C1 - Assignment 2 Report: Advection-Diffusion-Reaction equation.

Student Number: 1894945

November 17, 2018

Contents

1 Introduction

1.1 Boundary value problem

Definition 1. Let $\Omega \in \mathbb{R}^d$, for $d = \{1, 2\}$ be a bounded, simply connected, open domain. The Boundary Value Problem (BVP) that will be considered in the following is find, for f and v given, a function u such that

$$\mathcal{L}[u] = f \quad in\ \Omega, \qquad\qquad u = v \quad on\ \partial\Omega \tag{1}$$

for $\mathcal{L}[u] = -\Delta u(x) + \mathbf{p}(x)\nabla u(x) + q(x)u(x)$, with $\mathbf{p}$ and q given, smooth functions.

BVPs like this one are called Dirichlet problems. In what follows, for simplicity, everything will be referred to the case $d = 1$.

1.2 Finite difference methods

The discretisations that will be consider in the following are based on the finite difference (FD) approach. The spatial operator $\mathcal{L}$ is evaluated at a set of points $\{x_j\}_{j=1}^{J}$ and the derivatives are replaced with difference quotients of the approximate solution u, which are in turn obtained by truncating (at the desired order of accuracy) the Taylor expansions of the exact solution u(x) around the point x_j.

Let $\{x_j\}_{j=0}^{J+1}$ be a partition of the interval $(0, L)$, $L \in \mathbb{R}^+$, such that $0 = x_0 < x_1 < \cdots < x_{J+1} = L$. The j^{th} interval will be denoted by $I_j = (x_{j-1}, x_j)$ with mesh size $h_j = |I_j|$. For simplicity, a uniform mesh size will be considered and denoted by h, so that $x_j = jh$.
Moreover let u_j represents the approximation to the exact solution $u(x_j)$. Then, a simple discrete version of the operator Δ in (1) is, for $j = \{1, \cdots, J\}$, given by:

$$\Delta[u] = \frac{u_{j+1} - 2u_j + u_{j-1}}{h^2} \tag{2}$$

Similarly, the operator $\nabla[u]$ can be discretised as :

$$\Delta[u] = \frac{u_{j+1} - u_{j-1}}{2h} \tag{3}$$

Such choices guarantee that the differences between (2) and (3) and their respective continuous counterparts is, at most, $\mathcal{O}(h^2)$, as the following simple calculation explicitly shows.

$$u(x \pm h) = u(x) \pm hu'(x) + \frac{h^2}{2}u''(x) \pm \frac{h^3}{3!}u'''(x) + \frac{h^4}{4!}u^{(4)}(x) + \mathcal{O}(h^5) \tag{4}$$

Inserting Eqn.(4) into Eqn.2 and Eqn.(3), it is easy to see that:

$$\frac{u(x+h) - 2u(x) + u(x-h)}{h^2} = u''(x) + \frac{h^2}{12}u^{(4)}(x) + \mathcal{O}(h^4) \tag{5}$$

$$\frac{u(x+h) - u(x-h)}{2h} = u'(x) + \frac{h^2}{6}u'''(x) + \mathcal{O}(h^4) \tag{6}$$

The operators defined in Eqn.2 and Eqn.(3) are *central differences* and will be used for the implementation of the code.

The BVP (1), can thus be discretised as:

$$\begin{aligned} -\frac{u_{j+1}-2u_j+u_{j-1}}{h^2}+p_j\frac{u_{j+1}-u_{j-1}}{2h}+q_ju_j=f(x_j), \quad j\in\{1,\cdots,J\} \\ u_0=v(x_0), \quad u_{J+1}=v(x_{J+1}) \end{aligned} \tag{7}$$

1.3 Consistency, convergence and stability of FD methods

The following discrete function spaces will play an important role in the study of consistency, convergence and stability of FD methods:

$$\begin{aligned} \bar{\mathbb{U}}_h &= \{u : \bar{\Omega}_h \to \mathbb{R}\} \\ \bar{\mathbb{U}}_h^0 &= \{u \in \bar{\mathbb{U}}_h : u|_{\partial\Omega_h} = 0\} \\ \mathbb{U}_h &= \{u : \Omega_h \to \mathbb{R}\} \end{aligned}$$

where the domain $\bar{\Omega}_h \equiv \{x_j\}_{j=0}^{J+1}$, while Ω_h represents the set of points in the interior of Ω.
It also convenient to introduce the following *restriction operator* $r_h : C(\bar{\Omega}) \to \bar{\mathbb{U}}_h$ defined as:

$$[r_hv]_j = v(x_j), \quad j\in\{1,\cdots,J\}, \quad v\in C(\Omega)$$

Such an operator allows to compare $C(\bar{\Omega})$ functions with grid functions.

Clearly, the discrete problem (7) can be now stated as:

$$\bar{\mathcal{L}}_h[u] = f_h \tag{8}$$

where $\mathcal{L}_h : \bar{\mathbb{U}}_h \to \mathbb{U}_h$ represents both the discretisation of the the different operators involved in Eqn.(1) and the boundary conditions and $f_h \in \mathbb{U}_h$ is given by $[r_hf]$.

Definition 2. The consistency error of the method (8) relative to the exact solution u(x) in a suitable discrete norm is

$$e_c = ||\bar{\mathcal{L}}_h[r_hu] - f_h||_{\mathbb{U}_h}$$

The discretisation is said to be *consistent* if

$$e_c \to 0 \quad \text{as} \quad h \to 0 \tag{9}$$

Moreover, the discretisation is said to be consistent of order p if:

$$e_c \to \mathcal{O}(h^p) \quad \text{as} \quad h \to 0 \tag{10}$$

Definition 3. The discretisation is said to be *convergent* if

$$||r_h u - u_h||_{\bar{\mathbb{U}}_h} \to 0 \quad \text{as} \quad h \to 0 \tag{11}$$

The discretisation is said to be *convergent* of order p if

$$||r_h u - u_h||_{\bar{\mathbb{U}}_h} \to \mathcal{O}(h^p) \quad \text{as} \quad h \to 0 \tag{12}$$

Definition 4. The method is *stable* for some constant $C > 0$ if

$$||v_h - w_h||_{\bar{\mathbb{U}}_h} \leq C||\bar{\mathcal{L}}_h[v_h] - \bar{\mathcal{L}}_h[w_h]||_{\mathbb{U}_h} \quad \forall v_h, w_h \in \bar{\mathbb{U}}_h \tag{13}$$

The practical meaning of the previous definitions is as follows: Taylor's theorem gives a way of estimating the consistency error (given a sufficiently smooth exact solution and suitable discretisation of the right hand side), while stability guarantees that rounding errors occurring in the problem will not have an excessive effect on the final result (see [2]).

Theorem 1. *Let $u \in \bar{\mathbb{U}}_h$ solve the discrete problem $\mathcal{L}_h[u] = f_h$. If the method is stable and consistent, then it is convergent.*

Proof. Stability implies:

$$||r_h u - u||_{\bar{\mathbb{U}}_h} \leq C||\bar{\mathcal{L}}_h[r_h u] - \bar{\mathcal{L}}_h[u]||_{\bar{\mathbb{U}}_h} = ||\bar{\mathcal{L}}_h[r_h u] - f_h||_{\bar{\mathbb{U}}_h}$$

Hence, by consistency, the method is convergent. □

It is worth noticing that the calculation just presented also shows that the order of convergence is at least equal to the order of consistency.

1.4 The present case

In the present assignment, special cases of an 1D linear advection-diffusion-reaction problem are studied. Such equations describe the advection, diffusion and reaction (sometimes the term absorption is used) of a given quantity represented by $u(x)$: typical example can be picked from hydrodynamics and from physics in general.
The problem can be formulated as follows. Given $f \in C([0, L])$, find $u \in C^2(0, L)$ such that:

$$-u''(x) + pu'(x) + qu(x) = f, \qquad u(0) = a, u(L) = b \tag{14}$$

for $a, b, p, q \in \mathbb{R}$.

Just as the general method previously introduced, the problem can be represented in the matrix form:

$$A\mathbf{u} = \mathbf{f} \tag{15}$$

where $\mathbf{u} = (u_1, \cdots, u_J)^T$, $\mathbf{f} = (f(x_1), \cdots, f(x_J))^T$. In this context, the entries of A are given by:

$$a_{ij} = \begin{cases} -\frac{\alpha}{h^2} - \frac{\beta}{2h}, & \text{for } j = i-1 \\ \frac{\alpha}{h^2} + \gamma, & \text{for } j = i \\ -\frac{\alpha}{h^2} + \frac{\beta}{2h}, & \text{for } j = i+1 \\ 0, & \text{otherwise} \end{cases} \tag{16}$$

In this way, differential equations can be solved through the methods used in numerical linear algebra.

In what follows, it will be assumed that $f = 0$, $p = \frac{\beta}{\alpha}$, $q = \frac{\gamma}{\alpha}$ (for $\alpha, \beta, \gamma \in \mathbb{R}$), $u(0) = 0$ and $u(L) = 1$.

1.4.1 The advection-diffusion problem

If γ is set to 0 in (14), a simpler problem, composed of only the advection and diffusion terms, is obtained. The solution to such a problem, for $u(0) = 0$ and $u(L) = 1$ is easily obtained by means of standard techniques for second order ordinary differential equations and reads:

$$u(x) = \frac{1 - e^{\frac{\beta}{\alpha}x}}{1 - e^{\frac{\beta}{\alpha}L}}, \quad 0 \leq x \leq L \tag{17}$$

It is clear that the solution can be rewritten as a parametric function of the global Péclet number $\mathbb{P}_e = \frac{|\beta|L}{2\alpha}$, measures the dominance of the advective term over the diffusive one. Setting $L = 1$ as required by the assignment instructions and assuming $\beta > 0$ for simplicity, one has:

$$u(x) = \frac{1 - e^{2\mathbb{P}_e x}}{1 - e^{2\mathbb{P}_e}}, \quad 0 \leq x \leq 1$$

In order to have a more clear interpretation of the results presented in the following sections, it is worth studying the limit of both large and small global Péclet number.
If $\frac{\beta}{\alpha} << 1$, a simple Taylor expansion up to first order of (17) yields:

$$u(x) \approx \frac{\frac{\beta}{\alpha}x}{\frac{\beta}{\alpha}} = x \tag{18}$$

The dominance of α with respect to β makes the solution closer to the one of the problem $\alpha u''(x) = 0$ with the same boundary conditions (BC). On the other hand, for $\frac{\beta}{\alpha} >> 1$ one has:

$$u(x) \approx \frac{\frac{\beta}{\alpha}x}{\frac{\beta}{\alpha}} = e^{\frac{\beta}{\alpha}(x-1)} = e^{-\frac{\beta}{\alpha}(1-x)} \tag{19}$$

Since the exponent is big and negative the solution is almost equal to zero everywhere unless a small neighbourhood of the point $x = 1$ where the term $1 - x$ becomes very small and the solution joins the value 1 with an exponential behaviour. The width of the neighbourhood is of the order of $\frac{\alpha}{\beta}$ and thus it is quite small: in such an event, we say that the solution exhibits a boundary layer of width $\mathcal{O}\left(\frac{\alpha}{\beta}\right)$ at $x = 1$ (see Ref.[1]).

1.4.2 The diffusion-reaction problem

On the other hand, if β is set to zero, the solution to the corresponding problem (for the same BC) is given by:

$$u(x) = \frac{\sinh(\frac{\gamma}{\alpha}x)}{\sinh(\frac{\gamma}{\alpha}L)}, \quad 0 \leq x \leq L \tag{20}$$

In this case, the relevant parameter to study the dominance of the advection with respect to the response terms, is represented by $D_a = \frac{\gamma}{\alpha}$ and is called Damköhler number.

2 The program

2.1 Implementation

The program is built on the class ADR , which stands for Advection-Diffusion-Reaction. The class has 7 private variables: J (the number of points in the interior of $[0, L]$), α, β, γ (the parameters associated to the ADR equation at hand), L (the length of the interval), u_0 (the boundary condition at 0) and u_L (boundary condition at L). In this way, once the constructor is called, all the variables and parameters defining problem (14) are set.
As pointed out in §(1.4), BVPs can be solved through numerical methods for linear algebra: the member function *MatrixBuild*, which accepts no argument, constructs matrix A of Eqn.(15), whose entries are given by (16), which will be later inverted through the Gauss-Seidel algorithm. Such a construction takes places within the function *Solver*, that performs the tests required by the assignment instructions through the Gauss-Seidel method and outputs the results on different .txt files. For clarity reasons, each time a simulation is performed, a suitable success message containing the different parameters of the corresponding test, is printed on the terminal. *Solver* is a member function which accepts 4 arguments and implicitly returns the discretised solution *u_x*. The argument *itCheck* represents the number of iterations after which a check for stagnation of the method is performed, while *MaxIter* represents the maximum number of iterations the user is willing "wait" and *tol* the given tolerance.
The member function *An_sol* outputs the analytical solutions at each point of Ω_h, by simply computing either Eqn.(17) or Eqn.(20) at the given set of points. The choice between the two solutions is made by means of an *if* instruction, checking if $\alpha \neq 0, \gamma = 0$ or if $\alpha \neq 0, \beta = 0$. When none of the above conditions is satisfied, the program exits with an appropriate error message to the user.
Discretised and the analytical solutions are finally compared within the function *ADR_Test*. Such a function take as arguments the values of the private datas, the arguments of *Solver* and the different values of J the user wants to test on. The function loops over them and prints the final error between the numerical solution and the analytical one, in the L_∞ norm, on suitable .txt files, whose names are defined through the parameters the user has set for the test. The ratio between errors obtained for different mesh sizes is computed and printed, too.
Since, by means of simply changing the values provided in the main file, both class of equations can be tested with no further modifications to the code, the program can be regarded as particularly flexible.

2.2 Running the program

The program is run through the Makefile provided. The current optimisation is set to -Ofast and all following tests have been performed with this optimisation choice. Every possible error or warning has been explicitly checked by means of the instruction *-Wall -Wfatal-errors -pedantic* before proceeding to performing the different tests: no warnings appeared.
The program generates a fixed number of output files, in which the results of the tests have been stored. Some of these files are used to produce plots by means of six different plotscripts.

2.3 Testing correctness

In order to check the correctness of the implementation of the code, the tests requested by the assignment have been performed. The class of ADR equations chosen as a benchmark is represented by:

$$-\alpha u''(x) + \beta u'(x) = 0 \tag{21}$$

Nonetheless, the correctness of the code for the other class of equations is easy to verify by means of changing the parameters in the main file. Such a test, whose results will not be reported here, has been successfully performed in the construction of the code.
The solution to Eqn.(21) has been discussed in detail in §(1.4.1). However, from the numerical point of view, it is important to point out how the presence of a BC implies a suitable modifications of the vector $\mathbf{f}$ to invert against. In fact, if the matrix A is constructed to possess J columns and J rows, since the discrete Laplacian and gradient operators output the value at the point x_j by means of computing functions at the point x_{j+1} and x_{j-1}, the values of the function $u(x)$ at the boundary of Ω_h have to be inserted into the first and last entry of the vector $\mathbf{f}$. Such a construction is performed in the function *Solver* trough the following lines:

```
f[0] = u0_*( ( alpha_/(h*h) ) + ( beta_/(2*h) ) ); //first boundary condition
f[J_-1] = uL_*( ( alpha_/(h*h) ) - ( beta_/(2*h) ) ); //second boundary condition
```

Eqn.(15) then reads:

$$\begin{pmatrix} \frac{\alpha}{h^2}+\gamma & -\frac{\alpha}{h^2}-\frac{\beta}{2h} & 0 & 0 & 0 & \cdots & 0 & 0 \\ -\frac{\alpha}{h^2}+\frac{\beta}{2h} & \frac{\alpha}{h^2}+\gamma & -\frac{\alpha}{h^2}-\frac{\beta}{2h} & 0 & 0 & \cdots & 0 & 0 \\ 0 & -\frac{\alpha}{h^2}+\frac{\beta}{2h} & \frac{\alpha}{h^2}+\gamma & -\frac{\alpha}{h^2}-\frac{\beta}{2h} & 0 & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \ddots & \ddots & \cdots & \vdots & \vdots \\ 0 & 0 & \cdots & \cdots & \cdots & \cdots & \frac{\alpha}{h^2}+\gamma & -\frac{\alpha}{h^2}-\frac{\beta}{2h} \end{pmatrix} \begin{pmatrix} u_1 \\ \vdots \\ \\ \vdots \\ u_{J-1} \end{pmatrix} = \begin{pmatrix} u_0\left(-\frac{\alpha}{h^2}+\frac{\beta}{2h}\right) \\ 0 \\ \vdots \\ 0 \\ u_L\left(-\frac{\alpha}{h^2}-\frac{\beta}{2h}\right) \end{pmatrix}$$

where $\mathbf{u}$ is the initial guess required by the Gauss-Seidel algorithm to work. In the present case, each component of $\mathbf{u}$ has been set to 0, so that $u_j = 0 \quad \forall j \in \{1, \cdots, J\}$.
Solutions to Eqn.(21) have been obtained for different values of the mesh size $h = \frac{1}{J+1}$ and different values of the $\mathbb{P}_e$. The error, i.e the distance between the exact solution on the grid and the approximate one in the L_∞ norm, is computed for each solution. As previously mentioned, the

ratio between such errors for two successive solutions of the same problem (same $\mathbb{P}_e$, but different h) have been computed: the reason for this will be clear in the next sections.

2.4 Results of the tests

In the following, the results of the more significant tests performed will be reported and briefly discussed. The tolerance has been set to 10^{-6} for each of the tests, while the values of *itCheck* and *MaxIter* are kept fixed to 10^4 and 10^9 respectively.

2.4.1 Solutions to the AD equation

The *ADR_test* function has been shown to work correctly for the following Péclet numbers:

$$\mathbb{P}_e = \{0.0000, 0.0005, 0.0100, 1.0000, 10.5000\}$$

The solutions $u(x)$ obtained for the different cases, indexed by the value of J chosen, can be found in the files *P_numb_..._Solution_J_....txt* , where the dots stand for the different possible values of $\mathbb{P}_e$ and J. The residual error at every iteration, in the L_∞ norm, is stored in the files *P_numb_..._Residual_J_....txt*, which provide a direct proof of the general correctness of the code. Finally, the distance between the exact solution and the approximate one, in the L_∞ is printed on *Errors_P_numb_...txt* files. Unsurprisingly, the error becomes smaller as the mesh size decreases.
The correctness of the algorithm has been tested for different values of h, as required by the assignment. The values of the mesh size have been chosen to be:

$$h = \{10, 20, 40, 80, 160, 320, 640\}$$

Keeping the Péclet number fixed, the same equation has been solved for all the previous mesh sizes. The results are reported in the following.

2.4.2 Small Péclet numbers

Solutions to to Eqn.(21) have been studied in the case of small $\mathbb{P}_e$, as required by the assignment. The obtained results are reported in the following plots.

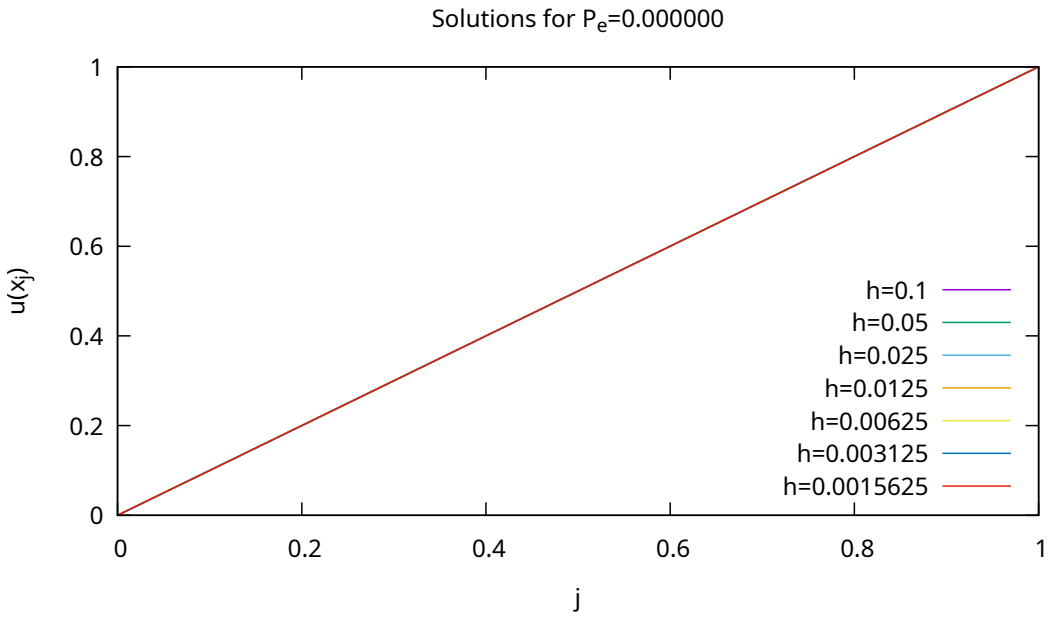

Figure 1: Graphical representation of the solution to Eqn.(21) for $\alpha = 1.0$, $\beta = 0.0$. The solutions for different mesh sizes overlap, so that only the last one is distinguishable.

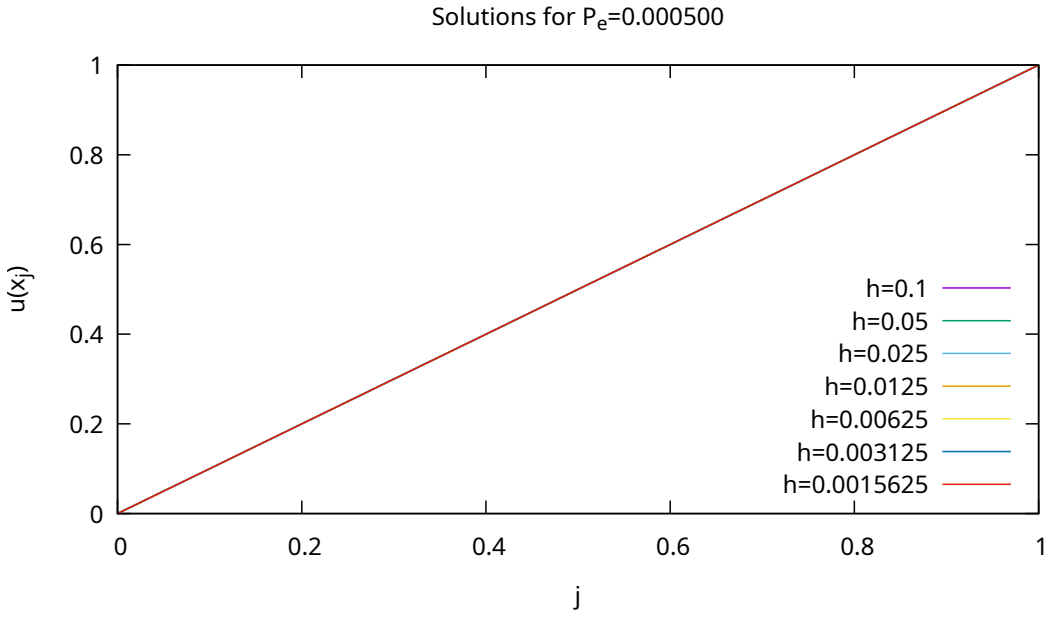

Figure 2: Graphical representation of the solution to Eqn.(21) for $\alpha = 1000$, $\beta = 1$. The solutions for different mesh sizes overlap, so that only the last one is distinguishable.

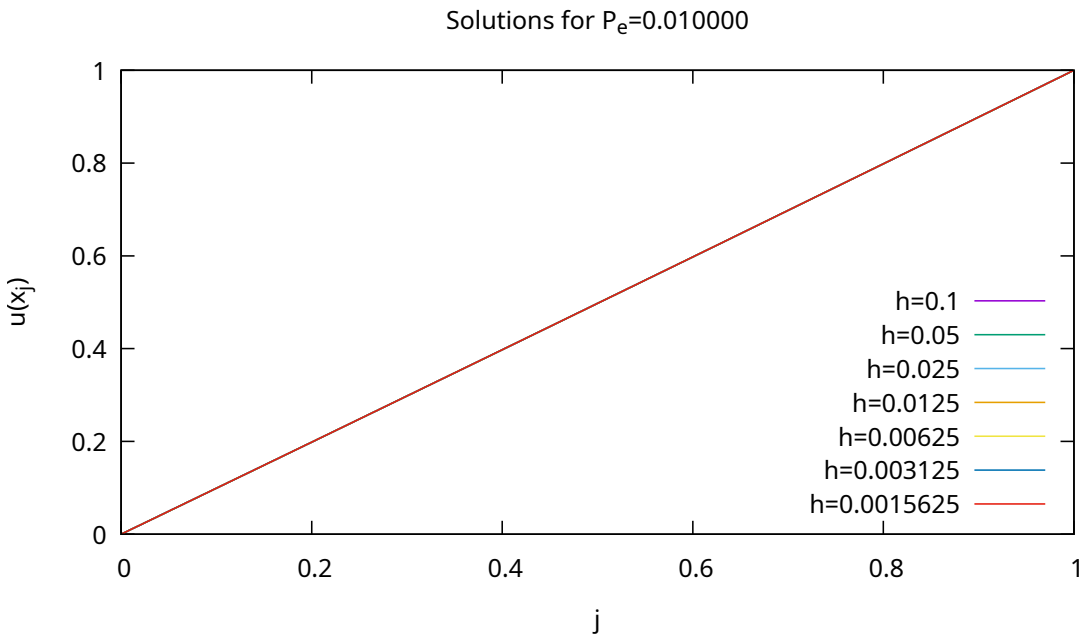

Figure 3: Graphical representation of the solution to Eqn.(21) for $\alpha = 10.0$, $\beta = 0.1$. The solutions for different mesh sizes overlap, so that only the last one is distinguishable.

The solutions appear to be straight line on the interval $[0, 1]$. Remembering what has been discussed in §(1.4.1), such a result does not appear surprising: Fig.2 and Fig.3 show the rightfulness of the Taylor expansion performed in Eqn.(18). Finally, Fig.1 shows that the code is able to find the correct solution to the problem also in the case of $\beta = 0$. In fact, under these circumstances, Eqn.(21) simplifies to:

$$-\alpha u''(x) = 0$$

which, for the given BCs, admits the unique solution $u(x) = x$.

2.4.3 Péclet number close or equal to 1

Solutions to Eqn.(21) have been studied in the case $\alpha = 0.25, \beta = 0.5$ as well. Such a choice of the parameters results in $\mathbb{P}_e = 1$. The result is reported in the following plot.

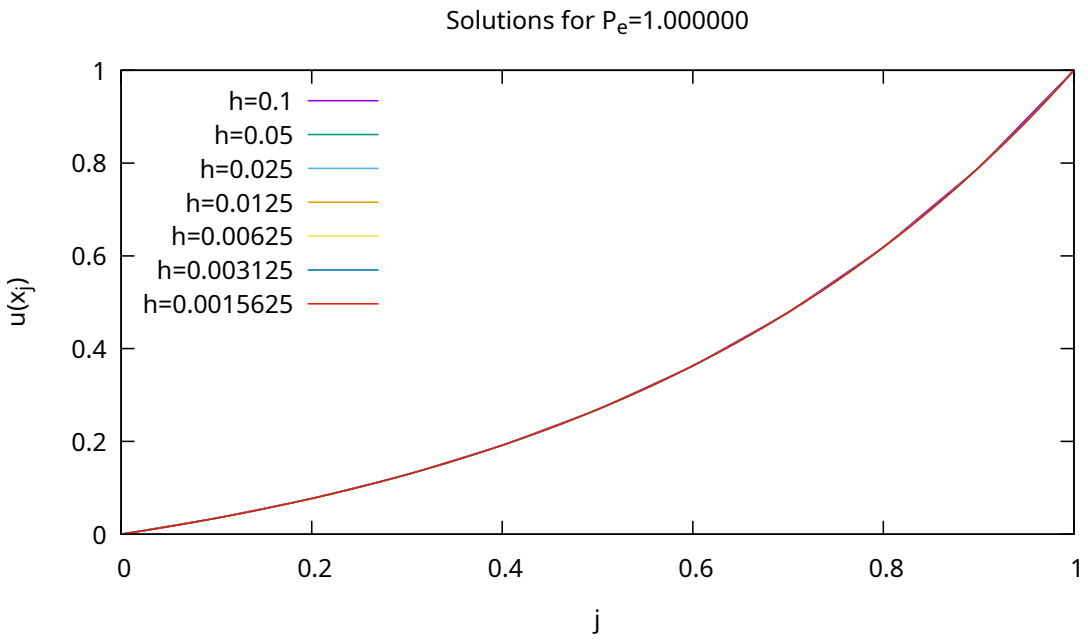

Figure 4: Graphical representation of the solution to Eqn.(21) for $\alpha = 0.25$, $\beta = 0.5$.

The exponential behaviour of the solution is now evident. In fact, in the present case, the Taylor expansion of the exponential function around 0 cannot be truncated after the first term being that $\frac{\beta}{\alpha} = 1$.
Solutions overlap in this case as well, but small differences between them are noticeable for big enough j: by means of a suitable zoom on the plot, it is possible to see that the red and blue curve present a slightly different profile.

2.4.4 Large Péclet number

Finally, solutions to Eqn.(21) have been studied in the case of large $\mathbb{P}_e$. The result is reported in the following plot.

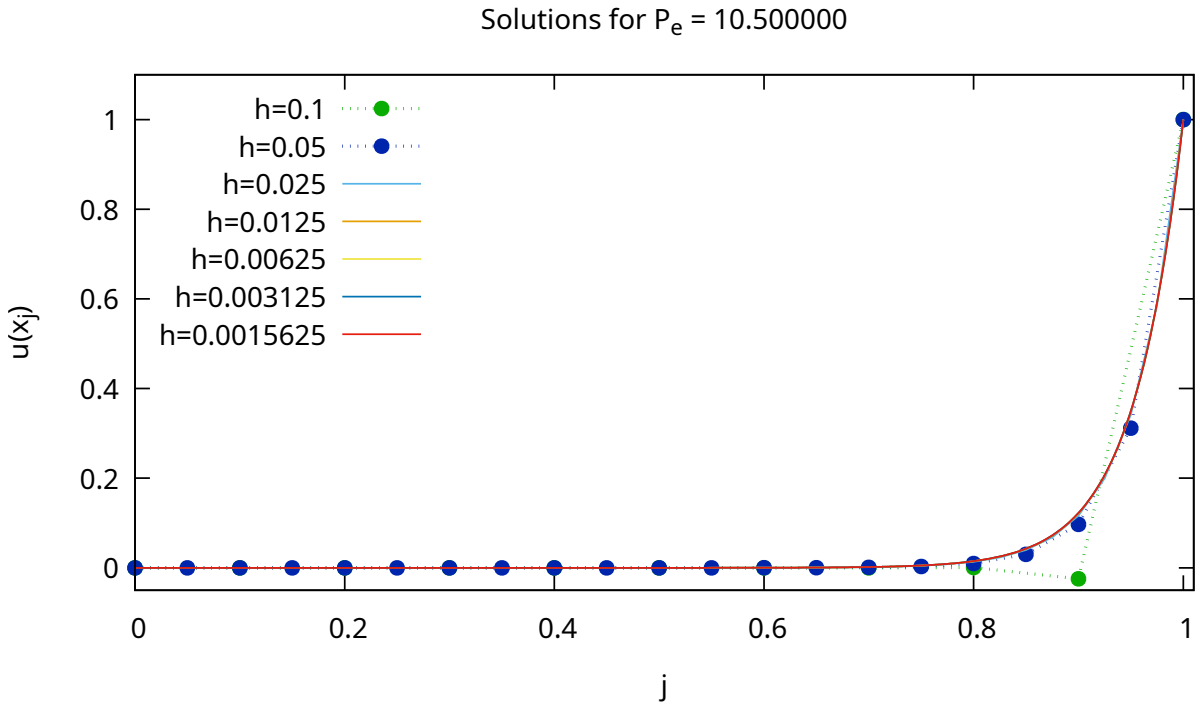

Figure 5: Graphical representation of the solution to Eqn.(21) for $\alpha = 1.0$, $\beta = 21.0$. The solutions overlap for $h \geq 0.025$.

The choice of the values of $\alpha$and β has been made through the following criterion.
A sufficient condition for the Gauss-Seidel method to converge is that the matrix A which the algorithm has to invert is strictly diagonally dominant (see [1]). In the present case, such a condition is satisfied if:

$$\left\| \frac{2\alpha}{h^2} \right\| > \left\| \frac{\alpha}{h^2} + \frac{\beta}{2h} \right\| + \left\| -\frac{2\alpha}{h^2} + \frac{\beta}{2h} \right\|$$

However:

$$\left\| \frac{\beta}{h} \right\| = \left\| -\frac{\alpha}{h^2} + \frac{\beta}{2h} + \frac{\alpha}{h^2} + \frac{\beta}{2h} \right\| \leq$$

So that, in order to have strict diagonally dominance, it must be:

$$\left\| \frac{\beta}{h} \right\| < \left\| \frac{2\alpha}{h^2} \right\|$$

Hence, for $L = 1$, remembering that $h = \frac{1}{J+1}$, one has that a sufficient condition for the Gauss-Seidel method to converge is:

$$J > \mathbb{P}_e - 1$$

The choice of $\alpha = 1$ and $\beta = 21$ satisfy the above condition. It was explicitly checked during the construction of the program that the choice of a great β greater (e.g. $\beta = 1000$) does not allow the algorithm to converge and the program exits with an appropriate warning to the user. Clearly, large values of β can be tested with large enough J, so that the previous inequality is satisfied.

2.5 Error analysis and order of convergence

As required by the assignment, errors on the solution have been computed and analysed. Not surprisingly, the distance between analytical and the approximate solution becomes smaller when the mesh size decreases.
The study of such errors becomes of great importance to determine the order of convergence of the method. Assume the FD method here used is consistent of order p, then:

$$\left\| \frac{\mathcal{L}_h[r_h u] - f_h}{\mathcal{L}_{\frac{h}{2}}\left[r_{\frac{h}{2}} u\right] - f_{\frac{h}{2}}} \right\| = \frac{Ch^p + \mathcal{O}(h^{p+1})}{C\left(\frac{h}{2}\right)^p + \mathcal{O}(h)^{p+1}} = 2^p$$

where $\mathcal{O}(h^{p+1})$ terms were neglected.

3 Conclusive remarks

In this section some additional comments and observation are presented.

3.1 Memory

When dynamically allocating memory, one has always to make sure to free the reserved locations as soon as they are no longer needed. With the help of the software *valgrind*, possible memory leaks have been checked. To do this, is it necessary to compile with *-g* and *-O1* optimisation. After having compiled the program, the following instruction has been used:

```
$ valgrind --leak-check=yes ./sparsematrix
```

The following output has been produced on the terminal:

```
==4200== Memcheck, a memory error detector
==4200== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4200== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==4200== Command: ./sparsematrix
==4200==
==4200==
==4200== HEAP SUMMARY:
==4200==     in use at exit: 0 bytes in 0 blocks
==4200==   total heap usage: 1,252,428 allocs, 1,252,428 frees, 989,616,540 bytes allocated
==4200==
==4200== All heap blocks were freed -- no leaks are possible
```

```
==4200==
==4200== For counts of detected and suppressed errors, rerun with: -v
==4200== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

One can clearly see that no memory has been lost, so that everything has been deleted correctly. It may seem a little strange at fist that all the heap blocks were freed, even though no explicit call to the destructor has been made. However, all instances are built by the function Gauss_Seidel_test, so that the deletion is automatically performed every time the programs exits the function.
The absence of memory leaks is however non trivial, since it proves that the destructor has been constructed correctly. This can be explicitly checked by means of commenting out the last two lines of the destructor: if *valgrind* is called again, some memory leak will be found.

3.2 Performances

In order to test performance, the following instructions have been used[1]:

```
$ valgrind --tool=callgrind ./sparsematrix
$ kcachegrind
```

The first command analyses the perfomances in the terms of load distrubution, while *kcachegrind* is used to visualise the load distribution results. Results are reported in the following screenshot.
It is clear that a better efficiency could have been obtained if the program had not relied so much on the *getValue* function, which is not particularly efficient.

3.2.1 Possible solutions

A possible way to optimise the code is to sort the indexes and values as soon as they are entered, so that each index can be associated to the corresponding value in a unique way and without the necessity to call a the *getValue*. Such a solution, however, has not been implemented, since even though it is clearly more efficient for the matrix given in the assignment, it is not immediately clear how faster and optimised the code could become in the case of denser sparse matrix. On the other hand, it is still reasonable to expect that such a choice will make the program faster, since the sorting takes place only at the beginning of the program and *getValue* is called a large number of times during the iterations.
Possibly, the best solution is to store the values and the indexes by means of the *insert* function instead that through a *push_back*: this way, the values will be automatically sorted.

[1]The test has been performed only on the part of the code concerned with the test over different λ and δ, commenting out the part on different N in the main file.

References

[1] A. Quateroni, R. Sacco, F. Saleri; *Numerical Mathematics*, Vol.37, Springer Verlag, (2007).

[2] T. Grafke; *Scientific Computing*, Lecture Notes, University of Warwick, (2018).