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

Student Number: 1894945

November 15, 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:

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

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:

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

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:

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

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

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

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* 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. The problem can be formulated as follows. Given $f \in C([0, L])$, find $u \in C^2(0, L)$ such that:

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

for $a, b, p, q \in \mathbb{R}$. In particular, in the present case, $f = 0$, $p = \frac{\beta}{\alpha}$, $q = \frac{\gamma}{\alpha}$ (for $\alpha, \beta, \gamma \in \mathbb{R}$), $L = 1$, $a = 0$ and $b = 1$.

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

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

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:

$$a_{ij} = \begin{cases} -\frac{\alpha}{h^2} - \frac{\beta}{2h}, & \text{for } j = i-1 \\ \frac{\alpha}{h^2} + \gamma, & \text{for } j = i \\ -\frac{\alpha}{h^2} + \frac{\beta}{2h}, & \text{for } j = i+1 \\ 0, & \text{otherwise} \end{cases} \tag{14}$$

In this way, differential equations can be solved through the methods used in numerical linear algebra, as the *Gauss-Seidel* method.

2 Implementation

The program is built on the class ADR , which stands for Advection-Diffusion-Reaction. The class has 7 private variables: J, α, β, γ, L, u_0 and $u_L = 1$, that represent respectively the number of points in the interior of $[0, L]$, the parameters associated to the equation at hand, the length of the interval and the boundary condition at 0 and L.
Such a construction allows for the implementation of a general 1D ADR problem, since the private variables contained in the class fully characterise the aforementioned problem.
The matrix associated to the method is built by a ...

The program has been built with the aim to minimise the memory used to store the matrices. Hence, following the instructions given in the assignment, the STL container *vector* has been used to store both the matrix entries and the corresponding column indexes. However, instead of using simple vectors, vectors of pointers have been used. Such a solution provides a more efficient storing of both entries and indexes, as it can be directly seen by comparing the memory allocated by the program with the case in which simple vectors are used.
The initialisation of an instance SparseMatrix is implemented in the following way: the constructor builds two vectors of vectors of pointers, setting the length of the first containers to the size of the matrix, but not initialising the content of the nested containers and setting the size of the rows and columns of the matrix. When a value and an index need to be stored, the *addEntry* function is called and if the row at which the value has to be inserted is non-existing, it is dynamically created: both the value and the index are pushed in. If the row exists already, a simple push_back is performed.

2.1 Construction of the Gauss-Seidel function

The core of the whole program is clearly represented by the function performing the Gauss-Seidel algorithm. Such a function has been chosen to be a member function and has been built to accept 7 parameters: the initial guess x_0, the vector to invert against b, the tolerance required by the method (*tol*), the number of iterations after which a check for stagnation has to be made (*itCheck*), two files on which the residual error at every iteration and the solution are printed respectively, the maximum number of iteration the user is willing to wait for the algorithm to converge (*MaxIter*).
Looking at Eqn. (**??**), it is clear that the elements of x_0 can be overwritten as they are computed in the algorithm and only one storage vector is needed: x_0 has then been passed by reference to the

function, so that no copy of it has to be produced.
It is worth pointing out that the matrix entries needed for overwriting the elements of x_0 are obtained through the member function $getValue$.[1].
In case the matrix is not a square matrix and the sizes of x_0 and b do not equal the row and column size of the matrix, the program exits with an appropriate warning to the user. If this is not the case, the iterations start. The program runs until either the residual error becomes smaller than the given tolerance or the maximum number of iterations *MaxIter* has been reached. In order to make sure that the program has not stagnated, every *itCheck* iterations, the current residual error, in the L_∞ norm, is compared with the one computed and stored *itCheck* iterations before: if the residual has not decreased, the program exits with an appropriate stagnation warning to the user. An appropriate warning is printed on the terminal if the program exist due to having reached the maximum number of iterations the user has fixed.
The sudo code for the implementation of the algorithm can be found either in Ref.[2] or on Wikipedia and will not be reported here.

2.2 Testing correctness

In order to check the correctness of the implementation of the Gauss-Seidel algorithm, the tests requested by the assignment have been performed: for this reason, a non-member function Gauss_Seidel_test has been built. Such a function takes 6 arguments: the size of the matrix A that has to be initialised to perform the test, the parameters δ and λ necessary to the implementation of the same tests, the fixed tolerance (*tol*), the number of iterations after which a check for stagnation has to be made (*itCheck*), the maximum number of iteration the user is willing to wait for the algorithm to converge (*MaxIter*). The last 3 parameters will be used by the member function Gauss_Seidel previously introduced, within the function Gauss_Seidel_test.
The vectors x_0, b, ω and D and the parameter $a = 4(\delta - 1)$ necessary to construct the matrix A are initialised by this function. The entries of A are added by means of the member function *addEntry.*
The files on which the residual error at every iteration and the corresponding final solution are printed, are identified by the value of δ, the value of λ, by the strings *Residual* or *solution* and the size of the matrix with on the output files.
The function prints on the terminal the error between the exact and the found solution, indexing it through the value of λ, δ and the size N of matrix A.

2.3 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* (that can still be found commented out in the Makefile) 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 two different plotscripts.

[1]More comments on this choice can be found in §3.2

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.

2.4.1 Varying the matrix size

The Gauss_Seidel_test function has been shown to work for the case of $N = 100$, $N = 1000$ and $N = 10000$, N being the size of the matrix, as required by the assignment. The time T_{10000} required to reach convergence in the case $N = 10000$ has been explicitly computed trough the following instructions:

```
auto start = std::chrono::high_resolution_clock::now();

auto finish = std::chrono::high_resolution_clock::now();
std::chrono::duration<double> elapsed = finish - start;
std::cout<< "Elapsed time: " << elapsed.count() << "s\n";
```

and found to be $T_{10000} \approx 83$ seconds. The algorithm has thus been shown to work in quite a short time, so that the program can be regarded as being reasonably efficient.[2] The solutions x_0 obtained for the three different cases can be found in the files *d_1.000000_l_0.000000_GSsolution_100.txt*, *d_1.000000_l_0.000000_GSsolution_1000.txt* and *d_1.000000_l_0.000000_GSsolution_10000.txt*. The residual error at every iteration has been printed in corresponding *Residual* files.
Finally, the distance between the exact solution and the approximate one is printed on the terminal. Unsurprisingly, the error becomes bigger as the size of the matrix grows.

2.4.2 Varying δ

The performance of the algorithm has been tested for different values of δ, as required by the assignment. Having shown that the algorithm converges in a reasonably short time even in the case of $N = 10000$, the size of the matrix for this series of tests has been fixed to $N = 100$.
In order to investigate the change in performance for different order of magnitudes, 10 different values of δ, ranging from 1.0 to 10^5 have been used.
It is clear that the desired closeness to the exact solution, measured in terms of the residual error, is reached independently of how large the initial residual is. As expected, the residual error is a monotone decreasing function of the iterations.
For $\delta \geq 50$, a concave region in the residual error profile is noticeable, signalling a change in the "velocity" of convergence. Interestingly, such a region is particularly steep for $\delta = 500$ and an overlap with the curves associated with smaller δ is present.

Clearly, the number of iterations required to have a residual error smaller than the fixed tolerance grows with δ, i.e. the code becomes less efficient when δ grows. In other words, the spectral radius of the matrix $P^{-1}N$ (see §**??**) gets smaller for increasing δ.

[2]More comments on the efficiency of the code can be found in §3.2.

2.4.3 Varying λ

The performance of the algorithm has been tested for different values of λ, as required by the assignment. Having shown that the algorithm converges in a reasonably short time even in the case of $N = 10000$, the size of the matrix for this series of tests has been fixed to $N = 100$.
In order to investigate the change in performance for different order of magnitudes, 10 different values of λ, ranging from 1.0 to 10^5 have been used.
It is clear that the presence of λ changes the efficiency of the algorithm. In particular, the plot shows that for $\lambda \geq 10^3$, the desired convergence is reached in just two iterations.
This can be understood by recalling what has been mentioned in §**??** and §**??** about the decomposition of A into $P_G = D - E$ and $N_G = F$. The presence of $\lambda \in \mathbb{R}^+$ modifies the diagonal component of A and consequently the matrix D. The preconditioner matrix P thereby obtained is different from the one of the initial problem: as a matter of fact, for $\lambda \in \mathbb{R}^+$ the matrix A becomes strictly diagonal dominant. Differently from the case of δ, the spectral radius of $P^{-1}N$ becomes smaller and a faster convergence is obtained.

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[3]:

```
$ 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.

[3]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).