Random Algorithms (RA-MIRI)
Assignment 1

Daniel Benedí

October 2022

1 Exercise 1: Throwing darts

The objective of this exercise is to compute π using a Monte Carlo method in which the area of a circle will be sampled randomly compared with the square that contains such circle. We take in advantage that the proportion of both areas is $\frac{\pi}{4}$:

$$\frac{\text{Area}_{\circ}}{\text{Area}_{\square}} = \frac{\pi r^2}{(2r)^2} = \frac{\pi}{4}$$

To do so, we will generate uniformly at random points in the square and check if the distance between the point and the center of the circle is lower or equal than the radius. If it is lower than the radius, we will count the point. And because of the law of large numbers, we expect that the proportion of points inside the circle and the total of points will be approximately $\frac{\pi}{4}$.

1.1 Implementation

We will generate points uniformly at random, $P = (X, Y)$, $X, Y \sim U[0, 1]$, therefore our circle will have its center at $C = (0.5, 0.5)$ and radius $r = 0.5$. With this definition, our condition will be:

$$d(P, C) = \sqrt{(P_X - C_X)^2 + (P_Y - C_Y)^2} \leq 0.5$$

In order to simplify the computations of this condition, instead of taking the square root, we can square the radius because its implementation has a lower cost. Also if we set radius $r = 1$ and center $C = (0, 0)$, we are removing the square of the radius because $1^2 = 1$ and the minus. To still be able to get $\frac{\pi}{4}$, we will still generate the point $P = (X, Y)$, $X, Y \sim U[0, 1]$ because we will take into account only one forth of the area of the circle. With this new conditions, our new condition will be:

$$P_X^2 + P_Y^2 \leq 1$$

We can design the experiment in pseudo-code:

Algorithm 1 Throwing darts

```
procedure EXPERIMENT($N$)                ▷ Monte Carlo method to compute $\frac{\pi}{4}$
    $in \leftarrow 0$                     ▷ It will have the points that lied inside the circle
    for all $i$ in $[1, N]$ do
        $X \leftarrow$ RANDOM(0,1)
        $Y \leftarrow$ RANDOM(0,1)
        if $X * X + Y * Y \leq 1$ then
            $in \leftarrow in + 1$
        end if
    end for
    return $\frac{\text{in}}{N}$    ▷ If we want $\pi$ instead, the value has to be multiplied by 4
end procedure
```

The implementation provided uses the standard library of C++ (further detail in appendix A). To generate the points we use a pseudo-random numbers engine provided by the standard library called `std::mt19937` (for implementation details check Makoto and Takuji paper [2]). We have chosen this engine instead of a linear congruential because with the right parameters it has the longest non-repeating sequence and we want to generate a lot of numbers, although it is slower and has greater state storage requirements. It makes use of `std::random_engine` which is a uniformly-distributed integer random number generator that produces non-deterministic random numbers or pseudo-random numbers if a non-deterministic source is not available. After generating the random (or pseudo-random) bits, they are passed to `std::uniform_real_distribution` so it can generate a decimal value in $[0, 1)$. To do this experiment we have generated almost 175 million points.

1.2 Results

Before analyzing any result we get doing this experiment, we have to check if there has been any error generating the random (or pseudo-random) points. To do so, we firstly can do a histogram over the X variable and the Y variable, figure 1. We can observe in that histogram that the variables X and Y are evenly distributed over $[0, 1)$.

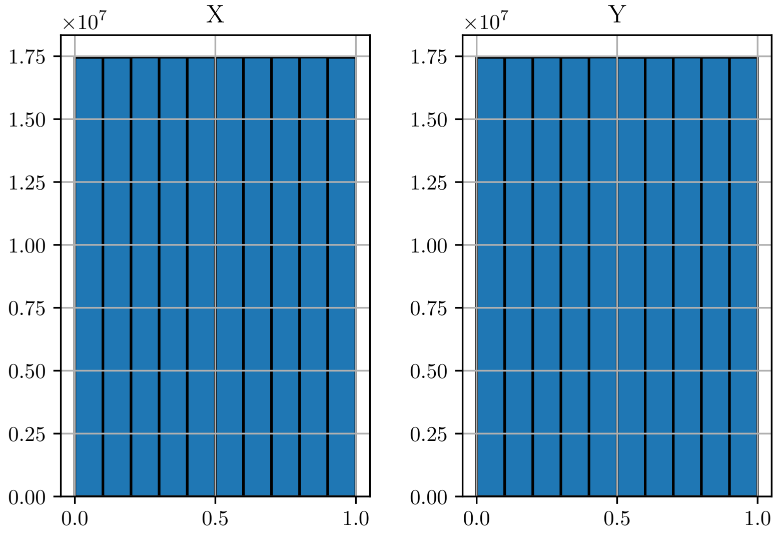


Figure 1: Histogram of the generated points over its variables, X and Y.

We should also check that there is no correlation between X and Y although we have used the same number generator. Each cell of the table 1 shows the number of points for which in each line their X is between that value and the next one and for each column their Y is between that value and the next one. For instance, the point $P = (0.245, 0.742369)$ will be in the intersection between the third row and the eight column. From this table, it seems that all the points are evenly distributed and that there is no hidden correlation.

X \ Y	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
0.0	1745145	1743955	1745232	1744564	1746407	1744605	1746542	1746672	1744476	1743516
0.1	1745270	1746324	1748509	1744806	1746836	1745933	1745279	1742781	1746230	1744721
0.2	1747144	1745078	1745527	1746533	1747825	1745591	1745113	1748146	1745903	1748439
0.3	1743395	1743625	1747289	1747116	1746891	1745744	1746801	1744843	1748323	1744737
0.4	1746659	1746695	1746828	1745925	1745895	1743369	1745108	1748394	1742910	1747143
0.5	1745416	1746147	1747978	1746176	1745469	1746341	1745014	1746914	1746220	1744766
0.6	1746182	1748644	1747573	1745971	1746631	1744745	1743806	1747284	1746923	1745694
0.7	1744587	1746206	1747853	1746838	1746575	1746781	1744927	1747990	1745125	1746856
0.8	1745560	1747070	1745350	1745748	1747400	1745993	1747877	1746417	1747316	1745785
0.9	1744578	1743407	1745620	1744546	1747630	1744052	1745102	1747261	1744886	1745353

Table 1: Number of points that belong to that range

Once we are sure that the generated points are useful, we can study how the proportion approximate $\frac{\pi}{4}$. All the showed plots have the number of generated points, or iteration, in the x axis. It is showed with logarithmic scale because there are more variance in the approximation at the beginning than at the end. Instead of taking all the values of the approximation, we only use them if it is distinct from the previous one; that is, we will use the i-th approximation if it is different from the previous one, $evolution[i] \neq evolution[i-1]$. This was a good point because making the plots was easier due to the fact the instead of processing 175 million points, we only used 250 thousand and we still kept the precision of the results and their evolution across iterations.

First of all, the figure 2 shows how the approximation of the value oscillates around the objective value, and as there are more and more points the approximation improves until it is quite close to $\frac{\pi}{4}$.

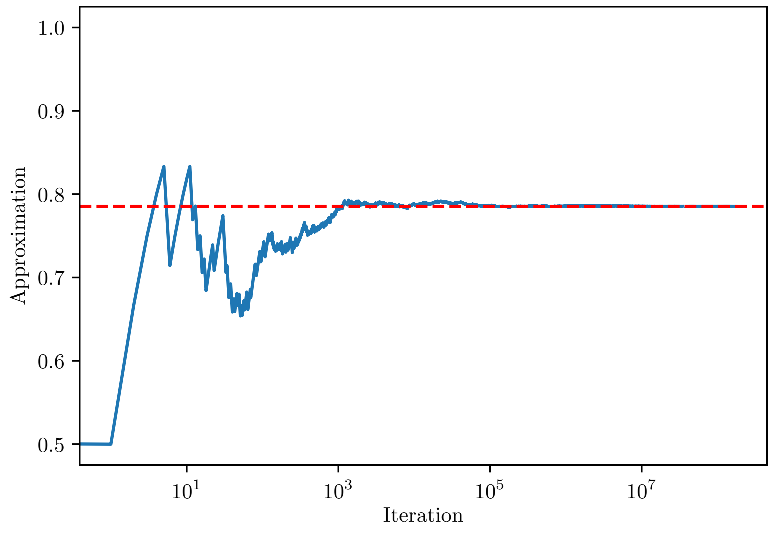


Figure 2: Evolution of the approximation of $\frac{\pi}{4}$

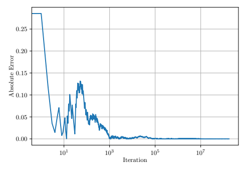
Although it is important to observe that the value it is quite close to the objective value, we also need to know how much error there is in our approximation. To do so, we propose three error functions $\epsilon(x)$, the absolute error (equation 1), the squared error (equation 2) and the percentage of error (equation 3). The main idea in the absolute error (equation 1) is that we can observe how much error there is, no matter its direction; this kind of error function is useful when the sign of the error does not matter like this case. The squared error function (equation 2) holds this idea of hiding the sign of the error, by the use of the square, and also has the property to magnify the big errors and minimize the small error; although it could be useful at the first iterations, it hides a lot of information as the number of iterations grows because the amount of error decreases. The third proposed error function, the percentage of error (equation 3), normalizes the error while keeping all kind of errors in the same magnitude, although with this implementation the sign of the error matters, it can be easily modified plugging the absolute value so it won't care. The main advantage from this error function is that the unit of the approximation does not care and the error will not be distorted by the magnitude of the population.

$$\epsilon(x) = \left|\frac{\pi}{4} - x\right| \tag{1}$$

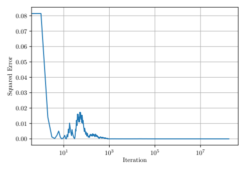
$$\epsilon(x) = \left(\frac{\pi}{4} - x\right)^2 \tag{2}$$

$$\epsilon(x) = \frac{x}{\frac{\pi}{4}} - 1 \tag{3}$$

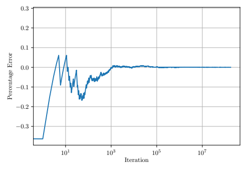
In figure 3 we can observe how this different error functions evaluate with our experiment. For instance, the absolute error (figure 3a) reports a higher error than the square error and it can be observed not only in the values, but also in the fact that the squared error is flat at 0 after 10^3 meanwhile the absolute error and the percentage they still oscillate. On the other hand, we observe that absolute error and percentage error has more or less the same shape even when it is zoomed in after the 10^5-th iteration (see appendix B). Overall, we can say that it is need a lot of iterations to get a not too bad approximation to $\frac{\pi}{4}$ and therefor to π.



(a) Absolute error of the approximation of $\frac{\pi}{4}$



(b) Squared error of the approximation of $\frac{\pi}{4}$



(c) Error percentage of the approximation of $\frac{\pi}{4}$

Figure 3: Error functions

In order to understand how the error evolves as the number of darts increases, we can plot (figure 4) the logarithmic value of the iterations and the logarithmic value of the error, also called log-log plot. The benefit of log-log plot is that the polynomial functions, $y = ax^k$, appears as straight line. From the figure 4, we observe that the error seems to approximate to a straight. If we fit a line, we will obtain a line with slope -0.89. The log-log plot has the property that a polynomial function of form $y = ax^k$ will appear as $y' = k * x' + b$, so we experimentally can define an upper bound of our error as $\epsilon(x) \in o(x^{-0.89})$.

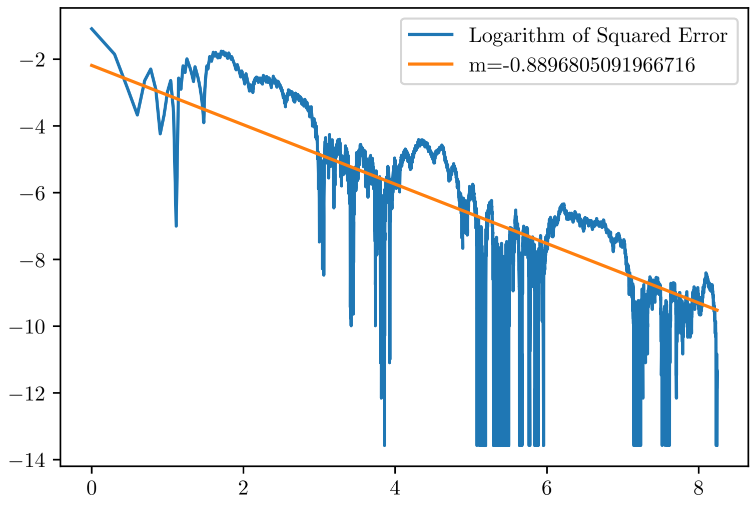


Figure 4: Log-Log plot of the squared error

It is quite common to compare the quality of a π approximation by the number of correct decimal places. The figure 5 shows how this decimal places evolve along the time. It has been computed with the following Python code:

```
import math

def similarity(a):
  a_str = str(a)
  b_str = str(math.pi/4)
  count = 0
  while(count < len(a_str) and
        count < len(b_str) and
        a_str[count] == b_str[count]):
    count += 1
  return count - 2
```

We can see that little by little the decimal precision improves taking some steps until it stabilises. In some iterations, we have been lucky with the result of our experiment and we get a better approximation, like in the iteration close to 10 that we go from 0 precision digits to up to 3. After 175 million iteration we have not been able to stabilise the precission to 6 digits, this notices us that maybe this method is not the best to approximate π. Modern procedures that do not make use of randomization have been able to achieve 10^{13} digits of precision [1].

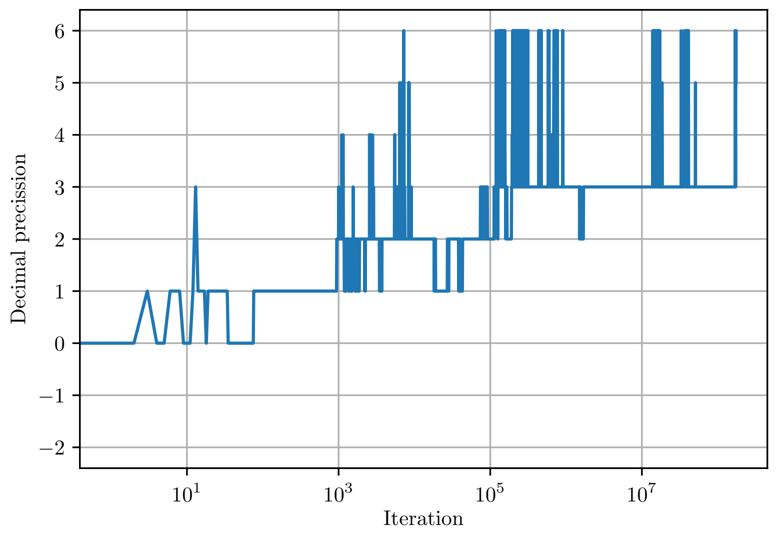


Figure 5: Correct decimal places of the approximation

2 Exercise 2: Buffon's needles

The objective of this exercise is to compute π using a Monte Carlo method as well. In this case, we will sample randomly the position and the orientation of a segment (or needle) of length l and check how many of them cross the half of the stripes of width t, with $l \leq t$. We will take in advantage that the probability of the segment crossing the stripes is:

$$\frac{2l}{t\pi}$$

So, to realize the experiment it will be generated uniformly at random the distance from the half of the segment to the closest stripe and then the positive acute angle of the segment. Then we will count the point if the length of the projection of the half of the segment with that angle is bigger than the distance to the stripe. The proportion of segments that cross a stripe over all the stripes will be approximately $\frac{2l}{t\pi}$.

2.1 Implementation

We will generate points uniformly at random, $P = (X, \theta)$, $X \sim U[0, \frac{t}{2}]$ and $\theta \sim U[0, \frac{\pi}{2}]$ [1]. To simplify our ratio we will force the size of the needle to be $l = \frac{t}{2}$ and for simplicity $t = 1$. Because of this assumption, the ratio we will get doing the experiment will be:

$$\frac{2l}{t\pi} = \frac{2\frac{t}{2}}{t\pi} = \frac{1}{\pi}$$

The following equation is used to define whether a needle will cross a stripe:

$$x \leq \frac{l}{2}\sin\theta = \frac{1}{4}\sin\theta$$

This equation comes from the fact that the projection of the half of the needle is computed from the sine of the angle and in case that projection is bigger than the distance to the closest stripe, obviously the needle will be crossing the stripe.

Note that there is a dependency with π to generate the angle uniformly at random. To solve this issue, in section 2.3 we propose a method generating a number between 0 and 1, but we were not able to get rid off the dependency with π, so we did not implement it as a method for this experiment.

The proposed experiment in pseudo-code is:

[1] In case we generate the angle between $[-\frac{\pi}{2}, \frac{\pi}{2}]$, it can be proven that the probability is $\frac{l}{t\pi}$.

Algorithm 2 Buffon's needles

```
1:  procedure EXPERIMENT($N$)                    ▷ Monte Carlo method to compute $\frac{1}{\pi}$
2:      $in \leftarrow 0$                        ▷ It will have the points that cross a stripe
3:      for all $i$ in $[1, N]$ do
4:          $X \leftarrow$ RANDOM($0, \frac{1}{2}$)
5:          $Y \leftarrow$ RANDOM($0, \frac{\pi}{2}$)
6:          if $X \leq \frac{1}{4} \sin Y$ then
7:              $in \leftarrow in + 1$
8:          end if
9:      end for
10:     return $\frac{\text{in}}{N}$   ▷ If we want $\pi$ instead, the value has to be the inverse, $\frac{N}{\text{in}}$
11: end procedure
```

The implementation provided uses, as the Dart's experiment, the standard library of C++ (further detail in appendix A). The code uses the pseudo-random numbers engine, same as in the previous experiment, called `std::mt19937`. All the details of the implementation and reasons are the same as in the darts experiment, so for further detail check them in section 1.1. In this case, we are using two different `std::uniform_real_distribution` with the random bits so we can generate decimal values in $[0, \frac{1}{2})$ and $[0, \frac{\pi}{2})$. We have also generated almost 175 million points, the same as in the previous experiment.

2.2 Results

Similarly to what it has been done in the previous experiment, we will firstly check the quality of the random (or pseudo-random) points before analysing any result that we got. In the histogram at figure 6 we can observe that the variable X and θ are evenly distributed over their ranges, $[0, \frac{1}{2})$ and $[0, \frac{\pi}{2})$ respectively.

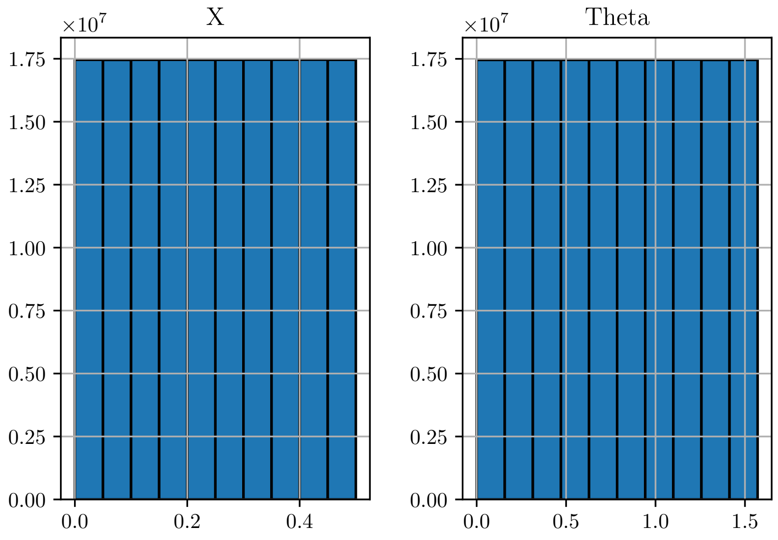


Figure 6: Histogram of the generated points over its variables, X and θ.

There should not be any correlation between the variables. As we have done before, we have generated a table (table 2 that in each cell shows the number of points for which in each line their X is in between that value and the next one and for each column their θ is between that value and the next one. For instance, the point $(X = 0.368, \theta = 0.12475)$ will have coordinates $(7, 0)$. Taking into account the table, it seems that all the points are evenly distributed and that there is no hidden correlation.

X \ θ	0.00000	0.15708	0.31416	0.47124	0.62832	0.78540	0.94248	1.09956	1.25664	1.41372
0.0	1745298	1745913	1744890	1749010	1744689	1747100	1744205	1746654	1746743	1744139
0.1	1746091	1746434	1744361	1745264	1745588	1745129	1745544	1745610	1743290	1747144
0.1	1745051	1743897	1746271	1746561	1749243	1743296	1745709	1745124	1745675	1747143
0.2	1747556	1746592	1746139	1744996	1742907	1744477	1748880	1746614	1746491	1745220
0.2	1745172	1746376	1744420	1749141	1745966	1747700	1744406	1745343	1744087	1745279
0.2	1745872	1745752	1747029	1746109	1744751	1743163	1746133	1748903	1745723	1744913
0.3	1745912	1746070	1745746	1743572	1747339	1747740	1744833	1748618	1746225	1746565
0.4	1746243	1744542	1745563	1744785	1746979	1747261	1745788	1747309	1746268	1747636
0.4	1745542	1746588	1744003	1745399	1746377	1747156	1745375	1747506	1747496	1748886
0.5	1745452	1746970	1746396	1744614	1745500	1745867	1748520	1746562	1745389	1743607

Table 2: Number of points that belong to that range

Once we are sure that the generated points are useful and they will fulfill our requirement, an study about how the proportion approximates $\frac{1}{\pi}$ can be done. Same as in the previous experiment, all the showed plots have the number of generated points, or iterations, in the x axis. It is showed with logarithmic scale because there are more variance in the approximation at the beginning that at the end. Instead of taking all the values of the approximation, we only use them if it is distinct from the previous one; that is, we will use the i-th approximation if it is different from the previous one, $evolution[i] \neq evolution[i-1]$. This was a good point because making the plots was easier due to the fact that instead of processing 1.5GB of points, we only used around 250 thousand while keeping the precision of the results and its evolution.

The figure 7 shows how the approximation of the value oscillates around the objective value, and as there are more and more points the approximation improves until it is quite close to the our objective, $\frac{1}{\pi}$.

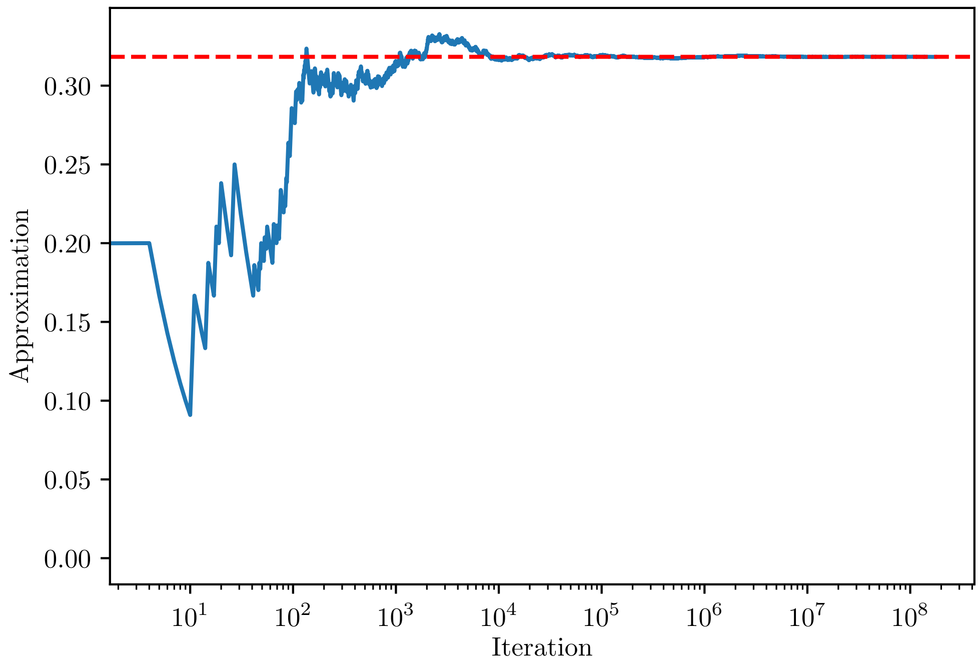
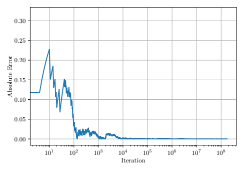


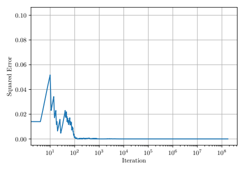
Figure 7: Evolution of the approximation of $\frac{1}{\pi}$.

Once we have observed how approximation tends to the objective value, we would like to quantify the quality of the approximation, that means to measure how much error there is. It has been used the same error functions as in the previous experiment because their properties are also interesting for this case: the absolute error (equation 1), the squared error (equation 2) and the normalized error (equation 3).

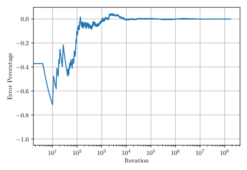
In the following figure, figure 8, we observe the 3 equations applied to this experiment. Comparing the absolute error (figure 8a) and the squared error (figure 8b), they report the same amount of error, but the squared error tends to flatten more the small error, this can be observed because the absolute error reports some significant error in iteration 10^2 and 10^4, but the squared error seems flat. If we use the normalized error (figure 8c) to compare this experiment with the previous one (figure 3c), it can be observed that the dart's experiment is a better approximation than the needle's experiment because it converge earlier to the real value. The same ideas can be observed from zoomed values (appendix C). The squared error flattens more the error for small values and the normalized error is 4 time bigger with the needles' experiment (figure 17) than the darts' experiment (figure 14).



(a) Absolute error of the approximation of $\frac{1}{\pi}$



(b) Squared error of the approximation of $\frac{1}{\pi}$



(c) Error percentage of the approximation of $\frac{1}{\pi}$

Figure 8: Error functions

When measuring how the error evolves depending on the number of rounds in the experiment, we firstly do a log-log plot, figure 9. In the plot, it can be observed that it seems to approximate with a straight line, so after fitting a straight line we get that it has a slope of $m = -1.21$. According to the properties of the log-log plot then the error evolves within a $o(n^{-1.21})$.

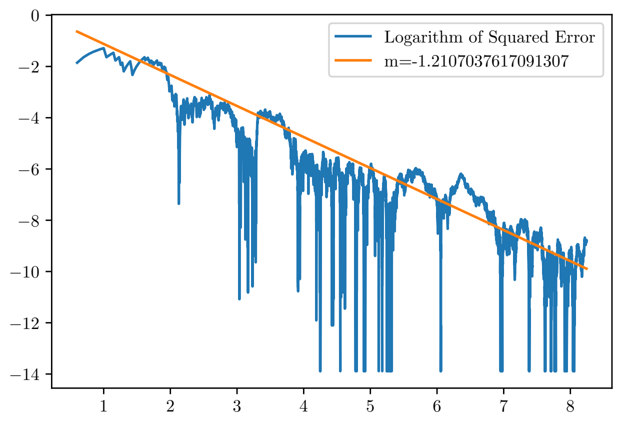


Figure 9: Log-Log plot of the squared error

The evolution curve of the needles' experiment and the darts' experiment should not be compared because they have different magnitudes because they are strongly dependent of their approximation value. To do a proper comparison, we provide the figure 10 in which it is shown the log-log plot of both experiment but in the absolute value of the normalized error which is not dependent of the approximation value and therefore it can be used to compare both approximations. We can observe that the error in the needles' experiment decreases faster than in the darts' experiment, therefore we could believe it is a better approximation because asymptotically will have less error earlier. But taking into account the y-intercept, it can be observed that at the beginning it has a bigger error so within the first 10^8 iterations the darts' experiment is much better.

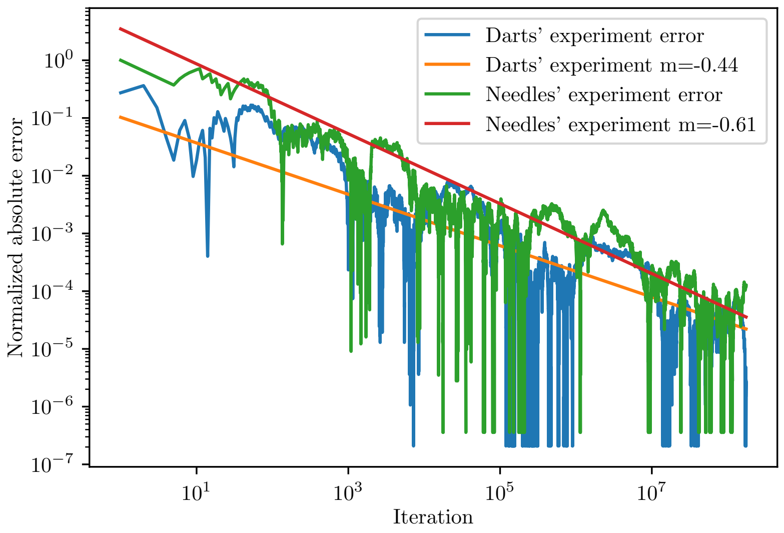


Figure 10: Comparison of normalized absolute error

Similarly to the analysis of correct decimal places it has been done in the previous experiment, it can be done with this approximation. We observe that at the beginning there is no correct decimal places until 10^2-th iteration. After that, it can be observed that the decimal places stabilises before with this method than the darts' method. So, as we have seen in the previous comparison, this method might be more stable than the other, but it has worse beginning.

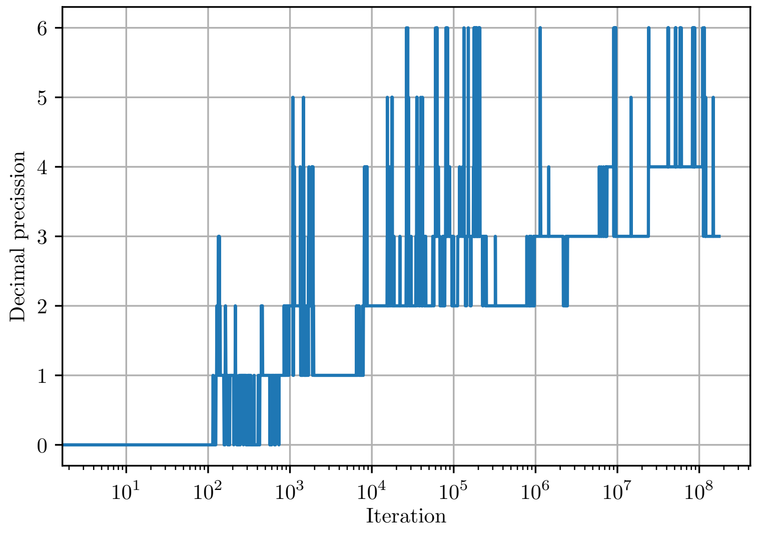


Figure 11: Correct decimal places of the approximation

2.3 Extra method

One can observe that using the previous method, there is a dependency with π to generate u.a.r the angle θ. To solve this dependency, instead of randomly generating θ that is bounded between $[0, \frac{\pi}{2}]$, we are going to randomly generate its cosine that is bounded between $[0, 1]$.

First, we need to check if the distribution of the cosine of a uniform distributed variable is also uniform. We know that $\Theta \sim U([0, \frac{\pi}{2}])$, so the probability density function is:

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

If we define the random variable $Y = \cos\Theta$, we know that $Y \in [0, 1]$ and also there is one solution for the equation $y = \cos\theta \to \theta = \cos^{-1} y$. Therefore, the density of $Y = \cos\Theta$ is given by the evaluation of f_Θ multiplied by the derivative of the inverse of our transformation (the inverse function of $y = cos\theta$):

$$\begin{aligned}
f_Y(y) &= f_\Theta(\cos^{-1} y) \left| \frac{d\theta}{dy} \right| \\
&= f_\Theta(\cos^{-1} y) \left| \frac{d(cos^{-1}y)}{dy} \right| \\
&= f_\Theta(\cos^{-1} y) \left| \frac{-1}{\sqrt{1-y^2}} \right| \\
&= \begin{cases} \frac{2}{\pi}\frac{1}{\sqrt{1-y^2}} & \text{if } \cos^- 1y \in [0, \frac{\pi}{2}] \\ 0 & \text{otherwise} \end{cases} \\
&= \begin{cases} \frac{1}{\pi\sqrt{1-y^2}} & \text{if } y \in [0,1] \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

We can observe that this probability distribution function is not uniformly distributed, so we can not generate $\cos\theta$ that simply. We are going to use inverse transform sampling. It is a method for pseudo-random number sampling from any probability distribution. Inverse transformation sampling takes uniform samples of a number between 0 and 1, and then returns the largest number from the domain of the distribution, in our case $P(Y)$.

This method requires the cumulative probability function. To obtain it, we need to do the following:

$$\begin{aligned}
F_Y(y) &= \int_{-\infty}^{y} f_Y(t)dt \\
&= \int_{0}^{y} \frac{2}{\pi\sqrt{1-t^2}}dt \quad \forall y \in [-\infty, 0], F_Y(y) = 0 \\
&= \frac{2\sin^{-1} t}{\pi}\Big|_0^y \\
&= \frac{2\sin^{-1} y}{\pi} - \frac{2\sin^{-1} 0}{\pi} = \frac{2\sin^{-1} y}{\pi} \\
F_Y(y) &= \begin{cases} 0 & \text{If } y < 0 \\ \frac{2\sin^{-1} y}{\pi} & \text{If } y \in [0,1] \\ 1 & \text{If } y > 1 \end{cases}
\end{aligned}$$

The inverse transform sampling method generates a random number $u, U \sim Unif(0,1)$. Then it uses the inverse of the desired CDF. Our random number in that distribution will be $Y = F_Y^{-1}(u)$. To find such inverse, we will solve the following equation $F_Y(F_Y^{-1}(u)) = u$.

$$\begin{aligned} F_Y(F_Y^{-1}(u)) &= u \\ \frac{2}{\pi}\sin^{-1}(F_Y^{-1}(u)) &= u \\ \sin^{-}1(F_Y^{-1}(u)) &= \frac{\pi u}{2} \\ F_Y^{-1}(u) &= \sin(\frac{\pi u}{2}) \end{aligned}$$

Our program will generate u.a.r number between $[0, 1]$ and using that number u, evaluate the function $F_Y^{-1}(u) = \sin(\frac{\pi u}{2})$. But we have the same problems as at the beginning, we are using π to generate π, so this is the reason we have not implemented this method.

Appendices

A C++ Implementation

```
//**************************************************
// File:    assignment1.cpp
// Author: Daniel Benedi Garcia
// Date:    02/10/2022
// Coms:    Implements both experiments for realizing
//          the experiments of assignment 1.
//**************************************************

#include <iostream>
#include <fstream>
#include <cstdlib>
#include <random>
#include <cmath>

void experiment(uint64_t N,
                double* rand(),
                bool eval(double* data,
                          std::ofstream& out),
                void header(std::ofstream& out)){
    std::ofstream evolution("evolution.csv");
    std::ofstream points("points.csv");

    uint64_t C = 0;
    header(points);
    for(uint64_t n = 1; n <= N; n++){
        double* data = rand();
        if(eval(data, points)){
            C++;
        }
        evolution << ((double)C)/((double)n);
        evolution << std::endl;
    }
}

bool eval_darts(double* data, std::ofstream& out){
    double x = data[0], y = data[1];
    delete[] data;
```

```cpp
    out << x << "," << y << std::endl;
    return 1 > (x*x + y*y);
}

double* rand_darts(){
    static std::random_device rd;
    static std::mt19937 gen(rd());
    static std::uniform_real_distribution<> dis(0.0,
                                                 1.0);

    double* data = new double[2];

    data[0] = dis(gen);
    data[1] = dis(gen);

    return data;
}

void header_darts(std::ofstream& out){
    out << "X,Y" << std::endl;
}

bool eval_needls(double* data, std::ofstream& out){
    double x = data[0], theta = data[1];
    delete[] data;
    out << x << "," << theta << std::endl;

    return x <= 0.25 * sin(theta);
}

double* rand_needls(){
    static std::random_device rd;
    static std::mt19937 gen(rd());
    static std::uniform_real_distribution<> disX(0.0,
                                                  0.5);
    static std::uniform_real_distribution<> disT(0,
                                                 M_PI/2.0);

    double* data = new double[2];

    data[0] = disX(gen);
    data[1] = disT(gen);

    return data;
}

```

```
void header_needls(std::ofstream& out){
    out << "X,Theta"  << std::endl;
}

int main(int argc, char* argv[]){
    int type = std::atoi(argv[1]);
    if(argc != 2 || ( type != 1 && type != 2 )){
        std::cout << "Usage: " << argv[0] << "type";
        std::cout << std::endl;
        std::cout << "type: Experiment to realize. ";
        std::cout << "1 for darts, ";
        std::cout << "2 for buffon's needles.";
        std::cout << std::endl;
    }

    uint64_t N;
    std::cout << "Insert number of runs: ";
    std::cout << std::flush;
    std::cin >> N;

    if(type == 1)
        experiment(N, rand_darts,
                   eval_darts, header_darts);
    else if(type == 2)
        experiment(N, rand_needls,
                   eval_needls, header_needls);
    return 0;
}
```

B Extra graphs of darts experiment

Zooming in error functions tails for more than 10^5 iterations.

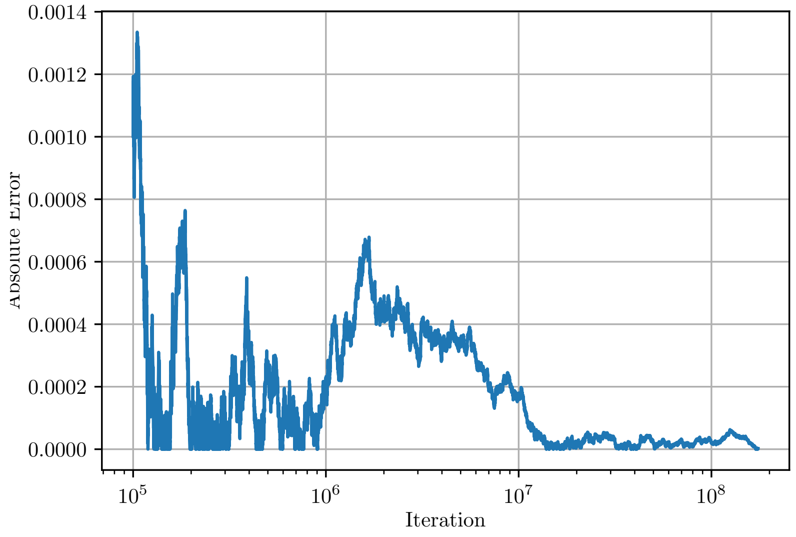


Figure 12: Zooming in the absolute error

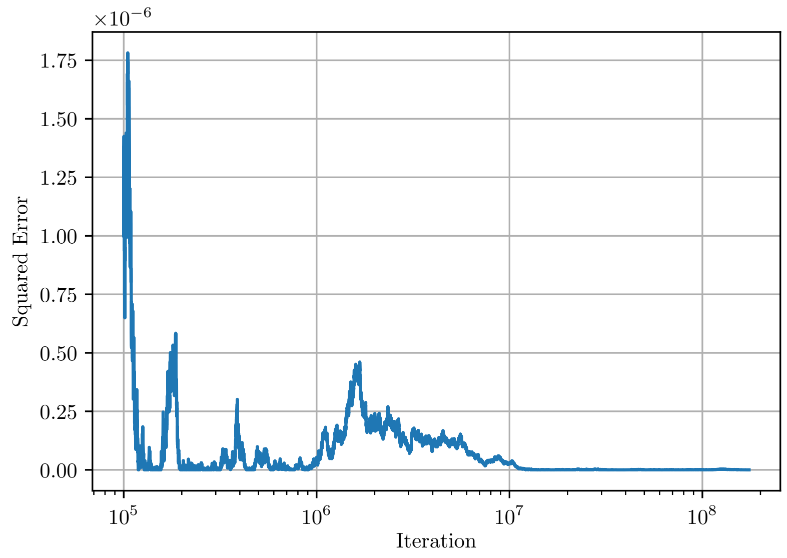


Figure 13: Zooming in the squared error

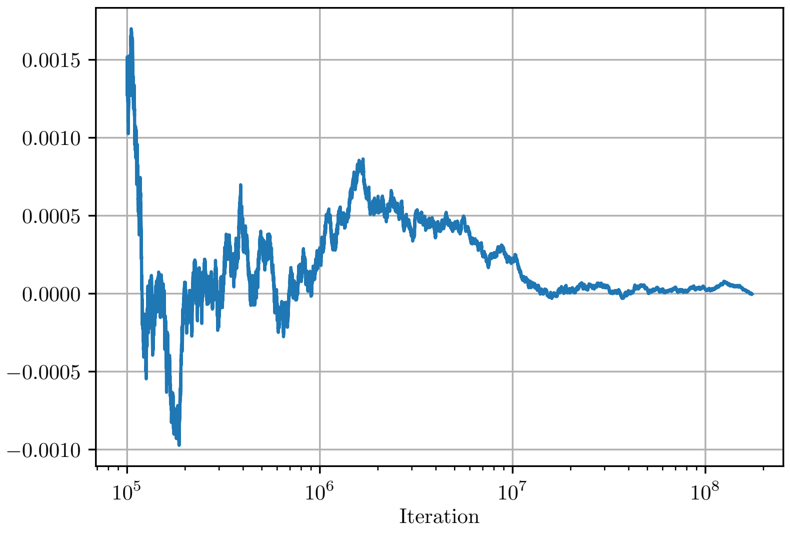


Figure 14: Zooming in the error percentage

C Extra graphs of needles experiment

Zooming in error functions tails for more than 10^5 iterations.

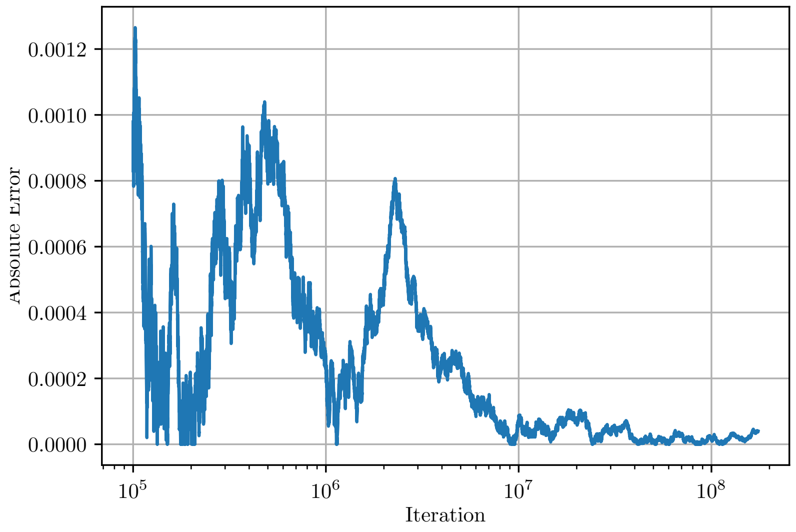


Figure 15: Zooming in the absolute error

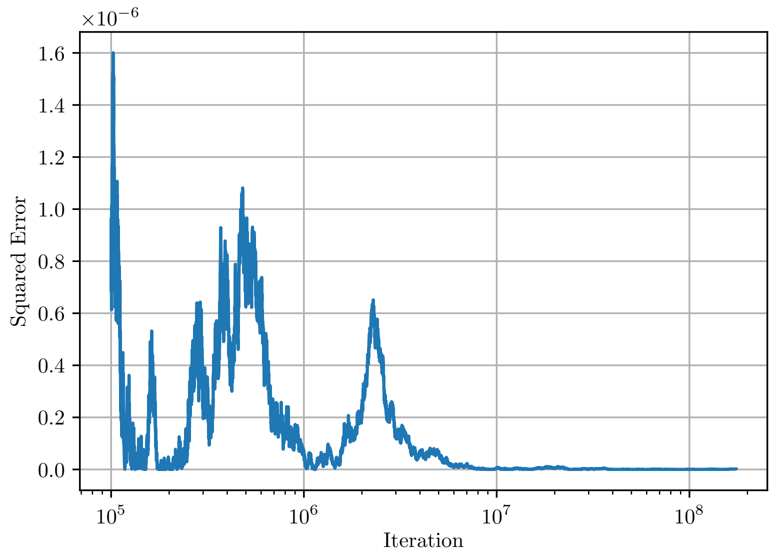


Figure 16: Zooming in the squared error

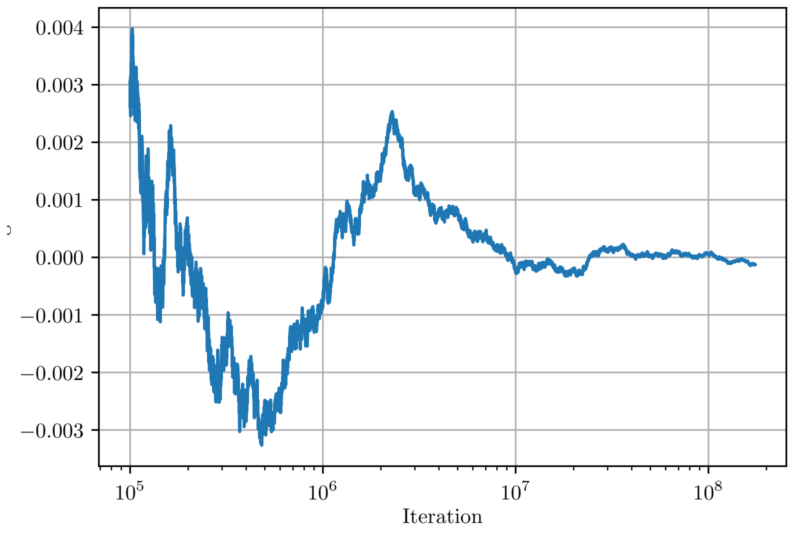


Figure 17: Zooming in the error percentage

References

[1] Emma Haruka Iwao. *Calculating 100 trillion digits of pi on google cloud — google cloud blog.* URL: `https://cloud.google.com/blog/products/compute/calculating-100-trillion-digits-of-pi-on-google-cloud`.

[2] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: *ACM Trans. Model. Comput. Simul.* 8.1 (Jan. 1998), pp. 3–30. ISSN: 1049-3301. DOI: `10.1145/272991.272995`. URL: `https://doi.org/10.1145/272991.272995`.