codes & further results

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.

```python
# first of all we insert the seed
seed = '0540'

# then we introduce some variables useful for the code
number = int(seed);
PR_list = []
counter = 0

# we implement the middle-square method loop
while number not in PR_list:
    # we count the number of steps the algorithm does before repeating itself
    counter += 1
    # we square the number
    squared = number * number
    # then we convert the squared number into a string
    squared = str(squared)
    # we insert enough zeros on the left to obtain a 8-digit number
    squared = squared.zfill(8)
    # we cut the string considering the central 4 digits
    squared = squared[2:6]
    # we append the number to the pseudo-random generated list
    PR_list.append(number)
    # finally we convert again the string into an integer
    number = int(squared)

print(f"We started with the seed {seed} and the algorithm generated"
      f"a list of {counter} pseudo-random numbers:\n"
      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.

```python
# we insert the seed
seed = '1'
# we then insert the three parameters
m = 2**32
a = 1103515245
c = 12345

# we introduce some variables useful for the code
number = int(seed);
PR_list = []
counter = 0

# we implement the LCG method loop
while (number not in PR_list) and (counter <= 300000):
    counter += 1
    # we append the number to the pseudo-random generated list
    PR_list.append(number)
    # we calculate the next number
    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.

```c
#include <stdint.h>
#include <stdlib.h>
#include <stdio.h>

//Parameters of the algorithm
#define a 24
#define b 16
#define c 37

//Global variable used to store temporary numbers
static unsigned long long int s[2];

//Function for the bitwise left rotation
static unsigned long long int rotl(const unsigned long long int x, int k){
    return (x<<k)|(x>>(64-k));
}

//Function that generates the next random number
unsigned long long int next(void){
    const unsigned long long int s0 = s[0];
    unsigned long long int s1 = s[1];

    //Next random number
    const unsigned long long int result = s0 + s1;

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

    return result;
}

int main(){
    int i, j, N;

    //Define the output of the code
    FILE *numbers;
    unsigned long long int *list;

    //Define the seed (first two numbers generated with LGC)
    s[0] = 1103527590;
    s[2] = 2524885223;

    //Define the list of the generated numbers (max N)
    N = 300000;
    list = malloc(N * sizeof(unsigned long long int));

    for(i=0; i<N; i++){
        // Generate through the 'next' function a new random number
        list[i] = next();
```

```c
            //Check if the number has already been extracted
            for(j=0; j<i; j++){
                if(list[i]==list[j]){
                    printf("The algorithm entered a loop after %d steps", i);
                    break;
                }
            }
        }
    return 0;
}
```

```
#=============================================================================#
#            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 test 207.  ntuple = 0
       dab_filltree|  32|  15000000|       1|0.67232847|  PASSED
       dab_filltree|  32|  15000000|       1|0.79648001|  PASSED
Preparing to run test 208.  ntuple = 0
      dab_filltree2|   0|   5000000|       1|0.08238921|  PASSED
      dab_filltree2|   1|   5000000|       1|0.67039797|  PASSED
Preparing to run test 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.

```python
import numpy as np
from scipy.stats import moyal
import matplotlib.pyplot as plt

# we open the file containing the number generated with the xoroshiro128+ algorithm
with open('xoroshiro_results', 'r') as f:
    list_num = f. read().splitlines()

# we split the list in half creating two lists
X = list_num[:150000]
Y = list_num[150000:]

# since we want to generate a Landau distribution with domain [-3,18] e codomain [0,0.25]
# we normalize the two lists to these intervals
div = 2**64-1   # this is the maximum number that can be produced with 64 bits
j = 0
for j in range(150000):
    X[j] = (X[j] / div)*(18+3)-3.
    Y[j] = (Y[j] / div)*0.25

# we can now implement the rejection-sampling algorithm
keep_list = np.array([])
for i in range(150000):
    xr = X[i]
    yr = Y[i]
    L = moyal.pdf(xr)
    if (yr > L): continue
    keep_list = np.append(keep_list, xr)
```

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

```python
import ROOT as r

# first of all we define the histogram of data and we fill it
h = r.TH1F("h",180,-3,18);

for i in range (150000):
    h.Fill(keep_list[i]);

# we then define the Landau function to fit the data
TF1 *landau_f = new TF1("landau_f", "[2]*TMath::Landau(x,[0],[1])", -3, 18);
landau_f->SetParameter(2,0.5);

# then we draw the histogram of the data and we fit it with the function described above
c = new TCanvas();
h->SetTitle("");
h->SetLineColor(kBlue);
h->SetLineWidth(1);
h->SetStats(0);
h->Fit("landau_f");
h->GetXaxis()->SetTitle("X");
h->GetYaxis()->SetTitle("Y");
```

```
h->Draw();
c->Draw();

# we acquire the chi2 and the NDF to calculate the reduced chi2
Double_t chi2 = landau_f->GetChisquare();
Double_t NDF = landau_f->GetNDF();
std::cout << "chi2/NDF = " << chi2/NDF << endl;

# we finally get the probability
Double_t p = landau_f->GetProb();
std::cout << "p = " << p << endl;
```

Exercise 2

Direct Summation

Below the code to implement the Direct Summation algorithm.

```c
/*******************************************************
 *
 *  Direct_Sum.c
 *
 *  Compute the Euler-Mascheroni constant with direct
 *  sum method.
 *
 *  gamma = (sum_i^N)(1/i) - ln(N)
 *
 *******************************************************/

#define MAIN_C
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

// FLOAT

float Direct_Sum_f(int n){
    // Compute the sum of 1/i with i in [1, n]

    float s = 0.;
    for (int i = 1; i<=n; i++){
        s += 1/(float)i;
    }
    return s;
}

float gamma_D_f(int n){
    // Compute an approximation of the gamma function up to n iterations

    float gamma_t = 0.;
    gamma_t = Direct_Sum_f(n) - logf((float)n);
    return gamma_t;
}
```

```c
// LONG DOUBLE

long double Direct_Sum_ld(int n){
    // Compute the sum of 1/i with i in [1, n]

    long double s = 0.;
    for (int i = 1; i<=n; i++){
        s += 1/(long double)i;
    }
    return s;
}

long double gamma_D_ld(int n){
    // Compute an approximation of the gamma function up to n iterations

    long double gamma_t = 0.;
    gamma_t = Direct_Sum_ld(n) - logl((long double)n);
    return gamma_t;
}

///////////////////////////////////////////////////////////////

int main (){

    // True values of gamma in the desired sizes
    long double gamma_true_ld = 0.57721566490153286;
    float       gamma_true_f  = 0.5772156;

    // Definitions of the output files
    FILE* errors;
    FILE* gammas;

    errors = fopen("Direct_Sum.txt", "w+");
    gammas = fopen("D_S_gammas.txt", "w+");

    // Definitions of some useful variables
    float       f, err_f;
    long double ld, err_ld;
    int         iter = 0;

    // Loop to implement the algorithm with 10^1, 10^2, ..., 10^9 iterations
    for (int i=1; i<=9; i++){

        iter = (int)pow(10, i);

        // FLOAT

        // Computing gamma and the relative error and write them in two txt files
        f     = gamma_D_f(iter);
        err_f = (f - gamma_true_f) / gamma_true_f;
        fprintf(errors, "%.70f\n", fabsf(err_f));
        fprintf(gammas, "%.70f\n", f);
```

```c
        // LONG DOUBLE

        // Computing gamma and the relative error and write them in two txt files
        ld     = gamma_D_ld(iter);
        err_ld = (ld - gamma_true_ld) / gamma_true_ld;
        fprintf(errors, "%.70Lf\n", fabsl(err_ld));
        fprintf(gammas, "%.70Lf\n", ld);
    }

}
```

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`, that are replaced by:

```c
float Sorted_Sum_f(int n){
    // Compute the sum of 1/i with i in [1, n]

    float s = 0.;
    for (int i = n; i>=1; i--){
        s += 1/(float)i;
    }
    return s;
}
```

```c
long double Sorted_Sum_ld(int n){
    // Compute the sum of 1/i with i in [1, n]

    long double s = 0.;
    for (int i = n; i>=1; i--){
        s += 1/(long double)i;
    }
    return s;
}
```

Pairwise Summation

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

```c
/*******************************************************
 *
 *  Pairwise_Sum.c
 *
 *  Compute the Euler-Mascheroni constant with pairwise
 *  sum method.
 *
 *  gamma = (sum_i^N)(1/i) - ln(N)
 *
 *******************************************************/

#define MAIN_C
#include <stdio.h>
#include <stdlib.h>
#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.

```c
float *nums_f;
long double *nums_ld;
float sf;
long double sld;
```

Let's analyse the Float functions of the algorithm.
First of all, we define a first function to compute the Euler-Mascheroni constant.

```c
// FLOAT

float gamma_P_f(int n){
    // Compute an approximation of the gamma function up to n iterations

    float gamma_t;
    Pairwise_Sum_f(n);
    gamma_t = sf - logf((float)n);
    return gamma_t;
}
```

As can be seen, this Function calls another function, namely `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 it with 10^8 or 10^9 numbers, since the occupied memory could become a problem. We then decide to sum the numbers in pairs and to fill `nums_f` with $\frac{n}{2}$ float numbers.

```c
void Pairwise_Sum_f(int n){
    int j = 0;

    // we dynamically allocate the memory for creating the array
    nums_f = (float*)calloc((int)(n/2), sizeof(float));

    if (nums_f == NULL) {
      printf("Error in creating the array nums_f!\n");
    }

    // we fill the array
    for(int i=1; i<=n; i=i+2){
      k = (float) i;
      nums_f[j] = 1/k + 1/(k+1.)
      j ++;
    }

    // we call the function PWf to perform the summation
    PWf(0, (int)(n/2)-1);
}
```

After having filled the array `nums_f`, we call the function `PWf`, defined and written below, to perform the Pairwise Summation algorithm. 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.

```c
void PWf(int start, int end){

    int len = end - start + 1;

    if (len<=2){
      for (int i = 0; i<len; i++) {
        sf += nums_f[start+i];
      }
    }
    else {
        int m = (int)(len/2);

        PWf(start,start+m-1);
        PWf(start+m,end);
    }
}
```

The same operations are performed 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 decide to divide it in four. In such way, however, for 10 iterations the algorithm breaks because it is impossible to divide 10 by 4. We decide then to consider $n = 10$ as a separate case, and so to fill the vector `nums_ld` with all 10 numbers of the series, since it was not a memory problem.

```c
void Pairwise_Sum_ld(int n){
    int j = 0;

    if (n == 10) {

      // allocating the memory for creating the array
      nums_ld = (long double*)calloc(n, sizeof(long double));

      if (nums_ld == NULL) {
          printf("Error in creating the array nums_ld!\n");
      }

      // filling the array
      for(int i=1; i<=n; i++){
          nums_ld[j] = 1/(long double)i;
          j ++;
      }

      // calling the PWld function to perform the summation
      PWld(0, n-1);

    } else {

      // allocating the memory for creating the array
      nums_ld = (long double*)calloc((int)(n/4), sizeof(long double));

      if (nums_ld == NULL) {
          printf("Error in creating the array nums_ld!\n");
      }

      // filling the array
      long double a, b, k;

      for(int i=1; i<=n; i=i+4){
          k = (long double) i;
          a = 1/k + 1/(k+1.);
          b = 1/(k+2.) + 1/(k+3.);

          nums_ld[j] = a+b;
          j ++;
      }

      // calling the PWld function to perform the summation
      PWld(0, (int)(n/4)-1);
    }
}
```

Concerning the `main`, it is actually equal to 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 lines below:

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

Kahan Summation

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

```c
/**********************************************************
 *
 *  Kahan_Sum.c
 *
 *  Compute the Euler-Mascheroni constant with Kahan
 *  sum method.
 *
 *  gamma = (sum_i^N)(1/i) - ln(N)
 *
 *********************************************************/

#define MAIN_C
#include <stdio.h>
#include <stdlib.h>
#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.

```c
// FLOAT

float gamma_K_f(int n){
    // Compute an approximation of the gamma function up to n iterations

    float gamma_t = 0.;
    gamma_t = Kahan_Sum_f(n) - logf((float)n);
    return gamma_t;
}
```

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

```c
float Kahan_Sum_f(int n){
    // Compute the sum of 1/i with i in [1, n]

    float s = 0., c = 0., t = 0., y = 0.;

    for (int i = 1; i<=n; i++){
        y = 1./(float)i - c;
        t = s + y;
        c = (t - s) - y;
        s = t;
    }
    return s;
}
```

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

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.

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

# number of iterations
N = 9

# read the files with the errors
with open('Direct_Sum.txt', 'r') as f:
    list_num_DS = f. read().splitlines()
with open('Sorted_Sum.txt', 'r') as f:
    list_num_SS = f. read().splitlines()
with open('Pairwise_Sum.txt', 'r') as f:
    list_num_PS = f. read().splitlines()
with open('Kahan_Sum.txt', 'r') as f:
    list_num_KS = f. read().splitlines()

# write the numbers in the files in two lists for each algorithm
F = np.arange(0,2*N-1,2)
D = np.arange(1,2*N,2)

    # direct summation
gammas_F_DS = []
gammas_LD_DS = []

for i in F:
    gammas_F_DS.append(abs(float(list_num_DS[i])))
for j in D:
    gammas_LD_DS.append(abs(float(list_num_DS[j])))

    # sorted summation
gammas_F_SS = []
gammas_LD_SS = []

for i in F:
    gammas_F_SS.append(abs(float(list_num_SS[i])))
for j in D:
    gammas_LD_SS.append(abs(float(list_num_SS[j])))

    # pairwise summation
gammas_F_PS = []
gammas_LD_PS = []

for i in F:
    gammas_F_PS.append(abs(float(list_num_PS[i])))
for j in D:
    gammas_LD_PS.append(abs(float(list_num_PS[j])))
```

```python
    # Kahan summation
gammas_F_KS = []
gammas_LD_KS = []

for i in F:
    gammas_F_KS.append(abs(float(list_num_KS[i])))
for j in D:
    gammas_LD_KS.append(abs(float(list_num_KS[j])))

# write an array containing the the ticks of the x axis
x = np.arange(1,N+1)

# write a list containing the name of the ticks of the x axis
x_name = []

for i in range(1,N+1):
    x_name.append(f'$10^{i}$')

# draw the plot for the errors for float numbers
plt.figure(figsize=(10,7))
plt.plot(x, gammas_F_DS, color='blue', marker='o', label='Direct Sum')
plt.plot(x, gammas_F_SS, color='magenta', marker='o', label='Sorted Sum')
plt.plot(x, gammas_F_PS, color='green', marker='o', label='Pairwise Sum')
plt.plot(x, gammas_F_KS, color='red', marker='o', label='Kahan Sum')
plt.xticks(x, x_name, fontsize=16)
plt.legend(fontsize=16)
plt.xlabel('Number of Iterations', fontsize=16)
plt.ylabel('$\epsilon_{alg}$', fontsize=16)
plt.yscale('log')
plt.title("Computation of $\epsilon_{alg}$ with Float numbers", fontsize=16)
plt.show()

# draw the plots for the errors for long double numbers
plt.figure(figsize=(10,7))
plt.plot(x, gammas_LD_DS, color='blue', marker='o', label='Direct Sum')
plt.plot(x, gammas_LD_SS, color='magenta', marker='o', label='Sorted Sum')
plt.plot(x, gammas_LD_PS, color='green', marker='o', label='Pairwise Sum')
plt.plot(x, gammas_LD_KS, color='red', marker='o', label='Kahan Sum')
plt.xticks(x, x_name, fontsize=16)
plt.legend(fontsize=16)
plt.xlabel('Number of Iterations', fontsize=16)
plt.ylabel('$\epsilon_{alg}$', fontsize=16)
plt.yscale('log')
plt.title("Computation of $\epsilon_{alg}$ with Long Double numbers", fontsize=16)
plt.show()
```

Exercise 3

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

```
#include "RooRealVar.h"
#include "RooConstVar.h"
#include "RooGaussian.h"
#include "RooArgusBG.h"
#include "RooAddPdf.h"
#include "RooDataSet.h"
#include "RooPlot.h"

using namespace RooFit;

// we first create the two observables (theta,phi) within their ranges [0,pi] and [0,2pi]
RooRealVar theta("theta","#theta",0,M_PI);
RooRealVar phi("phi","#phi",0,2*M_PI);

// we then define the three parameters
// alpha = 0.65
// beta = 0.06
// gamma = -0.18
RooRealVar alpha("alpha","#alpha",0.65,0.62,0.66);
RooRealVar bet("bet","#beta",0.06,0.05,0.075);
RooRealVar gam("gam","#gamma",-0.18,-0.2,-0.16);

// next we build the pdf function with the observables and the parameters previously defined
RooAbsPdf* pdf = RooClassFactory::makePdfInstance("pdf", "(3./(4.*M_PI))*(0.5*(1.-alpha) +
"(0.5)*(3.*alpha-1)*cos(theta)*cos(theta) - bet*sin(theta)*sin(theta)*cos(2.*phi)-
"sqrt(2.)*gam*sin(2.*theta)*cos(phi))", RooArgSet(theta,phi,alpha,bet,gam)) ;

// we draw the 2D histogram representing the pdf
TH1* hh_pdf = pdf->createHistogram("hh_model", theta, Binning(50), YVar(phi,Binning(50)));
TCanvas* c = new TCanvas("c","c",800,800);
hh_pdf->Draw("surf1");
c->Draw();

// we now generate the sample of 50000 MC events according to the previosuly defined pdf
RooDataSet* MC_ev = pdf->generate(RooArgSet(theta,phi),50000);

// we fit the MC events through the ML method with the pdf we have defined
// and we get the resulting parameters
RooFitResult *r_ML = pdf->fitTo(*MC_ev, Save());
r_ML->Print();
```

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

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

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

```
// we create the likelihood function starting from the MC events
RooAbsReal* nll = pdf->createNLL(*MC_ev, NumCPU(8));
// we then plot the likelihood function of the parameter analysed
RooPlot* frame = alpha.frame(Title("Likelihood #alpha"));
nll->plotOn(frame,ShiftToZero());
TCanvas* c2 = new TCanvas("c2","c2",800,800) ;
frame->Draw();
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 α, knowing that the codes used for the other two parameters are similar.

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

# we define the name of the ticks on the x axis
x = list(['ML', 'LS'])
# we fill two lists with the computed values of alpha and their errors
alpha = list([0.651,0.623])
alpha_err = list([0.003, 0.003])
# we plot these values
plt.errorbar(x, alpha, yerr=alpha_err, fmt='s', color = 'red')
# we plot the black dashed line which represent the true value of alpha
plt.plot((0,1),(0.65, 0.65), color='black', linestyle='--')
# we define the label for the y axis
plt.ylabel('$\alpha$', fontsize=16)
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.

```
// we first define some variables needed for the code
RooArgSet* pdfObs = pdf->getObservables(*MC_ev);
double log_lambda = 0;
double t, p;

// the value of the scalar-decay pdf is uniform, and so h0 is the same for all the data
double h0 = log(1/(4*M_PI));

// we loop over all the data
for (int i; i < 50000; i++) {
    // we first get theta and phi from the dataset
    auto ev = MC_ev->get(i);
    t = ev -> getRealValue("theta");
    p = ev -> getRealValue("phi");
    // we then obtain the value of the vector-pdf corresponding to the theta and phi just found
    *pdfObs = *MC_ev->get(i);
    double h1 = log(pdf->getVal());
    // we then sum the difference between the two log-likelihoods to the statistic
    log_lambda += h0 - h1;
}
```

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 $p \in [0, 10]$.

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

# we read and store the numbers generated with the xoroshiro128+ algorithm into a list
with open('xoroshiro_results', 'r') as f:
    list_num = f. read().splitlines()

# we convert the list of chars into an array of floats
list_arr = np.asarray(list_num)
list_arr = list_arr.astype(np.float)

# we then generate 4 lists, each containing 60000 random numbers
X = list_arr[:60000]            # x space coordinate
Y = list_arr[60000:120000]      # y space coordinate
Z = list_arr[120000:180000]     # z space coordinate
P = list_arr[180000:240000]     # momentum

# we redefine the range of the momenta in [0,10]
div = 2**64-1
for j in range(60000):
    P[j] = (P[j] / div)*10

# we now count the particles with high and low momenta
lowP = 0
highP = 0
for j in range (60000):
    if P[j] < 2 : lowP = lowP+1
    if P[j] > 8 : highP= highP+1

# we plot the uniform distribution of the momenta
plt.hist(P, bins = 240, color='c', density=True)
plt.xlabel("p (GeV)")
plt.ylabel("N")
plt.show()
```

```python
# we plot in a 3D space the space coordinates
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(15, 15))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(X, Y, Z, marker='o', color='red')
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')
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.

```python
# we generate another list of 60000 numbers
T = list_arr[240000:]

# we redefine the range of theta in [0, 2pi]
for j in range(60000):
    T[j] = (T[j] / div)*2*mt.pi

# we generate two arrays containing PL and PT
PL = np.array([])
PT = np.array([])
for j in range(60000):
    pl = abs(P[j]*mt.cos(T[j]))
    pt = P[j]*mt.sqrt(1-mt.cos(T[j])**2)
    PL = np.append(PL, pl)
    PT = np.append(PT, pt)

# we plot the histograms containing PL and PT
plt.hist(PL, bins = 240, color='c', density=True)
plt.xlabel("$p_L$ (GeV)")
plt.ylabel("N")
plt.show()

plt.hist(PT, bins = 240, color='c', density=True)
plt.xlabel("$p_T$ (GeV)")
plt.ylabel("N")
plt.show()

# we then count the particles with high and low PL and PT
lowPL = 0
highPL = 0
for j in range (60000):
    if PL[j] < 2 : lowPL = lowPL+1
    if PL[j] > 8 : highPL = highPL+1

lowPT = 0
highPT = 0
for j in range (60000):
    if PT[j] < 2 : lowPT = lowPT+1
    if PT[j] > 8 : highPT = highPT+1
```

```
# we show the first ten particles with their properties in a pandas DataFrame
import pandas as pd

data = pd.DataFrame(
    {
        'X': X,
        'Y': Y,
        'Z': Z,
        'P': P,
        '$\theta$': T,
        'PL':PL,
        'PT':PT
    }
)
data[:10]
```

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

```
import ROOT as r

# first of all, we plot the 2D histo of PT and PL
h = r.TH2F("h","h",100,0,10,100,0,10)
for i in range(60000):
    h.Fill(PL[i],PT[i])
c = r.TCanvas()
h.Draw("lego2")
h.GetXaxis().SetTitle("p_{L} (GeV)")
h.GetXaxis().SetLabelOffset(2)
h.GetYaxis().SetTitle("p_{T} (GeV)")
h.GetYaxis().SetLabelOffset(2)
h.SetTitle("")
h.SetStats(0)
c.Draw()

# subsequently, we plot <PT> in function of PL through the Profile method
PTPL = h.ProfileX("PTPL", 0, 100);
PTPL.GetXaxis().SetTitle("p_{L} (GeV)");
PTPL.GetXaxis().SetTitleOffset(1.2);
PTPL.GetYaxis().SetTitle("< p_{T}> (GeV)");
PTPL.SetStats(0)
c = r.TCanvas();
PTPL.Draw();
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 θ and p, uniformly distributed.

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

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

$$\begin{cases} p_T = p \cos\theta \\ p_L = p\sin\theta \end{cases} \tag{2}$$

Then we compute their inverses:

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

and we define the Jacobian matrix:

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

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

$$g(p_T;p_L) = |J| \cdot f\Big(\theta(p_T;p_L); p(p_T;p_L)\Big) \tag{5}$$

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

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

As regards θ, its pdf is written as follows.

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

So the joint function is:

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

and hence:

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

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.

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

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}.

```
// first of all we define the pdf of the transverse momentum to fit the data
TF1 *pdf_PT = new TF1("pdf_PT",
"[1]/(20*pi)*([2]*log(10+sqrt(pow([0]*x,2)+100))-log(10))", 0, 5);
pdf_PT->SetParameter(0,0.1);
pdf_PT->SetParameter(1,3000);
pdf_PT->SetParameter(2,0.5);

// then we draw the histogram of the data and we fit it with the function described above
c = new TCanvas();
h->SetTitle("");
h->SetLineColor(kBlue);
h->SetLineWidth(1);
h->SetStats(0);
h->Fit("pdf_PT");
h->Draw();
c->Draw();

// we acquire the chi2 and the NDF to calculate the reduced chi2
Double_t chi2 = pdf_PT->GetChisquare();
Double_t NDF = pdf_PT->GetNDF();
std::cout << chi2/NDF;
```

Exercise 5

Crude MC

Below the code to perform the Crude MC algorithm.

```
import numpy as np
import math as mt
import random
from matplotlib import pyplot as plt
from scipy import optimize
from IPython.display import clear_output
import matplotlib.mlab as mlab

# we define the integrand function
def f(x):
    return mt.e**x

# we then define the two arrays that will contain the values of Q_N and the relative variances
Integral = np.array([])
Vars = np.array([])

# we now implement the loop with which the values of Q_N and variances are computed
# we want to do 1000 iterations
for j in range (1000):
    # we define an array which will contain the values of f(x)
    # obtained with 128 random-extracted numbers in the interval (0,1)
    num = np.array([])
    for i in range (128):
        x = random.uniform(0,1)
        num = np.append(num, f(x))

    # we define the average of the values of f(x), namely Q_N
    m = num.mean()
    Integral = np.append(Integral, m)

    # we then calculate the variance of Q_N
    v = 0
    for i in range (128):
        v = v + (num[i]-m)**2
    v = v/(128-1)
    v = v/128
    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.

```
# we plot the values of Q_N
plt.hist(Integral, 25, color='blue')
plt.xlabel('$Q_N$', fontsize=16)
plt.ylabel('counts', fontsize=16)
plt.show()

# we plot the values of their variances
plt.hist(Vars, 25, color='blue')
plt.xlabel('$\hat{\sigma}^2$', fontsize=16)
plt.ylabel('Counts', fontsize=16)
plt.show()
```

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

```
m2 = np.mean(Integral)

S = 0
for i in range (1000):
    S += (Integral[i]-m2)**2
s_tilde = mt.sqrt(S/999)
s2 = s_tilde/mt.sqrt(1000)
print(f'For crude MC with 128 samples we have:\n'
      f'\tmean\t= {round(m2,4)}\n\terr\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 σ_I with different numbers of extractions. To compute σ_I we use the `np.var()` method directly on the array `Integral`.

```
mean_s = np.array([])
var_s = np.array([])

Ns = np.arange(100, 2500, 200)
for Ni in Ns:
    Integral = np.array([])
    for j in range (1000):
        num = np.array([])
        for i in range (Ni):
            x = random.uniform(0,1)
            num = np.append(num, f(x))
        Integral = np.append(Integral, num.mean())
    mean_s = np.append(mean_s, Integral.mean())
    var_s = np.append(var_s, Integral.var())
```

Finally we plot the values of σ_I in function of N and next we perform a fit of this data to obtain κ.

```
# here we plot the standard deviation of I in function of N
plt.plot(Ns, np.sqrt(var_s), 'go', color='green')
plt.xlabel('N', fontsize=16)
plt.ylabel('$\sigma_I$', fontsize=16)
plt.show()

# below we perform the fit
def test_func(x,a):
    return a / np.sqrt(x)

params, params_covariance = optimize.curve_fit(test_func, Ns, np.sqrt(var_s), p0=[2])

kappa = params[0]
print ('kappa = ', round(kappa,4))

err_kappa = np.sqrt(params_covariance[0][0])
print('err_kappa = ', round(err_kappa,4))
```

```
kappa =  0.485
err_kappa =  0.003
```

```
# finally we plot the data along with the fitted function
ran = np.arange(100, 2500, 1)

plt.plot(Ns, np.sqrt(var_s), 'go', color='green')
plt.plot(ran, test_func(ran, params), color='red')
plt.xlabel('N', fontsize=16)
plt.ylabel('$\sigma_I$', fontsize=16)
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.

```python
Integral = np.array([])
Vars = np.array([])

for j in range (1000):
    num1 = np.array([])
    num2 = np.array([])
    va = 0
    vb = 0
    for i in range (64):
        x1 = random.uniform(0,1/2.)
        x2 = random.uniform(1/2.,1)
        num1 = np.append(num1, f(x1))
        num2 = np.append(num2, f(x2))
    ma = num1.mean()
    mb = num2.mean()
    Integral = np.append(Integral, 0.5*(ma+mb))
    for i in range (64):
        va = va + (num1[i] - ma)**2
        vb = vb + (num2[i] - mb)**2
    va = va / ((64-1)*64)
    vb = vb / ((64-1)*64)

    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:

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

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

```python
def w1 (x):
    return (1+2*(np.e-1)*x)/np.e

def w2 (x):
    return (2.5+2.5*(np.e-1)*x**1.5)/(np.e+1.5)

xx = np.arange(0, 1, 0.01)

plt.plot(xx, f(xx), label='$y=e^x$')
plt.plot(xx, w1(xx), label='$y=w_1(x)$')
plt.plot(xx, w2(xx), label='$y=w_2(x)$')
plt.legend()
plt.xlabel('$x$')
plt.ylabel('$y$')
plt.show()
```

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

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

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

```
def s1 ():
    x = random.uniform(0, 1)
    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:

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

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.

```
for i in range (128):
    x = s1()
    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.

```
def s2 ():
    while(1):
        x=random.uniform(0, 1)
        y=random.uniform(0, np.e)
        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.

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

# then we show the values of I with their standard deviations along with the expected value
x = list(['Crude MC', 'Stratified\nsampling', 'Importance\nsampling\n$w_1$',
'Importance\nsampling\n$w_2$'])
y = list([1.7190, 1.7179, 1.7181, 1.7182])
err = list ( [0.0014, np.sqrt(v3)/np.sqrt(1000), np.sqrt(v4)/np.sqrt(1000),
np.sqrt(v5)/np.sqrt(1000)])
plt.errorbar(x, y, yerr=err, fmt='s', color = 'blue')
plt.plot((0,1,2,3),(1.7183, 1.7183, 1.7183, 1.7183), color='red', linestyle='--')
plt.ylabel('$I$', fontsize=16)
plt.show()
```

```

# finally, we show the values of kappa with their standard deviations
y = list([0.485, 0.254, 0.198, 0.037])
err = list ([0.3, 0.12, 0.02, 0.02])
plt.errorbar(x, y, yerr=err, fmt='s', color = 'blue')
plt.ylabel('$\kappa$', fontsize=16)
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.

```
import numpy as np
import matplotlib.pyplot as plt
import math as mt
import random
import scipy.special

# we define the parameters we need
l = 500 * 10^(-9)
a = 2 * 10^(-9)
k = 2*mt.pi/l

# we then generate 50000 numbers between 0 and 2*pi, namely the theta's
thetas = np.array([])
for i in range (50000):
    theta = random.uniform(0, mt.pi)
    thetas = np.append(thetas,theta)

# subsequently we define the argument of the Bessel function
z = np.array([])
for theta in thetas:
    z = np.append(z, k * a * mt.sin(theta))

# then we define the Bessel function itself
J1 = scipy.special.jv(1,z)

# finally we calculate the intensity spectrum we are looking for
I = np.array([])
for i in range(50000):
    I = np.append(I, 5*(2 * J1[i] / z[i])**2)

# we then plot the spectrum
fig = plt.figure(figsize=(10,7))
plt.hist(I, bins=70, density = True, fill = False, ec = 'green', range=(-1,1), histtype='step')
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.

```
# we implement the smearing for c = 0.4
c = 0.4
sigma = c * 0.03
smear04 = np.array([])
for j in range (50000):
    mean = list_arr[j]
    new = random.gauss(mean,sigma)
    smear04 = np.append(smear04, new)

# we implement the smearing for c = 0.9
c = 0.9
sigma = c * 0.03
smear09 = np.array([])
for j in range (50000):
    mean = list_arr[j]
    new = random.gauss(mean,sigma)
    smear09 = np.append(smear09, new)

# we draw the three distributions: original, c = 0.4 and c = 0.9
fig = plt.figure(figsize=(10, 7))
plt.hist(list_arr, bins=70, density = True, fill = False, ec = 'green', range=(-1,1), label = 'no sm
plt.hist(smear04, bins=70, density = True, fill = False, ec = 'red', range=(-1,1), label = 'c = 0.4
plt.hist(smear09, bins=70, density = True, fill = False, ec = 'blue', range=(-1,1), label = 'c = 0.
plt.legend(loc='upper right')
plt.show()

# we implement the smearing for c = 2
c = 2
sigma = c * 0.03
smear2 = np.array([])
for j in range (50000):
    mean = list_arr[j]
    new = random.gauss(mean,sigma)
    smear2 = np.append(smearTANTO, new)

# we draw the two distributions: original and c = 2
fig = plt.figure(figsize=(10, 7))
plt.hist(list_arr, bins=70, density = True, fill = False, ec = 'green', range=(-1,1), label = 'no sm
plt.hist(smearTANTO, bins=70, density = True, fill = False, ec = 'brown', range=(-1,1), label = 'c =
plt.legend(loc='upper right')
plt.show()
```

Unfolding

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

```python
import pyunfold

# we define the true and observed samples, we bin them
# and we compute the poissonian error on observed data
true_samples = list_arr
data_true, _ = np.histogram(list_arr, bins = 70)
observed_samples = smear09
data_observed, _ = np.histogram(smear09, bins = 70)
data_observed_err = np.sqrt(data_observed)

# we define as efficiencies 1 and as their errors 0.01
efficiencies = np.ones_like(data_observed, dtype=float)
efficiencies_err = np.full_like(efficiencies, 0.01, dtype=float)

# we define the response histogram and we plot it
response_hist, _, _ = np.histogram2d(observed_samples, true_samples, bins=70)
response_hist_err = np.sqrt(response_hist)

fig, ax = plt.subplots(figsize=(15,15))
im = ax.imshow(response_hist, origin='lower')
cbar = plt.colorbar(im, label='Counts')
ax.set(xlabel='Cause bins', ylabel='Effect bins')
plt.show()

# we then normalise the histogram to obtain the response matrix and we plot it
column_sums = response_hist.sum(axis=0)
normalization_factor = efficiencies / column_sums
response = response_hist * normalization_factor
response_err = response_hist_err * normalization_factor

fig, ax = plt.subplots(figsize=(15,15))
im = ax.imshow(response, origin='lower')
cbar = plt.colorbar(im, label='$P(E_i|C_{\mu})$')
ax.set(xlabel='Cause bins', ylabel='Effect bins', title='Normalizes response matrix')
plt.show()

# we define now the two callbacks, namely the logger and the regularizer
from pyunfold import callbacks

# the logger writes test statistic information for each unfolding iteration
logger = callbacks.Logger()

# the regularizer smooths the unfolded distribution at each iteration
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 (rmd), based on the ratio of absolute deviations of the observations from their class medians.

```
unfolded_results = iterative_unfold(data=data_observed,
                                    data_err=data_observed_err,
                                    response=response,
                                    response_err=response_err,
                                    efficiencies=efficiencies,
                                    efficiencies_err=efficiencies_err,
                                    ts = 'rmd',
                                    callbacks=[logger, regularizer]
    )
```

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

```
fig, ax = plt.subplots(figsize=(15, 15))
ax.step(np.arange(num_bins), data_true, where='mid', lw=3, alpha=0.7,
label='True distribution')
ax.errorbar(np.arange(num_bins), unfolded_results['unfolded'],
            yerr=unfolded_results['sys_err'],
            alpha=0.7,
            elinewidth=3,
            capsize=4,
            ls='None', marker='.', ms=10,
            label='Unfolded distribution')
ax.set(xlabel='X bins', ylabel='Counts')
plt.legend()
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.

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

# we define the parameters of the signal and background gaussians
mu_s = [0,0]
cov_s = [[0.09, 0.045], [0.045, 0.09]]

mu_b = [4,4]
cov_b = [[1., 0.4], [0.4, 1.]]

# we then generate the two gaussians
x_s, y_s = np.random.multivariate_normal(mu_s, cov_s, 50000).T
x_b, y_b = np.random.multivariate_normal(mu_b, cov_b, 50000).T

# finally we draw the two 2D gaussians and the two projections
plt.figure(figsize=(10, 10))
plt.plot(x_s, y_s, '.', color = 'blue', label = 'signal')
plt.plot(x_b, y_b, '.', color = 'red', label = 'background')
plt.legend()
plt.show()

plt.hist(x_s, bins=200, color = 'blue', histtype = 'step', label = 'signal')
plt.hist(x_b, bins=200, color = 'red', histtype = 'step', label = 'background')
plt.xlabel('X')
plt.legend()
plt.show()

plt.hist(y_s, bins=200, color = 'blue', histtype = 'step', label = 'signal')
plt.hist(y_b, bins=200, color = 'red', histtype = 'step', label = 'background')
plt.xlabel('Y')
plt.legend()
plt.show()

# we then build a dataframe containing all the data
x = np.append(x_s,x_b)
y = np.append(y_s,y_b)
```

```
# we fill the array s_b with 0.0 and 1.0 depending on the belonging class
s_b = np.array([])

# signal = 1
for i in range (50000):
    s_b = np.append(s_b,1)

# background = 0
for i in range (50000):
    s_b = np.append(s_b,0)

data = pd.DataFrame(
    {
        'x': x,
        'y': y,
        's_b': s_b
    }
)

data.head()
```

Binary Linear Classification

Below the code to perform the binary linear classification.

```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import LabelEncoder
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.metrics import log_loss

# we define the two target classes
classes = data["s_b"].unique()

# we then create two arrays containing features and target
X = data[['x', 'y']]
Y = data['s_b']

# we now create the train and test sets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3)

# the features are now standardised
ss = StandardScaler()
X_train = ss.fit_transform(X_train)
X_test = ss.transform(X_test)
```

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

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score

# we perform the logistic regression
lr = LogisticRegression(solver='lbfgs')
lr.fit(X_train, Y_train)
```

```python
# prediction of the target Y over X_test
Y_pred = lr.predict(X_test)
# probability of correct prediction
Y_pred_proba = lr.predict_proba(X_test)
# accuracy, computed seeing the differences between Y_test and Y_pred
A = accuracy_score(Y_test, Y_pred)
# log_loss, calculated seeing the probability that Y_pred is correct looking at Y_test
LL = log_loss(Y_test, Y_pred_proba)
# we then print both the metrics
print('ACCURACY = ', round(A,2))
print('LOG LOSS = ', round(LL,2))

# we allocate Y_test and Y_pred in a new dataframe to see which ones are different
data_new = pd.DataFrame(
    {
        'Y_test': Y_test,
        'Y_pred': Y_pred
    }
)

# calculate the number of Type I errors (false positive)
data_new.loc[(data_new.Y_test != data_new.Y_pred) & (data_new.Y_test == 0)]

# calculate the number of Type II errors (false negative)
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.

```python
# we first define a function to show the boundary line
def showBounds(X, Y, model, title=None):
    fig = plt.figure(figsize=(15,10))
    h = .02
    # we define the minimum and the maximum of each set
    x_min, x_max = X[:, 0].min(), X[:, 0].max()
    y_min, y_max = X[:, 1].min(), X[:, 1].max()

    # we obtain the coordinate matrices
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    # we plot the predicted line
    Z = model.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    plt.contourf(xx, yy, Z, cmap=plt.cm.jet)

    # we finally plot the two distributions
    X_m = X[Y==1]
    X_b = X[Y==0]
    plt.title(title)
    plt.scatter(X_b[:, 0], X_b[:, 1], marker='.', c='red')
    plt.scatter(X_m[:, 0], X_m[:, 1], marker='.', c='blue')
    plt.show()

# then we pass at the function the train and test sets
showBounds(X_train, Y_train, lr, title='Train set')
showBounds(X_test, Y_test, lr, title='Test set')
```