Monte Carlo Integration

Geimer Arno, Karst Philippe, Carvalho Bruno

University of Luxembourg

FSTC - BASI Mathematics

Under the supervision of Guendalina Palmirotta

Summer Term 2018

Abstract

This project is about numerical integration with an in-depth analysis of the Monte Carlo method. We start off by giving a motivation for the use of the Monte Carlo method followed by developing and explaining its mathematical background. Finally, we give examples and compare the Monte Carlo method to other deterministic methods for which we won't prove the necessary Theorems.

Contents

1 Introduction **3**
1.1 Motivation 3

2 Non-Deterministic approach **5**
2.1 Some words about integrals 5
2.2 Monte Carlo Integration 6

3 Application and examples **12**
3.1 Algorithms 14
3.2 Analysis of the function $\frac{\sin(x)}{x}$ 20
3.3 Analysis of the function $\frac{e^x}{x}$ 23
3.4 Analysis of the function $\sin\left(\frac{1}{x}\right)$ 25

4 Annex **28**
4.1 Code 28

1 Introduction

1.1 Motivation

It is well known that the mathematical constant π can be defined as the ratio of a circle's circumference to its diameter. However with this definition we still do not certainly know what the value of π is. Now our goal is to approximate the value with arbitrary precision. A naive method would be to put a string along a circle with diameter equal to 1 and then measure the length of the string, but with this method the result will probably be something like $\pi \approx 3.14$, if measured carefully. Since this method is not rigorous and the value is also not correct due to measuring error and the nonexistence of a perfect circle in the physical world, it is clear that we need a different way to approximate the value of π. Indeed, the Monte Carlo method (and also other methods) provides a simple and intuitive way of approximating π. From the definition of π it follows that the area bounded by a circle C is $\mathscr{A}_C := \pi r^2$, where r is the radius of C [1]. Setting $r = \frac{1}{2}$, we get $\pi = 4\mathscr{A}$. Furthermore we consider a 1×1 square in which we inscribe C as in Figure 1. Now we generate random points within that square and calculate the ratio of the amount of points that fall within the area bounded by C to the total amount of points. It makes intuitive sense that when the amount of points tend to infinity, the ratio approaches π, also one can support this claim by actually calculating π this way. One can see that this method is powerful in order to approximate different values, such as definite integrals, mathematical constants or limits numerically, but one may ask if it is possible to do better, perhaps by using a different method. And whether we are sure these methods give us the right value. We shall therefore investigate and prove the rigorousness of these methods in order to apply and compare them. But first we are going to treat some more examples where one can apply these methods.

- Monte Carlo integration: the main goal is to approximate integrals, especially in the case where the exact integral can't be calculated due to the impossibility of expressing the primitive [2] as a finite composition of elementary functions, i.e. constant functions, field operations $(+, -, \times, \div)$, powers, roots, exponential functions, logarithms, trigonometric functions and their inverses and hyperbolic functions.
- Finding the root of nonlinear algebraic equations [3] through Monte Carlo Method is often useful since in general it is very hard or even impossible to find such roots exactly.

[1] The proof and can be done with the following prerequisites: the Pythagorean Theorem, the trigonometric functions sine and cosine and basic knowledge on integration.

[2] Recall: $f : \mathbb{R} \to \mathbb{R}$ integrable, $a, b \in \mathbb{R}$, then $\int_a^b f(x) = F(b) - F(a)$ where F is the primitive of f.

[3] *Nonlinear algebraic equations*, which are also called polynomial equations, are defined by equating polynomials to zero.

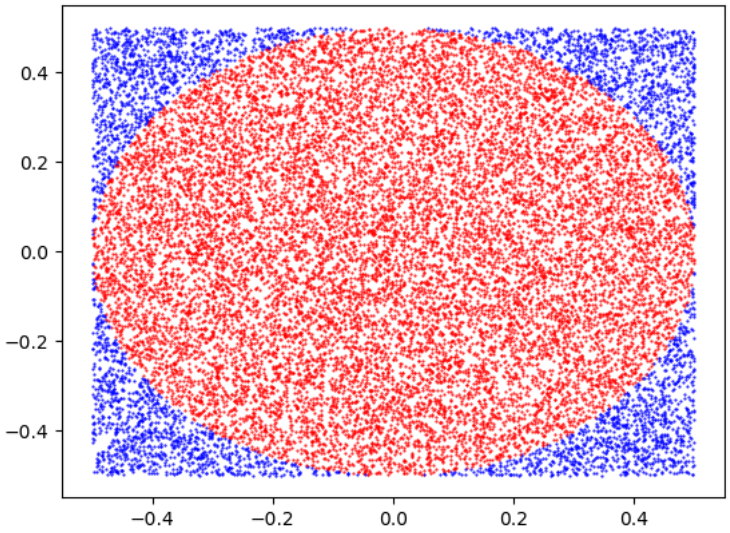

Figure 1: Illustration of the Monte Carlo method used to calculate π, red points are bounded by a circle of radius $\frac{1}{2}$ and blue points are outside the circle and within the unit square.

In this project, we are focusing on numerical integration of continuous functions $f : \mathbb{R} \to \mathbb{R}$, that is using either a stochastic (e.g. Monte Carlo Integration) or a deterministic approach (e.g. Trapezoidal Integration). In the following section we will elaborate on the non-deterministic approach.

2 Non-Deterministic approach

A non-deterministic approach or stochastic approach has some random properties [4] which may vary the outcome of an experiment. Even if the input stays the same, the output is determined by random variables and can thus not be predicted. An example would be a true random generator, in which it is impossible to truly predict the outcome even by knowing all initial conditions.

2.1 Some words about integrals

Definition 1. Let $[a,b] \subset \mathbb{R}$ and $\epsilon > 0$. We say that $f : [a,b] \to \mathbb{R}$ is *Riemann integrable* on $[a,b]$ if there exists $s \in \mathbb{R}$ and there exists a tagged partition $y_0, ..., y_m$ and $r_0, ..., r_{m-1}$ of $[a,b]$, such that for any tagged partition $x_0, ..., x_n$ and $t_0, ..., t_{n-1}$ which is a refinement of $y_0, ..., y_m$ and $r_0, ..., r_{m-1}$, such that

$$\left| \sum_{i=0}^{n-1} f(t_i)(x_{i+1} - x_i) - s \right| < \epsilon.$$

In this case s is the integral of f over $[a,b]$. Notatation: $s = \int_a^b f(x)dx$.

Proposition 1. *Let $[a,b] \subset \mathbb{R}$ and $f : [a,b] \to \mathbb{R}$ continuous, then f is integrable on $[a,b]$.* [5]

Let us now give a geometric, as well as an algebraic interpretation of the Riemann integral of a non-negative, real valued, continuous function on $[a,b] \subset \mathbb{R}$.

Geometrically, the integral of a continuous function $f : \mathbb{R} \to \mathbb{R}_{\geq 0}$ between two real points a and b can be viewed as the value of the area bounded by f, the lines $x = a$ and $x = b$ and the x-axis. In order to calculate this area, one approach is to slice it up into infinitely many subsections that tend to be infinitesimally small, and to add up their areas. One example for this approach is given by the definition of the Riemann integral, which slices up the function into rectangles as shown in Figure 2. For the algebraic interpretation we need the important theorem below.

Theorem 1 (Fundamental Theorem of Calculus)**.** [6] *Let f be a real-valued function on a closed interval $[a,b] \subset \mathbb{R}$ and F be a primitive of f in $[a,b]$, that is $f = F'$ on $[a,b]$. Then*

$$\int_a^b f(x)\,dx = F(b) - F(a).$$

Algebraically, the Fundamental Theorem of Calculus links the notion of a derivative to the integral. This gives an easy way for calculating a lot of different

[4] Being random is characterized by the lack of patterns or predictability.

[5] The proof will be omitted since it is an easy result from Analysis 2 and it is not relevant for this project.

[6] This proof will be omitted, since it is not relevant for this project.

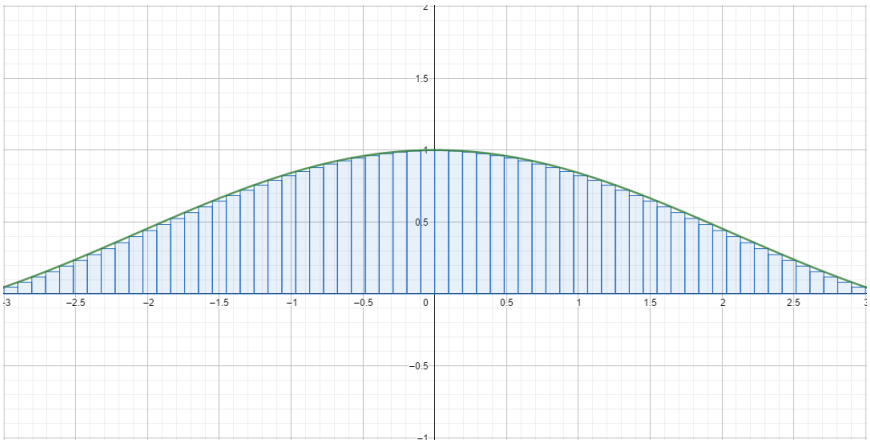

Figure 2: Approximation of the Riemann Integral.

integrals. However, some integrals can not be calculated that way since it is not possible to calculate their primitive even though they exist. Examples of these include: $x \mapsto x^x$, $x \mapsto e^{-x^2}$, $x \mapsto \sin(x^2)$, $x \mapsto \frac{\sin(x)}{x}$, $x \in \mathbb{R}$. Therefore, if one needs to integrate these functions, one usually approximates them. Some properties of Riemann-integrable functions are closure by addition as well as multiplication by a scalar. These properties will often be used in what follows.

2.2 Monte Carlo Integration

Monte Carlo integration is a technique for numerical integration using random numbers. It is a particular Monte Carlo method that numerically computes a definite integral. The key idea is that if one takes a large sample of some population, then the sample mean is close to the population mean. This seems to be somewhat obvious however in order to use this fact we need to formally state a theorem incorporating this idea as well as prove it. Indeed the theorem in question is called the "*Law Of Large Numbers*" (LLN). First, we will state and prove a Lemma and a Corollary that are used in the proof of the "*Weak Law of Large Numbers*" (WLLN), then we state and prove the WLLN and finally we will state and pove the "*Strong Law of Large Numbers*" (SLLN).

Definition 2. Let Ω be a sample space of a probability triple $(\Omega, \mathscr{A}, \mathbb{P})$ [7] and E a mesurable space (in this project $E = \mathbb{R}$). Then we call a function $X : \Omega \to E$ a *random variable*.

[7] $(\Omega, \mathscr{A}, \mathbb{P})$ means

- A sample space, Ω, which is the set of all possible outcomes,
- A set of events $\mathscr{A}$, where each event is a set containing zero or more outcomes,
- The assignment of probabilities to the events; i.e. a function $\mathbb{P}$ from events to probabilities.

Lemma 1 (Markov's inequality)**.** *Let X be a random variable and $h : [-\infty; +\infty] \to [0; +\infty]$ an increasing function. Then we have,*

$$P\{X \geq a\} \leq \frac{\mathbb{E}[h(X)]}{h(a)}, \ \forall a \in \mathbb{R} \text{ such that } h(a) > 0.$$

Proof.

$$\begin{aligned}
\mathbb{E}[h(X)] &= \sum_{x \in X(\Omega)} h(x)\mathbb{P}(X = x), \\
&\geq \mathbb{E}[h(X)\mathbf{1}_{\{X \geq a\}}]\mathbb{E}[h(X)\mathbf{1}_{\{X \geq a\}}] \\
&= \sum_{x \in \{X \geq a\}} h(x)\mathbb{P}(X = x) \\
&\geq h(a)\mathbb{E}[\mathbf{1}_{\{X \geq a\}}] \\
&= h(a) \sum_{x \in \{X \geq a\}} \mathbb{P}(X = x) \\
&= h(a)\mathbb{P}(X \geq a).
\end{aligned}$$

□

Corollary 1 (Chebychev-Markov inequality)**.** *Let $h(x) = (x \vee 0)^2 := max(x, 0)$. Then,*

$$\mathbb{P}\{|X| \geq a\} \leq \frac{\mathbb{E}|X|^2}{a^2}, \qquad \forall a > 0.$$

We will now define two different notions of convergence of random variables. This is crucial for understanding the difference between the WLLN and the SLLN.

Definition 3. Let $(X_n)_{n \geq 1}$ be a sequence of random variables on some probability space $(\Omega, \mathscr{A}, \mathbb{P})$.

- *convergence in probability*

$$\lim_{n \to \infty} \mathbb{P}\{|X_n - X| \geq \epsilon\} = 0, \quad \forall \epsilon > 0.$$

  "$X_n \to X$ in probability"

- *convergence almost surely*

$$\mathbb{P}\{\omega \in \Omega | \lim_{n \to \infty} X_n(\omega) = X(\omega)\} = 1.$$

  "$X_n \to X$ almost surely"

Theorem 2 (Weak law of large numbers)**.** *Let $X_1, ..., X_n$ be pairwise independent real valued random variables over $(\Omega, \mathscr{A}, \mathbb{P})$, $\mathbb{E}[X_i^2] < \infty$ and $\mathbb{E}[X_i] = \mathbb{E}[X_j] = \mu \ \forall i, j$. Then,*

$$\forall \epsilon > 0, \ \lim_{n\to\infty} \mathbb{P}\left\{\left|\frac{1}{n}\sum_{i=1}^{n}(X_i) - \mu\right| \geq \epsilon\right\} = \lim_{n\to\infty} \mathbb{P}\left\{\left|\frac{1}{n}S_n - \mu\right| \geq \epsilon\right\} = 0,$$

where $S_n := \sum_{i=1}^{n}(X_i)$.

Proof. Without restrictions we can suppose that $\forall i, \ E[X_i] = 0$. Using the Chebychev-Markov inequality, we obtain:

$$\begin{aligned}
\mathbb{P}\left\{\left|\frac{S_n}{n}\right| \geq \epsilon\right\} &= \mathbb{P}\left\{|S_n| \geq n\epsilon\right\} \\
&\leq \frac{\mathbb{E}|S_n|^2}{n^2\epsilon^2} \\
&= \frac{\mathbb{E}[S_n - \mathbb{E}[|S_n|]]^2}{n^2\epsilon^2} \\
&= \frac{var(S_n)}{n^2\epsilon^2} \\
&= \frac{1}{n^2\epsilon^2}\sum_{i=1}^{n} var(X_i) \to 0, \text{ for } n \to \infty.
\end{aligned}$$

□

We now have convergence in probability, but this is not enough in order to implement Monte Carlo Integration. Indeed let us treat an example.

Example 1. Let $(X_i)_{i\in\mathbb{N}} \in \{0,1\}$ be i.i.d. (independent and identically distributed) with $\mathbb{P}\{X_i = 0\} = \mathbb{P}\{X_i = 1\} = 0.5$. By applying the WLLN, we have that for any $\epsilon > 0$ there exists some $n \in \mathbb{N}$ such that $|\frac{S_n}{n} - 0.5| < \epsilon$. However for all $M > 0$ there may be $m > n$ such that $|\frac{S_m}{m} - 0.5| > M$.

This can't happen in almost sure convergence, which leads us to the SLLN. First we state two Lemmas that will be used in the proof of the Theorem.

Lemma 2 (Borel/Cantelli)**.** *Let* $A_1, A_2, ... \in \mathscr{A}$ *be a sequence of events and let*

$$A_\infty := \bigcap_n \bigcup_{k\geq n} A_k \hat{=} A_n \text{ infinitely often (i.o.)}$$

(1) If $\sum_k \mathbb{P}(A_k) < \infty$, *then* $\mathbb{P}(A_\infty) = 0$;
(2) If $\sum_k \mathbb{P}(A_k) = \infty$ *and* $A_1, A_2, \ldots$ *independant, then* $\mathbb{P}(A_\infty) = 1$.

Proof. (1)

$$\mathbb{P}(A_\infty) = \lim_{n\to\infty} \mathbb{P}\left(\bigcup_{k\geq n} A_k\right) \leq \lim_{n\to\infty} \sum_{k\geq n} \mathbb{P}(A_k) = 0.$$

(2) Firstly, we have:

$$\mathbb{P}(A_\infty^c) = \mathbb{P}\left(\bigcup_n \bigcap_{k\geq n} A_k^n\right) \leq \sum_n \mathbb{P}\left(\bigcap_{k\geq n} A_k^c\right).$$

For all n, we have:

$$\begin{aligned}
\mathbb{P}\left(\bigcap_{k\geq n} A_k^c\right) &= \lim_{m\to\infty} \mathbb{P}\left(\bigcap_{n\leq l\leq m} A_k^c\right) \\
&= \lim_{m\to\infty} \prod_{n\leq k\leq m} \mathbb{P}\left(A_k^c\right) \\
&= \lim_{m\to\infty} \prod_{n\leq k\leq m} \left(1-\mathbb{P}\left(A_k\right)\right) \\
&\leq \lim_{m\to\infty} \prod_{n\leq k\leq m} e^{-\mathbb{P}(A_k)} \\
&\leq \lim_{m\to\infty} e^{\sum_{k=n}^m \mathbb{P}(A_k)} = 0.
\end{aligned}$$

□

Lemma 3. *We have,*

$$X_n \to X \ \ a.s. \ \Leftrightarrow \forall \epsilon > 0, \mathbb{P}\left\{|X_n - X| \geq \epsilon \ \ i.o. \ \right\} = 0.$$

Proof. For $\{X_n \not\to X\} \equiv \{\omega : X_n(\omega) \not\to X(\omega)\}$ we can write

$$\{X_n \not\to X\} = \bigcup_{k\geq 1} \bigcap_{m\geq 1} \bigcup_{n\geq m} \left\{|X_n - X| \geq \frac{1}{k}\right\}.$$

Thus,

$$\begin{aligned}
\mathbb{P}\left\{|X_n - X| \not\to 0\right\} = 0 &\Leftrightarrow \mathbb{P}\left\{|X_n - X| \to 0\right\} = 1 \\
&\Leftrightarrow \mathbb{P}\left(\bigcap_{m\geq 1} \bigcup_{n\geq m} \left\{|X_n - X| \geq \frac{1}{k}\right\}\right) = 0 \\
&\Leftrightarrow \mathbb{P}\left\{|X_n - X| \geq \frac{1}{k} \text{ i.s. } \right\}.
\end{aligned}$$

□

Theorem 3 (Strong law of large numbers)**.** *Let $(X_i)_{i\geq 1}$ be a sequence of pairwise independant random variables such that $\mathbb{E}[X_i^2] \leq c < \infty$, $\forall i \in \mathbb{N}$. We set $S_n = X_1 + ... + X_n$. Then, if $\mathbb{E}[X_i] = \mathbb{E}[X_1]$ for all i,*

$$\frac{S_n}{n} \to \mathbb{E}[X_1] \ \ a.s.$$

Proof. Without loss of generality, we can assume that $\mathbb{E}[X_i] = 0$ for all i. Then, we have by hypothesis

$$\mathbb{E}[S_n^2] = \sum_{i=1}^{n} \mathbb{E}[X_i^2] + \sum_{\substack{i,j=1 \\ i \neq j}}^{n} \mathbb{E}[X_i X_j] \leq nc,$$

where $\mathbb{E}[X_i X_j] = 0$. By the Chebychev inequality, we get

$$\mathbb{P}\left\{|S_n| \geq n\epsilon\right\} \leq \frac{nc}{n^2\epsilon^2} = \frac{c}{n\epsilon^2},$$

and thus,

$$\sum_n \mathbb{P}\left\{|S_{n^2}| \geq n^2\epsilon\right\} \leq \sum_n \frac{c}{n^2\epsilon^s} < \infty.$$

By the Borel-Cantelli lemma, we have

$$\mathbb{P}\left\{|S_{n^2}| \geq n^2\epsilon \text{ i.o.}\right\} = 0,$$

and thus with the Lemma 3

$$\frac{S_{n^2}}{n^2} \to 0 \text{ a.s., when } n \to \infty.$$

Now it suffices to estimate the deviations of S_k for $n^2 < k < (n+1)^2$ in relation to S_{n^2}.
We define

$$D_n := \max_{n^2 \leq k < (n+1)^2} \mid S_k - S_{n^2} \mid, \;\; n \geq 1.$$

Thus,

$$\begin{aligned}
\mathbb{E}[D_n^2] &\leq \sum_{k=n^2}^{(n+1)^2-1} \mathbb{E}\left[\mid S_k - S_{n^2} \mid^2\right] \\
&= \sum_{k=n^2}^{(n+1)^2-1} \mathbb{E}\left[\left|\sum_{i=n^2+1}^{k} X_i\right|^2\right] \\
&= \sum_{k=n^2}^{(n+1)^2-1} \sum_{i=n^2+1}^{k} \mathbb{E}[X_i^2].
\end{aligned}$$

We obtain

$$\mathbb{P}\left\{D_n \geq n^2\epsilon\right\} \leq \frac{4n^2c}{n^4\epsilon^2} = \frac{4c}{\epsilon^2}\frac{1}{n^2}.$$

Thus, as above

$$\frac{D_n}{n^2} \to 0 \text{ a.s. , when } n \to \infty.$$

Finally, we have for $n^2 \leq k < (n+1)^2$:

$$\frac{\mid S_k \mid}{k} \leq \frac{\mid S_{n^2} \mid + D_n}{n^2} \to 0 \text{ a.s., when } n \to \infty.$$

□

The strong law of large numbers differs from the weak law in terms of convergence. The strong law suggests that we have almost sure convergence, while the weak law only gives us convergence in probability. We need the almost sure convergence to be sure that the Monte Carlo method works. Loosely speaking, the strong law tells us that, except for sets with measure zero, a sequence of random variables surely converges to its limit.
From these results, in particular from the SLLN, we obtain the following Corollary:

Corollary 2. *Let $(X_i)_{i\geq 1}$ a sequence of identically and independently distributed (i.i.d.) random variables and $f : \mathbb{R} \to \mathbb{R}$ be an integrable function such that $\mathbb{E}|f(X_i)| < \infty$. Then $(f(X_i))_{i\geq 1}$ is also a family of i.i.d. random variables, and by the SLLN:*

$$\frac{1}{n}\sum_{i=1}^{n} f(X_i) \to \mathbb{E}[f(X_i)] \text{ when } n \to \infty.$$

In particular, if X_i are uniformly distributed on $[a,b] \subset \mathbb{R}$ and $f : [a,b] \to \mathbb{R}$ is a continuous function. Then,

$$\frac{1}{n}\sum_{i=1}^{n} f(X_i) \to \frac{1}{b-a}\int_a^b f(x)\,dx \quad \text{almost surely.} \quad (2)$$

This Corollary describes what is known as Monte Carlo integration.
One may remark that the Corollary follows from the SLLN and not from the WLLN. This follows from the definition of an integrable function from measure theory, which we will not treat further here. It says that if a function is measurable, it is also measurable by adding or taking away sets of measure zero to its domain, with the its measure staying invariant. An example for a set of measure zero is a countable subset of $\mathbb{R}$. Almost sure convergence indeed gives us convergence up to sets of measure zero. That is $f(X_i) \not\to f(x)$ for countably many $x \in [a,b]$, i.e. a set of measure zero. This means

$$\lim_{n\to\infty} \frac{b-a}{n}\sum_{i=1}^{n} f(X_i) = \int_a^b f(x)\,dx.$$

3 Application and examples

In this section, we will study concrete examples of integrals in depth. In particular, we are comparing different methods of numerical integration and their respective efficiency. In the first example function, $\frac{\sin(x)}{x}$, we will provide explanations of the code and an elaborate analysis of the procedure. The other examples will be summarized in a table. The methods of numerical integration are:

- *MC_funct*: This method of integration is the standard Monte Carlo integration algorithm derived from formula (2). The method consists of generating random points on the interval $[a, b]$ and calculating their image by f. One takes the mean of all these image points and multiplies it by $(b - a)$ to get the integral approximation.
- *MC_points_1*: This method of integration is the same as the one used in the approximation of π in the motivation. It consists of putting random points in a rectangle around the function, as shown in Figure 3, and calculating the ratio of the total number of points and the number of points bounded by the graph of the function and the x-axis.

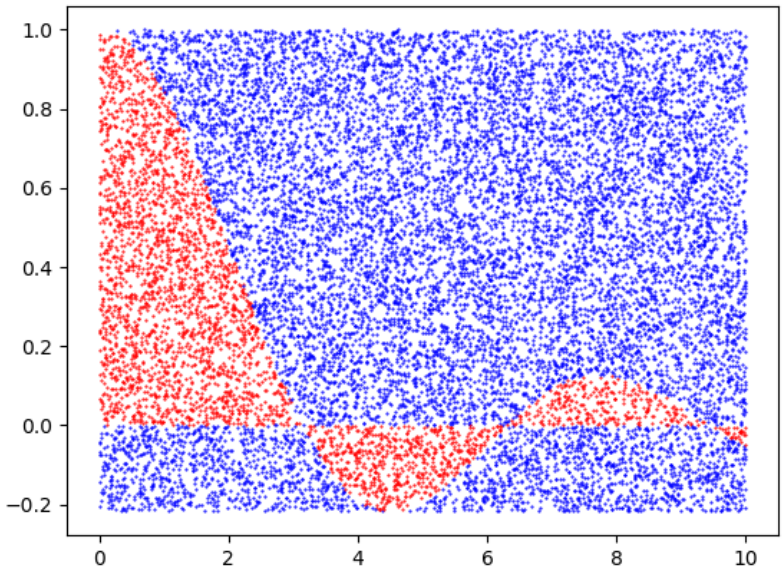

Figure 3: Set of points in $\mathbb{R}^2$, red if a point is bounded by the x-axis and the graph of $x \mapsto \sin(x)/x$ and blue otherwise.

- *MC_points_2*: This method of integration almost coincides with the method above. Indeed, we consider several smaller rectangles for the function instead of one rectangle, as shown in the Figure 4:

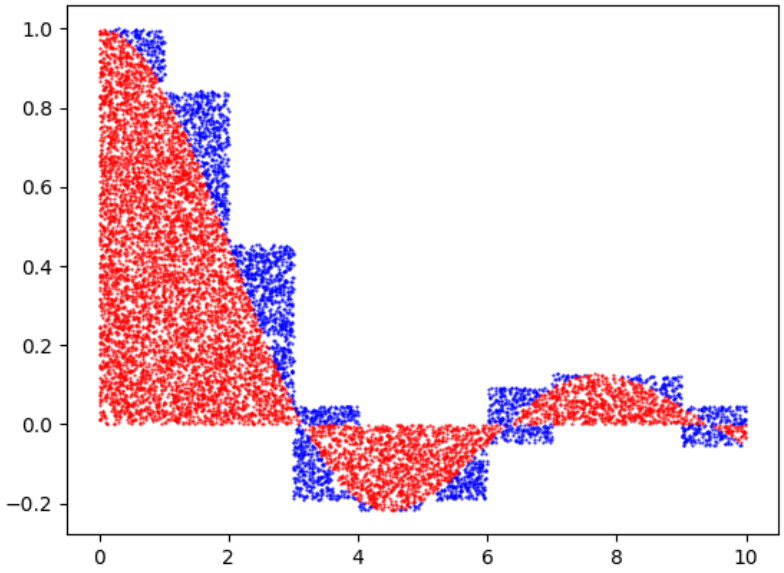

Figure 4: Same principle as for Figure 3 restricted to smaller rectangles covering the area bounded by the graph of $x \mapsto \sin(x)/x$ and the x-axis.

- *Riemann integration*: This method of integration is the Riemann integration as described in the geometric interpretation of the integral in subsection 2.1.
- *Trapezoidal integration*: This method of integration is essentially the same as Riemann integration, but one takes trapezes instead of rectangles, as shown in Figure 5. [8]

In a first step, we will explain the algorithms used for computation, since we are going to refer to different parts of the algorithm when treating the examples.

[8] We admit the that this method of integration is valid without stating the necessary theorems.

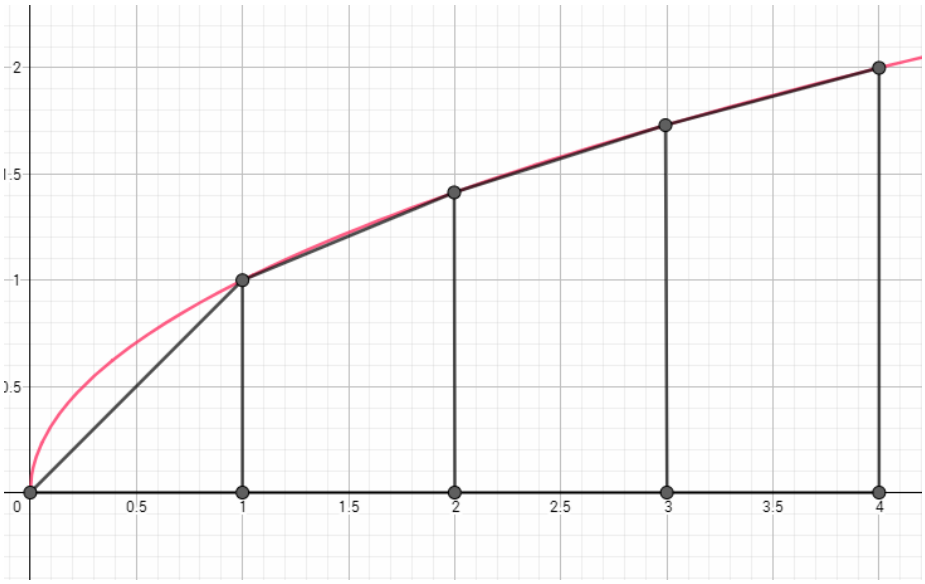

Figure 5: Trapezoidal integration of $\sqrt{x}$

3.1 Algorithms

All of the algorithms were written in the Python language. We used the Canopy environment, which provides us with necessary and practical packages we can use in the algorithms.
Below are all the auxiliary functions that are needed in the integration algorithms. To integrate a certain function, one needs to define *funct()* as desired. One must pay attention to the limit points of the function, especially when one divides by zero, as Python will show an error in the latter case. As in the example $\frac{\sin(x)}{x}$, one needs to assign a value to these limit points if one wants to include them in the integration domain.

```
import math
import random
import matplotlib.pyplot as plt
import multiprocessing as mult
from decimal import *

#the function we are integrating
def funct(x):
    if x == 0:
        return 1
    else:
        return (math.sin(x)/x)
```

```
#the maximum of two numbers
def maxi(x,y):
    if x > y :
        return x
    else:
        return y

#the minimum of two numbers
def mini(x,y):
    if x > y :
        return y
    else:
        return x


#the maximum of a function between a and b, up to a certain
#precision
def sup(a, b):
    l = 10000 #precision
    q = float(b)-float(a)
    k = funct(a)
    for i in range(l+1):
        k = maxi(funct(a+((q/l)*i)) , k)
    return k

#the minimum of a function between a and b, up to a certain
#precision
def inf(a, b):
    l = 10000 #precision
    q = float(b)-float(a)
    k = funct(a)
    for i in range(l+1):
        k = mini(funct(a+((q/l)*i)) , k)
    return k

#needed for the end result when calculating on all cores
def addition(list):
    f = 0
    for i in range(len(list)):
        f = f + list[i]
    return f
```

The following code is used to calculate the *MC_integral_funct*:
The *MC_integral_funct* algorithm puts x random points on the interval be-

tween u and v, giving $(X_1, ..., X_x)$. It then computes the images by *funct()* of these points and sums them up. This is the $\sum f(X_i)$ in the Monte Carlo integral formula (2). It then divides this sum by x and multiplies it by $(v - u)$. This gives the Monte Carlo integral between u and v of *funct()*, which is stored in a text file. The second part of the algorithm takes the number of threads of the processor (*cores*) and splits the integral domain into this number of sub-intervals. Then it lets each thread calculate the Monte Carlo integral on its respective sub-interval. These results are stored in text files created by the *MC_integral_funct* algorithm. The *result()* function sums the values in those text files, giving the final Monte Carlo integral between a and b.

Remark 1. The number x is the number of points on each sub-interval, which means that the total number of points depends on the number of threads of the processor and is thus a multiple of x.

```
#puts random points in the desired interval and approximates
#the integral of the function on that interval
def MC_integral_funct(x , u , v, j):
    k = 0
    text = open("document_MC_funct" + str(j) + ".txt", "w")
    for i in range(x):
        a = random.uniform(float(u), float(v))
        k = k + funct(a)
    q = ((k/x)*(v-u))
    text.write(str(q))


#divides the interval on the number of cores and lets each core
#write the integral in a text document
if __name__ == '__main__':
    x = int(input("how many numbers?\n"))
    u = float(input("lower bound?\n"))
    v = float(input("upper bound?\n"))
    cores = mult.cpu_count()
    s = (v-u)/cores
    res = 0

    for j in range(cores):
        m = u+(j*s)
        n = u+(j+1)*s
        p = mult.Process(target = MC_integral_funct,
args = (x , m , n, j))
        p.start()

#adds together the values from the text documents to get
#the end result
```

```
def result():
    cores = mult.cpu_count()
    res = 0
    for j in range(cores):
        text = open("document_MC_funct" + str(j) + ".txt", "r")
        res = res + float(text.readline())

    print(res)
```

The following code is used for the *MC_points_1* and the *MC_points_2* integral. The only difference is that in the first case, one only takes 1 interval, while in the second case, one takes as many intervals as one wants. The algorithm now calculates the rectangle with the largest area and generates x random points in that rectangle. The number of points generated in the other rectangles is proportional to the number of points in the biggest rectangle, so that the ratio of points per rectangle to the area of the rectangle stays the same for all the rectangles. The algorithm now counts the points between the x-axis and the function. It calculates the ratio of these points to x and multiplies this number by the area of the corresponding rectangle. This gives the *MC_points_1* integral of *funct()* on the interval $[u, v]$. In the case of the *MC_points_2* integral, the algorithm now sums all of these values to get the integral from a to b of *funct()*.

```
def integral_points():
    t = int(input("How many intervals?\n"))
    x = int(input("Maximal number of points per interval?\n"))
    u = int(input("Left bound?\n"))
    v = int(input("Right bound?\n"))
    s = ((v-u)/t)
    c = 0
    k = 0
    for j in range(t):
        k = maxi(k , ((maxi(sup(u+(j*s), u+(j+1)*s) , 0) -
            mini((inf(u+j*s , u+(j+1)*s)) , 0))))
    for j in range(t):
        g = 1  #>0
        q = 0  #>0 and below the graph
        h = 1  #<0
        r = 0  #<0 and over the graph
        m = maxi(0 , sup(u+(j*s), u+(j+1)*s))
        n = mini(0 , inf( u+(j*s) , u+(j+1)*s ))
        d = int((((m-n))/k)*x)
        for i in range(d):
            a = random.uniform(float(u+(j*s)), float(u+((j+1)*s)))
```

```
            b = random.uniform(float(n), float(m))
            for j in range(10000):
                plt.plot(j/1000, funct(j/1000), 'go',
                markersize = 0.5)
            if b >= 0:
                g = g+1
                if b <= funct(a):
                    q = q+1
                    plt.plot(a, b, 'ro', markersize = 0.5)
                else:
                    plt.plot(a, b, 'bo', markersize = 0.5)
            else:
                h = h+1
                if b >= funct(a):
                    r = r+1
                    plt.plot(a, b, 'ro', markersize = 0.5)
                else:
                    plt.plot(a, b, 'bo', markersize = 0.5)
        c = c + ((q/g)*(s)*(m)) + ((r/h)*(s)*(n))

    return c
```

The following code is used for the *Riemann integration*. It splits the interval into x sub-intervals. On each sub-interval $[u, v]$, it calculates the area of the rectangle whose height is the mean of *funct(u)* and *funct(v)* and whose width is $(v - u)$. It then sums all the areas of the x rectangles, to give the final integral approximation of the function.
The second and third part of the code distribute the integral on the processor threads, just as in the *MC_integral_funct* algorithm, and then sum up all of the results.

```
#calculates the Riemann integral on a certain interval [u,v]
#and writes the result on a document
def integral(x , u , v, j):
    k = 0
    a = (v-u)/x
    text = open("documentriemann" + str(j) + ".txt", "w")
    for i in range(x):
        k = k + a*((funct(u+i*a))+funct(u+(i+1)*a))*0.5
    text.write(str(k))

#divides the interval on the number of cores and lets each core
#write the calculated integral in a text document
if __name__ == '__main__':
```

```
    x = int(input("How many rectangles?\n"))
    u = float(input("Left bound?\n"))
    v = float(input("Right bound?\n"))
    cores = mult.cpu_count()
    s = (v-u)/cores
    res = 0

    for j in range(cores):
        m = u+(j*s)
        n = u+(j+1)*s
        p = mult.Process(target = integral, args = (x , m , n, j))
        p.start()

#adds together the values from the text documents to get the end
#result
def result():
    cores = mult.cpu_count()
    res = 0
    for j in range(cores):
        text = open("documentriemann" + str(j) + ".txt", "r")
        res = res + float(text.readline())

    print(res)
```

The following code is used for the *Trapezoidal Integration*. It is essentially the same as the algorithm of the *Riemann Integration*, the only difference being that it does not operate with rectangles, but with trapezes.

```
#the trapezoidal integral on a certain interval
def integrale(x, u ,v):
    k = 0
    m = (v-u)/x #sub-interval length
    for i in range(x):
        a = funct(u+i*m)
        b = funct(u+(i+1)*m)
        z = mini(a,b)*m #the lower rectangle part of the trapeze
        y = (maxi(a,b)-mini(a,b))*m*0.5 #the triangle part
        k = k+z+y
    print(k)
```

3.2 Analysis of the function $\frac{\sin(x)}{x}$

In the following, we are computing approximations of

$$\int_0^{100} \frac{\sin(x)}{x} dx = Si(100)^{9} \approx 1.56223^{10}$$

- The Monte Carlo algorithms will use 1600, 8000 and 16000 points (the inputs are 100, 500 and 1000 but the processor has 16 threads).
- In the *MC_points_2* algorithm, we split $[0, 100]$ into 100 parts.
- The *Riemann* and *Trapezoidal* algorithms each split $[0, 100]$ into 100, 500 and 1000 parts.

Computation time is not relevant, since it stays under $0.1s$ for each algorithm. For the Monte Carlo methods, we will consider the mean value of 5 computations to get a better approximation.

Method	Nb. of points or rectangles	result1	result2	result3	result4	result5
MC_funct	1600	1.21995	1.24420	1.56836	1.42908	1.52449
	8000	1.58909	1.48418	1.61442	1.57334	1.53020
	16000	1.58955	1.50829	1.42579	1.54710	1.63362
MC_points_1	1600	0.99099	1.53288	2.24349	2.15251	1.96304
	8000	1.58563	1.70513	1.53319	1.34555	1.83188
	16000	1.93041	1.67292	1.42071	1.31917	1.72065
MC_points_2	1600	1.44677	1.37224	1.32672	1.47627	1.50025
	8000	1.63509	1.65950	1.55798	1.46689	1.64606
	16000	1.63099	1.57185	1.52758	1.59463	1.47733
Riemann integral	100	1.56223				
	500	1.56223				
	1000	1.56223				
Trapezoidal integral	100	1.56223				
	500	1.56223				
	1000	1.56223				

TABLE 1: green digits coincide with the expected value

We will now compare the corresponding error terms for each algorithm. Recall that the expected value is ≈ 1.56223.

[9] $Si(t) := \int_0^t \frac{\sin(x)}{x} dx$.
[10] Computation by Wolfram Alpha.

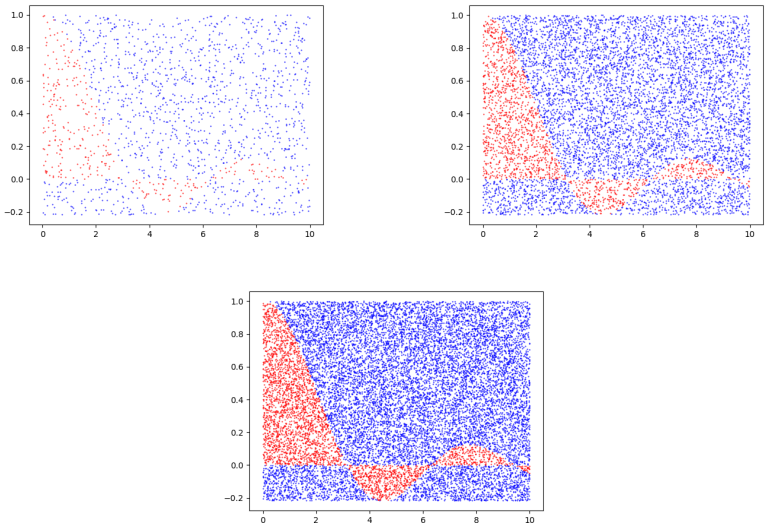

Figure 6: An example of the MC_points_1 algorithm from the table

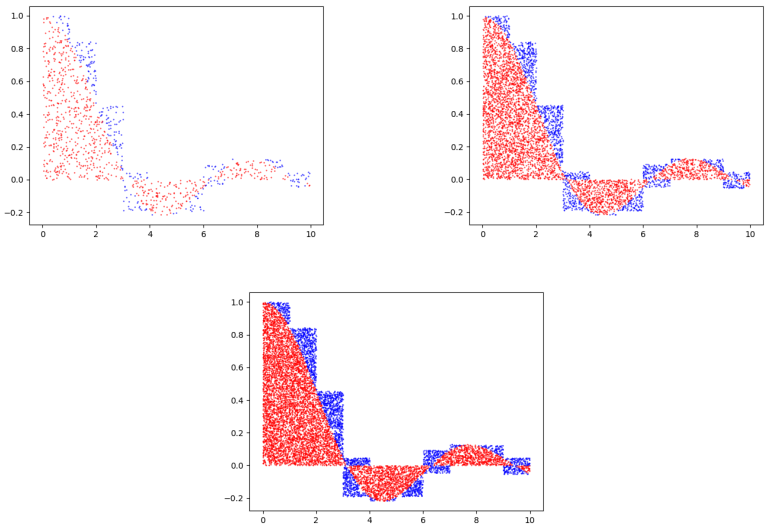

Figure 7: An example of the MC_points_2 algorithm from the table

Method	Nb. of points or rectangles	mean result	error term
MC_funct	1600	1.39721	0.16501
	8000	1.55824	0.00398
	16000	1.54087	0.02136
MC_points_1	1600	1.77658	0.21435
	8000	1.60027	0.03805
	16000	1.61277	0.05054
MC_points_2	1600	1.42445	0.13778
	8000	1.59314	0.03087
	16000	1.56047	0.00754
Riemann integral	100	1.56223	<0.00001
	500	1.56223	<0.00001
	1000	1.56223	<0.00001
Trapezoidal integral	100	1.56223	<0.00001
	500	1.56223	<0.00001
	1000	1.56223	<0.00001

TABLE 2: the greatest error is red, the smallest green

This shows that, as expected, the error term becomes smaller as the algorithm uses more points. The following graph shows the error term:

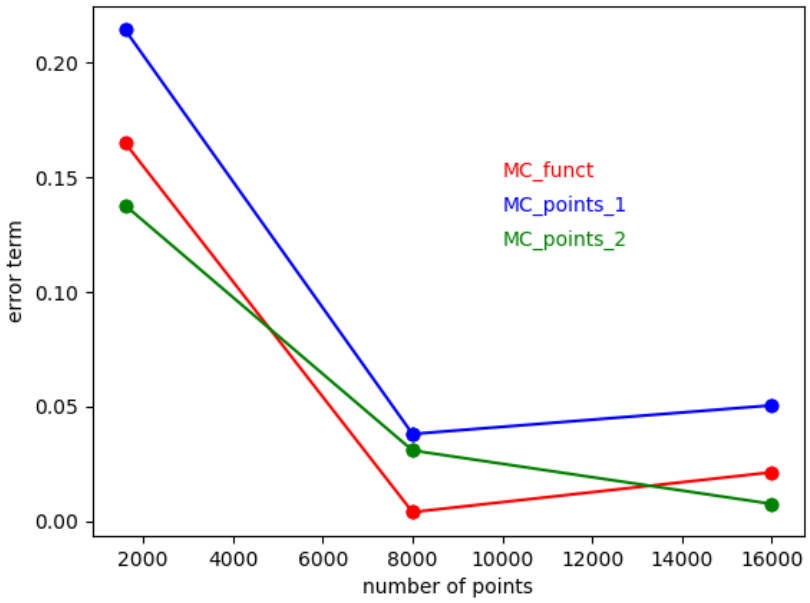

Figure 8: Visual representation of the evolution of the error term of the three different non-deterministic methods as the number of random points (i.e. random variables) increases.

3.3 Analysis of the function $\frac{e^x}{x}$

In the following, we will analyze the approximate value of

$$\int_1^2 \frac{e^x}{x} dx = \mathrm{Ei}(2)^{11} - \mathrm{Ei}(1) \approx 3.05912^{12}$$

which was given to us as an example by our supervisor, Mrs Guendalina Palmirotta. The number of points and rectangles stays the same as in **Section 3.2**.

[11] $\mathrm{Ei}(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t} dt$
[12] Computation by Wolfram Alpha

Method	Nb. of points or rectangles	result1	result2	result3	result4	result5
MC_funct	1600	3.05900	3.05891	3.05893	3.05974	3.05923
	8000	3.05904	3.05902	3.05950	3.05935	3.05887
	16000	3.05899	3.05909	3.05924	3.05942	3.05919
MC_points_1	1600	3.18242	3.18242	3.07268	3.10926	3.03609
	8000	3.24469	3.03822	3.08984	3.06771	2.97185
	16000	3.03387	2.96051	2.98509	3.02649	3.08185
MC_points_2	1600	2.94838	2.94612	2.94612	2.94615	2.94278
	8000	3.03733	3.03479	3.03641	3.03595	3.03664
	16000	3.04581	3.04857	3.04765	3.04799	3.04731
Riemann integral	100	3.05912				
	500	3.05912				
	1000	3.05912				
Trapezoidal integral	100	3.05912				
	500	3.05912				
	1000	3.05912				

TABLE 3: green digits coincide with the expected value

We will now compare the corresponding error terms for each algorithm. Reminder: the expected value is 3.05912.

Method	No of points or rectangles	mean result	error term
MC_funct	1600	3.05916	0.00004
	8000	3.05916	0.00004
	16000	3.05919	0.00007
MC_points_1	1600	3.11657	0.05745
	8000	3.08246	0.0233
	16000	3.01756	0.04156
MC_points_2	1600	2.94591	0.11321
	8000	3.03622	0.02290
	16000	3.04747	0.01165
Riemann integral	100	3.05912	<0.00001
	500	3.05912	<0.00001
	1000	3.05912	<0.00001
Trapezoidal integral	100	3.05912	<0.00001
	500	3.05912	<0.00001
	1000	3.05912	<0.00001

TABLE 4: the greatest error is red, the smallest green

3.4 Analysis of the function $\sin\left(\frac{1}{x}\right)$

By the above examples, the Riemann and Trapezoidal integration techniques are more accurate than the Monte Carlo methods. Therefore, in the following, we will analyze the approximate value of

$$\int_0^{10^{-6}} \sin\left(\frac{1}{x}\right) dx \approx 9.36751 \times 10^{-13} \qquad [13]$$

which we think is an example of a function for which the Monte Carlo integral is more accurate than Riemann or Trapezoidal integration. The number of points and rectangles stays the same as in **Section 3.2**.

Method	Nb. of points or rectangles	result1	result2	result3	result4	result5
MC_funct	1600	-4.38296×10^{-8}	-3.31255×10^{-8}	-2.05145×10^{-8}	-3.83488×10^{-8}	5.14855×10^{-8}
	8000	8.95669×10^{-9}	-1.18392×10^{-8}	2.58952×10^{-9}	-9.63238×10^{-9}	-3.05111×10^{-9}
	16000	-3.55284×10^{-9}	8.59746×10^{-10}	6.77858×10^{-9}	-1.32190×10^{-9}	1.953479×10^{-9}
MC_points_1	1600	-1.11801×10^{-7}	-1.87524×10^{-7}	-1.15741×10^{-7}	5.88235×10^{-8}	5.95238×10^{-8}
	8000	-4.41487×10^{-8}	-2.05619×10^{-8}	-4.87307×10^{-8}	9.61906×10^{-9}	-3.98405×10^{-9}
	16000	7.59556×10^{-9}	-1.22986×10^{-8}	-4.46207×10^{-8}	3.30011×10^{-8}	-1.25901×10^{-8}
MC_points_2	1600	1.44557×10^{-8}	-1.00307×10^{-8}	-1.15709×10^{-8}	-8.08912×10^{-9}	-2.145355×10^{-8}
	8000	1.00098×10^{-9}	8.62734×10^{-9}	4.45880×10^{-9}	1.29751×10^{-8}	1.12534×10^{-8}
	16000	-1.36737×10^{-9}	-3.76345×10^{-9}	-7.38404×10^{-9}	4.46885×10^{-9}	-9.18610×10^{-10}
Riemann integral	100	-1.18404×10^{-8}				
	500	-7.41097×10^{-9}				
	1000	-1.66418×10^{-9}				
Trapezoidal integral	100	-1.18404×10^{-8}				
	500	-7.41097×10^{-9}				
	1000	-1.64418×10^{-9}				

TABLE 5: green digits coincide with the expected value

We will now compare the corresponding error terms for each algorithm. Reminder: the expected value is 9.36751×10^{-13}.

[13] Computation by Wolfram Alpha

Method	No of points or rectangles	mean result	error term
MC_funct	1600	-1.68666×10^{-8}	1.68656×10^{-8}
	8000	-2.59542×10^{-9}	2.59446×10^{-9}
	16000	9.43413×10^{-10}	9.424762×10^{-10}
MC_points_1	1600	-5.93437×10^{-8}	5.93427×10^{-8}
	8000	-2.15613×10^{-8}	2.15603×10^{-8}
	16000	-2.38222×10^{-9}	2.38128×10^{-9}
MC_points_2	1600	7.33771×10^{-9}	7.33677×10^{-9}
	8000	7.66312×10^{-9}	7.66218×10^{-9}
	16000	-1.79292×10^{-9}	1.79198×10^{-9}
Riemann integral	100	-1.18404×10^{-8}	1.18395×10^{-8}
	500	-7.41097×10^{-9}	7.41003×10^{-9}
	1000	-1.66418×10^{-9}	1.66324×10^{-9}
Trapezal integral	100	-18404×10^{-8}	1.18395×10^{-8}
	500	-7.41097×10^{-9}	7.41003×10^{-9}
	1000	-1.64418×10^{-9}	1.64324×10^{-9}

TABLE 6: the greatest error is red, the smallest green

Conclusion

As a conclusion, we can say that numerical integration is often useful to easily approximate non elementary integrals [14]. The first two examples suggest that the deterministic methods (Riemann and Trapezoidal integration) give better approximations than the Monte Carlo method. However for $\sin\left(\frac{1}{x}\right)$ the MC_funct algorithm, which is based on a Monte Carlo method, gives a better approximation on an interval $[0,\epsilon]$, for ϵ small. Due to this observation, one can question whether for functions behaving wildly the Monte Carlo methods outperforms the Riemann as well as the Trapezoidal integration methods. More precisely, for a function $f: I \to J$ such that there exists $v \in \bar{I}$ such that for all $\epsilon > 0$, $f([v-\epsilon, v+\epsilon])$ is dense in J, I and J intervals.
Monte Carlo integration is a problem which is not hard to imagine and to visualize, but one needs some theorems that are not easy to formulate in a mathematical way, as well as to prove them. The timing of the project was very fortunate, since we worked on it while actually doing the theory in our probability and statistics course.
Overall, we appreciated the project, as it is oftentimes straightforward, especially the programming part, while still being challenging. There are some open questions left, namely for which functions Monte Carlo Integration works better than Riemann (we only found one), as well as if there is a way to further optimize the algorithms used in the project.

[14] A non elementary integral is an integral of a function which has a non elementary primitive.

References

[1] https://en.wikipedia.org/wiki/Integral 26/05/2018

[2] https://en.wikipedia.org/wiki/Riemann_integral 26/05/2018

[3] https://en.wikipedia.org/wiki/Fundamental_theorem_of_calculus 26/05/2018

[4] https://en.wikipedia.org/wiki/Nonelementary_integral 26/05/2018

[5] https://en.wikipedia.org/wiki/Logarithmic_integral_function 26/05/2018

[6] https://en.wikipedia.org/wiki/Exponential_integral 26/05/2018

[7] http://www.businessdictionary.com/definition/deterministic-model.html 26/05/2018

[8] https://www.researchgate.net/post/What_is_the_difference_among_Deterministic_model_Stochastic_model_and_Hybrid_model 26/05/2018

[9] http://www.businessdictionary.com/definition/deterministic-model.html 26/05/2018

[10] Prof. Martin Olbrich, *Analysis 3/4 script*, 2017/2018, University of Luxembourg

[11] Prof. Anton Thalmaier, *Probability & Statistics Course for Bachelor students in the second year*, University of Luxembourg http://math.uni.lu/thalmaier/ 26/05/2018

4 Annex

4.1 Code

Code for the approximation of π:

```
def piapprox():
    import random
    import matplotlib.pyplot as plt
    x = int(input("How many numbers?\n"))
    y = int(input("Plot? 1=Yes, 0=No \n"))
    q = 0
    for i in range(x):
            a = random.uniform(float(-0.5), float(0.5))
            b = random.uniform(float(-0.5), float(0.5))
            if y == 1:
                if ((a*a)) + (b*b) <= 0.25:
                    q = q+1
                    plt.plot(a, b, 'ro', markersize = 0.5)
                else:
                    q = q
                    plt.plot(a, b, 'bo', markersize = 0.5)
            else:
                if (a)*(a) + (b)*(b) <= 0.25:
                    q = q+1
                else:
                    q = q
    return (4*q)/x
```

Code for the MC_funct:

```
# -*- coding: utf-8 -*-
import math
import random
import multiprocessing as mult
from decimal import *

getcontext().prec = 50

#the function we are integrating
def funct(x):
    if x == 0:
        return 0
    else:
```

```
        return (math.sin(1/x))

#needed for the end result
def addition(list):
    f = 0
    for i in range(len(list)):
        f = f + list[i]
    return f

#puts random points in the desired interval and approximates
#the integral of the function on that interval
def MC_integral_funct(x , u , v, j):
    k = 0
    text = open("document_MC_funct" + str(j) + ".txt", "w")
    for i in range(x):
        a = random.uniform(float(u), float(v))
        k = k + funct(a)
    q = ((k/x)*(v-u))
    text.write(str(q))


#divides the interval on the number of cores and lets each core
#write the integral in a text document
if __name__ == '__main__':
    x = int(input("how many numbers?\n"))
    u = float(input("lower bound?\n"))
    v = float(input("upper bound?\n"))
    cores = mult.cpu_count()
    s = (v-u)/cores
    res = 0

    for j in range(cores):
        m = u+(j*s)
        n = u+(j+1)*s
        p = mult.Process(target = MC_integral_funct,
args = (x , m , n, j))
        p.start()

#adds together the values from the text documents to
#get the end result
def result():
    cores = mult.cpu_count()
    res = 0
    for j in range(cores):
        text = open("document_MC_funct" + str(j) + ".txt", "r")
```

```
        res = res + float(text.readline())

    print(res)
```

Code for MC_points_1 and MC_points_2

```
import math
import random
import matplotlib.pyplot as plt


def funct(x):
    if x == 0:
        return 1
    else:
        return (math.sin(x)/x)

def maxi(x,y):
    if x > y :
        return x
    else:
        return y

def mini(x,y):
    if x > y :
        return y
    else:
        return x


def sup(a, b):
    l = 10000 #precision
    q = float(b)-float(a)
    k = funct(a)
    for i in range(l+1):
        k = maxi(funct(a+((q/l)*i)) , k)
    return k

def inf(a, b):
    l = 10000 #precision
    q = float(b)-float(a)
    k = funct(a)
    for i in range(l+1):
        k = mini(funct(a+((q/l)*i)) , k)
```

```
    return k



def integral_points():
    t = int(input("How many intervals?\n"))
    x = int(input("Maximal number of points per interval?\n"))
    u = int(input("Left bound?\n"))
    v = int(input("Right bound?\n"))
    s = ((v-u)/t)
    c = 0
    k = 0
    for j in range(t):
        k = maxi(k , ((maxi(sup(u+(j*s), u+(j+1)*s) , 0) -
mini((inf(u+j*s , u+(j+1)*s)) , 0))))
    for j in range(t):
        g = 1  #>0
        q = 0  #>0 and below the graph
        h = 1  #<0
        r = 0  #<0 and over the graph
        m = maxi(0 , sup(u+(j*s), u+(j+1)*s))
        n = mini(0 , inf( u+(j*s) , u+(j+1)*s ))
        d = int((((m-n))/k)*x)
        for i in range(d):
            a = random.uniform(float(u+(j*s)), float(u+((j+1)*s)))
            b = random.uniform(float(n), float(m))
            for j in range(10000):
                plt.plot(j/1000, funct(j/1000),
'go', markersize = 0.5)
            if b >= 0:
                g = g+1
                if b <= funct(a):
                    q = q+1
                    plt.plot(a, b, 'ro', markersize = 0.5)
                else:
                    plt.plot(a, b, 'bo', markersize = 0.5)
            else:
                h = h+1
                if b >= funct(a):
                    r = r+1
                    plt.plot(a, b, 'ro', markersize = 0.5)
                else:
                    plt.plot(a, b, 'bo', markersize = 0.5)
       c = c + ((q/g)*(s)*(m)) + ((r/h)*(s)*(n))
```

```
    return c
```

Code for the Riemann Integration

```
import math
import random
import matplotlib.pyplot as plt
import multiprocessing as mult
from decimal import *

getcontext().prec = 50

#the function we are integrating
def funct(x):
    if x == 0:
        return 1
    else:
        return (math.sin(1/x))

#needed for the end result
def addition(list):
    f = 0
    for i in range(len(list)):
        f = f + list[i]
    return f

#calculates the Riemann integral on a certain interval [u,v]
#and writes the result on a document
def integral(x , u , v, j):
    k = 0
    a = (v-u)/x
    text = open("documentriemann" + str(j) + ".txt", "w")
    for i in range(x):
        k = k + a*((funct(u+i*a))+funct(u+(i+1)*a))*0.5
    text.write(str(k))

#divides the interval on the number of cores and lets each core
#write the calculated integral in a text document
if __name__ == '__main__':
    x = int(input("How many rectangles?\n"))
    u = float(input("Left bound?\n"))
    v = float(input("Right bound?\n"))
    cores = mult.cpu_count()
    s = (v-u)/cores
```

```
    res = 0

    for j in range(cores):
        m = u+(j*s)
        n = u+(j+1)*s
        p = mult.Process(target = integral, args = (x , m , n, j))
        p.start()

#adds together the values from the text documents to
#get the end result
def result():
    cores = mult.cpu_count()
    res = 0
    for j in range(cores):
        text = open("documentriemann" + str(j) + ".txt", "r")
        res = res + float(text.readline())

    print(res)
```

The code for the Trapezoid Integral

```
# -*- coding: utf-8 -*-
import math
import random
import multiprocessing as mult
from decimal import *

getcontext().prec = 50

#the basic function we are integrating
def funct(x):
    if x == 0:
        return 1
    else:
        return (math.sin(x)/x)

#the maximum of two numbers
def maxi(x,y):
    if x > y :
        return x
    else:
        return y

#the minimum of two numbers
```

```
def mini(x,y):
    if x > y :
        return y
    else:
        return x

#the trapezial integral on a certain interval
def integrale(x, u ,v):
    k = 0
    m = (v-u)/x #sub-interval length
    for i in range(x):
        a = funct(u+i*m)
        b = funct(u+(i+1)*m)
        z = mini(a,b)*m #the lower rectangle part of the trapeze
        y = (maxi(a,b)-mini(a,b))*m*0.5 #the triangle part
        k = k+z+y
    print(k)
```