT->IsBatch()) TMVA::TMVAGui( outputFile );
return 0;
}

```

Prima di andare ad analizzare i grafici prodotti da TMVAGui, riportiamo i risultati da terminale del training e del test del modello:

```

: Evaluation results ranked by best signal efficiency and purity (area)

: -----
: DataSet      MVA
: Name:        Method:      ROC-integ
: NN_classification MLP      : 1.000
: -----
: 
: Testing efficiency compared to training efficiency (overtraining check)
: -----
: DataSet      MVA      Signal efficiency: from test sample (from training sample)
: Name:        Method:    @B=0.01      @B=0.10      @B=0.30
: 
: NN_classification MLP      : 1.000 (1.000)   1.000 (1.000)   1.000 (1.000)
: -----

```

Vediamo che anche in questo caso l'accuracy è perfetta anche se l'architettura è relativamente piccola.

TMVA supporta delle funzioni di plot automatiche per analizzare sia il train del modello che l'output sul test set per controllare l'over training.

Il primo grafico da guardare riguarda il valore della loss al variare delle epoch di train. Ci aspettiamo che la loss sia monotona decrescente sia per il train che per il test. In caso di overfitting dei dati esso si manifesterà come un incremento della loss di test mentre la loss sul train continuerà a decrescere. Il modello, al posto di generalizzare la realtà, sta imparando a memoria il dataset di train ottimizzandosi su di esso e perdendo di generalità. La loss apparirà quindi convessa con un punto di minimo che indica l'inizio dell'overfitting.

Metodi di regolarizzazione come Dropout, EarlyStopping sono implementati da altre librerie come **Keras** o **Tensorflow** o Pytorch ma non da TMVA.

Riportiamo lo status del train nel grafico seguente:

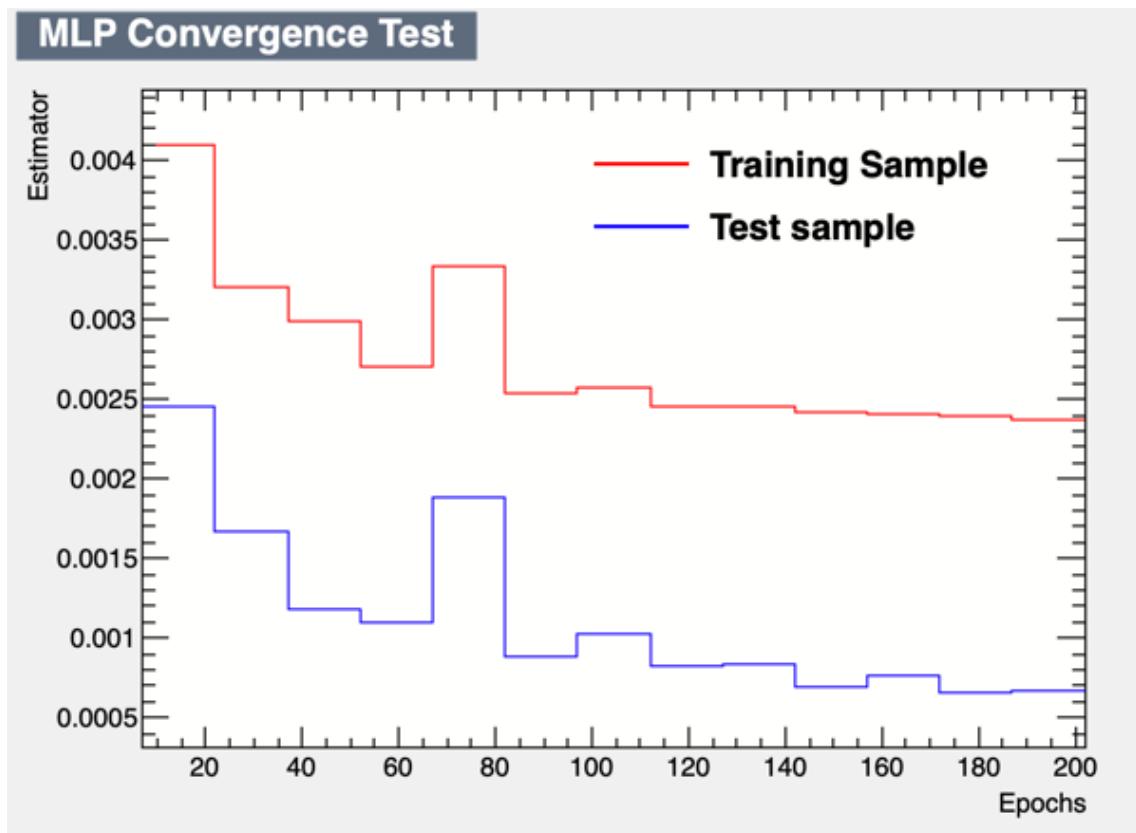


Figura 7.8: Plot della loss per il train e per il test al variare delle epoch.

Si osserva che l'andamento è decrescente, il che vuol dire che il modello si è allenato in una situazione di leggero underfitting. Avremmo potuto spingere il training ulteriormente ma non è necessario in quanto l'accuracy è perfetta anche così.

Possiamo quindi plottare la distribuzione degli output di segnale e di fondo. Essendo l'accuracy a 1 questi grafici sono ideali e poco significativi, li riportiamo per chiarezza d'esposizione dei risultati:

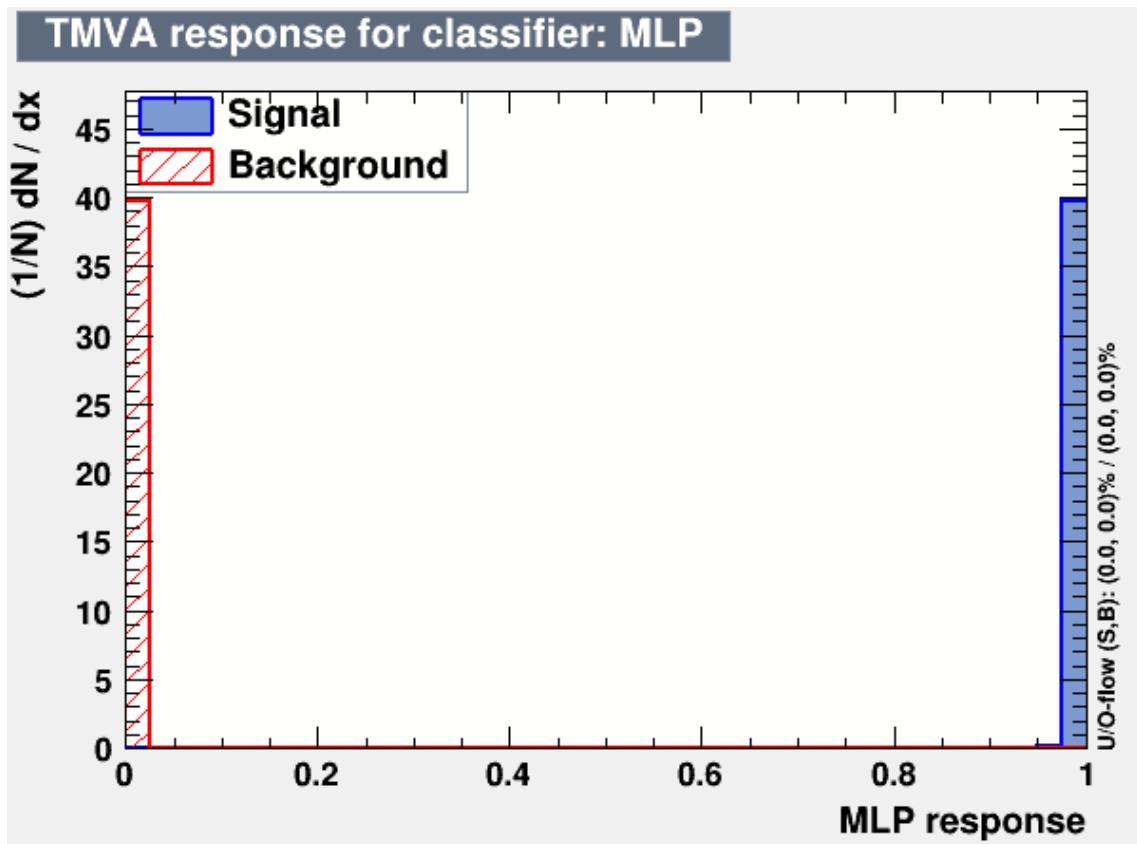


Figura 7.9: Plot della distribuzione delle probabilità [0,1] dove 0=signal e 1=background con probabilità 1 per il test set.

Il grafico della distribuzione della significatività in funzione del cut è generato:

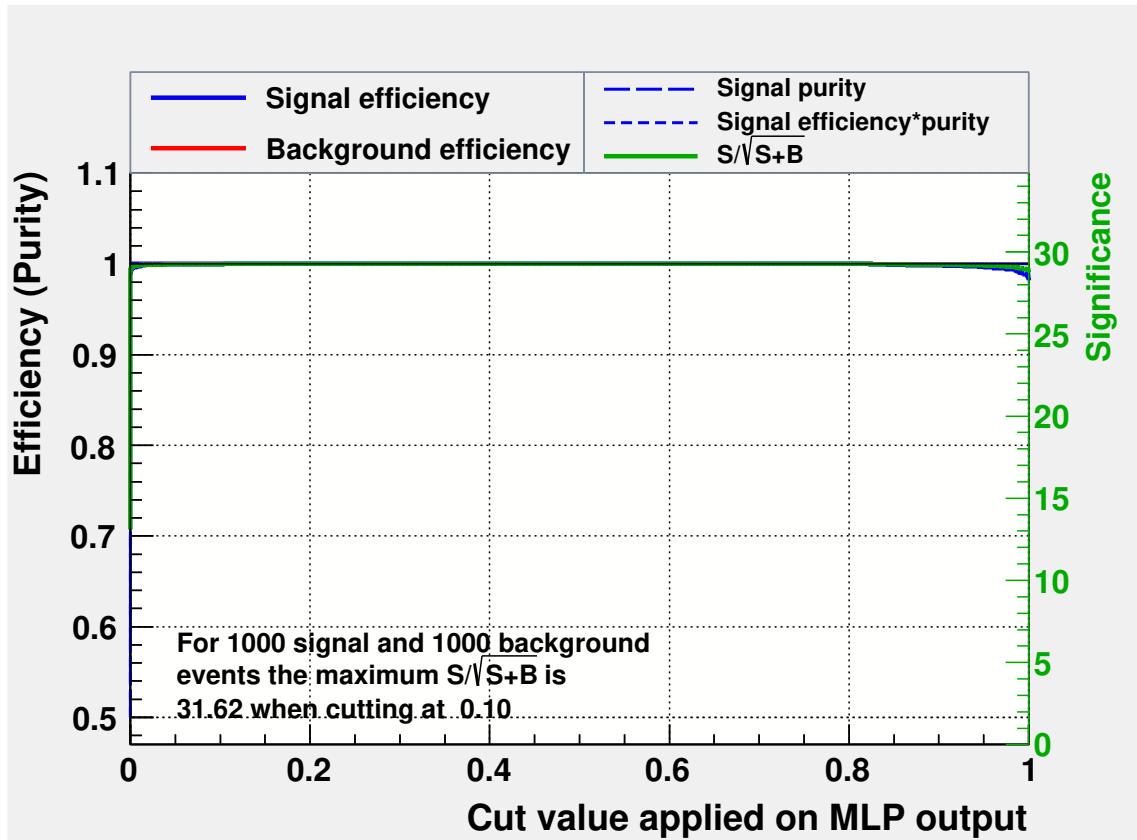


Figura 7.10: Efficienza di segnale e background e significatività in funzione del cut sui dati di test.

TMVA non supporta il plot per il decision boundary. Scriviamo un piccolo script in Python per vedere come lo stesso modello si comporta. Da notare che introduciamo un bias nell’analisi per la differenza negli algoritmi di Tensorflow e di TMVA tuttavia è utile osservare come si comporta l’algoritmo di classificazione:

```

from sklearn.preprocessing import StandardScaler
import keras
import tensorflow
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.callbacks import EarlyStopping
from sklearn.model_selection import train_test_split

dataset = []
for i in range(0, 10000):
    dataset.append([x_s[i], y_s[i], 0])
    dataset.append([x_b[i], y_b[i], 1])

scaler = StandardScaler()
dataset = np.array(dataset)

y = dataset[:, 2]

```

```

x = dataset[:, :2]
x = StandardScaler().fit_transform(x)
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size=0.2, random_state=42)

#model
model = Sequential()
model.add(Dense(2, input_dim=x_train.shape[1], activation='tanh'))
model.add(Dense(units=5, activation='tanh'))
model.add(Dense(units=1, activation='sigmoid'))

early_stop = EarlyStopping(monitor='val_loss', min_delta=1e-5, patience=5,
verbose=1, mode='auto', baseline=None)

model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

history = model.fit(x_train, y_train, epochs=1000,
validation_data=(x_test, y_test), callbacks= [early_stop])

y_pred_proba = model.predict(x_test)
y_pred = []
for pred_score in y_pred_proba:
y_pred.append(round(pred_score[0]))

x = np.linspace(-2, 3, 50)
y = np.linspace(-2, 3, 50)
xx, yy = np.meshgrid(x, y)
labels = model.predict(np.c_[xx.ravel(), yy.ravel()])
z = labels.reshape(xx.shape)

#just to plot sig and bkg with diff colors
x_signal = []
x_back = []

for i in range(0, x_test.shape[0]):
    if(y_test[i] == 0):
        x_signal.append(x_test[i])
    else:
        x_back.append(x_test[i])

x_signal = np.array(x_signal)
x_back = np.array(x_back)

fig = plt.figure(figsize=(13,10))
plt.contourf(xx, yy, z, cmap='bwr', alpha=0.3)
plt.scatter(x_signal[:,0], x_signal[:,1], c="red" , s =0.5, label =

```

```

'Signal Prediction')
plt.scatter(x_back[:,0], x_back[:,1], c="blue" , s =0.5, label =
'Background Prediction')
plt.xlabel('X[a.u.]', fontsize=15)
plt.ylabel('Y[a.u.]', fontsize=15)
plt.title('Scatter plot result of classification NN ', fontsize=15)
legend = plt.legend(loc = 'upper left', fontsize=15)
legend.legendHandles[0]._sizes = [30]
legend.legendHandles[1]._sizes = [30]

```

Plottiamo i risultati del training ovvero l'andamento delle loss e dell'accuracy al variare delle epoch e il plot del risultato della classificazione sull'intero dataset con il decision boundary:

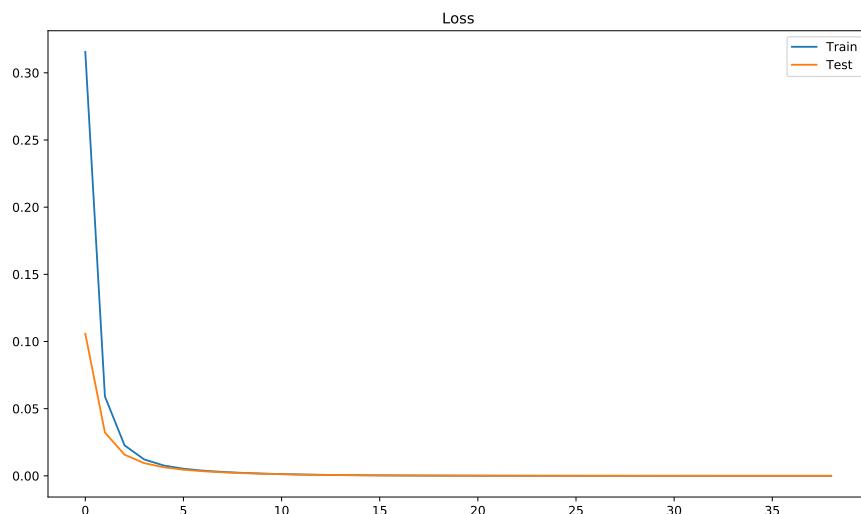


Figura 7.11: Loss della classificazione binaria al variare delle epoch utilizzando il pacchetto Tensorflow con API Keras per costruire il MLP come nell'esempio di TMVA

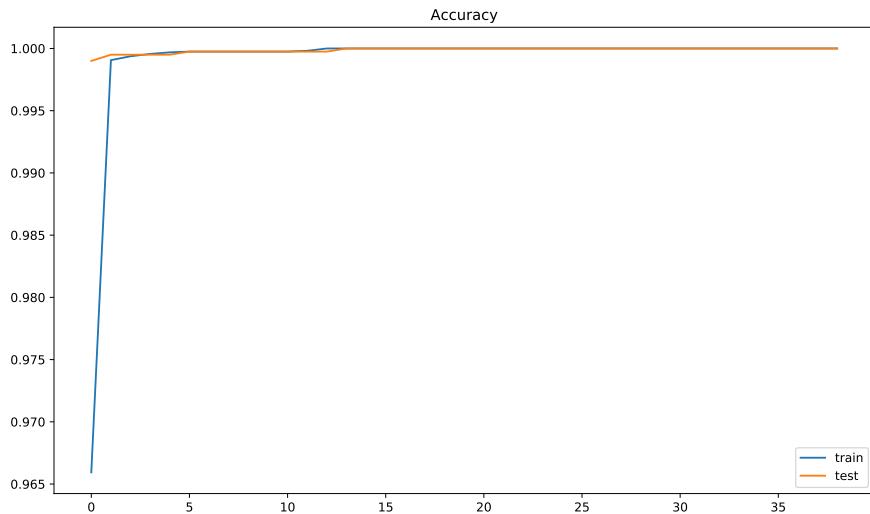


Figura 7.12: Accuracy della classificazione binaria al variare delle epoch utilizzando il pacchetto **Tensorflow** con API **Keras** per costruire il MLP come nell'esempio di **TMVA**

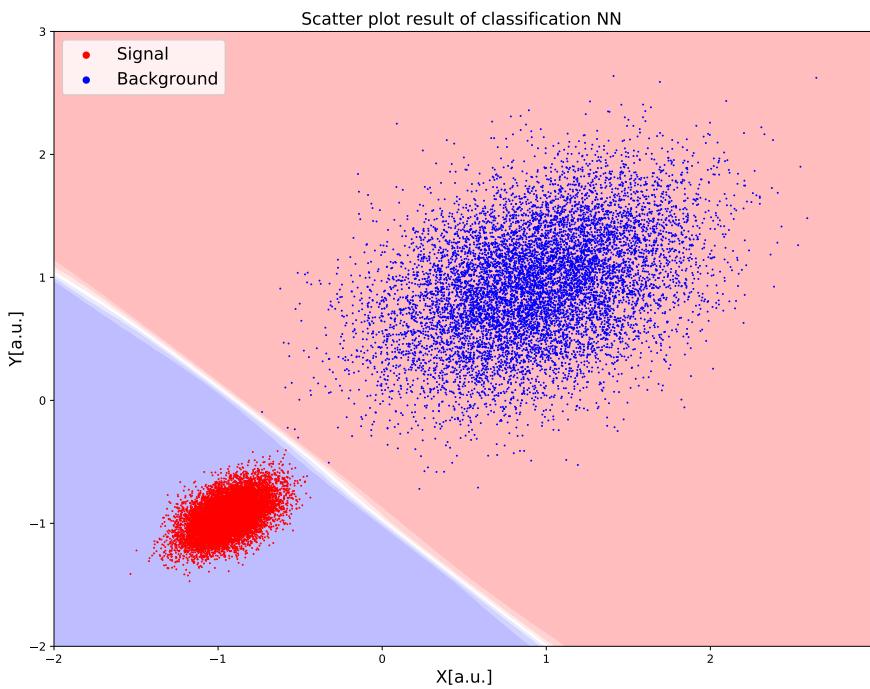


Figura 7.13: Plot del confine decisionale del MLP implementato con **Tensorflow**. Sono anche rappresentati i dati per intero anche se il modello è stato allenato solo su una parte di essi.

# Bibliografia

- [BV18] David Blackman e Sebastiano Vigna. *Scrambled Linear Pseudorandom Number Generators*. 2018. eprint: [arXiv:1805.01407](https://arxiv.org/abs/1805.01407).
- [CDF] CDF. *CDF*, [https://www-cdf.fnal.gov/physics/statistics/notes/cdf8032\\_trandompitfalls.pdf](https://www-cdf.fnal.gov/physics/statistics/notes/cdf8032_trandompitfalls.pdf)
- [HK95] Andreas Hocker e Vakhtang Kartvelishvili. *SVD Approaching to Data Unfolding*. 1995. eprint: [arXiv:9509307](https://arxiv.org/abs/9509307).
- [J.H02] Nicholas J.Higham. *Accuracy and Stability of Numerical Algorithms*. 2<sup>a</sup> ed. Philadelphia, PA: Society for Industrial e Applied Mathematics, 2002. ISBN: 0898715210.
- [Pre+07] William H. Press et al. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. 3<sup>a</sup> ed. New York, NY, USA: Cambridge University Press, 2007. ISBN: 0521880688, 9780521880688.
- [Wik] Wikipedia. *LGC*, [https://en.wikipedia.org/wiki/Linear\\_congruential\\_generator](https://en.wikipedia.org/wiki/Linear_congruential_generator).