Université de Strasbourg

Report - practical works 4 to 6

M1 Physique

Axel ROELLINGER

Practical work n°4

Exercise 1 :

The objective in this exercise is to study the efficiency of two differentiation schemes.

$$\frac{\mathrm{d}f}{\mathrm{d}x} \approx \phi(h) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2)$$

and

$$\frac{\mathrm{d}f}{\mathrm{d}x} \approx \frac{4}{3}\phi(h/2) - \frac{1}{3}\phi(h) + O(h^4)$$

To implement these schemes, I dedicated two functions which return the value of the derivative at a given point x for a given h. Then, I combined them in a function called 'deriv()' which allows the user to choose which order to use by using an int in the arguments which acts as a criterion for a switch.

The functions we will study are the following :

$$f(x) = x - \sin^2\left(\frac{3\pi x}{2}\right)$$

and

$$g(x) = \tan(x)$$

We propose ourselves to use the interval [0, π] to study these functions and their derivatives. As they have a periodicity in π, the behavior will remain the same on larger intervals. However, we can notice that the value of $\frac{\pi}{2}$ will cause some trouble for $g(x)$ doesn't exist, and so does its derivative.

To display the evolution of the error in relation to h for different points on the interval [0,π], I chose to create 2D-tables. I created one per order and per function, so 4 tables.

I took my code from the first exercise about communicating vases and modified it to fit my needs. I named the function 'error_varyX'.

The function in itself takes as arguments the aforementioned four 2-D tables, the analytical derivatives and the 'deriv()' function. The variation of x is pre-implemented to cover 11 going from 0 to π, with a step of $\pi/10$. Thus I estimate the interval to be enough split to study the two functions $f(x)$ and $g(x)$. The criterion which will be left to the user's choice is the number of values of h. It is defined as $1/\mathrm{pow}(10,i)$, giving straight lines when using a logarithmic scale for plotting, with i being the counter of the "for" loop in the 'error_varyX' function. However, to make plotting easier, my tables won't only store the computation results. The first case of each column stores the value of x for which we are computing the derivatives, which is then used for the key of the graphs. Also, the first column is dedicated to the values of h to fill the x-axis of the graphs. Therefore the [0][0] and [13][0] elements are empty. The last column of each table is dedicated to the storage of the power of h which fits the model.

To synthetize, we obtain this kind of table :

0	x_0	...	x_N	0
h_0				h^2
...		*Values*		*or*
h_N				h^4

FIGURE 1 – Table obtained at the end for each differentiation order for each function

The zeros won't affect the data as they are not linked to any other value except when plotting h on a Cartesian scale. The use of the log scale avoids this problem.

As we can see on the graph of $f(x)$, the point x = 0 seems to work without any noticeable error, without any algorithm dependence, as the curve can't even be seen for this value.

On the opposite, for $g(x)$, as we declared it above, the point $x = \pi/2$ causes the error to be enormous, caused by the non-existence of $\tan(x)$.

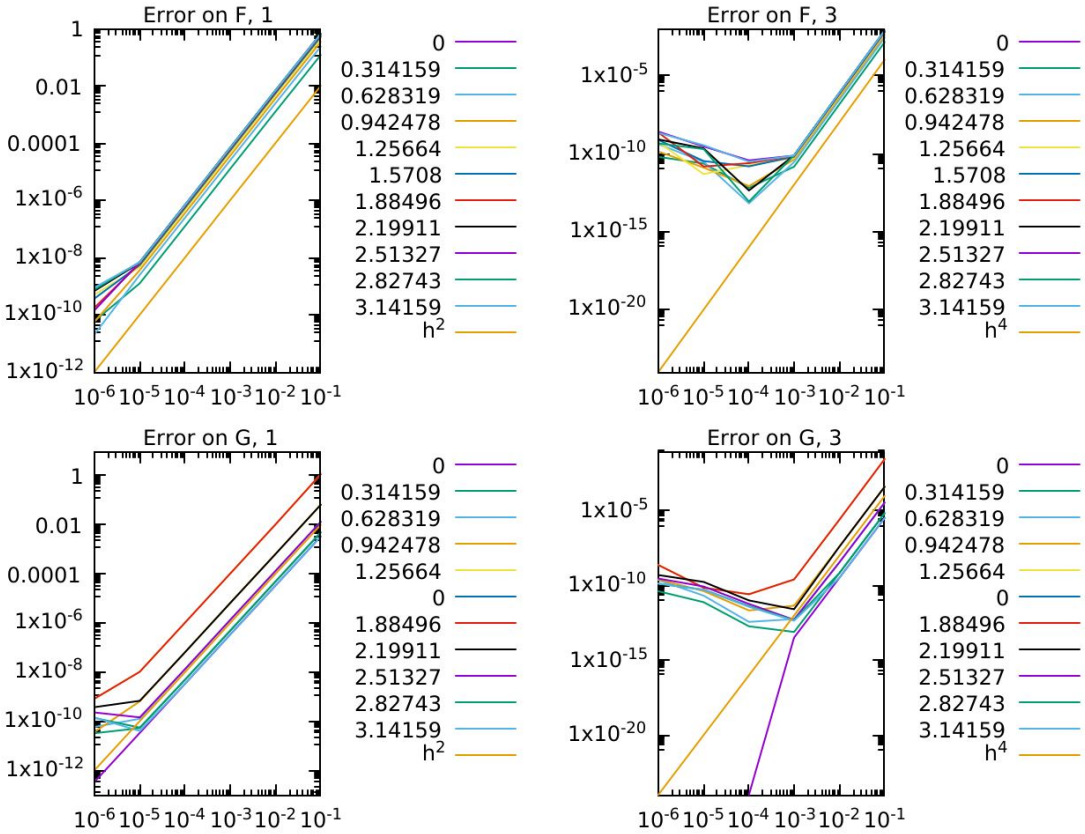

FIGURE 2 – Final errors on the two schemes in function of the step h for several given x values in the keys. We can see that numerical errors tend to be more present for the third order's scheme past $h = 10^{-3}$

The graph displays an error in h^2 for the first order's scheme and in h^4 for the third order's scheme. I chose not to display the whole plausible power laws on this graph in order to keep it a bit more readable. The values for $x = \pi/2$ are not displayed for $g(x)$, as shown in figure [3] because its value is disproportionate as the derivative of $g(x)$ doesn't exist for this point. The full graph is the figure [13] in appendix.

```
for(int j = 1; j < 12; j++)
{
    if(M_PI*double(j-1)/10 != M_PI/2)
    {
        F1[0][j] = M_PI*double(j-1)/10.;
        F3[0][j] = M_PI*double(j-1)/10.;
        G1[0][j] = M_PI*double(j-1)/10.;
        G3[0][j] = M_PI*double(j-1)/10.;
    }
    else
    {
        F1[0][j] = M_PI*double(j-1)/10.;
        F3[0][j] = M_PI*double(j-1)/10.;
        G1[0][j] = 0
        G3[0][j] = 0
    }
}
```

FIGURE 3 – Condition used to filter the data to put in the files. 'M_PI*double(j-1)/10' represents the value x

Exercise 2 : Accelerated Richardson convergence

This exercise will put in use the algorithm of the accelerated Richardson's convergence. The principle is the same as for the exercise 3 in the practical work n°1 : to avoid tons of computations to find a satisfying value of our estimation. The algorithm requires to create a 2D table, whom only the first line must be filled before running the algorithm. In our case, the first row must feature values of the derivative we want to calculate with different h steps. The step at the k-th column is given by $h_0/2^k$. h_0 is the initial step. To fill the table, we use the following equation :

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

with j the row number (starting at $j = 1$) and k the column number (starting at $k = 0$).
To fill the rest of the table, I implemented 2 'for' loops, the first one varying from 1 to 3 (for the rows), and the second one varying from 0 to N+1-i, which stops the algorithm from scanning cases where no value is intended. There is no point to go through the whole tab, only the blocks which need to be filled.

I fixed h_0 as 0.1, giving us a fair view of the algorithm's power.
I went through the whole interval to study how my results evolve. The algorithm is very precise, and the step for which the error between the estimated value and the analytical value is less than 0.1% is either on the first row or on the second row for the functions we study. It may vary for some others. Also, this is conditioned by the initial h_0, as it will influence on the precision of the first row's values.

Nevertheless, the algorithm tends to obtain a good value of $\frac{\mathrm{d}g}{\mathrm{d}x}$ on the second row, meaning that $1 + \tan^2(x)$ seems harder to evaluate than a product of sine and cosine functions.

If we had to run the differentiation algorithms of the first exercise, the most probable step is around 0.05 for $f(x)$ and 0.01 for $g(x)$. These are not computation-heavy steps, but remain more demanding than the Richardson's algorithm, which allows to split the $x - axis$ in lots of subdivisions, allowing for a more precise point-to-point derivative.

One of the possible studies would be to evaluate how many operations (in other terms the rank of the value) are needed to recover a value with less than 0.1% of error compared to the analytical value. To do so, I implemented a function called 'error_percent' which computes the rank of the corresponding value. Overall, the value is between the third one of the first row, and the second one of the second row (when reading from left to right, top to bottom), meaning that this algorithm tends to be fairly precise with a few amount of steps.

As another way to study the precision of this algorithm, I computed the average h required to obtain the same precision using the first scheme of the first exercise. Here is the final graph :

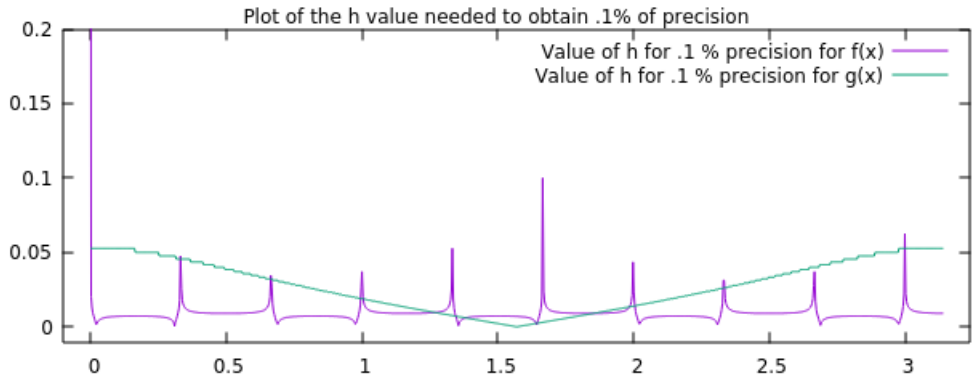

FIGURE 4 – Graph of the required h to obtain 0.1 % of precision compared to the analytical value

What is interesting to see is the periodicity of the functions is also present in the one for the value of h, which looks to be directly correlated to the value of x.

To see if the final result of the Richardson's algorithm could be used as a reliable value, I decided to compute the absolute difference between this and the analytical value. On the whole interval, the average absolute difference is 0.000253993 for $f(x)$ and 1679.648321292 for $g(x)$. These values fluctuate because of computation errors, but these fluctuations are proportional to 10^{-5}. The results are satisfying for $f(x)$. However, it can't be safely used for $g(x)$, especially for values around $\frac{\pi}{2}$, which cause such a mean error value. If we ignore the values of x in the interval $[\pi/2 - 0.05,\ \pi/2 + 0.05]$, the mean error value is 12.283307709, which is better, but not enough. To obtain the same range of mean error, we must define an interval of avoidance such as $[\pi/2 - 0.2,\ pi/2 + 0.2]$, which we could rename as the interval of "unreliability", assuming the new value of 0.000944948 is sufficiently small for the user.

Practical work n°5

Exercise 1 : integration with a first order Eulerian scheme

The goal of this exercise is to implement two integration schemes based on a series for which each element is determined by the previous one. The goal is to reconstruct a function $f(y,t)$ on an interval [a,b].

The differential equation to solve is :

$$y'(t) = \frac{\mathrm{d}y(t)}{\mathrm{d}t} = f(t,y) = -\Gamma \times y(t)$$

with Γ being a constant factor from the interval [0,10], and where t is a value on the x-axis.

The first method is the following :

$$y_{i+1} - y_i = h \times f(t_i, y_i).$$

with $h = (b-a)/N$, $i = 0, 1, ..., N$ and $y(t_i) = y_i$. The second one, also called delayed Euler method is :

$$y_i - y_{i-1} = h \times f(t_i, y_i)$$

Such an algorithm strongly reminds me of the Riemann integration, where the integral was computed as the sum of small rectangle of length h and height $y(a + i \times h)$, i, being the number of the rectangle.

To analyze the correctness of our algorithm, the next step was to compute the exact solution of the differential equation, which is :

$$y(t) = y_0 \times e^{\Gamma(t_0 - t)}$$

y_0 and t_0 correspond to the coordinates of the starting point of the integration. This implies that we can't start our integration at the point (0,0), which is logical, as the equation $e^x = 0$ has no solution.

The first step to build the two schemes, as it is the common point between the two, is to implement a function for $f(t,y)$. As specified above, this function returns, for a given t, $-\Gamma \times y(t)$. Thus I made a function which takes as arguments Γ and t and returns what $f(y,t)$, which can be seen as $f(y(t))$.

I created two functions for each scheme, each returning a 1D table filled with the different values of y. For an interval split in N subdivisions, the returned table will be $N+1$ long to go from a to b. Both of the functions are based on simple 'for' loops, the difference being the index. On the first one, as we run the algorithm to get the $(i+1)$-th value, we can start at $i = 0$. On the second one, we need to start at $i = 1$, as we use the $(i-1)$-th value to get the i-th value. Also, $f(y,t)$ can't be used as such for the delayed scheme :

$$
\begin{aligned}
y_i - y_{i-1} &= -h \times \Gamma \times y_i \\
y_i + h \times \Gamma \times y_i &= y_{i-1} \\
y_i(1 + h \times \Gamma) &= y_{i-1} \\
y_i &= \frac{y_{i-1}}{1 + h \times \Gamma}
\end{aligned}
$$

The last line is what I used for the delayed scheme.

From this, I can compute the whole integration process and study the error for varying h. As the algorithm allows to compute a whole function, the only way to do so s to pick a value on the x-axis and study the error on this one. I chose to study the error on the last point, as it would be interesting to observe the results of every cumulated numerical errors (due to the iterative scheme, each error impacts the value of the next y-value, up to the final one). To do so, I implemented a loop going from 1 to 8, with $h = 1/10^i$.

To see if the study of the error in one point gives us a valid error law, we have to study the stability of the scheme. We will introduce an error on y_0 and see if the error on the final term is proportional to the first one.

$$
y_0 = y(t = a) \longrightarrow y_0 + \delta y_0
$$

then the solution in $t = b$ becomes :

$$
y(t = b) = y_N \longrightarrow y_N + K\delta y_0
$$

with K a constant which doesn't vary when N » 1.

The objective is to compute the value of K over a number of iterations of the differentiation schemes.

To do so, I created a function named 'stable', which takes as arguments Γ, y_0 and $y_0 + \delta$, which I renamed y0_delta in my prototype. This function runs the two schemes on a given set of subdivisions, to study how K stands along the value of N, which will here be set at 5000.

From this, I can extract the value of K :

```
double h = 1/double(i); //Step
double* f1 = Euler_1(gamma, y0_delta, h, i); //Studies the scheme 1 an offset delta on y0
double* f2 = Euler_2(gamma, y0_delta, h, i); //Studies the scheme 2 with the same offset

double K1 = (f1[i] - y0)/(f1[0] - y0); //Value of K stored in a tab, found with the equations given in the problem sheet
double K2 = (f2[i] - y0)/(f2[0] - y0);

stable_plot << i << " " << K1 << " " << K2 << endl;
```

FIGURE 5 – One loop turn of the 'stable' function, used to collect the value of K for each scheme.

I chose to run the algorithm of each scheme to fill tables of size i. I then extract the last value of the table to get the value of K, which is finally stored in a file used for plotting.

The figure [6] is the final graph. As we can observe it, the error follows a linear line for both schemes, with a little "bump" for $h = 10^{-5}$, which I can't explain otherwise than with some calculation errors. Concerning the third graph, we can deduce that both schemes are stable past 1000 iterations, making those schemes reliable.

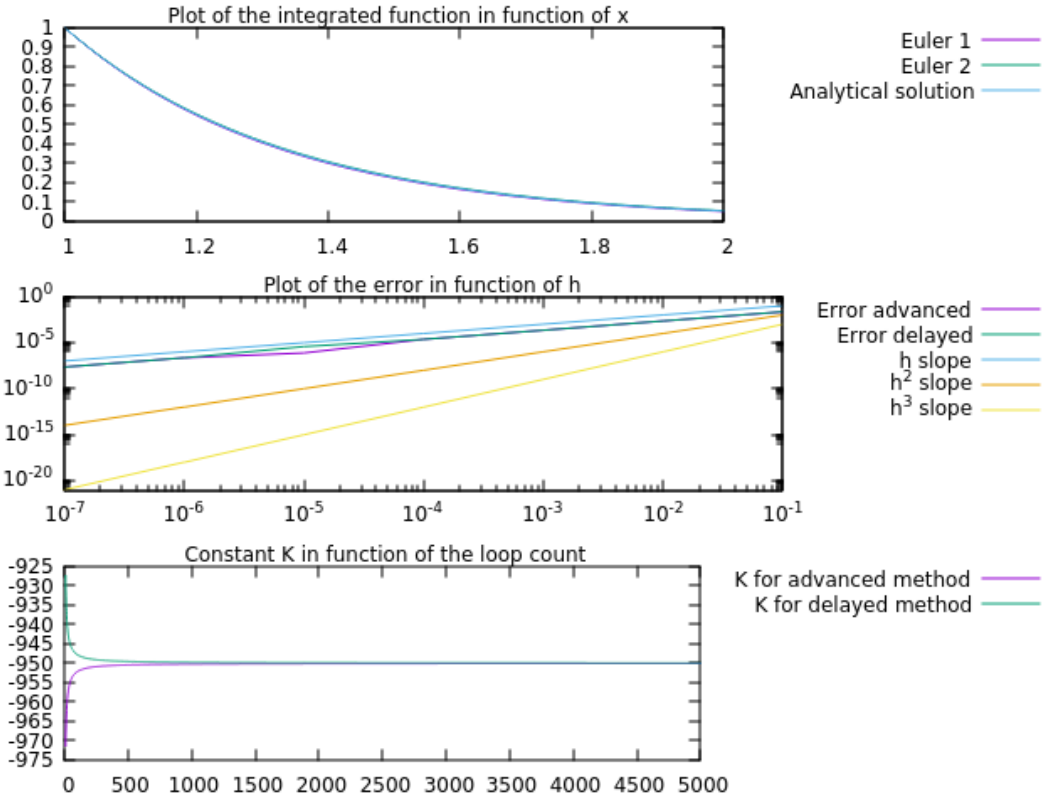

FIGURE 6 – Final graph using displaying the final integration, the error, and the stability of the scheme. This scheme was obtained with N = 150 for the initial subdivisions of the curve, and $\delta y_0 = 0.001$

Exercise 2 : second order Eulerian scheme and ABM scheme

In this exercise, we will use two other patterns which are supposedly being more reliable due to their smaller error while using the same type of functions. The two schemes are the following :
Second order Eulerian scheme :

$$y_{i+1} - y_{i-1} = 2hf(t_i, y_i) + O(h^3)$$

and the ABM scheme :

$$\tilde{y}_{i+1} - y_i = \frac{h}{12}[23f(t_i, y_i) - 16f(t_{i-1}, y_{i-1}) + 5f(t_{i-2}, y_{i-2})] + O(h^4)$$

$$y_{i+1} - y_i = \frac{h}{12}[5f(t_{i+1}, \tilde{y}_{i+1}) + 8f(t_i, y_i) - f(t_{i-1}, y_{i-1})] + O(h^4)$$

To implement the first scheme, I built a table whose first 2 elements were made using the Euler scheme of previous exercise, non delayed, as the second order scheme can't be used otherwise. For the rest of the table, I implemented the formula above in a 'for' loop.

Concerning the ABM scheme, I created another table named y_tilde, whose I filled with the formula for $\tilde{y}$, and I then rewrote the content of the table used in the second order Eulerian scheme to plot the final result. i could allow this as I have already stored the data of the second order scheme in a file.

To study the error, I arbitrarily chose $x = 1.5$. Then, I implemented a loop making the step h vary as $1/pow(10, i)$, i being the loop counter. I run the full algorithm to determine the error on the second order Eulerian scheme and the ABM scheme. To plot it, I fill a text file, one per scheme, with the absolute value of "numerical value - analytical value". The final figure is the figure [7].

As we can observe, there is a huge surge of both errors at 10^{-5}. I can't explain what is the cause of it, as it is located at the same place than the anomaly on the error graph of exercise 1, needing some deeper study. Nonetheless, we can still safely establish that the error of the second order Eulerian scheme's error is following a law of power of h^2, and the ABM scheme's is following a power law of h^3. In conclusion, for a high enough step h, the ABM scheme is more reliable in terms of calculations, while being heavier because it requires to fill 2 1D tables to be used

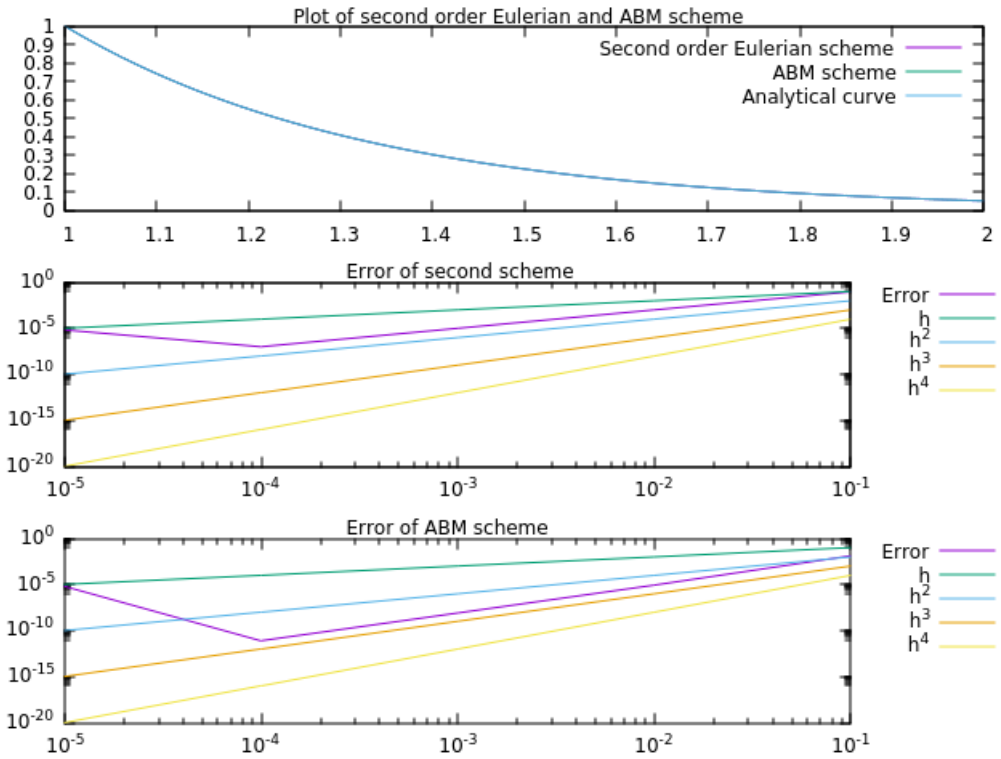

FIGURE 7 – Final graph obtained when using $\Gamma = 3$ and $y_0 = 1$. We can see that both methods allow for a good integration scheme, as the three graphs at the top can't be distinguished.

Practical work n°6 : zero and extrema of functions

Exercise 1 : finding extrema with the bisection method

The objective of this exercise is to find all the extrema (maxima or minima) of a given function over a given interval. The two functions we want to test are the following :

$$f(x) = \sin^2\left(\frac{3\pi x}{2}\right)$$

and

$$g(x) = x - \sin^2\left(\frac{3\pi x}{2}\right)$$

The bisection method consists in finding a given array of four points (x_1, x_2, x_3 and x_4) on the x axis, for they might verify one of the following conditions :

— If $f(x_1) > f(x_{2or3})$ and $f(x_{2or3} < f(x_4)$, then a minimum lies somewhere in this array

— If $f(x_1) < f(x_{2or3})$ and $f(x_{2or3} > f(x_4)$, then a maximum lies somewhere in this array

To find such an array, I decided to fill a 1D table with 10001 points of the x-axis, to go from one end of the interval to the other one. This value can be changed by the user, but I set it to 10001 (so, N = 10000) as it gives a good discretization of the interval. There is one table per function.

The next step is to find four points verifying one of the above conditions. To do so, I implemented a function which goes through the generated tables and checks for each loop turn if one of these criteria is fulfilled. If so , it returns the four needed points, which constitute the baseline to find the extremum's position.

The objective will be to implement an algorithm to redefine the initial array with a condition of proximity on x_1 and x_4. Indeed, if we shorten enough the distance between these points, around 10^{-4}, which represent the limits of our presence interval for the extremum, we can assume that our approximation for $x_{extremum}$ is fairly precise, as the error will be located on the seventh digit of its mantissa.

We should define the interval in the following way :

$$\frac{x_3 - x_2}{x_4 - x_1} = z$$
$$\frac{x_2 - x_1}{x_4 - x_1} = w$$
$$z = 1 - 2w = \sqrt{5} - 2 = 0.23606$$
$$\frac{3 - \sqrt{5}}{2} = w$$

Using these relations, we can calculate x_2 and x_3. Once it is done, we have to find a sub-interval in between x_2 and x_3 if x_3 - $x_2 > 10^{-4}$, if not, the value x_2 would be assigned to x_1 and the value of x_3 to x_4, and a last computation of x_2 and x_3 would be then needed.

To implement this algorithm, I created a function called 'find_subinterval' which takes in arguments an initial array of four points, and an int N. The function will then take the interval between the second and third value of the array and split it in N sub-divisions.

This function has no stop flag, the selection criterion is mentioned in another function : 'find_min_max', in which 'find_subinterval' redefines the interval only if the last one doesn't respect the rule of x_4 - x_1 $< 10^{-4}$.

```
double* find_min_max(double* newPts, double eps, int N)
{
    if(abs(newPts[3] - newPts[0]) > eps) //Precision criterion
    {
        double* newPts2 = find_subinterval(newPts, N);

        find_min_max(newPts2, eps, N); //Rerun the function with the same values to validate the epsilon's condition
    }
    else
    {
        return newPts; //Return the list as is
    }
}
```

FIGURE 8 – Function used to redefine the interval if the boundaries of the array are too widely spaced. It works recursively, as it will call itself until the condition is verified

Once this is done, a last function is used, invoking the following rules :

— A minimum is present in the sub-interval (x_1, x_2, x_3) if $f(x_2) < f(x_3)$ or in the sub-interval (x_2, x_3, x_4) elsewise

— A minimum is present in the sub-interval (x_1, x_2, x_3) if $f(x_2) > f(x_3)$ or in the sub-interval (x_2, x_3, x_4) elsewise

Considering the subdivisions give us a precise estimation of our value, I chose to designate either x_2 or x_3 as $x_{extremum}$.

From now, the remaining part is to plot the results. To do so, I used 3 different files : one for the functions, one for the extrema of $f(x)$ and the last for the extrema of $g(x)$. I chose to split the extrema in separate files to colorize them accordingly to the function they are linked to. To display the lines on the graph, I used the option "with vectors" in gnuplot, allowing me to draw straight lines. However, I went through minor problems, which could be problematic on larger scales, as some extrema would be calculated twice by my algorithm. The origin of this bug is unknown, but could lead to over-estimations of results.

The final result is the following :

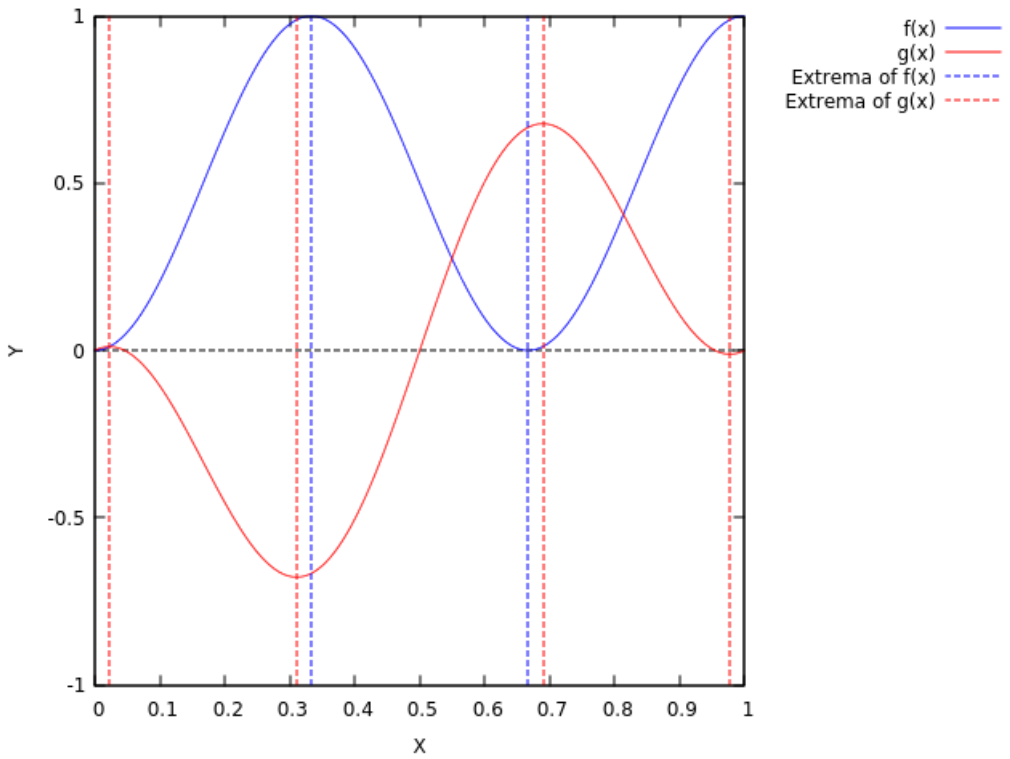

FIGURE 9 – Final representation with the extrema of each function. I matched the colors of the extrema with the one of their respective function for a better readability

Exercise 2 : finding zeros with the Newton-Raphson method

The final objective ob this exercise is the same as above, but with a different technique. The Newton-Raphson method assumes that two points x_1 and x_2 have been identified, such as $f(x_1) \times f(x_2) < 0$. In other words, a zero of the function is in between x_1 and x_2. From there we can assume that one or the other point is a good guess for our zero. The algorithm of Newton-Raphson proceeds by approaching the function at first order in this point based on the fact that the function is asymptotically equal to its tangent in $x = x_2$, such as $f(x) \approx f(x_2) + f'(x_2)(x - x_2)$. From this, the idea is to find the intersection of the tangent with the x axis, resolving the equation $f(x_2) + f'(x)(x - x_2) = 0$ for x. This will give us a new estimation x_3 from which we will repeat the same process until finding a good estimation of the location of the zero on the x-axis.

In practice, we will start from a given approximation of the zero at $x = x_k$, and the algorithm proceeds by iteration to find the next x value with the following formula :

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Starting from this information, the first objective is to find two points which fulfil the first criterion of $f(x_1) \times f(x_2) < 0$. To do so, I first have to define a subdivision criterion for the x-axis. I decided to start at N = 100, giving us a step $h = 1/N = 0.01$. Knowing this, I implemented a function called 'find_zero_interval' which takes as an argument one of the functions to study, the step h, and an int i. This function will return two points x_1 and x_2 in a table if we verify :

```
if(func(double(i)*h)*func(double(i+1)*h) < pow(10,-6.5))
```

FIGURE 10 – Condition used to select intervals, the value of -6.5 was arbitrarily chosen.

with 'func' representing the studied function. I am not comparing the product to zero, otherwise some zeros wouldn't appear. Indeed, this condition of the product of function values being strictly negative

implies that the x-axis is fully crossed, and not only reached, which means that functions like $y = x^2$ would have no zeros according to this algorithm. I then chose a small condition to display all of the zero values.

The int i serves the purpose for the function to be used in a loop, returning every zero-containing intervals, otherwise it would stop at the first one.

Once this function returns the two initial points, the Newton-Raphson algorithm can be used. To do so, I use the table returned by the 'find_zero_interval' which will the first two elements of a table initialized at the start of the function. The table's size is determined as an int called "precision", which also represents the number of loop-turns which will be made to determine the zero. I use a loop to fill the tab using the Newton-Raphson algorithm. The function will then return the last value of the table, which is the estimation of the zero's position. To obtain all of the zeros, I then created a function called 'find_all_zeros' which repeats the whole process throughout the whole interval (in our case, [0,1]) This is the general way to go, however, due to some computation errors, I had to implement a workaround.

As the function 'find_zero_interval' can't work if it doesn't return a table, I decided that, if the condition on the product was not verified, the table will be filled with "NaN" values. The next step of using the Newton-Raphson will still go on, but it will return a "NaN" value as well. To counter this, I implemented the following condition :

```
if (!isnan(estimXFunc) && !isnan(tabInitFunc[0]))
```

FIGURE 11 – Condition to consider the value of the zero or not. I used it because I encountered some errors with "NaN" values to evaluate, which is impossible.

with "estimXFunc" being the estimation of the zero's position and "tabInitFunc" the table containing the two initial values of the interval. If these two values are different from "NaN", in other words, exploitable, they will then be used in the plotting of the function. Otherwise, they are just dumped. There is only one value for which "estimXFunc" will be evaluated as "NaN", but acting on the values of the table allows not to declare another empty table in my 'main()' loop and directly use what 'find_zero_interval' will return.

To plot the figures, I used a '.plt' script, which I call from within my program through a "system" line. This script uses 3 different files : one to plot the two functions $f(x)$ and $g(x)$, one to display the zeros of $f(x)$ using vectors colored in blue, and a third one to display the zeros of $g(x)$ with vectors too, but colored in red, as these colors correspond to the trace of their respective function.

This is the final figure :

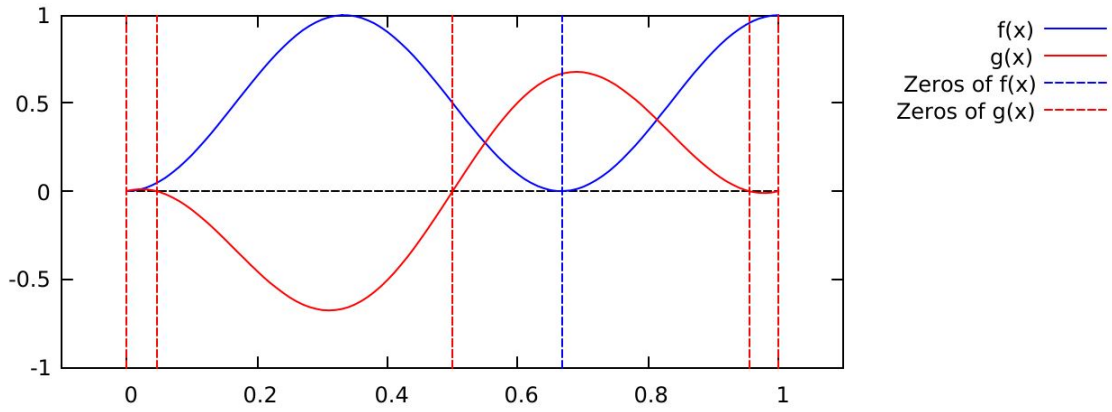

FIGURE 12 – Final graph with zeros for $f(x)$ and $g(x)$ using the Newton-Raphson algorithm. This graph was obtain with a step $h = 0.01$ and 'precision' set to 100

As we can see, both methods (extrema and zeros) give us relatively precise results from a graphical point of view, the last error in the chain being the resolution of the screen. However, it would be interesting to study the numerical errors that intervened during the computation and study how much the number of initial subdivisions would impact it for example. I presented a method to do so in appendix.

Appendix

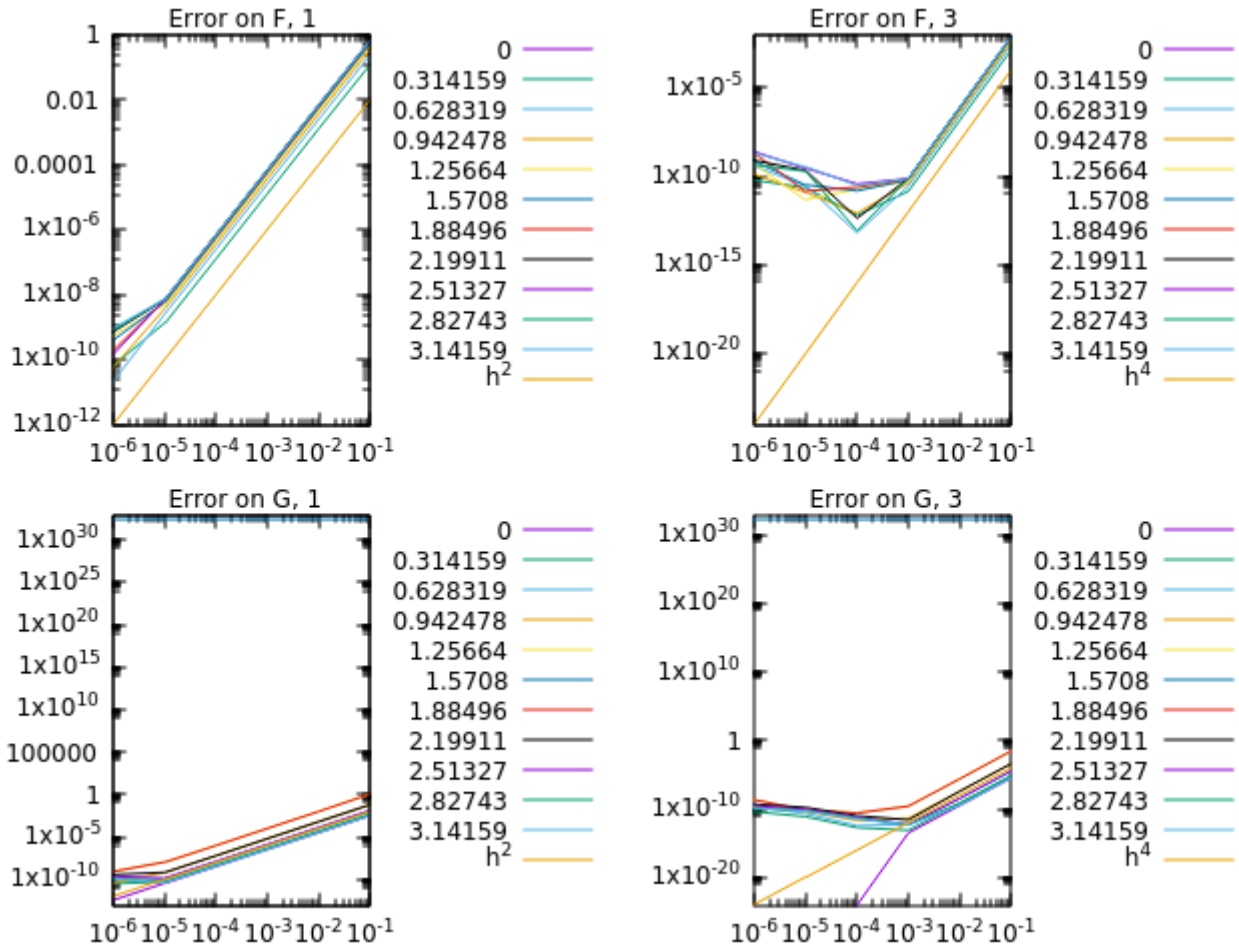

FIGURE 13 – Errors on the full interval between 0 and π for $f(x)$ and $g(x)$. As we can see, the error for $x = \pi/2$ is very important for $g(x)$ due to the fact that $\tan(\pi/2)$ doesn't exist.

Calculating the errors for the last practical work

Concerning the errors, I would proceed as the following. I would first analytically calculate the x values of the extrema and zeros of the functions to store them in different 1D tables. I would then pass them as arguments of my main functions, here 'stable' and the main loop of the second exercise, then a counter should be implemented, starting at -1 and that would be incremented of 1, each time an extrema or a zero is found. Then the error would be calculated by comparing the numerical x value that was just found with the value of the corresponding table at the rank [counter]. Starting at -1 will then allow the first point of interest to be compared to value of rank [0] of the corresponding table. Repeating these tasks for different steps would allow for a plot of the evolution of the error in function of the step. Plus, the number of loopturns of the Newton-Raphson gives an other interesting point of view, which can show that even a high number of subdivisions can be useless if this algorithm isn't sufficiently exploited. Still, a quick study allows me to tell that there is a noticeable graphical error for the extrema when the number of subdivisions of the initial interval is lower than 100. Concerning the zeros, defects start to appear for a number of subdivisions under 100 with a low number of loopturns for the Newton-Raphson algorithm, which plays a keypoint in terms of precision.