

codes & further results

Bianca Sofia Pinolini

31 January 2020

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

# Exercise 1

## Middle-Square Method

Below we show an example of the middle-square method algorithm with the seed 0540, that is one of the seeds that breaks the algorithm in few steps.

```
1  # first of all we insert the seed
2  seed = '0540'
3
4  # then we introduce some variables useful for the code
5  number = int(seed);
6  PR_list = []
7  counter = 0
8
9  # we implement the middle-square method loop
10 while number not in PR_list:
11     # we count the number of steps the algorithm does before repeating itself
12     counter += 1
13     # we square the number
14     squared = number * number
15     # then we convert the squared number into a string
16     squared = str(squared)
17     # we insert enough zeros on the left to obtain a 8-digit number
18     squared = squared.zfill(8)
19     # we cut the string considering the central 4 digits
20     squared = squared[2:6]
21     # we append the number to the pseudo-random generated list
22     PR_list.append(number)
23     # finally we convert again the string into an integer
24     number = int(squared)
25
26 print(f"We started with the seed {seed} and the algorithm generated"
27       f"a list of {counter} pseudo-random numbers:\n"
28       f"{PR_list}")
```

We started with the seed 0540 and the algorithm generated a list of 3 pseudo-random numbers:  
[2916, 5030, 3009]

# Linear Congruent Generator

We then show a basic implementation of a LCG algorithm through which we have generated  $3 \cdot 10^5$  pseudo-random numbers.

```

1  # we insert the seed
2  seed = '1'
3  # we then insert the three parameters
4  m = 2**32
5  a = 1103515245
6  c = 12345
7
8  # we introduce some variables useful for the code
9  number = int(seed);
10 PR_list = []
11 counter = 0
12
13 # we implement the LCG method loop
14 while (number not in PR_list) and (counter <= 300000):
15     counter += 1
16     # we append the number to the pseudo-random generated list
17     PR_list.append(number)
18     # we calculate the next number
19     number = (a * number + c) % m

```

We show the results of the diehard tests over the  $3 \cdot 10^5$  LCG generated numbers.

dieharder version 3.31.1 Copyright 2003 Robert G. Brown						
rng_name	filename		rands/second			
mt19937	list_num		1.17e+08			
test_name	ntup	tsamples	psamples	p-value	Assessment	
diehard_birthdays	0	100	100	0.81483922	PASSED	
diehard_operm5	0	1000000	100	0.99031984	PASSED	
diehard_rank_32x32	0	40000	100	0.65008259	PASSED	
diehard_rank_6x8	0	100000	100	0.49192625	PASSED	
diehard_bitstream	0	2097152	100	0.59717560	PASSED	
diehard_opso	0	2097152	100	0.66814623	PASSED	
diehard_oqso	0	2097152	100	0.77412657	PASSED	
diehard_dna	0	2097152	100	0.62912602	PASSED	
diehard_count_1s_str	0	256000	100	0.64644330	PASSED	
diehard_count_1s_byt	0	256000	100	0.32705120	PASSED	
diehard_parking_lot	0	12000	100	0.72516127	PASSED	
diehard_2dsphere	2	8000	100	0.05783582	PASSED	
diehard_3dsphere	3	4000	100	0.06780386	PASSED	
diehard_squeeze	0	100000	100	0.80187447	PASSED	
diehard_sums	0	100	100	0.05368740	PASSED	
diehard_runs	0	100000	100	0.86886197	PASSED	
diehard_runs	0	100000	100	0.41828659	PASSED	
diehard_craps	0	200000	100	0.98627108	PASSED	
diehard_craps	0	200000	100	0.22121224	PASSED	
marsaglia_tsang_gcd	0	10000000	100	0.21589127	PASSED	
marsaglia_tsang_gcd	0	10000000	100	0.91959953	PASSED	
sts_monobit	1	100000	100	0.55909931	PASSED	

sts_runs	2	100000	100	0.97976036	PASSED
sts_serial	1	100000	100	0.05043560	PASSED
sts_serial	2	100000	100	0.89906202	PASSED
sts_serial	3	100000	100	0.16905092	PASSED
sts_serial	3	100000	100	0.47947275	PASSED
sts_serial	4	100000	100	0.29247656	PASSED
sts_serial	4	100000	100	0.55415936	PASSED
sts_serial	5	100000	100	0.78512023	PASSED
sts_serial	5	100000	100	0.76622129	PASSED
sts_serial	6	100000	100	0.86496127	PASSED
sts_serial	6	100000	100	0.87126971	PASSED
sts_serial	7	100000	100	0.24034433	PASSED
sts_serial	7	100000	100	0.99560203	WEAK
sts_serial	8	100000	100	0.24716774	PASSED
sts_serial	8	100000	100	0.19368405	PASSED
sts_serial	9	100000	100	0.94698903	PASSED
sts_serial	9	100000	100	0.10267715	PASSED
sts_serial	10	100000	100	0.82003813	PASSED
sts_serial	10	100000	100	0.64803908	PASSED
sts_serial	11	100000	100	0.92197857	PASSED
sts_serial	11	100000	100	0.17148599	PASSED
sts_serial	12	100000	100	0.14731918	PASSED
sts_serial	12	100000	100	0.05733969	PASSED
sts_serial	13	100000	100	0.05148452	PASSED
sts_serial	13	100000	100	0.20896485	PASSED
sts_serial	14	100000	100	0.02235022	PASSED
sts_serial	14	100000	100	0.06535739	PASSED
sts_serial	15	100000	100	0.02153552	PASSED
sts_serial	15	100000	100	0.54350954	PASSED
sts_serial	16	100000	100	0.01493225	PASSED
sts_serial	16	100000	100	0.13836832	PASSED
rgb_bitdist	1	100000	100	0.32037734	PASSED
rgb_bitdist	2	100000	100	0.91768410	PASSED
rgb_bitdist	3	100000	100	0.96704664	PASSED
rgb_bitdist	4	100000	100	0.41044607	PASSED
rgb_bitdist	5	100000	100	0.09446943	PASSED
rgb_bitdist	6	100000	100	0.89860343	PASSED
rgb_bitdist	7	100000	100	0.72806879	PASSED
rgb_bitdist	8	100000	100	0.86520785	PASSED
rgb_bitdist	9	100000	100	0.79495436	PASSED
rgb_bitdist	10	100000	100	0.28514686	PASSED
rgb_bitdist	11	100000	100	0.60352297	PASSED
rgb_bitdist	12	100000	100	0.53590102	PASSED
rgb_minimum_distance	2	10000	1000	0.45267143	PASSED
rgb_minimum_distance	3	10000	1000	0.97542148	PASSED
rgb_minimum_distance	4	10000	1000	0.45664619	PASSED
rgb_minimum_distance	5	10000	1000	0.47773072	PASSED
rgb_permutations	2	100000	100	0.88800107	PASSED
rgb_permutations	3	100000	100	0.04704121	PASSED
rgb_permutations	4	100000	100	0.19903813	PASSED
rgb_permutations	5	100000	100	0.24460847	PASSED
rgb_lagged_sum	0	1000000	100	0.81313342	PASSED
rgb_lagged_sum	1	1000000	100	0.05785824	PASSED
rgb_lagged_sum	2	1000000	100	0.84216172	PASSED
rgb_lagged_sum	3	1000000	100	0.35172939	PASSED
rgb_lagged_sum	4	1000000	100	0.10564525	PASSED

rgb_lagged_sum	5	1000000	100	0.41653619	PASSED
rgb_lagged_sum	6	1000000	100	0.85024757	PASSED
rgb_lagged_sum	7	1000000	100	0.65139417	PASSED
rgb_lagged_sum	8	1000000	100	0.58143401	PASSED
rgb_lagged_sum	9	1000000	100	0.21014439	PASSED
rgb_lagged_sum	10	1000000	100	0.26884122	PASSED
rgb_lagged_sum	11	1000000	100	0.42882978	PASSED
rgb_lagged_sum	12	1000000	100	0.80806841	PASSED
rgb_lagged_sum	13	1000000	100	0.58272134	PASSED
rgb_lagged_sum	14	1000000	100	0.99693404	WEAK
rgb_lagged_sum	15	1000000	100	0.97187243	PASSED
rgb_lagged_sum	16	1000000	100	0.04553690	PASSED
rgb_lagged_sum	17	1000000	100	0.22088126	PASSED
rgb_lagged_sum	18	1000000	100	0.62223508	PASSED
rgb_lagged_sum	19	1000000	100	0.47534612	PASSED
rgb_lagged_sum	20	1000000	100	0.43828479	PASSED
rgb_lagged_sum	21	1000000	100	0.99957506	WEAK
rgb_lagged_sum	22	1000000	100	0.52747275	PASSED
rgb_lagged_sum	23	1000000	100	0.89178547	PASSED
rgb_lagged_sum	24	1000000	100	0.97914370	PASSED
rgb_lagged_sum	25	1000000	100	0.86003129	PASSED
rgb_lagged_sum	26	1000000	100	0.30854697	PASSED
rgb_lagged_sum	27	1000000	100	0.31856673	PASSED
rgb_lagged_sum	28	1000000	100	0.94969432	PASSED
rgb_lagged_sum	29	1000000	100	0.66753010	PASSED
rgb_lagged_sum	30	1000000	100	0.18311290	PASSED
rgb_lagged_sum	31	1000000	100	0.30699678	PASSED
rgb_lagged_sum	32	1000000	100	0.94151934	PASSED
rgb_kstest_test	0	10000	1000	0.66478661	PASSED
dab_bytedistrib	0	51200000	1	0.61783759	PASSED
dab_dct	256	50000	1	0.49181118	PASSED
Preparing to run test	207.	ntuple = 0			
dab_filltree	32	15000000	1	0.67538312	PASSED
dab_filltree	32	15000000	1	0.51325969	PASSED
Preparing to run test	208.	ntuple = 0			
dab_filltree2	0	5000000	1	0.15033650	PASSED
dab_filltree2	1	5000000	1	0.81586192	PASSED
Preparing to run test	209.	ntuple = 0			
dab_monobit2	12	65000000	1	0.60917254	PASSED

## The xoroshiro128+ Algorithm

Below are shown the xoroshiro128+ algorithm implemented by us and the results of the diehard test made on the  $3 \cdot 10^5$  numbers generated with the algorithm.

```
1  #include <stdint.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  //Parameters of the algorithm
6  #define a 24
7  #define b 16
8  #define c 37
9
10 //Global variable used to store temporary numbers
11 static unsigned long long int s[2];
12
13 //Function for the bitwise left rotation
14 static unsigned long long int rotl(const unsigned long long int x, int k){
15     return (x<<k)|(x>>(64-k));
16 }
17
18 //Function that generates the next random number
19 unsigned long long int next(void){
20     const unsigned long long int s0 = s[0];
21     unsigned long long int s1 = s[1];
22
23     //Next random number
24     const unsigned long long int result = s0 + s1;
25
26     //Upgrade of s0 and s1
27     s1 ^= s0;
28     s[0] = rotl(s0, a) ^ s1 ^ (s1 << b);
29     s[1] = rotl(s1, c);
30
31     return result;
32 }
33
34 int main(){
35     int i, j, N;
36
37     //Define the output of the code
38     FILE *numbers;
39     unsigned long long int *list;
40
41     //Define the seed (first two numbers generated with LGC)
42     s[0] = 1103527590;
43     s[1] = 2524885223;
44
45     //Define the list of the generated numbers (max N)
46     N = 300000;
47     list = malloc(N * sizeof(unsigned long long int));
48
49     for(i=0; i<N; i++){
50         // Generate through the 'next' function a new random number
51         list[i] = next();
```

```

52      //Check if the number has already been extracted
53      for(j=0; j<i; j++){
54          if(list[i]==list[j]){
55              printf("The algorithm entered a loop after %d steps", i);
56              break;
57          }
58      }
59  }
60  return 0;
61 }

```

```

#=====#
#      dieharder version 3.31.1 Copyright 2003 Robert G. Brown      #
#=====#

```

```

#      rng_name      |      filename      | rands/second |
#      mt19937      | xoroshiro_results_e5 | 9.48e+07      |
#=====#

```

```

#      test_name      | ntup | tsamples | psamples | p-value | Assessment
#=====#

```

diehard_birthdays	0	100	100	0.44410025	PASSED
diehard_operm5	0	1000000	100	0.74915042	PASSED
diehard_rank_32x32	0	40000	100	0.40957517	PASSED
diehard_rank_6x8	0	100000	100	0.97949975	PASSED
diehard_bitstream	0	2097152	100	0.22928778	PASSED
diehard_opso	0	2097152	100	0.43138734	PASSED
diehard_oqso	0	2097152	100	0.94547019	PASSED
diehard_dna	0	2097152	100	0.44075379	PASSED
diehard_count_1s_str	0	256000	100	0.95057358	PASSED
diehard_count_1s_byt	0	256000	100	0.99977004	WEAK
diehard_parking_lot	0	12000	100	0.83367635	PASSED
diehard_2dsphere	2	8000	100	0.97940937	PASSED
diehard_3dsphere	3	4000	100	0.18355613	PASSED
diehard_squeeze	0	100000	100	0.94554620	PASSED
diehard_sums	0	100	100	0.01634319	PASSED
diehard_runs	0	100000	100	0.87568350	PASSED
diehard_runs	0	100000	100	0.07946039	PASSED
diehard_craps	0	200000	100	0.89123459	PASSED
diehard_craps	0	200000	100	0.54721842	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.17436311	PASSED
marsaglia_tsang_gcd	0	10000000	100	0.09858488	PASSED
sts_monobit	1	100000	100	0.73867451	PASSED
sts_runs	2	100000	100	0.87874252	PASSED
sts_serial	1	100000	100	0.77210997	PASSED
sts_serial	2	100000	100	0.44736700	PASSED
sts_serial	3	100000	100	0.62900522	PASSED
sts_serial	3	100000	100	0.70830223	PASSED
sts_serial	4	100000	100	0.70052035	PASSED
sts_serial	4	100000	100	0.71526063	PASSED
sts_serial	5	100000	100	0.14252726	PASSED
sts_serial	5	100000	100	0.16449485	PASSED
sts_serial	6	100000	100	0.45336555	PASSED
sts_serial	6	100000	100	0.23647977	PASSED
sts_serial	7	100000	100	0.78122401	PASSED
sts_serial	7	100000	100	0.44969079	PASSED
sts_serial	8	100000	100	0.23608312	PASSED
sts_serial	8	100000	100	0.89400452	PASSED



sts_serial	9	100000	100	0.90597438	PASSED
sts_serial	9	100000	100	0.80743205	PASSED
sts_serial	10	100000	100	0.91763411	PASSED
sts_serial	10	100000	100	0.42908270	PASSED
sts_serial	11	100000	100	0.84173165	PASSED
sts_serial	11	100000	100	0.51882790	PASSED
sts_serial	12	100000	100	0.76489961	PASSED
sts_serial	12	100000	100	0.95859060	PASSED
sts_serial	13	100000	100	0.45676442	PASSED
sts_serial	13	100000	100	0.93863561	PASSED
sts_serial	14	100000	100	0.31173789	PASSED
sts_serial	14	100000	100	0.65965546	PASSED
sts_serial	15	100000	100	0.38216396	PASSED
sts_serial	15	100000	100	0.95557148	PASSED
sts_serial	16	100000	100	0.62328738	PASSED
sts_serial	16	100000	100	0.68453701	PASSED
rgb_bitdist	1	100000	100	0.99069364	PASSED
rgb_bitdist	2	100000	100	0.52824802	PASSED
rgb_bitdist	3	100000	100	0.67588244	PASSED
rgb_bitdist	4	100000	100	0.11074895	PASSED
rgb_bitdist	5	100000	100	0.80043171	PASSED
rgb_bitdist	6	100000	100	0.82136035	PASSED
rgb_bitdist	7	100000	100	0.49837238	PASSED
rgb_bitdist	8	100000	100	0.98271587	PASSED
rgb_bitdist	9	100000	100	0.58418802	PASSED
rgb_bitdist	10	100000	100	0.67102697	PASSED
rgb_bitdist	11	100000	100	0.16370726	PASSED
rgb_bitdist	12	100000	100	0.74917838	PASSED
rgb_minimum_distance	2	10000	1000	0.44620377	PASSED
rgb_minimum_distance	3	10000	1000	0.19767294	PASSED
rgb_minimum_distance	4	10000	1000	0.77940120	PASSED
rgb_minimum_distance	5	10000	1000	0.24966853	PASSED
rgb_permutations	2	100000	100	0.58058134	PASSED
rgb_permutations	3	100000	100	0.33079407	PASSED
rgb_permutations	4	100000	100	0.13753410	PASSED
rgb_permutations	5	100000	100	0.47927026	PASSED
rgb_lagged_sum	0	1000000	100	0.17002736	PASSED
rgb_lagged_sum	1	1000000	100	0.36803170	PASSED
rgb_lagged_sum	2	1000000	100	0.16028478	PASSED
rgb_lagged_sum	3	1000000	100	0.68917786	PASSED
rgb_lagged_sum	4	1000000	100	0.91254205	PASSED
rgb_lagged_sum	5	1000000	100	0.47073143	PASSED
rgb_lagged_sum	6	1000000	100	0.83116063	PASSED
rgb_lagged_sum	7	1000000	100	0.39987552	PASSED
rgb_lagged_sum	8	1000000	100	0.95168673	PASSED
rgb_lagged_sum	9	1000000	100	0.49580511	PASSED
rgb_lagged_sum	10	1000000	100	0.95693624	PASSED
rgb_lagged_sum	11	1000000	100	0.67050464	PASSED
rgb_lagged_sum	12	1000000	100	0.93988451	PASSED
rgb_lagged_sum	13	1000000	100	0.05299184	PASSED
rgb_lagged_sum	14	1000000	100	0.18974408	PASSED
rgb_lagged_sum	15	1000000	100	0.74372142	PASSED
rgb_lagged_sum	16	1000000	100	0.23996687	PASSED
rgb_lagged_sum	17	1000000	100	0.76985958	PASSED
rgb_lagged_sum	18	1000000	100	0.35855000	PASSED
rgb_lagged_sum	19	1000000	100	0.97449153	PASSED

rgb_lagged_sum	20	1000000	100	0.98261383	PASSED
rgb_lagged_sum	21	1000000	100	0.28171518	PASSED
rgb_lagged_sum	22	1000000	100	0.22621883	PASSED
rgb_lagged_sum	23	1000000	100	0.83387295	PASSED
rgb_lagged_sum	24	1000000	100	0.95490084	PASSED
rgb_lagged_sum	25	1000000	100	0.82830350	PASSED
rgb_lagged_sum	26	1000000	100	0.56810954	PASSED
rgb_lagged_sum	27	1000000	100	0.08259279	PASSED
rgb_lagged_sum	28	1000000	100	0.39608992	PASSED
rgb_lagged_sum	29	1000000	100	0.46941056	PASSED
rgb_lagged_sum	30	1000000	100	0.60976842	PASSED
rgb_lagged_sum	31	1000000	100	0.11799127	PASSED
rgb_lagged_sum	32	1000000	100	0.51092186	PASSED
rgb_kstest_test	0	10000	1000	0.42010989	PASSED
dab_bytedistrib	0	51200000	1	0.25709360	PASSED
dab_dct	256	50000	1	0.50775262	PASSED
Preparing to run <b>test</b>	207.	ntuple = 0			
dab_filltree	32	15000000	1	0.67232847	PASSED
dab_filltree	32	15000000	1	0.79648001	PASSED
Preparing to run <b>test</b>	208.	ntuple = 0			
dab_filltree2	0	5000000	1	0.08238921	PASSED
dab_filltree2	1	5000000	1	0.67039797	PASSED
Preparing to run <b>test</b>	209.	ntuple = 0			
dab_monobit2	12	65000000	1	0.75766266	PASSED

## The Rejection Sampling

It is shown here the rejection sampling algorithm to build a Landau distribution obtained with PR-numbers generated through the xoroshiro128+ algorithm.

```
1  import numpy as np
2  from scipy.stats import moyal
3  import matplotlib.pyplot as plt
4
5  # we open the file containing the number generated with the xoroshiro128+ algorithm
6  with open('xoroshiro_results', 'r') as f:
7      list_num = f.read().splitlines()
8
9  # we split the list in half creating two lists
10 X = list_num[:150000]
11 Y = list_num[150000:]
12
13 # since we want to generate a Landau distribution with domain [-3,18] e codomain [0,0.25]
14 # we normalize the two lists to these intervals
15 div = 2**64-1 # this is the maximum number that can be produced with 64 bits
16 j = 0
17 for j in range(150000):
18     X[j] = (X[j] / div)*(18+3)-3.
19     Y[j] = (Y[j] / div)*0.25
20
21 # we can now implement the rejection-sampling algorithm
22 keep_list = np.array([])
23 for i in range(150000):
24     xr = X[i]
25     yr = Y[i]
26     L = moyal.pdf(xr)
27     if (yr > L): continue
28     keep_list = np.append(keep_list, xr)
```

After having generated the desired distribution, we can fit it.

```
29 import ROOT as r
30
31 # first of all we define the histogram of data and we fill it
32 h = r.TH1F("h",180,-3,18);
33
34 for i in range (150000):
35     h.Fill(keep_list[i]);
36
37 # we then define the Landau function to fit the data
38 TF1 *landau_f = new TF1("landau_f", "[2]*TMath::Landau(x,[0],[1])", -3, 18);
39 landau_f->SetParameter(2,0.5);
40
41 # then we draw the histogram of the data and we fit it with the function described above
42 c = new TCanvas();
43 h->SetTitle("");
44 h->SetLineColor(kBlue);
45 h->SetLineWidth(1);
46 h->SetStats(0);
47 h->Fit("landau_f");
48 h->GetXaxis()->SetTitle("X");
49 h->GetYaxis()->SetTitle("Y");
```

```

50  h->Draw();
51  c->Draw();
52
53  # we acquire the chi2 and the NDF to calculate the reduced chi2
54  Double_t chi2 = landau_f->GetChisquare();
55  Double_t NDF = landau_f->GetNDF();
56  std::cout << "chi2/NDF = " << chi2/NDF << endl;
57
58  # we finally get the probability
59  Double_t p = landau_f->GetProb();
60  std::cout << "p = " << p << endl;

```

# Exercise 2

## Direct Summation

Below the code to implement the Direct Summation algorithm.

```
1  *****
2  *
3  * Direct_Sum.c
4  *
5  * Compute the Euler-Mascheroni constant with direct
6  * sum method.
7  *
8  * gamma = (sum_i^N)(1/i) - ln(N)
9  *
10 *****/
11
12 #define MAIN_C
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <math.h>
16
17 // FLOAT
18
19 float Direct_Sum_f(int n){
20     // Compute the sum of 1/i with i in [1, n]
21
22     float s = 0.;
23     for (int i = 1; i<=n; i++){
24         s += 1/(float)i;
25     }
26     return s;
27 }
28
29 float gamma_D_f(int n){
30     // Compute an approximation of the gamma function up to n iterations
31
32     float gamma_t = 0.;
33     gamma_t = Direct_Sum_f(n) - logf((float)n);
34     return gamma_t;
35 }
```

```

36 // LONG DOUBLE
37
38 long double Direct_Sum_ld(int n){
39     // Compute the sum of 1/i with i in [1, n]
40
41     long double s = 0.;
42     for (int i = 1; i<=n; i++){
43         s += 1/(long double)i;
44     }
45     return s;
46 }
47
48 long double gamma_D_ld(int n){
49     // Compute an approximation of the gamma function up to n iterations
50
51     long double gamma_t = 0.;
52     gamma_t = Direct_Sum_ld(n) - logl((long double)n);
53     return gamma_t;
54 }
55
56 //////////////////////////////////////
57
58 int main (){
59
60     // True values of gamma in the desired sizes
61     long double gamma_true_ld = 0.57721566490153286;
62     float gamma_true_f = 0.5772156;
63
64     // Definitions of the output files
65     FILE* errors;
66     FILE* gammas;
67
68     errors = fopen("Direct_Sum.txt", "w+");
69     gammas = fopen("D_S_gammas.txt", "w+");
70
71     // Definitions of some useful variables
72     float f, err_f;
73     long double ld, err_ld;
74     int iter = 0;
75
76     // Loop to implement the algorithm with 10^1, 10^2, ..., 10^9 iterations
77     for (int i=1; i<=9; i++){
78
79         iter = (int)pow(10, i);
80
81         // FLOAT
82
83         // Computing gamma and the relative error and write them in two txt files
84         f = gamma_D_f(iter);
85         err_f = (f - gamma_true_f) / gamma_true_f;
86         fprintf(errors, "%.70f\n", fabsf(err_f));
87         fprintf(gammas, "%.70f\n", f);

```

```

88         // LONG DOUBLE
89
90         // Computing gamma and the relative error and write them in two txt files
91         ld      = gamma_D_ld(iter);
92         err_ld = (ld - gamma_true_ld) / gamma_true_ld;
93         fprintf(errors, "%.70Lf\n", fabs1(err_ld));
94         fprintf(gammas, "%.70Lf\n", ld);
95     }
96
97 }

```

## Sorted Summation

The Sorted Summation algorithm is actually the same as the Direct Summation one, except for the functions `Direct_Sum_f` and `Direct_Sum_ld`, since the summation is now performed from the smallest number to the largest one, i.e. from  $i = n$  to  $i = 1$ .

```

1  float Sorted_Sum_f(int n){
2      // Compute the sum of 1/i with i in [1, n]
3
4      float s = 0.;
5      for (int i = n; i>=1; i--){
6          s += 1/(float)i;
7      }
8      return s;
9  }

10 long double Sorted_Sum_ld(int n){
11     // Compute the sum of 1/i with i in [1, n]
12
13     long double s = 0.;
14     for (int i = n; i>=1; i--){
15         s += 1/(long double)i;
16     }
17     return s;
18 }

```

## Pairwise Summation

The Pairwise Summation algorithm works in a very different way with respect to the Direct and Sorted Summation ones. Let's analyse the code step by step.

```
1  /*****
2  *
3  *  Pairwise_Sum.c
4  *
5  *  Compute the Euler-Mascheroni constant with pairwise
6  *  sum method.
7  *
8  *  gamma = (sum_i^N)(1/i) - ln(N)
9  *
10 *****/
11
12 #define MAIN_C
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <math.h>
```

First of all, we define some global variables:

- `nums_f` and `nums_ld`: The two arrays which contain the numbers of the series for float and long double representations – respectively.
- `sf` and `sld`: The two numbers that represent the summations of the float and long double numbers contained in the two arrays above.

```
16 float *nums_f;
17 long double *nums_ld;
18 float sf;
19 long double sld;
```

Let's first analyse the Float functions of the algorithm.

First of all, we define a function that computes the Euler-Mascheroni constant up to  $n$  iterations.

```
20 // FLOAT
21
22 float gamma_P_f(int n){
23     // Compute an approximation of the gamma function up to n iterations
24
25     float gamma_t;
26     Pairwise_Sum_f(n);
27     gamma_t = sf - logf((float)n);
28     return gamma_t;
29 }
```

As can be seen, this function calls another function `Pairwise_Sum_f`, that is the one which fills the `nums_f` array with the numbers of the series. We don't want to fill this array with  $N = 10^8$  or  $N = 10^9$  numbers, since the occupied memory could become a problem. We therefore decide to first sum the numbers in pairs and then to fill `nums_f` with these summations, i.e. with  $\frac{N}{2}$  float numbers.



```

30 void Pairwise_Sum_f(int n){
31     int j = 0;
32
33     // we dynamically allocate the memory for creating the array
34     nums_f = (float*)calloc((int)(n/2), sizeof(float));
35
36     if (nums_f == NULL) {
37         printf("Error in creating the array nums_f!\n");
38     }
39
40     // we fill the array
41     for(int i=1; i<=n; i=i+2){
42         k = (float) i;
43         nums_f[j] = 1/k + 1/(k+1.);
44         j ++;
45     }
46
47     // we call the function PWf to perform the summation
48     PWf(0, (int)(n/2)-1);
49 }

```

After having filled the array `nums_f` with the numbers of the series, we call the function `PWf` that performs the Pairwise Summation itself through a recursive method. This function takes as input the first and the last addresses of the array `nums_f` and halves it sequentially until it reaches vectors of at most 2 numbers. At this point, the summation is performed and `sf` is updated.

```

50 void PWf(int start, int end){
51
52     int len = end - start + 1;
53
54     if (len<=2){
55         for (int i = 0; i<len; i++) {
56             sf += nums_f[start+i];
57         }
58     }
59     else {
60         int m = (int)(len/2);
61
62         PWf(start,start+m-1);
63         PWf(start+m,end);
64     }
65 }

```

These functions are defined in the same way for Long Doubles numbers, except for the `Pairwise_Sum_f` function, that is a bit different. In fact, with Long Double numbers, even  $\frac{N}{2}$  entries were too much, so we decided to divide it in four parts, not in two. To do so, we had to consider  $N = 10$  as a separate case, since it is impossible to divide 10 by 4. We decided then to fill the vector `nums_ld` with all the 10 numbers of the series, since it was not a memory problem. Below the code of the function `Pairwise_Sum_ld` is shown in details.

```

66 void Pairwise_Sum_ld(int n){
67     int j = 0;
68
69     if (n == 10) {
70
71         // allocating the memory for creating the array
72         nums_ld = (long double*)calloc(n, sizeof(long double));
73
74         if (nums_ld == NULL) {
75             printf("Error in creating the array nums_ld!\n");
76         }
77
78         // filling the array
79         for(int i=1; i<=n; i++){
80             nums_ld[j] = 1/(long double)i;
81             j ++;
82         }
83
84         // calling the PWld function to perform the summation
85         PWld(0, n-1);
86
87     } else {
88
89         // allocating the memory for creating the array
90         nums_ld = (long double*)calloc((int)(n/4), sizeof(long double));
91
92         if (nums_ld == NULL) {
93             printf("Error in creating the array nums_ld!\n");
94         }
95
96         // filling the array
97         long double k;
98
99         for(int i=1; i<=n; i=i+4){
100             k = (long double) i;
101             nums_ld[j] = 1/k + 1/(k+1.) + 1/(k+2.) + 1/(k+3.);
102             j ++;
103         }
104
105         // calling the PWld function to perform the summation
106         PWld(0, (int)(n/4)-1);
107     }
108 }

```

Concerning the main, it is actually the same as the one defined in the lines 58-99 in the Direct Summation algorithm, excepted for the fact that in the for-cycle we added the following lines:

```

sf = 0.;
sld = 0.;
free(nums_f);
free(nums_ld);

```

## Kahan Summation

We now show the code implemented to perform the Kahan Summation algorithm in details.

```
1  /*****
2  *
3  *   Kahan_Sum.c
4  *
5  *   Compute the Euler-Mascheroni constant with Kahan
6  *   sum method.
7  *
8  *   gamma = (sum_i^N)(1/i) - ln(N)
9  *
10 *****/
11
12 #define MAIN_C
13 #include <stdio.h>
14 #include <stdlib.h>
15 #include <math.h>
```

Since the implementation for Float numbers and for Long Double numbers is actually the same, we report only the code written for Float numbers.

First of all, we define a function that computes the Euler-Mascheroni constant up to  $n$  iterations.

```
16 // FLOAT
17
18 float gamma_K_f(int n){
19     // Compute an approximation of the gamma function up to n iterations
20
21     float gamma_t = 0.;
22     gamma_t = Kahan_Sum_f(n) - logf((float)n);
23     return gamma_t;
24 }
```

This first function calls on its turn the function `Kahan_Sum_f`, which actually compute the summation we need through the Kahan Summation algorithm.

```
25 float Kahan_Sum_f(int n){
26     // Compute the sum of 1/i with i in [1, n]
27
28     float s = 0., c = 0., t = 0., y = 0.;
29
30     for (int i = 1; i<=n; i++){
31         y = 1./(float)i - c;
32         t = s + y;
33         c = (t - s) - y;
34         s = t;
35     }
36     return s;
37 }
```

The `main` is then the same as the one written for the Direct Summation algorithm.

## Results

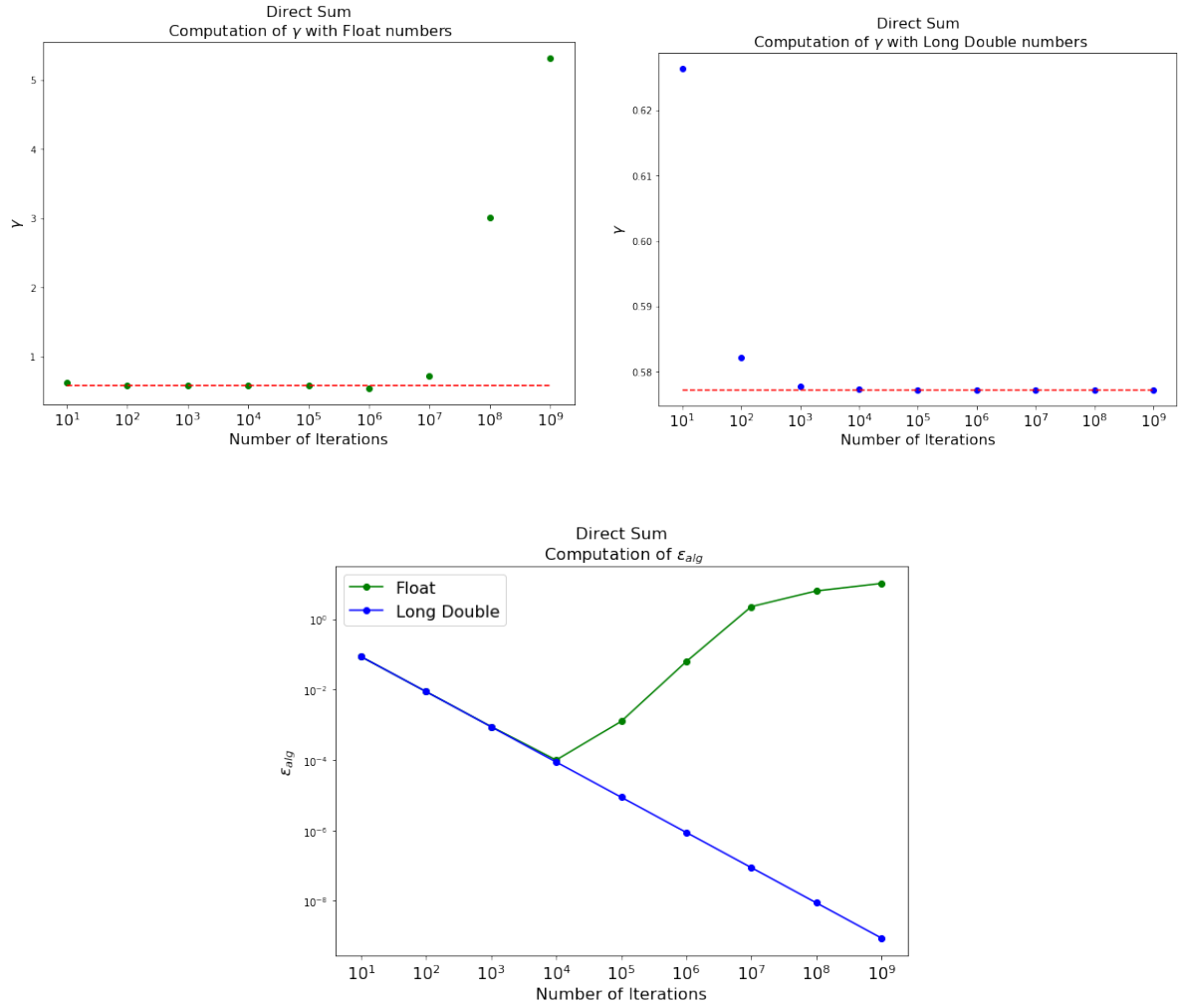
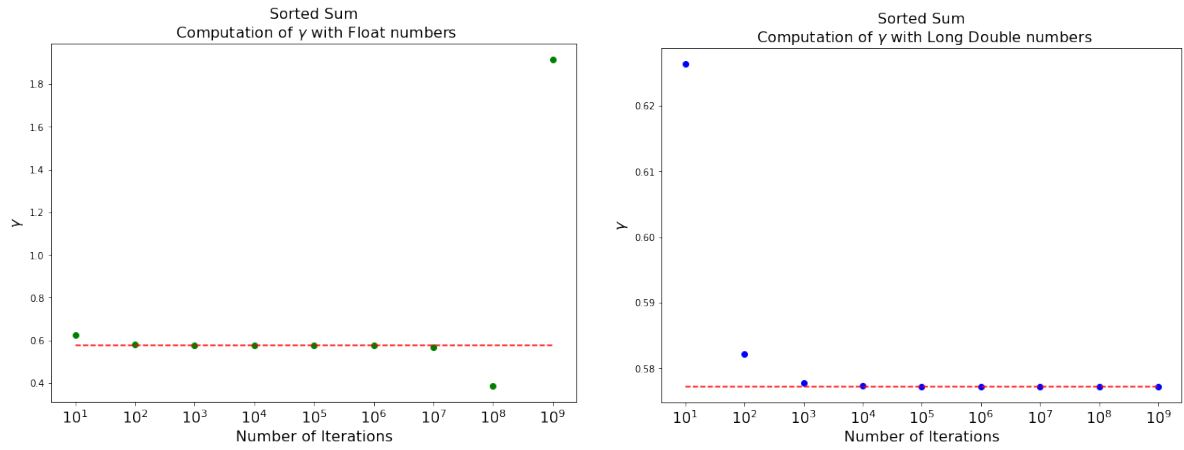


Figure 1: Computation of  $\gamma$  and  $\epsilon_{alg}$  with the Direct Summation algorithm with both Float and Long Double numbers. The correct value of  $\gamma$  is represented with a red dashed line.



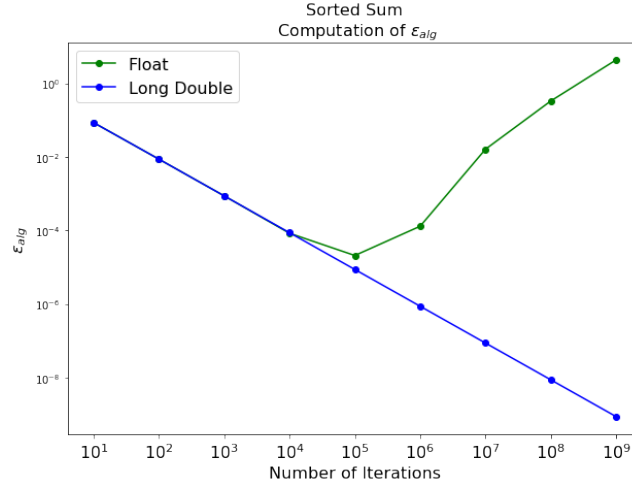


Figure 2: Computation of  $\gamma$  and  $\epsilon_{alg}$  with the Sorted Summation algorithm with both Float and Long Double numbers. The correct value of  $\gamma$  is represented with a red dashed line.

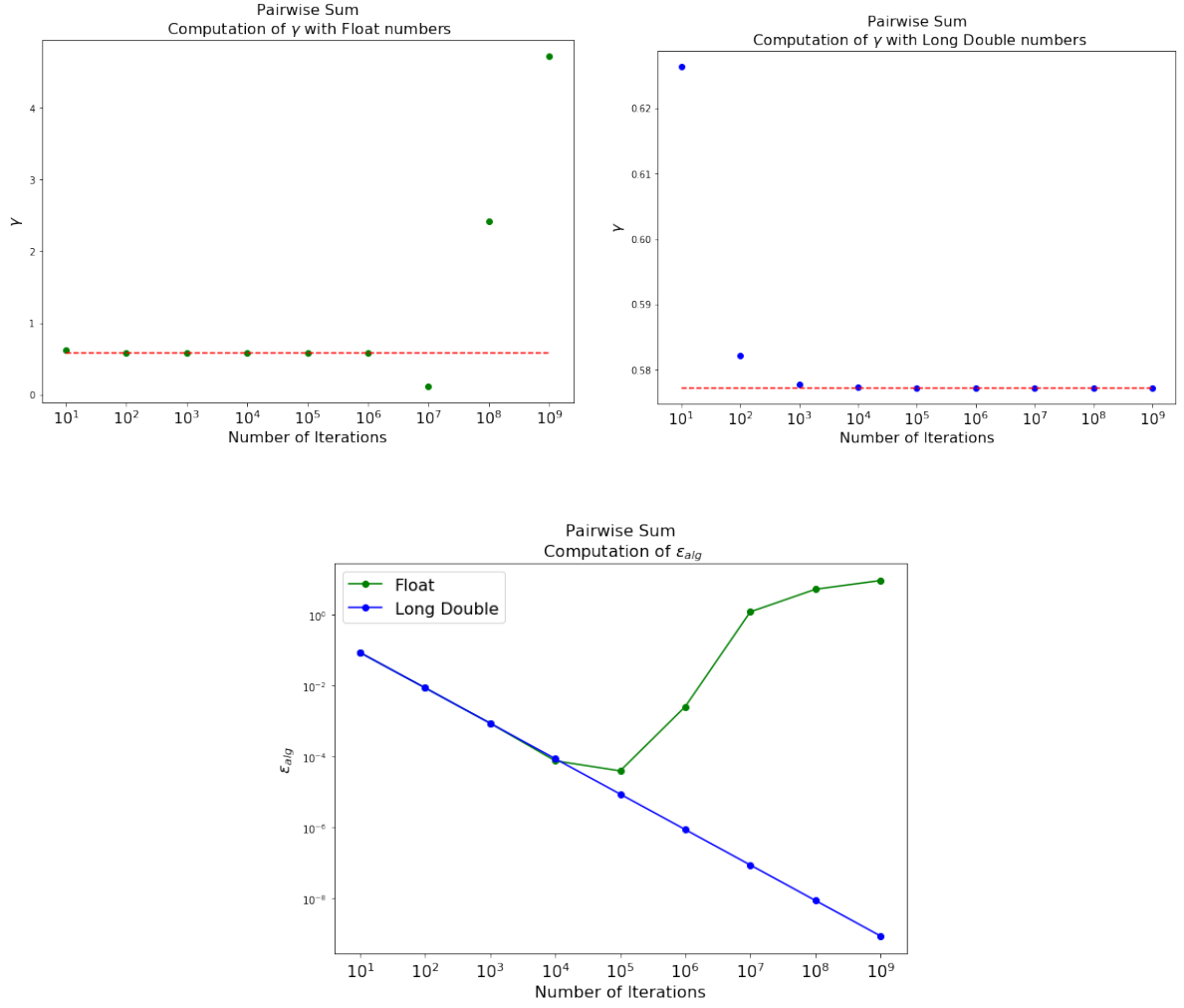


Figure 3: Computation of  $\gamma$  and  $\epsilon_{alg}$  with the Pairwise Summation algorithm with both Float and Long Double numbers. The correct value of  $\gamma$  is represented with a red dashed line.

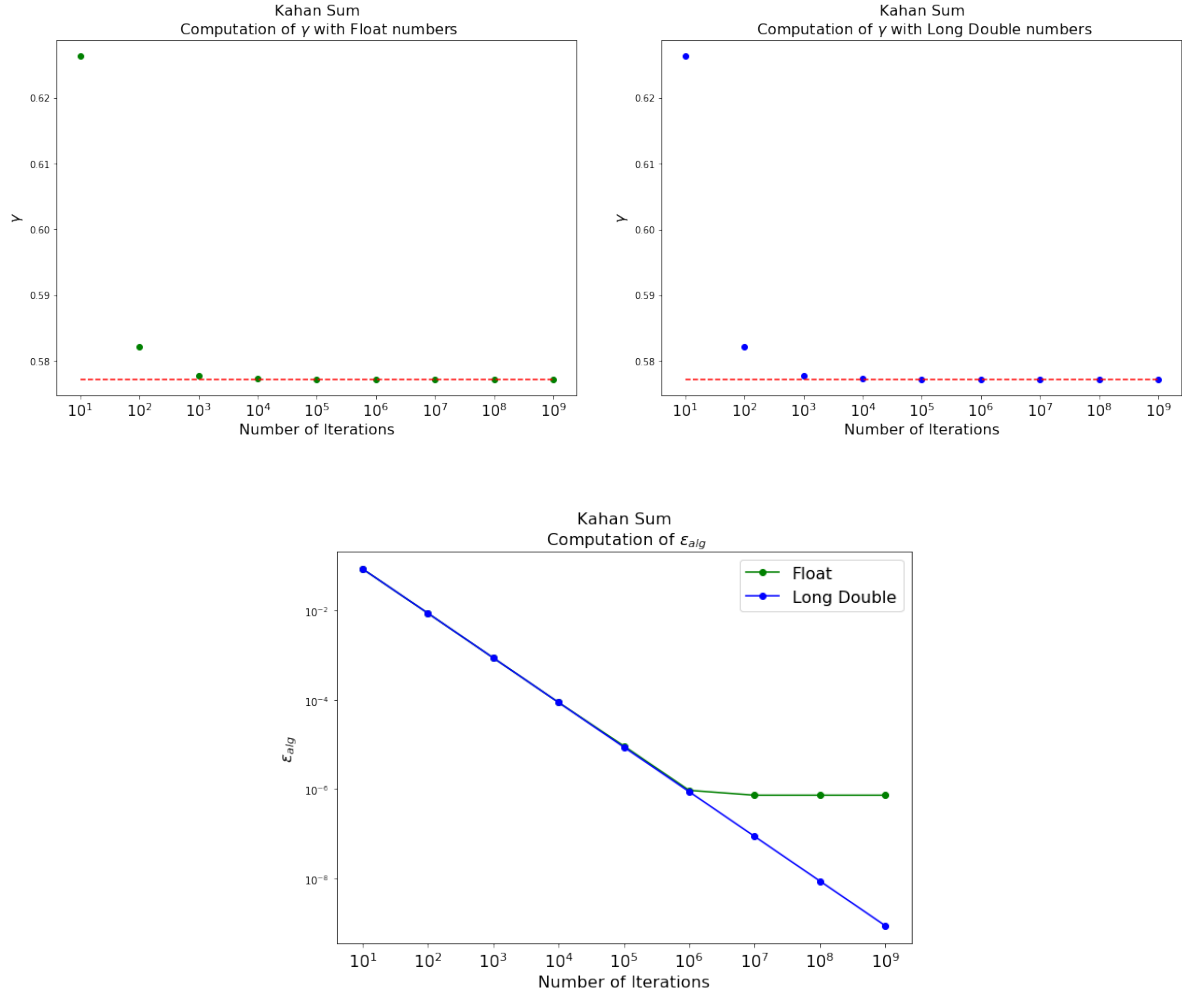


Figure 4: Computation of  $\gamma$  and  $\epsilon_{alg}$  with the Kahan Summation algorithm with both Float and Long Double numbers. The correct value of  $\gamma$  is represented with a red dashed line.

## Comparison

Now we show the code written to draw the plots used to compare the various algorithm. The codes to draw the singular plots are analogous.

```

1  import matplotlib.pyplot as plt
2  import numpy as np
3
4  # exponent of 10 - number of iterations
5  N = 9
6
7  # read the files with the errors
8  with open('Direct_Sum.txt', 'r') as f:
9      list_num_DS = f.read().splitlines()
10  with open('Sorted_Sum.txt', 'r') as f:
11      list_num_SS = f.read().splitlines()
12  with open('Pairwise_Sum.txt', 'r') as f:
13      list_num_PS = f.read().splitlines()
14  with open('Kahan_Sum.txt', 'r') as f:
15      list_num_KS = f.read().splitlines()
16

```

```

17 # write the errors in two lists, one for the float numbers and one for the long double ones
18 F = np.arange(0,2*N-1,2)
19 D = np.arange(1,2*N,2)
20
21 # direct summation
22 gammas_F_DS = []
23 gammas_LD_DS = []
24
25 for i in F:
26     gammas_F_DS.append(abs(float(list_num_DS[i])))
27 for j in D:
28     gammas_LD_DS.append(abs(float(list_num_DS[j])))
29
30 # sorted summation
31 gammas_F_SS = []
32 gammas_LD_SS = []
33
34 for i in F:
35     gammas_F_SS.append(abs(float(list_num_SS[i])))
36 for j in D:
37     gammas_LD_SS.append(abs(float(list_num_SS[j])))
38
39 # pairwise summation
40 gammas_F_PS = []
41 gammas_LD_PS = []
42
43 for i in F:
44     gammas_F_PS.append(abs(float(list_num_PS[i])))
45 for j in D:
46     gammas_LD_PS.append(abs(float(list_num_PS[j])))
47
48 # Kahan summation
49 gammas_F_KS = []
50 gammas_LD_KS = []
51
52 for i in F:
53     gammas_F_KS.append(abs(float(list_num_KS[i])))
54 for j in D:
55     gammas_LD_KS.append(abs(float(list_num_KS[j])))
56
57 # write an array containing the the ticks of the x axis
58 x = np.arange(1,N+1)
59
60 # write a list containing the name of the ticks of the x axis
61 x_name = []
62
63 for i in range(1,N+1):
64     x_name.append(f'$10^{{i}}$')

```

```

65  # draw the plot for the errors for float numbers
66  plt.figure(figsize=(10,7))
67  plt.plot(x, gammas_F_DS, color='blue', marker='o', label='Direct Sum')
68  plt.plot(x, gammas_F_SS, color='magenta', marker='o', label='Sorted Sum')
69  plt.plot(x, gammas_F_PS, color='green', marker='o', label='Pairwise Sum')
70  plt.plot(x, gammas_F_KS, color='red', marker='o', label='Kahan Sum')
71  plt.xticks(x, x_name, fontsize=16)
72  plt.legend(fontsize=16)
73  plt.xlabel('Number of Iterations', fontsize=16)
74  plt.ylabel('$\epsilon_{alg}$', fontsize=16)
75  plt.yscale('log')
76  plt.title("Computation of  $\epsilon_{alg}$  with Float numbers", fontsize=16)
77  plt.show()
78
79  # draw the plots for the errors for long double numbers
80  plt.figure(figsize=(10,7))
81  plt.plot(x, gammas_LD_DS, color='blue', marker='o', label='Direct Sum')
82  plt.plot(x, gammas_LD_SS, color='magenta', marker='o', label='Sorted Sum')
83  plt.plot(x, gammas_LD_PS, color='green', marker='o', label='Pairwise Sum')
84  plt.plot(x, gammas_LD_KS, color='red', marker='o', label='Kahan Sum')
85  plt.xticks(x, x_name, fontsize=16)
86  plt.legend(fontsize=16)
87  plt.xlabel('Number of Iterations', fontsize=16)
88  plt.ylabel('$\epsilon_{alg}$', fontsize=16)
89  plt.yscale('log')
90  plt.title("Computation of  $\epsilon_{alg}$  with Long Double numbers", fontsize=16)
91  plt.show()

```



# Exercise 3

## Generation of the MC data-sample, Estimation of the parameters through ML and LS methods

```
1  #include "RooRealVar.h"
2  #include "RooConstVar.h"
3  #include "RooGaussian.h"
4  #include "RooArgusBG.h"
5  #include "RooAddPdf.h"
6  #include "RooDataSet.h"
7  #include "RooPlot.h"
8
9  using namespace RooFit;
10
11  // we first create the two observables (theta,phi) within their ranges [0,pi] and [0,2pi]
12  RooRealVar theta("theta","#theta",0,M_PI);
13  RooRealVar phi("phi","#phi",0,2*M_PI);
14
15  // we then define the three parameters
16  // alpha = 0.65
17  // beta = 0.06
18  // gamma = -0.18
19  RooRealVar alpha("alpha","#alpha",0.65,0.62,0.66);
20  RooRealVar bet("bet","#beta",0.06,0.05,0.075);
21  RooRealVar gam("gam","#gamma",-0.18,-0.2,-0.16);
22
23  // next we build the pdf function with the observables and the parameters previously defined
24  RooAbsPdf* pdf = RooClassFactory::makePdfInstance("pdf", "(3./(4.*M_PI))*(0.5*(1.-alpha) +
25  "(0.5)*(3.*alpha-1)*cos(theta)*cos(theta) - bet*sin(theta)*sin(theta)*cos(2.*phi)-
26  "sqrt(2.)*gam*sin(2.*theta)*cos(phi))", RooArgSet(theta,phi,alpha,bet,gam)) ;
27
28  // we draw the 2D histogram representing the pdf
29  TH1* hh_pdf = pdf->createHistogram("hh_model", theta, Binning(50), YVar(phi,Binning(50)));
30  TCanvas* c = new TCanvas("c","c",800,800);
31  hh_pdf->Draw("surf1");
32  c->Draw();
33
34  // we now generate the sample of 50000 MC events according to the previously defined pdf
35  RooDataSet* MC_ev = pdf->generate(RooArgSet(theta,phi),50000);
36
37  // we fit the MC events through the ML method with the pdf we have defined
38  // and we get the resulting parameters
39  RooFitResult *r_ML = pdf->fitTo(*MC_ev, Save());
40  r_ML->Print();
```

We then repeat the same procedure for the LS method by using:

```

41 // we first bin the data
42 TH1* hh_data = MC_ev->createHistogram("hh_data", theta, Binning(100),YVar(phi,Binning(100)));
43 RooDataHist binData ("binData", "binData", RooArgList(theta,phi),hh_data);
44 // and then we fit the data through the LS method
45 RooFitResult *r_chi2 = pdf-> chi2FitTo(binData, Save());
46 r_chi2->Print();

```

Moreover, we plot the likelihood functions of the three parameters as follows:

```

47 // we create the likelihood function starting from the MC events
48 RooAbsReal* nll = pdf->createNLL(*MC_ev, NumCPU(8));
49 // we then plot the likelihood function of the parameter analysed
50 RooPlot* frame = alpha.frame(Title("Likelihood #alpha"));
51 nll->plotOn(frame,ShiftToZero());
52 TCanvas* c2 = new TCanvas("c2","c2",800,800) ;
53 frame->Draw();
54 c2->Draw();

```

Then we wrote a brief code to plot the three parameters computed with ML and LS methods with their errors together with the true values. Below, we show the code used to plot  $\alpha$ , knowing that the codes used for the other two parameters are similar.

```

1 import numpy as np
2 from matplotlib import pyplot as plt
3
4 # we define the name of the ticks on the x axis
5 x = list(['ML', 'LS'])
6 # we fill two lists with the computed values of alpha and their errors
7 alpha = list([0.651,0.623])
8 alpha_err = list([0.003, 0.003])
9 # we plot these values
10 plt.errorbar(x, alpha, yerr=alpha_err, fmt='s', color = 'red')
11 # we plot the black dashed line which represent the true value of alpha
12 plt.plot((0,1),(0.65, 0.65), color='black', linestyle='--')
13 # we define the label for the y axis
14 plt.ylabel('$\alpha$', fontsize=16)
15 plt.show()

```

## The Likelihood-Ratio Test

After having defined all the variables and having generated the data `MC_ev`, we can perform the likelihood-ratio test as follows.

```
1  // we first define some variables needed for the code
2  RooArgSet* pdfObs = pdf->getObservables(*MC_ev);
3  double log_lambda = 0;
4  double t, p;
5
6  // the value of the scalar-decay pdf is uniform, and so h0 is the same for all the data
7  double h0 = log(1/(4*M_PI));
8
9  // we loop over all the data
10 for (int i; i < 50000; i++) {
11     // we first get theta and phi from the dataset
12     auto ev = MC_ev->get(i);
13     t = ev -> getRealValue("theta");
14     p = ev -> getRealValue("phi");
15     // we then obtain the value of the vector-pdf corresponding to the theta and phi just found
16     *pdfObs = *MC_ev->get(i);
17     double h1 = log(pdf->getVal());
18     // we then sum the difference between the two log-likelihoods to the statistic
19     log_lambda += h0 - h1;
20 }
```

# Exercise 4

## Generation of the MC Data-Sample

Below we show the code written to generate a sample of 60000 particles with uniform distribution in space and with uniform momenta  $p \in [0, 10]$ .

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import math as mt
4
5  # we read and store the numbers generated with the xoroshiro128+ algorithm into a list
6  with open('xoroshiro_results', 'r') as f:
7      list_num = f.read().splitlines()
8
9  # we convert the list of chars into an array of floats
10 list_arr = np.asarray(list_num)
11 list_arr = list_arr.astype(np.float)
12
13 # we then generate 4 lists, each containing 60000 random numbers
14 X = list_arr[:60000]          # x space coordinate
15 Y = list_arr[60000:120000]    # y space coordinate
16 Z = list_arr[120000:180000]   # z space coordinate
17 P = list_arr[180000:240000]    # momentum
18
19 # we redefine the range of the momenta in [0,10]
20 div = 2**64-1
21 for j in range(60000):
22     P[j] = (P[j] / div)*10
23
24 # we now count the particles with high and low momenta
25 lowP = 0
26 highP = 0
27 for j in range(60000):
28     if P[j] < 2 : lowP = lowP+1
29     if P[j] > 8 : highP = highP+1
30
31 # we plot the uniform distribution of the momenta
32 plt.hist(P, bins = 240, color='c', density=True)
33 plt.xlabel("p (GeV)")
34 plt.ylabel("N")
35 plt.show()
```

```

36 # we plot in a 3D space the space coordinates
37 from mpl_toolkits.mplot3d import Axes3D
38
39 fig = plt.figure(figsize=(15, 15))
40 ax = fig.add_subplot(111, projection='3d')
41 ax.scatter(X, Y, Z, marker='o', color='red')
42 ax.set_xlabel('x')
43 ax.set_ylabel('y')
44 ax.set_zlabel('z')
45 plt.show()

```

After having generated the 60000 particles, each with its own momentum  $p \in [0, 10]$  GeV, we project the momentum in two components, longitudinal and transverse, by generating  $\theta \in [0, 2\pi]$  from a uniform distribution.

```

46 # we generate another list of 60000 numbers
47 T = list_arr[240000:]
48
49 # we redefine the range of theta in [0, 2pi]
50 for j in range(60000):
51     T[j] = (T[j] / div)*2*mt.pi
52
53 # we generate two arrays containing PL and PT
54 PL = np.array([])
55 PT = np.array([])
56 for j in range(60000):
57     pl = abs(P[j]*mt.cos(T[j]))
58     pt = P[j]*mt.sqrt(1-mt.cos(T[j])**2)
59     PL = np.append(PL, pl)
60     PT = np.append(PT, pt)
61
62 # we plot the histograms containing PL and PT
63 plt.hist(PL, bins = 240, color='c', density=True)
64 plt.xlabel("$p_L$ (GeV)")
65 plt.ylabel("N")
66 plt.show()
67
68 plt.hist(PT, bins = 240, color='c', density=True)
69 plt.xlabel("$p_T$ (GeV)")
70 plt.ylabel("N")
71 plt.show()
72
73 # we then count the particles with high and low PL and PT
74 lowPL = 0
75 highPL = 0
76 for j in range (60000):
77     if PL[j] < 2 : lowPL = lowPL+1
78     if PL[j] > 8 : highPL = highPL+1
79
80 lowPT = 0
81 highPT = 0
82 for j in range (60000):
83     if PT[j] < 2 : lowPT = lowPT+1
84     if PT[j] > 8 : highPT = highPT+1

```

```

85  # we show the first ten particles with their properties in a pandas DataFrame
86  import pandas as pd
87
88  data = pd.DataFrame(
89      {
90          'X': X,
91          'Y': Y,
92          'Z': Z,
93          'P': P,
94          '$\theta$': T,
95          'PL': PL,
96          'PT': PT
97      }
98  )
99  data[:10]

```

After having generated all the properties we are interested in, we can make some plots.

```

100 import ROOT as r
101
102 # first of all, we plot the 2D histo of PT and PL
103 h = r.TH2F("h", "h", 100, 0, 10, 100, 0, 10)
104 for i in range(60000):
105     h.Fill(PL[i], PT[i])
106 c = r.TCanvas()
107 h.Draw("lego2")
108 h.GetXaxis().SetTitle("p_{L} (GeV)")
109 h.GetXaxis().SetLabelOffset(2)
110 h.GetYaxis().SetTitle("p_{T} (GeV)")
111 h.GetYaxis().SetLabelOffset(2)
112 h.SetTitle("")
113 h.SetStats(0)
114 c.Draw()
115
116 # subsequently, we plot <PT> in function of PL through the Profile method
117 PTPL = h.ProfileX("PTPL", 0, 100);
118 PTPL.GetXaxis().SetTitle("p_{L} (GeV)");
119 PTPL.GetXaxis().SetTitleOffset(1.2);
120 PTPL.GetYaxis().SetTitle("< p_{T}> (GeV)");
121 PTPL.SetStats(0)
122 c = r.TCanvas();
123 PTPL.Draw();
124 c.Draw();

```

## Computation of $f_{p_T}$ and $f_{p_L}$

We now show how to obtain the pdf of  $p_T$  and  $p_L$ . First of all, we recall the definition of these two variables in function of  $\theta$  and  $p$ , uniformly distributed.

$$\begin{cases} p_T = |p \sin \theta| \\ p_L = |p \cos \theta| \end{cases} \quad (1)$$

To compute the pdfs, we use a change-of-variable algorithm. Starting from the variables  $\theta$  and  $p$ , we need invertible transformations, so we reduce the domain of  $\theta$  to  $[0, \frac{\pi}{2}]$  so that the equations (1) become:

$$\begin{cases} p_T = p \cos \theta \\ p_L = p \sin \theta \end{cases} \quad (2)$$

Then we compute their inverses:

$$\begin{cases} \theta = \arctan \frac{p_T}{p_L} \\ p = \sqrt{p_T^2 + p_L^2} \end{cases} \quad (3)$$

and we define the Jacobian matrix:

$$\begin{aligned} J &= \begin{pmatrix} \frac{\partial p}{\partial p_T} & \frac{\partial p}{\partial p_L} \\ \frac{\partial \theta}{\partial p_T} & \frac{\partial \theta}{\partial p_L} \end{pmatrix} \\ &= \begin{pmatrix} \frac{p_T}{\sqrt{p_T^2 + p_L^2}} & \frac{p_L}{\sqrt{p_T^2 + p_L^2}} \\ \frac{-p_L}{p_T^2 + p_L^2} & \frac{p_T}{p_T^2 + p_L^2} \end{pmatrix} \end{aligned} \quad (4)$$

So, we find the joint distribution  $g(p_T; p_L)$  as:

$$g(p_T; p_L) = |J| \cdot f(\theta(p_T; p_L); p(p_T; p_L)) \quad (5)$$

where  $|J|$  is the determinant of the Jacobian matrix and  $f(\theta(p_T; p_L); p(p_T; p_L))$  is the joint distribution of  $\theta$  and  $p$  evaluated in  $p = p(p_T; p_L)$  and  $\theta = \theta(p_T; p_L)$  according to (3). Since  $\theta$  and  $p$  are independent variables, the joint function is just the product of the two pdfs.

The pdf of the momentum  $p$  reads as:

$$f_p(p) = \begin{cases} \frac{1}{10} & 0 \leq p \leq 10, \\ 0 & \text{otherwise} \end{cases} \quad (6)$$

As regards  $\theta$ , its pdf is written as follows.

$$f_\theta(\theta) = \begin{cases} \frac{1}{\pi/2} & 0 \leq \theta \leq \frac{\pi}{2}, \\ 0 & \text{otherwise} \end{cases} \quad (7)$$

So the joint function is:

$$f(\theta; p) = f_p(p) \cdot f_\theta(\theta) = \frac{1}{\pi/2} \cdot \frac{1}{10} = \frac{1}{5\pi} \quad (8)$$

and hence:

$$g(p_T; p_L) = \frac{1}{5\pi} \frac{1}{\sqrt{p_T^2 + p_L^2}} \quad (9)$$

In order to compute the marginal distributions for one of the variables, namely  $p_T$  and  $p_L$ , we integrate the joint distribution over the domain of the other one.

$$\begin{aligned} g(p_T) &= \int_0^{10} g(p_T; p_L) dp_L \\ &= \frac{1}{5\pi} \left[ \ln \left( 10 + \sqrt{p_T^2 + 100} \right) + \ln 10 \right] \end{aligned} \quad (10)$$

Finally we have to divide by 4 to obtain the right results in the initial domain, and so:

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

We then fit the histogram of  $p_T$  with its pdf  $f_{p_T}$ .

```
1  // first of all we define the pdf of the transverse momentum to fit the data
2  TF1 *pdf_PT = new TF1("pdf_PT",
3  "[1]/(20*pi)*([2]*log(10+sqrt(pow([0]*x,2)+100))-log(10))", 0, 5);
4  pdf_PT->SetParameter(0,0.1);
5  pdf_PT->SetParameter(1,3000);
6  pdf_PT->SetParameter(2,0.5);
7
8  // then we draw the histogram of the data and we fit it with the function described above
9  c = new TCanvas();
10 h->SetTitle("");
11 h->SetLineColor(kBlue);
12 h->SetLineWidth(1);
13 h->SetStats(0);
14 h->Fit("pdf_PT");
15 h->Draw();
16 c->Draw();
17
18 // we acquire the chi2 and the NDF to calculate the reduced chi2
19 Double_t chi2 = pdf_PT->GetChisquare();
20 Double_t NDF = pdf_PT->GetNDF();
21 std::cout << chi2/NDF;
```



# Exercise 5

## Crude MC

Below the code to perform the Crude MC algorithm.

```
1  import numpy as np
2  import math as mt
3  import random
4  from matplotlib import pyplot as plt
5  from scipy import optimize
6  from IPython.display import clear_output
7  import matplotlib.mlab as mlab
8
9  # we define the integrand function
10 def f(x):
11     return mt.e**x
12
13 # we then define the two arrays that will contain the values of Q_N and the relative variances
14 Integral = np.array([])
15 Vars = np.array([])
16
17 # we now implement the loop with which the values of Q_N and variances are computed
18 # we want to do 1000 iterations
19 for j in range(1000):
20     # we define an array which will contain the values of f(x)
21     # obtained with 128 random-extracted numbers in the interval (0,1)
22     num = np.array([])
23     for i in range(128):
24         x = random.uniform(0,1)
25         num = np.append(num, f(x))
26
27     # we define the average of the values of f(x), namely Q_N
28     m = num.mean()
29     Integral = np.append(Integral, m)
30
31     # we then calculate the variance of Q_N
32     v = 0
33     for i in range(128):
34         v = v + (num[i]-m)**2
35     v = v/(128-1)
36     v = v/128
37     Vars = np.append(Vars, v)
```

At the end of this loop we have two arrays `Integral` and `Vars` with 1000 values of  $Q_N$  and their relative variances  $\hat{\sigma}^2$ . We then plot their histograms with the following code.

```

38 # we plot the values of Q_N
39 plt.hist(Integral, 25, color='blue')
40 plt.xlabel('$Q_N$', fontsize=16)
41 plt.ylabel('counts', fontsize=16)
42 plt.show()
43
44 # we plot the values of their variances
45 plt.hist(Vars, 25, color='blue')
46 plt.xlabel('$\hat{\sigma}^2$', fontsize=16)
47 plt.ylabel('Counts', fontsize=16)
48 plt.show()

```

Next we want to obtain the values of  $I$  and its standard deviation  $\sigma_I$ . To do so we use the following code.

```

49 m2 = np.mean(Integral)
50
51 S = 0
52 for i in range (1000):
53     S += (Integral[i]-m2)**2
54 s_tilde = mt.sqrt(S/999)
55 s2 = s_tilde/mt.sqrt(1000)
56 print(f'For crude MC with 128 samples we have:\n'
57       f'\tmean\t= {round(m2,4)}\n\tterr\t= {round(s2,4)}\n')

```

For crude MC with 128 samples we have:

```

mean = 1.7190
err   = 0.0014

```

Next we want to compute  $I$  and  $\sigma_I$  with different numbers of extractions. To compute  $\sigma_I$  we use the `np.var()` method directly on the array `Integral`.

```

58 mean_s = np.array([])
59 var_s = np.array([])
60
61 Ns = np.arange(100, 2500, 200)
62 for Ni in Ns:
63     Integral = np.array([])
64     for j in range (1000):
65         num = np.array([])
66         for i in range (Ni):
67             x = random.uniform(0,1)
68             num = np.append(num, f(x))
69         Integral = np.append(Integral, num.mean())
70     mean_s = np.append(mean_s, Integral.mean())
71     var_s = np.append(var_s, Integral.var())

```

Finally we plot the values of  $\sigma_I$  in function of  $N$  and next we perform a fit of this data to obtain  $\kappa$ .

```
72 # here we plot the standard deviation of I in function of N
73 plt.plot(Ns, np.sqrt(var_s), 'go', color='green')
74 plt.xlabel('N', fontsize=16)
75 plt.ylabel('$\sigma_I$', fontsize=16)
76 plt.show()
77
78 # below we perform the fit
79 def test_func(x,a):
80     return a / np.sqrt(x)
81
82 params, params_covariance = optimize.curve_fit(test_func, Ns, np.sqrt(var_s), p0=[2])
83
84 kappa = params[0]
85 print('kappa = ', round(kappa,4))
86
87 err_kappa = np.sqrt(params_covariance[0][0])
88 print('err_kappa = ', round(err_kappa,4))
89
90 kappa = 0.485
91 err_kappa = 0.003
92
93 # finally we plot the data along with the fitted function
94 ran = np.arange(100, 2500, 1)
95
96 plt.plot(Ns, np.sqrt(var_s), 'go', color='green')
97 plt.plot(ran, test_func(ran, params), color='red')
98 plt.xlabel('N', fontsize=16)
99 plt.ylabel('$\sigma_I$', fontsize=16)
100 plt.show()
```

## Stratified Sampling

To perform the stratified sampling, the algorithm is different from the one used in the Crude MC, while the other estetic things are equal. We then report just the first step, namely the algorithm itself.

```
1  Integral = np.array([])
2  Vars = np.array([])
3
4  for j in range (1000):
5      num1 = np.array([])
6      num2 = np.array([])
7      va = 0
8      vb = 0
9      for i in range (64):
10         x1 = random.uniform(0,1/2.)
11         x2 = random.uniform(1/2.,1)
12         num1 = np.append(num1, f(x1))
13         num2 = np.append(num2, f(x2))
14     ma = num1.mean()
15     mb = num2.mean()
16     Integral = np.append(Integral, 0.5*(ma+mb))
17     for i in range (64):
18         va = va + (num1[i] - ma)**2
19         vb = vb + (num2[i] - mb)**2
20     va = va / ((64-1)*64)
21     vb = vb / ((64-1)*64)
22
23     Vars = np.append(Vars, 0.25*(va + vb))
```

## Importance Sampling

Here we have two important codes, namely those with which we extract random numbers from  $w_1$  and  $w_2$ . The algorithm is then the same used in the Crude MC. First of all, we have the code to plot the shape of  $f(x) = e^x$  and of:

$$w_1(x) = \frac{1}{e} [1 + 2(e - 1)x] \quad (11)$$

$$w_2(x) = \frac{1}{e + 1.5} [2.5 + 2.5(e - 1)x^{1.5}] \quad (12)$$

```
1  def w1 (x):
2      return (1+2*(np.e-1)*x)/np.e
3
4  def w2 (x):
5      return (2.5+2.5*(np.e-1)*x**1.5)/(np.e+1.5)
6
7  xx = np.arange(0, 1, 0.01)
8
9  plt.plot(xx, f(xx), label='$y=e^x$')
10 plt.plot(xx, w1(xx), label='$y=w_1(x)$')
11 plt.plot(xx, w2(xx), label='$y=w_2(x)$')
12 plt.legend()
13 plt.xlabel('$x$')
14 plt.ylabel('$y$')
15 plt.show()
```

To extract random numbers from  $w_1$ , we first have to compute its CDF and its inverse:

$$W(x) = \int_0^x w_1(t)dt = \frac{x}{e} + \left(1 - \frac{1}{e}\right) \cdot x^2 \quad W^{-1}(x) = \frac{1 - \sqrt{4e^2x - 4ex + 1}}{2 - 2e} \quad (13)$$

The numbers will then be extracted starting from this last function, as can be seen in the code below.

```

16 def s1 ():
17     x = random.uniform(0, 1)
18     return (1 - np.sqrt(4*x*np.e**2 - 4*np.e*x + 1)/(2 - 2*np.e))

```

We then want to compute the value of  $Q_N$  as follows:

$$Q_N^{imp} = \frac{1}{N} \sum_{i=1}^N \frac{f(x_i)}{w_1(x_i)} \quad (14)$$

Where  $x_i$  are the numbers extracted with the function  $W^{-1}(x)$ . The code implemented to do so is the same as the one used in the Crude MC, with the only difference in the following for-cycle.

```

19 for i in range (128):
20     x = s1()
21     num = np.append(num, f(x)/w1(x))

```

The rejection sampling method used to extract numbers distributed as  $w_2$  is based on the function written in the following code.

```

22 def s2 ():
23     while(1):
24         x=random.uniform(0, 1)
25         y=random.uniform(0, np.e)
26         if (y<w2(x)): return x

```

The rest of the code is then the same as for the  $w_1$  function.

## Comparison

Finally, we report the codes used to show the final comparison.

```

1 # first of all, we plot the standard deviation of I in function of N for all the method used
2 plt.plot(Ns, np.sqrt(var_s), 'bo', label='Crude MC')
3 plt.plot(Ns, np.sqrt(varns3), 'go', label='Stratified Sampling')
4 plt.plot(Ns, np.sqrt(varns4), 'ro', label='Importance Sampling $w_1$')
5 plt.plot(Ns, np.sqrt(varns5), 'yo', label='Importance Sampling $w_2$')
6 plt.xlabel('$N$', fontsize=16)
7 plt.ylabel('$\sigma_{\{I\}}$', fontsize=16)
8 plt.legend()
9 plt.show()

```

```

10 # then we show the values of I with their standard deviations along with the expected value
11 x = list(['Crude MC', 'Stratified\nsampling', 'Importance\nsampling\n$w_1$',
12 'Importance\nsampling\n$w_2$'])
13 y = list([1.7190, 1.7179, 1.7181, 1.7182])
14 err = list ( [0.0014, np.sqrt(v3)/np.sqrt(1000), np.sqrt(v4)/np.sqrt(1000),
15 np.sqrt(v5)/np.sqrt(1000)])
16 plt.errorbar(x, y, yerr=err, fmt='s', color = 'blue')
17 plt.plot((0,1,2,3),(1.7183, 1.7183, 1.7183, 1.7183), color='red', linestyle='--')
18 plt.ylabel('$I$', fontsize=16)
19 plt.show()
20
21 # finally, we show the values of kappa with their standard deviations
22 y = list([0.485, 0.254, 0.198, 0.037])
23 err = list ([0.3, 0.12, 0.02, 0.02])
24 plt.errorbar(x, y, yerr=err, fmt='s', color = 'blue')
25 plt.ylabel('$\kappa$', fontsize=16)
26 plt.show()

```

# Exercise 6

## Generation of the MC Data-Sample

Below we show the code implemented to generate a sample of 50000 particles with a distribution given by the Fraunhofer diffraction for circular apertures.

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import math as mt
4  import random
5  import scipy.special
6
7  # we define the parameters we need
8  l = 500 * 10(-9)
9  a = 2 * 10(-9)
10 k = 2*mt.pi/l
11
12 # we then generate 50000 numbers between 0 and 2*pi, namely the theta's
13 thetas = np.array([])
14 for i in range(50000):
15     theta = random.uniform(0, mt.pi)
16     thetas = np.append(thetas, theta)
17
18 # subsequently we define the argument of the Bessel function
19 z = np.array([])
20 for theta in thetas:
21     z = np.append(z, k * a * mt.sin(theta))
22
23 # then we define the Bessel function itself
24 J1 = scipy.special.jv(1, z)
25
26 # finally we calculate the intensity spectrum we are looking for
27 I = np.array([])
28 for i in range(50000):
29     I = np.append(I, 5*(2 * J1[i] / z[i])**2)
30
31 # we then plot the spectrum
32 fig = plt.figure(figsize=(10,7))
33 plt.hist(I, bins=70, density = True, fill = False, ec = 'green', range=(-1,1), histtype='step')
34 plt.show()
```

## Smearing

Below we implement the smearing algorithm with  $c = 0.4$ ,  $c = 0.9$  and  $c = 2$  and then we plot the three distributions along with the original data.

```
1  # we implement the smearing for c = 0.4
2  c = 0.4
3  sigma = c * 0.03
4  smear04 = np.array([])
5  for j in range(50000):
6      mean = list_arr[j]
7      new = random.gauss(mean, sigma)
8      smear04 = np.append(smear04, new)
9
10 # we implement the smearing for c = 0.9
11 c = 0.9
12 sigma = c * 0.03
13 smear09 = np.array([])
14 for j in range(50000):
15     mean = list_arr[j]
16     new = random.gauss(mean, sigma)
17     smear09 = np.append(smear09, new)
18
19 # we draw the three distributions: original, c = 0.4 and c = 0.9
20 fig = plt.figure(figsize=(10, 7))
21 plt.hist(list_arr, bins=70, density = True, fill = False, ec = 'green', range=(-1,1), label = 'no s
22 plt.hist(smear04, bins=70, density = True, fill = False, ec = 'red', range=(-1,1), label = 'c = 0.4
23 plt.hist(smear09, bins=70, density = True, fill = False, ec = 'blue', range=(-1,1), label = 'c = 0.
24 plt.legend(loc='upper right')
25 plt.show()
26
27 # we implement the smearing for c = 2
28 c = 2
29 sigma = c * 0.03
30 smear2 = np.array([])
31 for j in range(50000):
32     mean = list_arr[j]
33     new = random.gauss(mean, sigma)
34     smear2 = np.append(smearTANT0, new)
35
36 # we draw the two distributions: original and c = 2
37 fig = plt.figure(figsize=(10, 7))
38 plt.hist(list_arr, bins=70, density = True, fill = False, ec = 'green', range=(-1,1), label = 'no s
39 plt.hist(smearTANT0, bins=70, density = True, fill = False, ec = 'brown', range=(-1,1), label = 'c :
40 plt.legend(loc='upper right')
41 plt.show()
```



## Unfolding

To unfold the data we use the PyUnfold library implemented in Python. Below the code is shown step by step.

```
1  import pyunfold
2
3  # we define the true and observed samples, we bin them
4  # and we compute the poissonian error on observed data
5  true_samples = list_arr
6  data_true, _ = np.histogram(list_arr, bins = 70)
7  observed_samples = smear09
8  data_observed, _ = np.histogram(smear09, bins = 70)
9  data_observed_err = np.sqrt(data_observed)
10
11 # we define as efficiencies 1 and as their errors 0.01
12 efficiencies = np.ones_like(data_observed, dtype=float)
13 efficiencies_err = np.full_like(efficiencies, 0.01, dtype=float)
14
15 # we define the response histogram and we plot it
16 response_hist, _, _ = np.histogram2d(observed_samples, true_samples, bins=70)
17 response_hist_err = np.sqrt(response_hist)
18
19 fig, ax = plt.subplots(figsize=(15,15))
20 im = ax.imshow(response_hist, origin='lower')
21 cbar = plt.colorbar(im, label='Counts')
22 ax.set(xlabel='Cause bins', ylabel='Effect bins')
23 plt.show()
24
25 # we then normalise the histogram to obtain the response matrix and we plot it
26 column_sums = response_hist.sum(axis=0)
27 normalization_factor = efficiencies / column_sums
28 response = response_hist * normalization_factor
29 response_err = response_hist_err * normalization_factor
30
31 fig, ax = plt.subplots(figsize=(15,15))
32 im = ax.imshow(response, origin='lower')
33 cbar = plt.colorbar(im, label='$P(E_i|C_{\mu})$')
34 ax.set(xlabel='Cause bins', ylabel='Effect bins', title='Normalizes response matrix')
35 plt.show()
36
37 # we define now the two callbacks, namely the logger and the regularizer
38 from pyunfold import callbacks
39
40 # the logger writes test statistic information for each unfolding iteration
41 logger = callbacks.Logger()
42
43 # the regularizer smooths the unfolded distribution at each iteration
44 regularizer = callbacks.SplineRegularizer(smooth=0.95)
```

Regulariser is used as a means to ensure that unfolded distributions do not suffer from growing fluctuations potentially arising from the finite binning of the response matrix.

We call now the `iterative_unfold` method to actually unfold the observed distribution. The test statistics we use is the Ratio Mean Deviations (`rmc`), based on the ratio of absolute deviations of the observations from their class medians.

```

45  unfolded_results = iterative_unfold(data=data_observed,
46                                     data_err=data_observed_err,
47                                     response=response,
48                                     response_err=response_err,
49                                     efficiencies=efficiencies,
50                                     efficiencies_err=efficiencies_err,
51                                     ts = 'rmc',
52                                     callbacks=[logger, regularizer]
53     )

```

After 63 iterations we obtain the unfolded distribution and we plot it along with the true distribution.

```

54  fig, ax = plt.subplots(figsize=(15, 15))
55  ax.step(np.arange(num_bins), data_true, where='mid', lw=3, alpha=0.7,
56  label='True distribution')
57  ax.errorbar(np.arange(num_bins), unfolded_results['unfolded'],
58             yerr=unfolded_results['sys_err'],
59             alpha=0.7,
60             elinewidth=3,
61             capsize=4,
62             ls='None', marker='.', ms=10,
63             label='Unfolded distribution')
64  ax.set(xlabel='X bins', ylabel='Counts')
65  plt.legend()
66  plt.show()

```

# Exercise 7

## Generation of the MC Data-Sample

Below we show the code implemented to generate a sample of 50000 signal particles and 50000 background particles distributed according to two 2D gaussians.

```
1  import matplotlib.pyplot as plt
2  import numpy as np
3  import pandas as pd
4
5  # we define the parameters of the signal and background gaussians
6  mu_s = [0,0]
7  cov_s = [[0.09, 0.045], [0.045, 0.09]]
8
9  mu_b = [4,4]
10 cov_b = [[1., 0.4], [0.4, 1.]]
11
12 # we then generate the two gaussians
13 x_s, y_s = np.random.multivariate_normal(mu_s, cov_s, 50000).T
14 x_b, y_b = np.random.multivariate_normal(mu_b, cov_b, 50000).T
15
16 # finally we draw the two 2D gaussians and the two projections
17 plt.figure(figsize=(10, 10))
18 plt.plot(x_s, y_s, '.', color = 'blue', label = 'signal')
19 plt.plot(x_b, y_b, '.', color = 'red', label = 'background')
20 plt.legend()
21 plt.show()
22
23 plt.hist(x_s, bins=200, color = 'blue', histtype = 'step', label = 'signal')
24 plt.hist(x_b, bins=200, color = 'red', histtype = 'step', label = 'background')
25 plt.xlabel('X')
26 plt.legend()
27 plt.show()
28
29 plt.hist(y_s, bins=200, color = 'blue', histtype = 'step', label = 'signal')
30 plt.hist(y_b, bins=200, color = 'red', histtype = 'step', label = 'background')
31 plt.xlabel('Y')
32 plt.legend()
33 plt.show()
34
35 # we then build a dataframe containing all the data
36 x = np.append(x_s,x_b)
37 y = np.append(y_s,y_b)
```

```

38 # we fill the array s_b with 0.0 and 1.0 depending on the belonging class
39 s_b = np.array([])
40
41 # signal = 1
42 for i in range (50000):
43     s_b = np.append(s_b,1)
44
45 # background = 0
46 for i in range (50000):
47     s_b = np.append(s_b,0)
48
49 data = pd.DataFrame(
50     {
51         'x': x,
52         'y': y,
53         's_b': s_b
54     }
55 )
56
57 data.head()

```

## Binary Linear Classification

Below the code to perform the binary linear classification.

```

1  from sklearn.model_selection import train_test_split
2  from sklearn.preprocessing import StandardScaler
3  from sklearn.preprocessing import LabelEncoder
4  from sklearn.linear_model import LogisticRegression
5  from sklearn.metrics import accuracy_score
6  from sklearn.metrics import log_loss
7
8  # we define the two target classes
9  classes = data["s_b"].unique()
10
11 # we then create two arrays containing features and target
12 X = data[['x', 'y']]
13 Y = data['s_b']
14
15 # we now create the train and test sets
16 X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3)
17
18 # the features are now standardised
19 ss = StandardScaler()
20 X_train = ss.fit_transform(X_train)
21 X_test = ss.transform(X_test)

```

Now that we have the features and the target, we can proceed to perform the Binary Linear Classification.

```

22 from sklearn.linear_model import LogisticRegression
23 from sklearn.metrics import accuracy_score
24
25 # we perform the logistic regression
26 lr = LogisticRegression(solver='lbfgs')
27 lr.fit(X_train, Y_train)

```

```

28 # prediction of the target Y over X_test
29 Y_pred = lr.predict(X_test)
30 # probability of correct prediction
31 Y_pred_proba = lr.predict_proba(X_test)
32 # accuracy, computed seeing the differences between Y_test and Y_pred
33 A = accuracy_score(Y_test, Y_pred)
34 # log_loss, calculated seeing the probability that Y_pred is correct looking at Y_test
35 LL = log_loss(Y_test, Y_pred_proba)
36 # we then print both the metrics
37 print('ACCURACY = ', round(A,2))
38 print('LOG LOSS = ', round(LL,2))
39
40 # we allocate Y_test and Y_pred in a new dataframe to see which ones are different
41 data_new = pd.DataFrame(
42     {
43         'Y_test': Y_test,
44         'Y_pred': Y_pred
45     }
46 )
47
48 # calculate the number of Type I errors (false positive)
49 data_new.loc[(data_new.Y_test != data_new.Y_pred) & (data_new.Y_test == 0)]
50
51 # calculate the number of Type II errors (false negative)
52 data_new.loc[(data_new.Y_test != data_new.Y_pred) & (data_new.Y_test == 1)]

```

After having checked that the model is well-performing we can print the line the model has learnt in a plot.

```

53 # we first define a function to show the boundary line
54 def showBounds(X, Y, model, title=None):
55     fig = plt.figure(figsize=(15,10))
56     h = .02
57     # we define the minimum and the maximum of each set
58     x_min, x_max = X[:, 0].min(), X[:, 0].max()
59     y_min, y_max = X[:, 1].min(), X[:, 1].max()
60
61     # we obtain the coordinate matrices
62     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
63
64     # we plot the predicted line
65     Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
66     Z = Z.reshape(xx.shape)
67     plt.contourf(xx, yy, Z, cmap=plt.cm.jet)
68
69     # we finally plot the two distributions
70     X_m = X[Y==1]
71     X_b = X[Y==0]
72     plt.title(title)
73     plt.scatter(X_b[:, 0], X_b[:, 1], marker='.', c='red')
74     plt.scatter(X_m[:, 0], X_m[:, 1], marker='.', c='blue')
75     plt.show()
76
77 # then we pass at the function the train and test sets
78 showBounds(X_train, Y_train, lr, title='Train set')
79 showBounds(X_test, Y_test, lr, title='Test set')

```