Université de Strasbourg

Report - practical works 1 to 3

M1 Physique

Axel ROELLINGER

Practical work n°1

1. Communicating vases

The goal of this exercise is to code a way to obtain the values of a numerical suite for three variables, a, b and c, as a function of an integer varying in the range [0, N]. This suite consists in giving a fraction α of a quantity to the next element of the list. The cycle is over when the last element of the list passes a fraction α to the first one.
In our case of three variables, the following figure describes the suite :

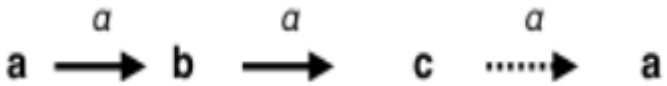

FIGURE 1 – Illustration of the suite. b gets a portion α from a. Then, c gets a portion α from b. Finally, a gets a portion *alpha* from c. The cycle is then over.

To compute this suite, I decided to proceed to a simple "for" loop, going from 0 to N, in which I would represent the different transfers between a, b and c. To observe the evolution of these values according to alpha, I created a function 'vases' which takes as arguments 3 2D-tables, which collect the evolution of the values a, b and c, for a given α. The row corresponds to a cycle and the column to a given alpha. To re-use the values of α in my plot, they are written at the first line of the file. Two other int, i and N, are used to fill the tables and limit the 'for' loop with the desired number of cycles.

```
void vases(double** aVal, double** bVal, double** cVal, int N, double alpha, int i)
{
    double a = 5;
    double b = 10;
    double c = 15;
    double sum = 0;

    aVal[0][i] = a;
    bVal[0][i] = b;
    cVal[0][i] = c;

    for(int j = 0; j < N; j++)
    {
        b += alpha*a;
        a -= alpha*a;

        c += alpha*b;
        b -= alpha*b;

        a += alpha*c;
        c -= alpha*c;

        aVal[j][i] = a;
        bVal[j][i]= b;
        cVal[j][i]= c;
    }
}
```

FIGURE 2 – Function "vases", which computes the evolution of a, b and c through different sums. The tables are used to fill in the different files for the plotting.

By running the program once without changing α, we observe that the sum of a, b and c remains constant. We will study the impact of α on the values.

At each loop cycle, the variables' evolution is collected in a 2D-table, one per variable, with a row being equivalent to one cycle, and the column being the change of α, to study the evolution of each value and how α influences their behavior. The sum of a, b and c is also collected in a 2D-table to

observe the influence α may have on it. After running my program, I observed that, no matter what parameter would be changed, the sum would remain the same.

However, what we can observe is that for the smallest variable, the value will always go up, the growing speed depending on the value of alpha.
The largest one will always decrease, and once again, alpha determines the stiffness of the initial slope. The most interesting behavior is the one of the value in between. At first, a decrease will be noticed, then, after a few cycles, it will increase and reach a plateau.

To display all of my graphs, I chose to use a '.plt' script that is ran from the program itself by a 'system' instruction. The same method applies to generate every graph.

The following figure describes the expected type of figure :

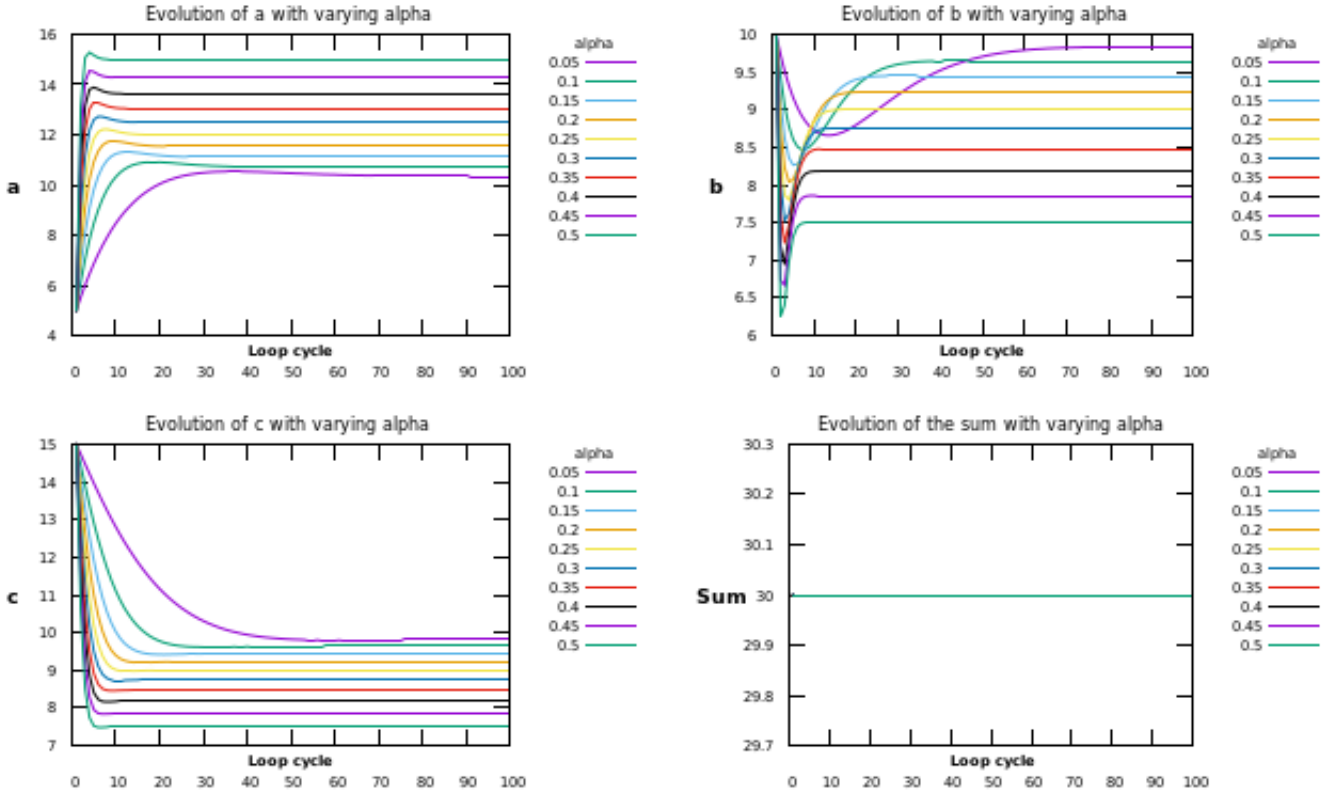

FIGURE 3 – Evolution of a, b, c and sum according to the number of loop cycles and the value of alpha. Here, $a = 5$, $b = 10$, $c = 15$

2. Riemann integration

The objective is now to implement a solution to get the approximate result of the integral of a given function.

The theoretical formula is the following :

$$I = \int_a^b f(x)\mathrm{d}x = \sum_i \int_{x_i}^{x_{i+1}} f(x)\mathrm{d}x = \sum_i P_{0,i} \times h + r_n(h)$$

We define the sum term as :

$$P_{0,i} = f(x_i), x_i = a + i \times h,\ i = 0, 1, 2, ..., N$$

with

$$h = \frac{b-a}{N}$$

and N the number of points used.

The first step is to define a function that returns the value of $f(x)$ with a given x, making it easier if we only want to change the input function.

```
double fx(double x)
{
    if(x < a || x > b) //Validity criteria
    {
        cout << "Error ! x is not in the interval [a,b]" << 2*x << endl;
    }
    return 2*x;
}
```

FIGURE 4 – Function to compute the wanted $f(x)$ in a given interval. Boundary conditions are applied to ensure that $f(x)$ stays within the integration interval

To compute such a calculation, the main idea is to implement a 'for' loop for the sum, and multiply the sum by h.

As a and b, the integral's limits, were said not to be invoked as arguments of the function, I chose to insert them right after the 'using namespace std ;' line, making them global variables for the whole program. input.

```
double solve_Riemann(int j, int N, ofstream& errEvo)
{
    int N1 = N*j; //Number of subdivisions, goes up with j for the evolution of h
    double x[N1], y[N1]; //Tabs
    double h = (b - a)/N1; //Step
    double sum = 0;

    for(int i = 1; i < N1; i++) //Computes the sum
    {
        x[i] = a + double(i)*h;
        y[i] = fx(a + double(i)*h); //Uses the function described above
        sum += y[i];
    }

    errEvo << h << " " << abs(sum*h - Ifx()) << endl;
    cout << "h : " << h << " Integral : " << abs(sum*h - Ifx()) << endl;
}
```

FIGURE 5 – Function used to solve the Riemann's integral. The int j and the file are implemented for the evolution of the residual error. j allows for the function to be used in a loop to compute the integral for several values of h, and the file collects the residual error with the corresponding h

```
for(int j = 1; j < REVal; j++) //Loop to make h vary
{
    solve_Riemann(j, N, errEvo);
}
```

FIGURE 6 – Loop to fill the input file. REVal defines the number of values in the file and the step h

To estimate the convergence rate of this method, the idea is to compute the sum for different values of h, plot the residual error (absolute value of the analytical value minus the numerical value), fill a file with this value and the corresponding h, and plot it Thus, the graphical analysis will provide us an answer.

To do so, I chose to insert my calculation loop inside a function called 'solve_Riemann' [5] which I have given as arguments an int j, which is the index of the loop, an int N, which is the starting point

for the subdivisions, and finally an output file to collect the residual error after each loop. The loop is encapsulated in a loop for a varying h, as the number of subdivisions is equal to a base number N (user-defined) multiplied by j. [6].

The plot of the error as a function of h gives us the following graph :

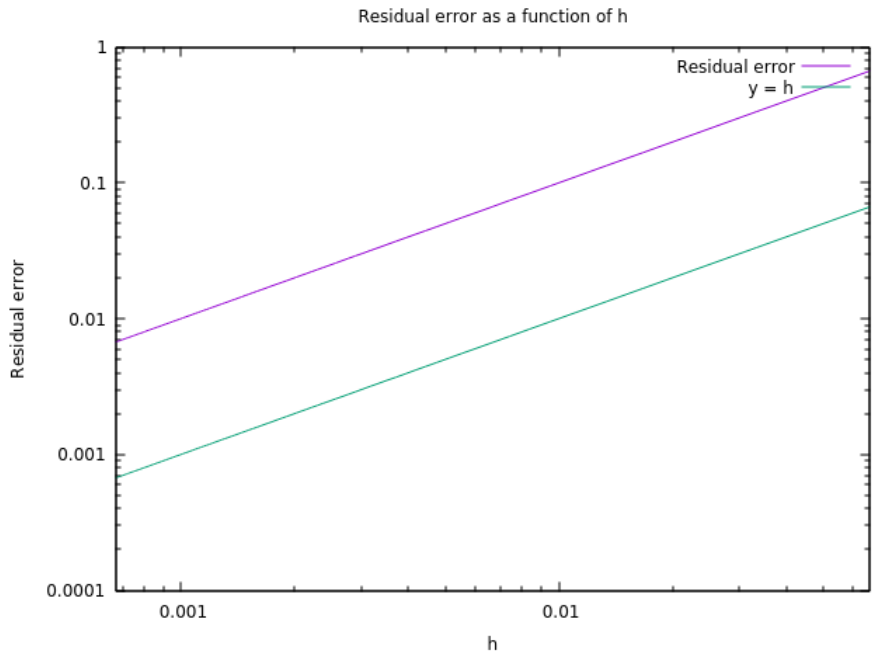

FIGURE 7 – Residual error as a function of h for the function $y = 2x$. Thanks to the *log*-scale, we can observe that the error behaves as a linear function of h

3. Accelerated Richardson convergence

As an improvement of the numerical scheme of the Riemann integration, we can compute the algorithm of the accelerated Richardson convergence. This algorithm doesn't require loads of calculations to obtain a precise estimation of the integral, saving a lot of computation time and resources.

Our starting point is the Riemann estimation of our integral with a step h_0, labeled $I = Q_0(h_0)$. We can compare these results to those obtained with different values $h = h_k$ to build a set of results $Q_j(h_k)$, $j = 1$ and $k = 0$ being our reference value.

Richardson obtained the following recurrence scheme :

$$Q_{j+1}(h_k) = \frac{2^j Q_j(h_{k+1}) - Q_j(h_k)}{2^j - 1}$$

where $h_k = h/2^k$, with k = 0,1,2,...,N

Each $Q_j(h_k)$ can be interpreted as a matrix element, with j being the row and k the column.

Therefore, the first line must be filled with values of the Riemann integral, only with the step h changing. Once done, the rest of the tab is filled with the formula above by using the loop in figure [8].

To compute this algorithm, the first step is to create the 2D-table. I built a 4 by 4 table as recommended, and filled the first line by using my 'solve_Riemann' function, with h varying.

To fill the rest of the lines, I applied a double loop, which only considers the elements of the tab to fill, making my column loop stop at the $(4 - j)^{th}$ column.

```
for(int i = 1; i < N+1; i++) //rows
{
    for(int j = 0; j < N+1-i; j++) //columns
    {
        tab[i][j] = (pow(2,double(i))*tab[i-1][j+1] - tab[i-1][j])/(pow(2,i) - 1); //Matrix element
    }
}
```

FIGURE 8 – Loop to fill the table of the Richardson's algorithm. The formula used is the one above

As my loop starts at i = 1 for the rows (corresponding to j-1), I had no need to tweak the formula to fulfill its purpose.

The error rate for low integral values (i.e. less than 100000) is very low, as the error rate with the analytical solution is less than 0.1% for the first value of the second line. For high values, such as e^{25}, the second value of the second line usually fills this criteria. This means that this algorithm allows very fast computation of integrals, as the first line is the only one which requires to compute the Riemann integration. The final step was to compare the Richardson's accelerated convergence to a standard Riemann algorithm. For the integral $I = \int_0^{25} e^x \mathrm{d}x$, the required number of steps to fall under the .1% of error is around 160000, which means far more calculations than the Richardson's algorithm, which preserves the same error convergence, but only requires 4 important computations.

The type error remains constant between the methods, as they will both converge linearly, for every function that I've tried ($\sin(x)$, $\tan(x)$, polynoms,...).

Practical work n°2

The objective of this second practical work is to study two other integration techniques : the trapezoid rule and the method of Simpson, and determine their efficiency.

1. Trapezoidal rule

Two functions are given to integrate :

$$f(x) = \frac{1}{4} + x - \frac{3x^2}{4} + \frac{3x^3}{32}$$

$$g(x) = \sin\left(1 + \pi \cos\left(x\right)\right)$$

The trapezoidal rule is a geometrical integration method which works by approximating the region under the curve by the sum of trapezoids. The total integration interval $[a, b]$ is divided in N sub intervals, defining at the same time the step $h = \frac{b-a}{N}$.

The formula to implement is the following :

$$\int_a^b f(x)\mathrm{d}x \approx \sum_{i=0}^{N-1} \frac{f(x_{i+1}) + f(x_i)}{2} \times h$$

$h = x_{i+1} - x_i$, therefore, it is the same as stating $h = \frac{b-a}{N}$. We stop at N-1, because the computation method would then use the value b + h, which is out of the integration interval. To implement this method, I created a function called 'trapez_solver' which takes in argument the function to evaluate, under the form of a pointer to said function, the integration limit values (thus, a and b), and N, the number of steps, to be able to use this function in a loop with a varying discretization value.

To do so, I implemented the function in a 'for' loop, and filled a file with the error to the analytical solution to display it using 'gnuplot' and study the order of convergence.

The final graph is the following :

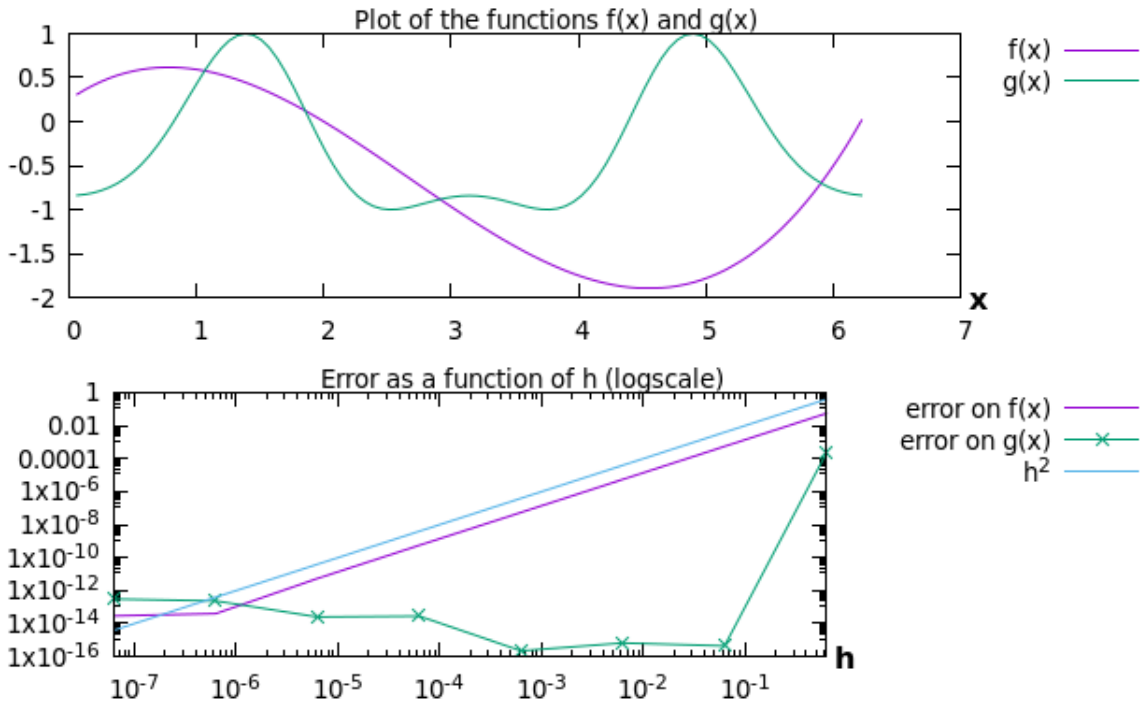

FIGURE 9 – Plot of the functions and the errors. As we can see, the error will follow a steady slope for $f(x)$, but the one for $g(x)$ describes no recognizable pattern

As we can observe it, the error for $f(x)$ decreases as a function of h^2. However, I wasn't able to find how the error for $g(x)$ converges. My hypothesis is that it converges too fast to be identified as a polynomial form of any sort. I decided to test my code on other trigonometric functions (see [17], [18] in appendix), the high convergence seems to be linked to the cosine part of the function. As we can see for $f(x) = 1 - \tanh^2(x)$, the error tends to follow the path of a quadratic function, before rising up, which must be related to numerical errors. For the second figure, the error is too constant to be considered as a polynomial form, but once again errors are piling up. This algorithm is still reliable for important values of N, but a value selection must be made, as the error should decrease for higher computations.

2. Method of Simpson

This second method is based on approximate a given function $f(x)$ with a polynomial function $P(x)$, which is easier to integrate. The numerical scheme to implement was the following :

$$\int_a^b f(x)\mathrm{d}x \approx \frac{h}{3}\left(f(a) + 4\sum_{\text{i odd}} f(x_i) + 2\sum_{\text{i even}} f(x_i) + f(b)\right)$$

To compute this sum, the first step I took was to code a loop which uses the function fx of the previous exercise, and distinguishes the cases between even and odd is, with the condition $if\ (i\%2 == 0)$, meaning that I consider the remainder of the euclidean division of i with 2. Then each sum is used in the formula below, which outputs the integral's value. The loop starts at 1 and ends at $N-1$ to avoid counting the function's value in a and b twice. I encapsulated it in a function called 'Simpson_solver', which takes in argument the function to integrate, the integration boundaries and an int N, used to define the step h. By putting N as an argument, I can implement the function in a loop and collect the residual error with the corresponding h.

This method allows a very fast convergence of value, as precision is very important even with a high step value compared to the one required for the Riemann integration technique.

In conclusion, the two methods offer a precise value, assuming an error in 10^{-14} can be tolerated, but are not reliable without a selection criterion on the computed values of the integral. The only thing

```
double Simpson_solver(double (*func)(double), double a, double b, int N)
{
    double h = (b-a)/double(N);
    double sum_even = 0; //Sum for values of the function where i is even
    double sum_odd = 0;

    for(int i = 1; i < N; i++) //Starts at func(a + h) and stops at func(b - h)
    {
        if(i%2 == 0)
        {
            sum_even += func(a + double(i)*h);
        }
        else
        {
            sum_odd += func(a + double(i)*h);
        }
    }
    return (h/3.)*(func(a) + 4*sum_odd + 2*sum_even + func(b));
}
```

FIGURE 10 – Function used to compute the integral. Two values (sum_*) are initialized to compute the two different sums required by the method. The counter starts at 1 to avoid computing func(a) twice and stops at N-1 to avoid computing func(b) twice.

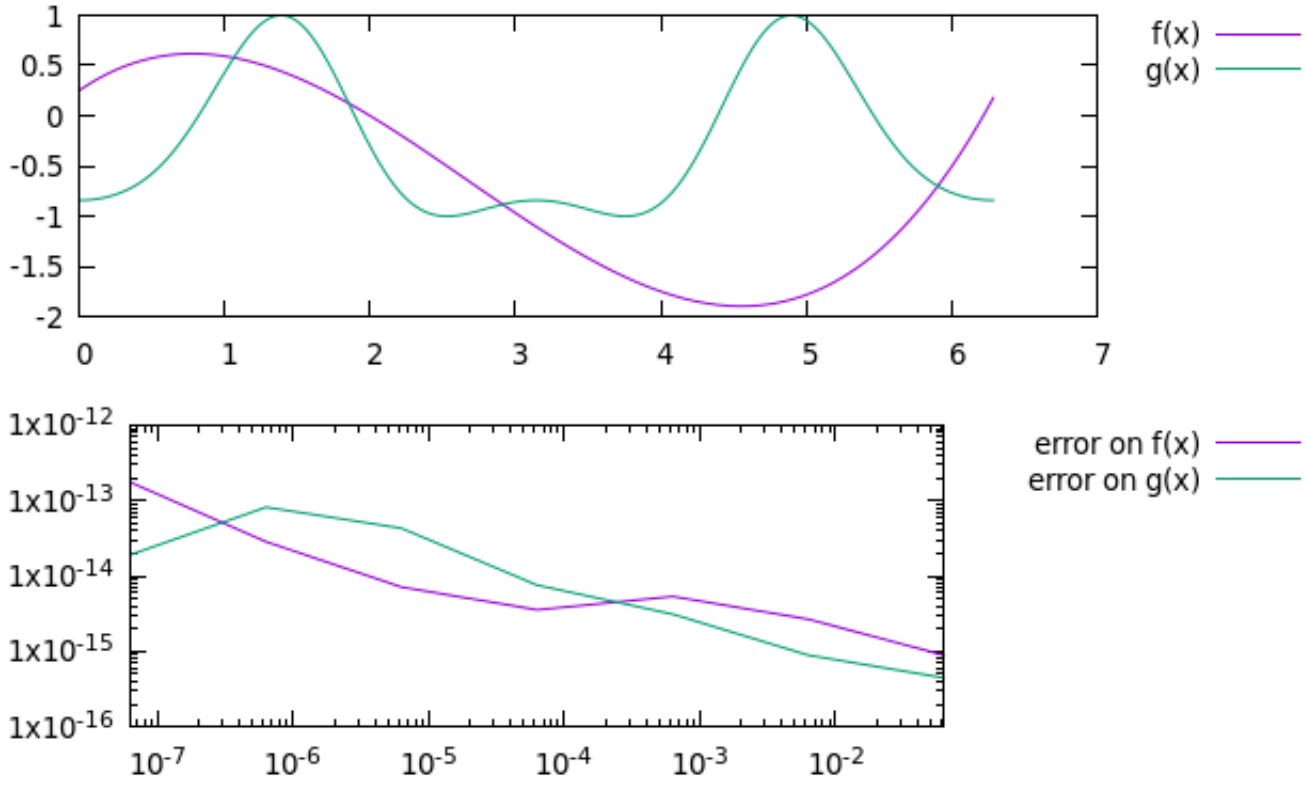

FIGURE 11 – Residual error's evolution. As witnessed in the figure [19], numerical errors pile up and cause the error to grow, while remaining low.

I couldn't identify is that the error grows back with the same value. This might consist in reproducible errors, or something linked to the method itself.

Practical work n°3

The objective of this third practical work is to use the von Neumann/Ulam method to compute the integral of a given function.
This method consists in finding the maximum of a function over the integration interval. This defines a rectangle h × L, with L the integration' interval length and h the function maximum in this interval. The next step is to generate a given number N of random coordinates which fall within the aforementioned rectangle's area. Then, we must count the points that fall under (N_{good}) and above (N_{bad}) the curve. The integral is then approximated by :

$$I \approx \frac{N_{\text{good}}}{N} \times h \times L$$

which corresponds, for an important number of points, to the area of the rectangle under the curve.To do so, we first have to implement a function to find the maximum of the function over our integration interval. The function to integrate is defined as follows :

$$D_E(E) = \begin{cases} 0 & \text{if} \quad E < 2 \\ \sqrt{2} - 1 + \exp(2 - E) & \text{if} \quad 2 < E < 4 \\ \exp(4 - E) & \text{if} \quad E \geq 4 \end{cases}$$

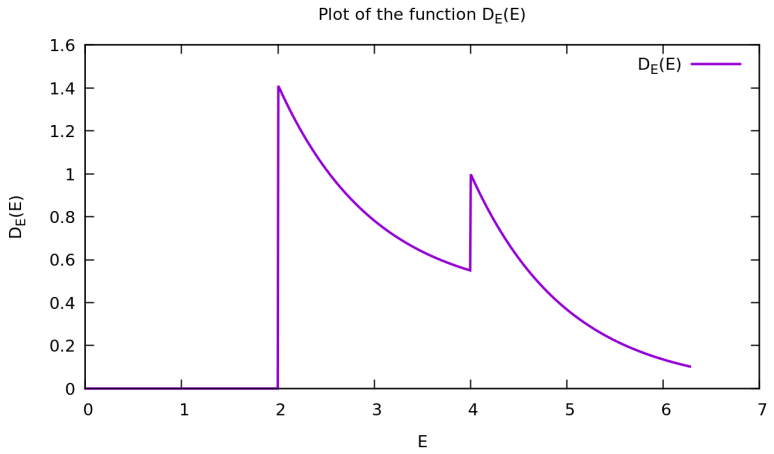

FIGURE 12 – Plot of $D_E(E)$. The function here has been plotted with 10^6 points for a good precision, required for the next steps.

This function is plotted on the figure above.
Analytically, the obtained value for this integral of this function over the interval $[0,\ 2 \times \pi]$ is 2.59113292307.
The first step to define the rectangle is to find the maximum of the function. Thus, I chose to compute the function with a very small step on the x-axis. I sub-divided it in 10^6 sub values, for I estimated the precision to be sufficient. I collected every y-axis equivalent value, and used a double-type variable to collect the biggest value while scanning through the tab.

The maximum will then be established as 1.41421285. This value then allows me to define the rectangle's area $L \times h$, which will be used to compute the integral. To obtain these values, the user should run my program and specify as an input the number of subdivisions named N_0.
The second step was to implement the random function to generates coordinates which fall within our rectangle's area. To do so, I chose to use the Mersenne twister engine, known as a good pseudo-random generator. The seed is also randomly generated by the built-in function 'random_device'.

The random values are generated in a function, which takes in argument a 2D-table and the number of wanted points.

```
double maxfunc( double (*func)(double), int N) //find the maximum of a given function
{
    double x = 0;

    for(int i = 0; i < N; i++)
    {
        double y = 2*M_PI*double(i)/double(N);

        if (func(y) > x)
        {
            x = func(y);
        }
    }
    return x;
}
```

FIGURE 13 – Function to compute the maximum of a given function 'func', the step defined by the number N is the critical factor concerning the precision of the returned double.

Number of subdivisions	Maximum value
10^3	1.40365045
10^4	1.40988684
10^5	1.41364752
10^6	1.41421285
10^7	1.41421285
10^8	1.41421348

TABLE 1 – Precision depending on the number of subdivisions

```
double rdm(double N)
{
    random_device rd{}; //Random 32-bit seed
    mt19937 generator{rd()};
    uniform_real_distribution<double>dist(0.0, N); //goes to a given value N
    return dist(generator);
}

void rdm_gen(double** valRdm, double L, double fMax, int N)
{
    for(int i = 0; i < N; i++)
    {
        valRdm[i][0] = rdm(L); //random from 0 to L
        valRdm[i][1] = rdm(fMax);
    }
}
```

FIGURE 14 – Functions used to generate random points with boundary conditions. The first function generates a random number by using the Mersenne twister engine, shaped through a uniform distribution. The second one fills a 2D-table used for point counting and plotting the final graph.

To plot the points, I chose to use two different files, which makes it easier for the color attribution.

After running the algorithm for 800 random points, we obtain the following graph (see figure [15]). The obtained value of the integral in the figure is way too far from the analytical solution to be used, but serves as a good example to show that the algorithm works, without overloading the figure with points. Past 1.7 million points, the integral starts to stabilize. However, and due to my hardware limitations, it took way too much time to try to compute the integral above 2.4 million points. However, the array of values is pretty satisfying, with the residual error being proportional to 10^{-3} for the last values I computed. The value of the integral doesn't vary before the 2^{nd} decimal once passed the barrier of 10^5 generated points. To try and reproduce (within an uncertainty due to randomness) my result, the user must simply run my program as it is. This approximation is a good starting point to evolve towards a more precise one, but the minimum required number of points to obtain a steady estimation is very

important. This algorithm tends to be resource-heavy. it would be interesting to run it on more powerful computers to observe the stabilization of the integral's value.

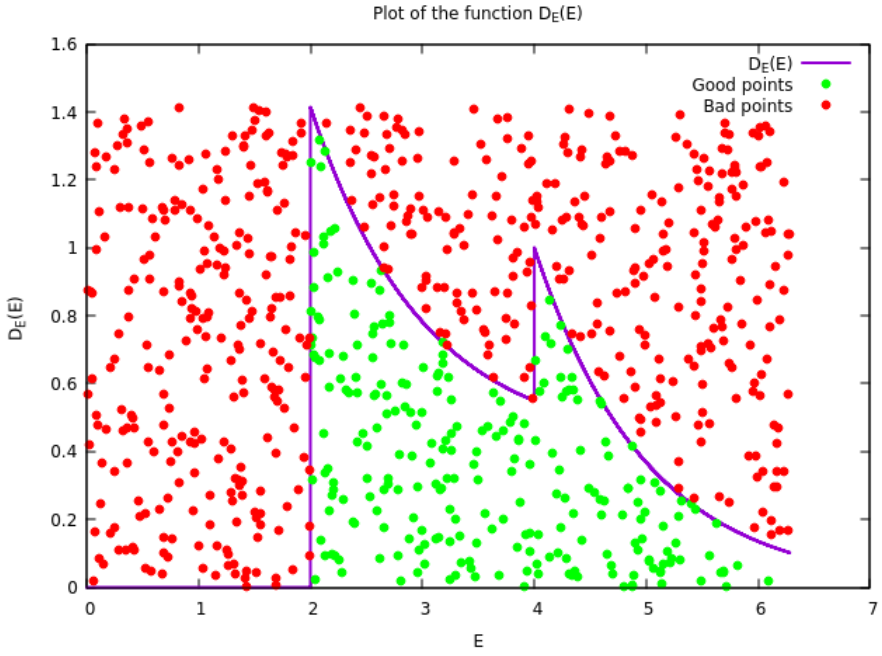

FIGURE 15 – $D_E(E)$ plotted with the different types of points. There are 209 good points, giving us an integral of 2.32140516

As a final result, I decided to compare every integration method for the following calculation :

$$I = \int_0^{25} e^x \, \mathrm{d}x$$

The value N will be defined as $N = 10^6$ and the analytical value will be set as : $7.20048993363858 \times 10^{10}$. I chose to let this much digits to get a precise estimation of the error. One exception being made for the Richardson algorithm, for which the starting number of steps will be 100. Concerning the von Neumann/Ulam method, we will use N generated points.

Method	Value	Error
Riemann	$7.20039992788947 \times 10^{10}$	$9.0005749118042 \times 10^{5}$
Richardson	$7.20049054183593 \times 10^{10}$	$6.0819734039306 \times 10^{3}$
Trapezoid	$7.20048993401363 \times 10^{10}$	3.7503814697265
Simpson	$7.20048993363863 \times 10^{10}$	0.0004119873047
von Neumann/Ulam	$7.18500888038105 \times 10^{10}$	$1.54810532575317 \times 10^{8}$

TABLE 2 – Comparison table for the integral I

At first sight, the Simpson method is undoubtedly better than all of the other techniques. The only one which could be better, with a bit patience, is the von Neumann/Ulam method, as a good row of points can give us a fairly precise approximation, but this probability is very low. As a matter of calculation safety, the Simpson method is the more appropriate.

Appendix

Practical work n°1

```
6.337892237e+10 6.75983251e+10 6.97781838e+10 7.088568246e+10
7.181772784e+10 7.19580425e+10 7.199318112e+10 0
7.200481406e+10 7.2004894e+10 0 0
7.200490542e+10 0 0 0
```

FIGURE 16 – Filled tab with the Richardson's algorithm for $y = e^x$ between 0 and 25

Practical work n°2

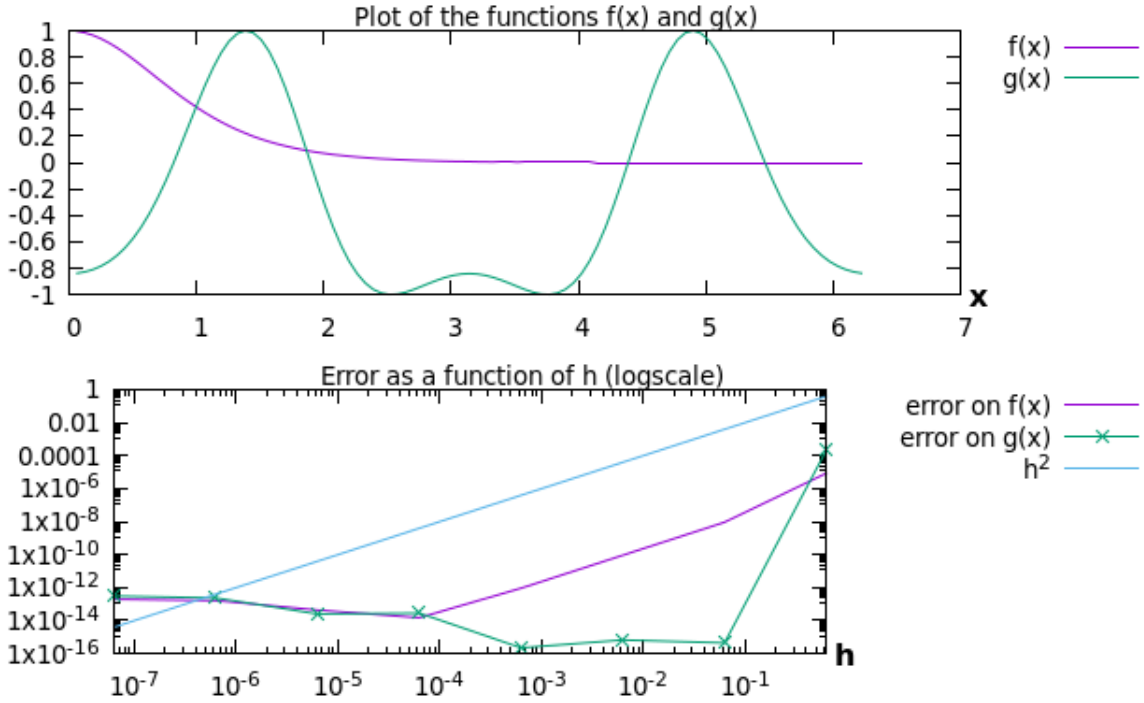

FIGURE 17 – Error for $f(x) = 1 - \tanh^2(x)$ with the trapezoidal rule. The error was evaluated for powers of 10.

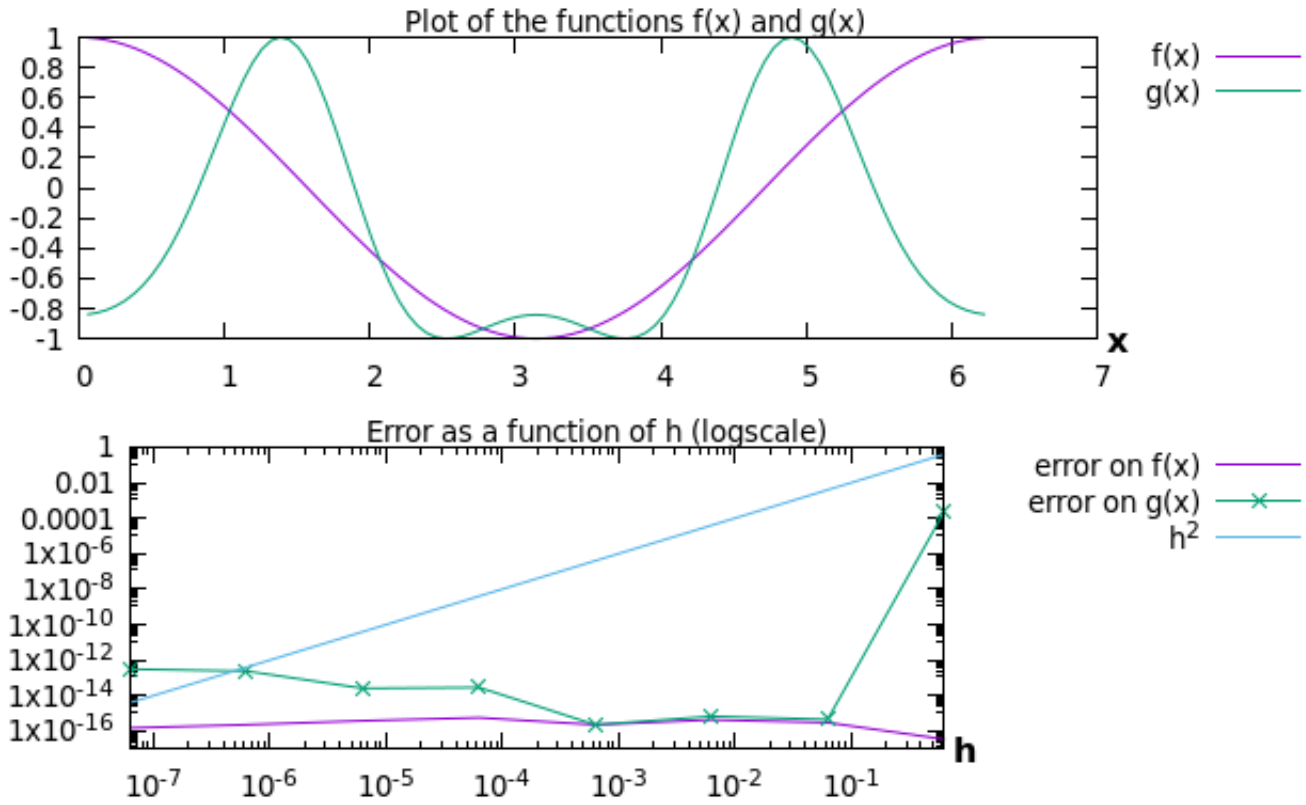

FIGURE 18 – Error for $f(x) = \cos(x)$ with the trapezoidal rule. The error was evaluated for powers of 10.

```
N : 100
fVal : -4.174139093875112 error : 8.881784197001252e-16
gVal : -1.60856433277385 error : 4.440892098500626e-16
N : 1000
fVal : -4.174139093875115 error : 2.664535259100376e-15
gVal : -1.608564332773849 error : 8.881784197001252e-16
N : 10000
fVal : -4.174139093875118 error : 5.329070518200751e-15
gVal : -1.608564332773847 error : 3.108624468950438e-15
N : 100000
fVal : -4.174139093875109 error : 3.552713678800501e-15
gVal : -1.608564332773857 error : 7.549516567451064e-15
N : 1000000
fVal : -4.174139093875105 error : 7.105427357601002e-15
gVal : -1.608564332773807 error : 4.285460875053104e-14
N : 10000000
fVal : -4.174139093875141 error : 2.842170943040401e-14
gVal : -1.608564332773768 error : 8.149037000748649e-14
N : 100000000
fVal : -4.174139093874937 error : 1.749711486809247e-13
gVal : -1.608564332773831 error : 1.909583602355269e-14
```

FIGURE 19 – Numerical issue for the Simpson's algorithm. The values converge very fast, but due to a high number of steps, errors pile up and produce a growing, but small nonetheless, residual error. The evolution is the same for the trapezoidal rule, with a low increase in the error.