UNIVERSITÉ LIBRE DE BRUXELLES

ECOLE POLYTECHNIQUE DE BRUXELLES

Numerical Methods Project Report

MATH-H401 - Numerical Methods

Author :
STÉVINS Alexander

Academic year :
2023-2024

1 Introduction

The project was completed succesfully : all questions, from number 1 to number 4 were completed. In this report, the convergence plots and the performance of the program will be discussed, as well as the optimizations that were used and the features of the program. The code is available on the GitHub repository of the project.

2 Features

The program that was developped for the MATH-H401 course project solves the stationnary heat equation, with the boundary conditions of room number 23. The code supports resolving the equation with 4 different techniques : a two-grid method, a multigrid V-cycle method, a multigrid W-cycle method, and a flexible preconditioned conjugate gradient method (using either a V-cycle or a W-cycle as preconditioner).

To run the program, execute the `main` binary in the directory where the program files are installed after compiling the program using `make`.

The behavior of the program is controlled by modifying a few key variables in the main.c file. The solution method is chosen by (un)commenting the solution methods on lines 118 to 121. The most important variables are the following :

— `m` : controls the discretization of the problem. Due to the geometry of the room, this number must be of the form $m = 11n + 1$, where $n \in N$. In the case of the two-grid method, this is restricted to $m = 22n + 1$. In the case of the multigrid method, each restriction doubles the step size, meaning that this is further restricted to $m = 2^p \cdot q \cdot 11 + 1$, where $p \in N$ is the maximum recursion level of the multigrid method, and $q \in N$ is an extra factor : if we don't recurse all the way to the coarsest level possible, m doesn't need to be entirely expressed in powers of 2 (times 11).

— `max_recursion` : controls how many recursion levels the multigrid methods will go through, effectively controlling at which coarse level the direct solver will act. For the construction of the matrices necessary to a two-grid method, this parameter should be set to 0.

— `tolerance` : determines the threshold for the stopping criterion of the program. Once the residual of the computed solution goes below the `tolerance` parameter, the program stops applying the selected iterative method.

— `problem_size_iterations` : controls on how many problem sizes the program will iterate, starting from the lowest one possible (m_0 = 23). From then on, $m_n = 2m_{n-1} - 1$, $n \leq$ `problem_size_iterations`. If set to 0, the program will only solve the problem size it is

set to with m. Iterating over problem sizes was mainly useful for the performance benchmarks in section 4.

The code is structured in files that contain functions that are called from the `main()` function. The main functions are the following :

— `void *generate_multigrid_problem(int max_recursion, int m, int **ia_ptr, int **ja_ptr, double **a_ptr, double **b_ptr)`

generates all of the a, ia, ja arrays necessarry to describe the multigrid problem, as well as the LU factorization of the coarsest level problem on which the UMFPACK solver will act. A pointer to each of the different a arrays is then stored in the `a_ptr array`, and so forth for the other arrays that describe the system in CSR format. Using this function with `max_recursion` equal to 0 will generate the matrices necessary to a two-grid problem (which is consistent with the fact that a v-cycle with no recursion is equivalent to a two-grid method). Computing all of this in a separate function which is only called once makes the code much faster, as it eliminates the need for reconstructing these matrices using `prob` or factorizing the same matrix over and over when calling `solve_umfpack`.

— `int two_grid_method(int n, int m, double L, int **ia_ptr, int **ja_ptr, double **a_ptr, double *b, double *x, void *Numeric)`

applies the two grid method to the solution vector x. Typically, this function is called in a loop until the residual is satisfactory.

— `int v_cycle(int max_recursion, int c, int n, int m, double L, int **ia_ptr, int **ja_ptr, double **a_ptr, double *b, double *x, void *Numeric)`

applies the v-cycle multigrid method to the solution vector x. The v-cycle is implemented recursively.

— `int w_cycle(int max_recursion, int c, int n, int m, double L, int **ia_ptr, int **ja_ptr, double **a_ptr, double *b, double *x, void *Numeric)`

applies the w-cycle multigrid method to the solution vector x. The w-cycle is implemented recursively.

— `int flexible_cg(int max_recursion, int n, int m, double L, int **ia_ptr, int **ja_ptr, double **a_ptr, double *b, double *x, double *r, double *d, void *Numeric, int v_w)`

applies the flexible preconditioned conjugate gradient method to the solution vector x.

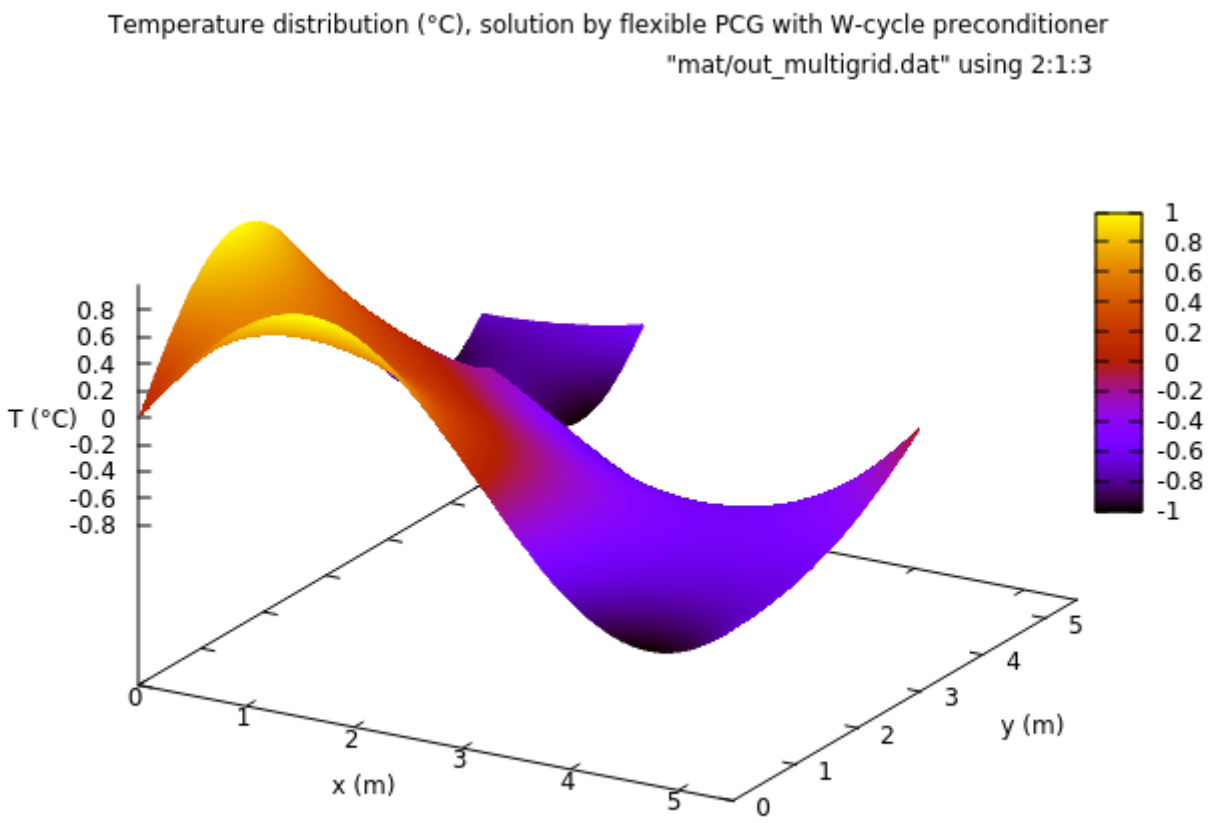

FIGURE 1 – Plot of the solution of the problem (at m = 705) obtained by 17 iterations of the flexible preconditioned conjugate gradient algorithm, using a W-cycle as a preconditioner. All of the methods produce plots that are identical to the naked eye.

The code only features a flexible preconditioned conjugate gradient method. A standard version is available in the code, however some error that wasn't debugged in time makes it converge very slowly (over 150 iterations for the relative residual to reach 10^{-14}). The flexible version version seems to converge a bit slower (17 iterations for this implementation versus 12 for theirs) than the implementations of other students following this course, which seems to indicate that there is a slight error in the implementation of the algorithm, however it was not found in time for the project deadline.

3 Convergence of the methods

The plots of the evolution of the residual of the method for m = 2817 (so 5302785 unknowns) are shown below in figures 2, 3, 4, 5 and 6, with the number of iterations of the method on the x-axis, and the residual after the iteration (in semi-logarithmic scale) on the y-axis. The multigrid methods were used with the maximum recursion level possible (7 recursion levels in this case). The durations given in the captions of the figures measure the total execution time, also taking into account the

time necessary for constructing and storing all of the system matrices, as well as executing the `umfpack_di_numeric` routine for the coarsest system and storing the resulting LU factorization in the Numeric variable.

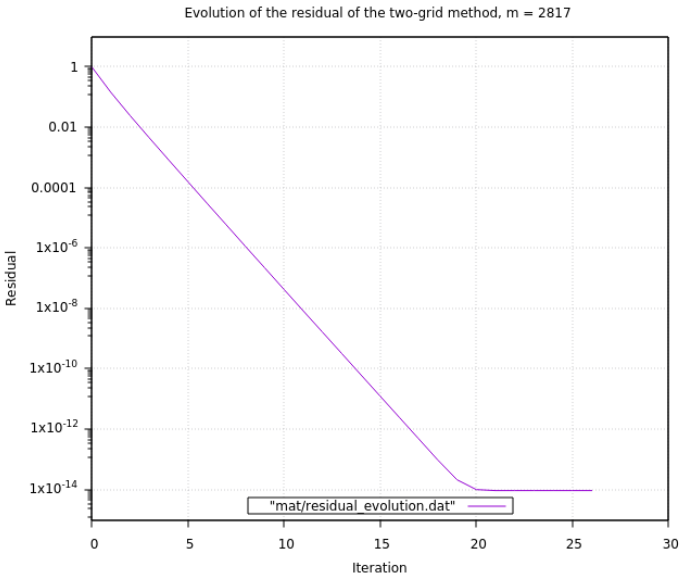

FIGURE 2 – Evolution of the residual of the two-grid method for m = 2817. Final residual : 9.2858641981e-15, achieved in 21 iterations, lasting for 48.48 s total. Most of the time is taken by the LU factorization, as the 21 iterations only take 10.12 s.

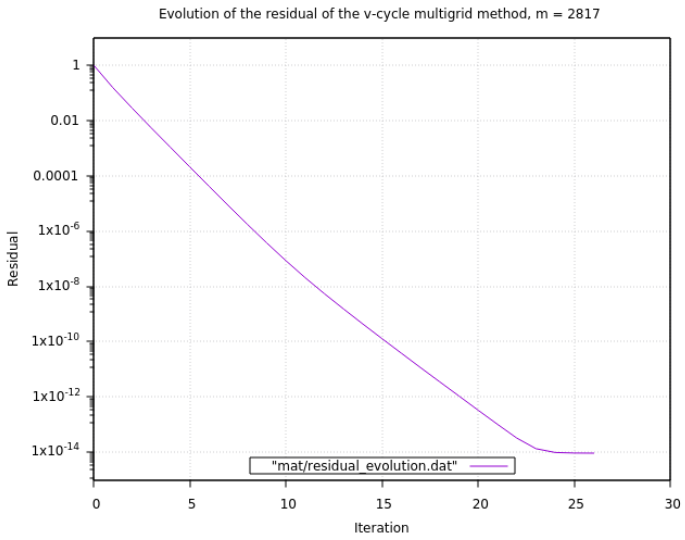

FIGURE 3 – Evolution of the residual of the v-cycle multigrid method for m = 2817. Final residual : 9.2873476759e-15, achieved in 25 iterations, lasting for 8.08 s total.

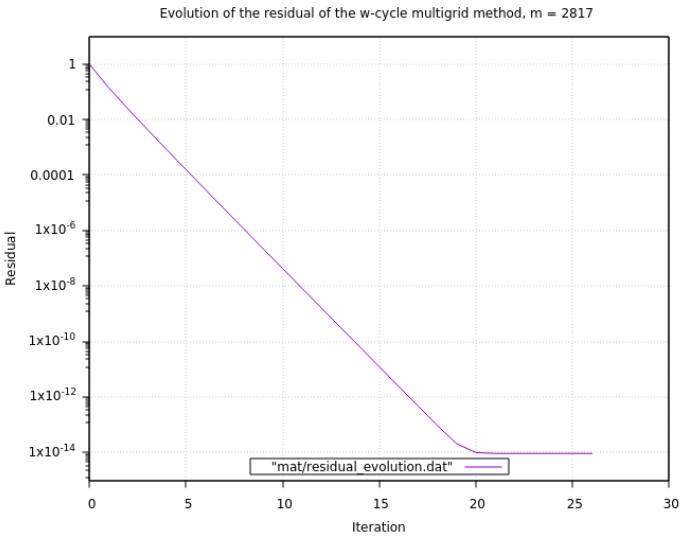

FIGURE 4 – Evolution of the residual of the w-cycle multigrid method for m = 2817. Final residual : 9.2522054500e-15, achieved in 21 iterations, lasting 9.92 s total.

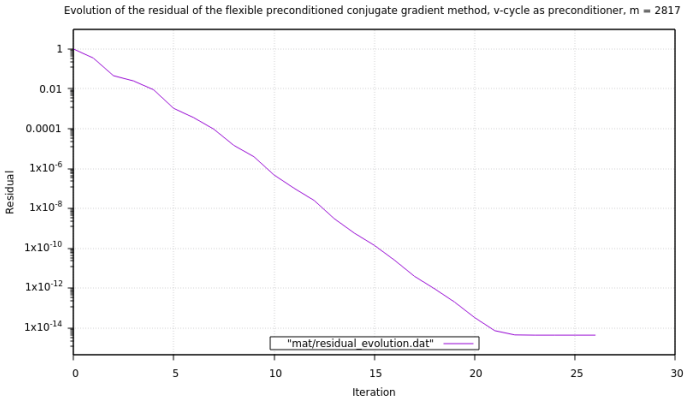

FIGURE 5 – Evolution of the residual of the flexible preconditioned conjugate gradient with a v-cycle multigrid preconditioner for m = 2817. Final residual : 4.3488457608e-15, achieved in 23 iterations, lasting 12.55 s total.

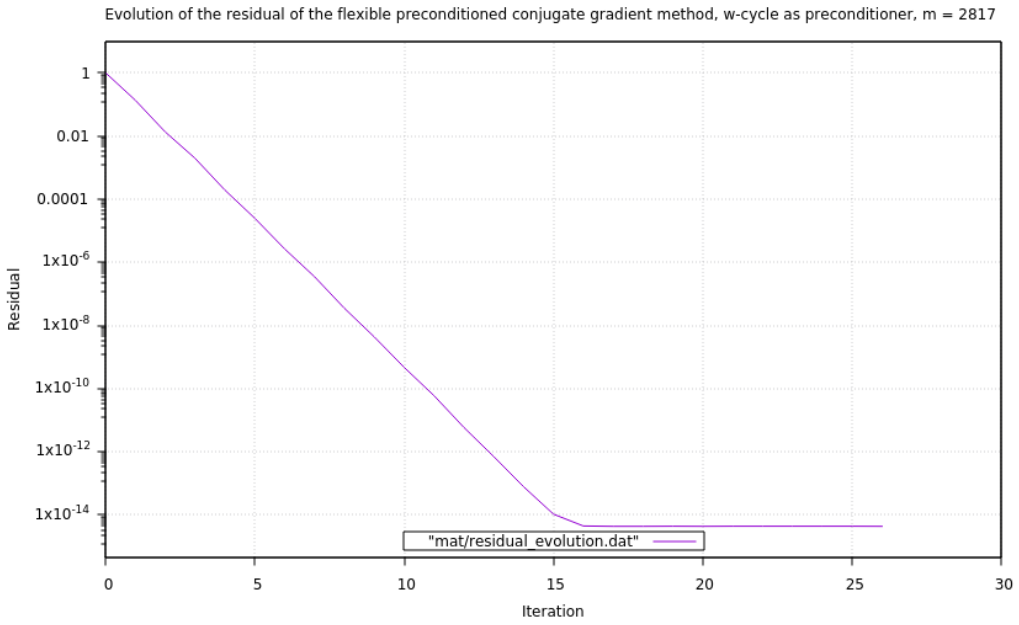

FIGURE 6 – Evolution of the residual of the flexible preconditioned conjugate gradient method with a w-cycle multigrid preconditioner for m = 2817. Final residual : 4.3411661713e-15, achieved in 17 iterations, lasting 12.13 s total.

4 Performance

The code's performance is reasonablee, averaging less than 2 seconds resolution time per million of unknows for the V-cycle method, a little over 2 seconds per million unknowns for the W-cycle and the flexible preconditioned conjugate gradient with a W-cycle preconditioner, and a little over 3 seconds per million unknowns for the flexible preconditioned conjugate gradient with a V-cycle as preconditioner. For reference, plots of the evolution time of the solution times in function of the number of unknowns for each of these methods can be found on figures 7, 8, 9 and 10. The two-grid method is not shown here because it does not scale as well with problem size (probably because it needs to factorize a much larger system matrix than the others).

These execution times were achieved by passing objects by reference to functions (only pointers are copied), and not recomputing all of the matrices at every stage of the two-grid and multigrid methods : they are computed once at the start of the program and reused every time. This is also true for the numeric factorization of the system matrix done by UMFPACK stored in the Numeric variable in the code : Numeric is stored in a variable in the scope of the `main` function and passed as argument to the functions that need to solve the coarsest system. The implementation of the Gauss-Seidel algorithm is also particularised to the problem, eliminating the need for an if check to find the diagonal element at every iteration when iterating over the matrix elements and unknowns to be added to the sum.

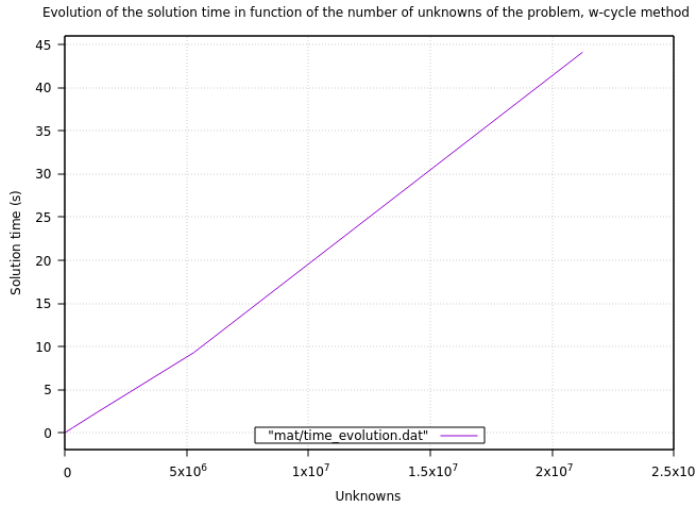

FIGURE 7 – Evolution of the solution time for the W-cycle method in function of the number of unknowns.

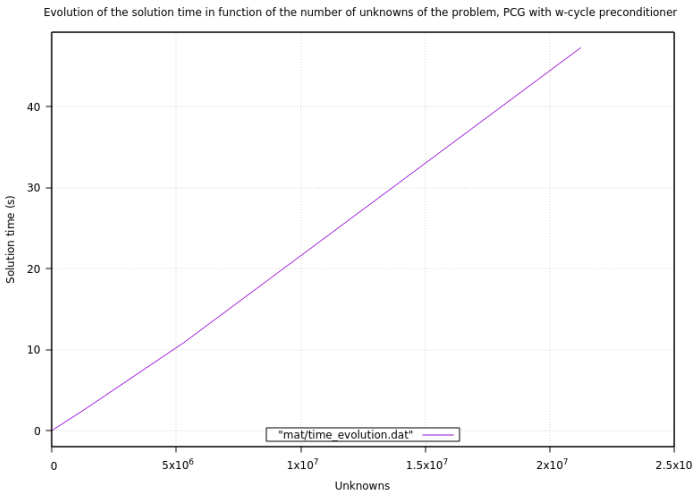

FIGURE 8 – Evolution of the solution time for the preconditioned conjugate gradient method, with a W-cycle as preconditoner.

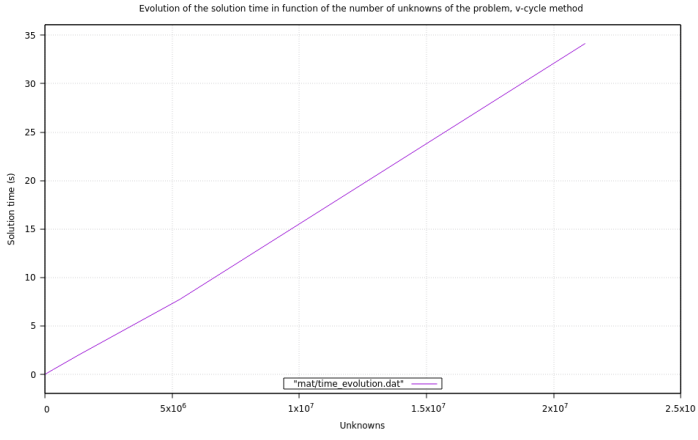

FIGURE 9 – Evolution of the solution time for the V-cycle method in function of the number of unknowns.

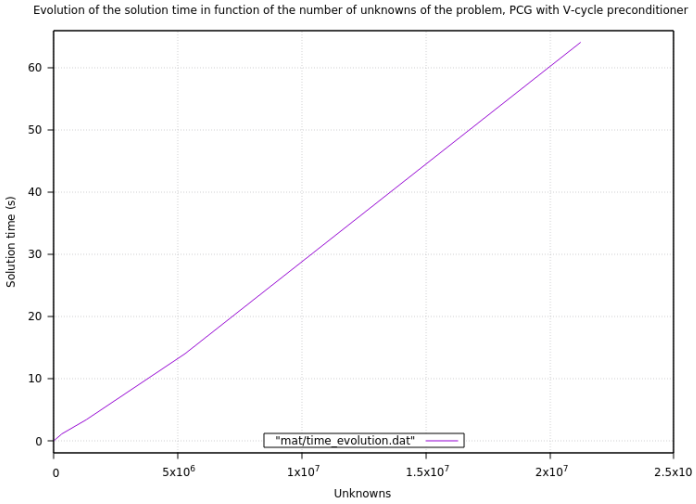

FIGURE 10 – Evolution of the solution time for the preconditioned conjugate gradient method, with a V-cycle as preconditoner.