Project C: Two-dimensional Poisson equation and the method of manufactured solutions

Morteza Namvar

November 10, 2021

Contents

1 Introduction **3**

2 Governing equations **3**

3 Numerical solution **4**

4 Implementation approach **4**

4.1 Class:*Grid* 5

4.2 Class:*Field* 5

4.3 Class:*PoisonSolver* 6

4.4 Implemented classes to solve System of equations 6

4.4.1 Class:*DenseSolver* 7

4.4.2 Class:*JacobiSolver* 8

5 Results and discussion **9**

5.1 Verification 9

5.2 Order of accuracy 11

5.3 Performance analysis 12

5.4 Memory usage analysis 14

6 Conclusion **15**

1 Introduction

Poisson's equation is an elliptic partial differential equation with many applications in mathematics and numerical methods. For example, this equation is widely used in unstructured grid generation. In this project, the numerical solutions of this equation are implemented. In many cases, we need just a rough estimation of this equation, but in other cases, for example, plasma actuator simulation, we need a more accurate solution. So in this project, we will use an implicit method that is more efficient in terms of execution time. However, the main problem of this approach is memory consumption. Another challenge of any implicit method is solving a system of equations which is time-consuming. In this project, we will use the ArrayFire [1] library for solving the system of equations.

2 Governing equations

The two-dimensional Poisson equation is in the form of:

$$u_{xx} + u_{yy} = f(x, y) \tag{1}$$

Like any other PDE, to solve this equation, analytically or numerically, it is required to define a domain and the boundary condition values at that domain. In this project, we will use the method of manufactured solutions [2] to generate an analytical solution for verification purposes of the implemented code. So, if the analytical solution was chosen as:

$$f(x, y) = sin(x) + cos(y) \tag{2}$$

Then the right hand side of Eq. 1 constructed as:

$$f(x, y) = -sin(x) - cos(y) \tag{3}$$

and the boundary condition based on the analytical solution is:

$$\begin{cases} u(x_{min}, y) = sin(x_{min}) + cos(y) \\ u(x_{max}, y) = sin(x_{max}) + cos(y) \\ u(x, y_{min}) = sin(x) + cos(y_{min}) \\ u(x, y_{max}) = sin(x) + cos(y_{max}) \end{cases} \tag{4}$$

where the domain is defined in such a way that $(x, y) \in [x_{min}, x_{max}] \times [y_{min}, y_{max}]$.

3 Numerical solution

First of all in this project we use the finite-difference method (FDM) for numerical modeling. So, the following notation is used to show the discretized position:

$$\begin{cases} x_i = x_{min} + i\Delta x \\ y_j = y_{min} + j\Delta y \end{cases} \tag{5}$$

here i and j are the steps for x, and y respectively.

In this project, for spatial discretization, a second-order centered finite difference scheme is used by exploiting the stencil shown in Fig. 1.

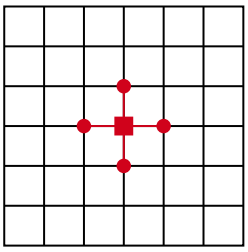


Figure 1: The stencil used for spatial discretization by second order scheme.

So, the discretized form of Eq. 1 would be as:

$$\frac{u_{i+1,j} - 2u_{i,j} + u_{i-1,j}}{\Delta x^2} + \frac{u_{i,j+1} - 2u_{i,j} + u_{i,j-1}}{\Delta y^2} = f(x, y) \tag{6}$$

After some algebraic operations this equation would be as:

$$\Delta y^2 \left(u_{i+1,j} - 2u_{i,j} + u_{i-1,j}\right) + \Delta x^2 \left(u_{i,j+1} - 2u_{i,j} + u_{i,j-1}\right) = \Delta y^2 \Delta x^2 f(x, y) \tag{7}$$

To solve the Eq. 7 by implicit approach, it assumed all the variables *i.e.* u are unknowns for all of the non-boundaries nodes. In ref. [3], constructing the coefficient and right-hand-side of the above equation are explained perfectly. So, in the next section, we just explain how to program this method by using object-oriented programming and the ArrayFire library for parallel programming.

4 Implementation approach

In this implementation, we used the object-oriented programming (OOP) paradigm to enhance the capability of the code to implement and test a different approach. For programming in the paradigm, the best practice is to create different characters (classes) with suitable attributes(functions or

methods) and finally create a program by composing these characters. the class diagram illustrated in Fig. 2 shows the based code. Let's explain these classes briefly in the following sections.

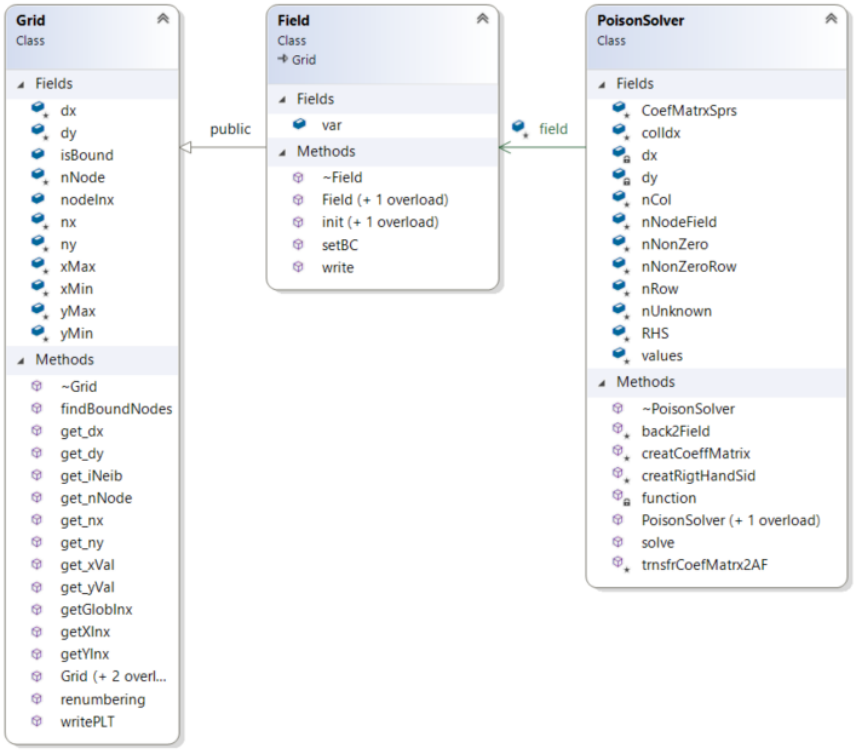


Figure 2: Class diagram of the implemented code.

4.1 Class: *Grid*

The logic to develop this class is: we want to solve a PDE numerically. So, we need a class for spatial discretization. But the first step for designing this class is to ask ourselves which characteristics this class should have? We can answer this question to design the class. For example, any domain has limits, it has some nodes, the spacing in x and y direction should be defined, any nodes have a type: boundary or non-boundary, any nodes have an index. Now we can develop some methods to set and get these variables outside the class. In addition, we can define some actions for this class to do some tasks *i.e. methods*. For example for the class *Grid* can plot itself.

4.2 Class: *Field*

If we want to define a field in the discrete space, we can use the class *Grid*. So, we can use the OOP capabilities and create a class by deriving the class *Grid* as depicted in Fig. 2. This class just has

a variable that stores the values at each node of the grid. As illustrated in Fig. 2 the filed variable could be initialized or the boundary condition could be set on the domain.

4.3 Class:*PoisonSolver*

Now we can design a class to solve a PDE by using the class *Field*. For example in Fig. 2 we designed a class for Poison's equation. In this implementation, this class has the following methods:

- *PoisonSolver(Field)*: A constructor which get an object of class *Field*.
- *double function(double, double)*: the function which is the right-hand-side of Poison's equation.
- *void creatCoeffMatrix()*: to create the coefficient matrix in implicit method.
- *void creatRigtHandSid()*: to create the right-hand-side in implicit method.
- *void trnsfrCoefMatrx2AF()*: to transfer the coefficient matrix to the arrayfire library.
- *void back2Field(af::array)*: to transfer the results to the main variable of class *Field*.
- *virtual void solve()*: this method is a virtual method that should be implemented to have a different solution for Poison's equation such as LU, Jacobi, and SOR.

Since the coefficient matrix is big in the field of CFD, we need to use efficiently the memory. Besides, we know this matrix is a sparse one. So, storing this matrix in a sparse shape is one of the methods of using memory efficiently. Hence, in this project we are will store this matrix in CSR [4] format.

4.4 Implemented classes to solve System of equations

As explained above, we had an abstract class that would be implemented to develop a solver for a system of equations. In this project, we implemented two classes including the ArrayFire solver and Jacobi method as illustrated in Fig. 3.

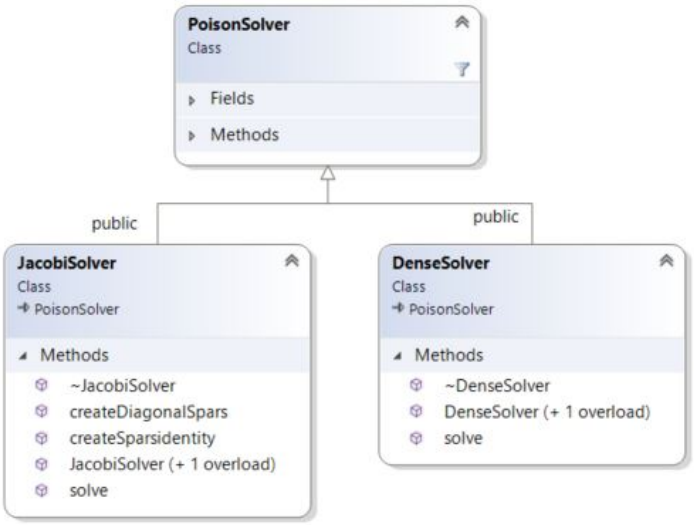


Figure 3: Class diagram of the extended solvers from *PoisonSolver* abstract class.

4.4.1 Class:*DenseSolver*

This class is a derived (child) class of *PoisonSolver*. So, here we just implement the *solve* method. As we store the coefficient matrix in sparse shape we need to convert this matrix to a dense shape to be able to use the solver of ArrayFire. The implemented code is shown in Listing 1.

Listing 1: The *solve* method which correspond to solve the system of equation.

```
1
2         //Creating RHS of Poison's equation
3         creatRigtHandSid();
4
5         //Creating coefficient matrix in naive C++
6         creatCoeffMatrix();
7
8         //Trasfering coefficient matrix to arayfire
9         trnsfrCoefMatrx2AF();
10
11        //Creatig a dense matrix (Note the coefficient matrix was in sparse shape)
12        af::array A = af::dense(CoefMatrxSprs);
13
14        //Trasfering RHS to arrayfire
15        af::array b = af::array(nUnknown, RHS);
16
17        //Free unnecessary allocated memory
18        delete[] RHS;
19
20        //solving the system of equations by using arrayfire library
21        af::array x = af::solve(A, b);
22
23        //Transfering data from the matrix to the discrete domain
24        back2Field(x);
25
```

The drawback of this method for solving a system of equations is the amount of required memory as we need to use a dense matrix. So, for the large-scale problem, this method would not be efficient in terms of memory usage.

4.4.2 Class:*JacobiSolver*

In numerical linear algebra, the Jacobi method is an iterative method for finding the solution of a system of linear equations. The concept of this method could be found in the literatures[5]. But, briefly the formula of this algorithm is:

$$\mathbf{x}^{k+1} = D^{-1}(\mathbf{b} - (L+U)\mathbf{x}^{k}) \tag{8}$$

where $\mathbf{x}^{(k)}$ is the kth approximation or iteration of $\mathbf{x}$ and $\mathbf{x}^{(k+1)}$ is the next or $k+1$ iteration of $\mathbf{x}$. The inverse of a diagonal matrix could be calculated by replacing each element in the diagonal with its reciprocal [6]. Note as we develop code by using ArrayFire and since there is not any function to create a sparse unity matrix, we develop a function to create that. The implemented code of the Jacobi method is shown in Listing 2.

Listing 2: The *solve* method of class *JacobiSolver* which correspond to solve the system of equation.

```
1
2         //Creating RHS of Poison's equation
3         creatRigtHandSid();
4
5         //Creating coefficient matrix in naive C++
6         creatCoeffMatrix();
7
8         //get the value of element on the diagonal
9         double diagonalVal = values[0];
10
11        //coefficient matrix to arayfire
12        trnsfrCoefMatrx2AF();
13
14        //Trasfering RHS to arrayfire
15        af::array b = af::array(nUnknown, RHS);
16
17        //Set the initial value in jacobi method (However, this values could be \\
18        //initilize more accurate for better convergence)
19        af::array x0 = b;
20
21        //Creating LU part
22        af::array I = createDiagonalSpars(diagonalVal, nUnknown);
23        af::array LU = CoefMatrxSprs - I;
24
25
26        //solving the system of equation by using arrayfire library and jacobi method
27        af::array x;
28        int it = 0;
29        double diff = 10e6;
30        while ( it < 10e5 || diff < 10e-4) {
31
32                x = 1 / diagonalVal * (b - af::matmul(LU, x0));
33
34                double diff = af::sum<double>(af::abs(x - x0));
35
36                x0 = x;
37
38                it++;
39        }
40
41        //Transfering data from the matrix to the discrete domain
42        back2Field(x);
43
```

Note on parallel programming: In parallel programming, the first step is to find the bottleneck of the code and then the attempt should be put on these parts to be paralleled to decrease the execution time. Since we are using ArrayFire for parallel programming we need to use this library at those parts of the code which need to be more efficient in terms of execution time. But, using ArrayFire or any other library in the whole code will reduce the portability of the code. Moreover,

imagine we want to use another library for parallel programming, if we use ArrayFire just at the appropriate parts we can easily develop new code independent of ArrayFire. So, in the implementation of this project, we used ArrayFire just to solve a system of equations, and in the other parts, we just used naive C++.

5 Results and discussion

In this section as the first step, the code is verified by comparing the results of the dense solver to the analytical solution. Then, the accuracy of the discretization method is investigated. To verify the spare matrix solver (Jacobi solver) just the *L2 error* is compared to the dense solver. After that, the performance of different solvers is studied. All of the simulations were executed with double-precision floating-point numbers. The computer used for the simulations was equipped with an Intel(R) Core(TM)i7-7700 @2.80GHz processor and a 16GB of main memory and the graphic card was GTX-1080 with 6GB of memory.

5.1 Verification

The contour of initial condition is depicted in Fig. 4. In this approach all the values in the domain are set as zero. But initially the boundary conditions are applied at each boundary. For verification, the analytical results are needed which is plotted in Fig. 5. These values are obtained from Eq. 2.

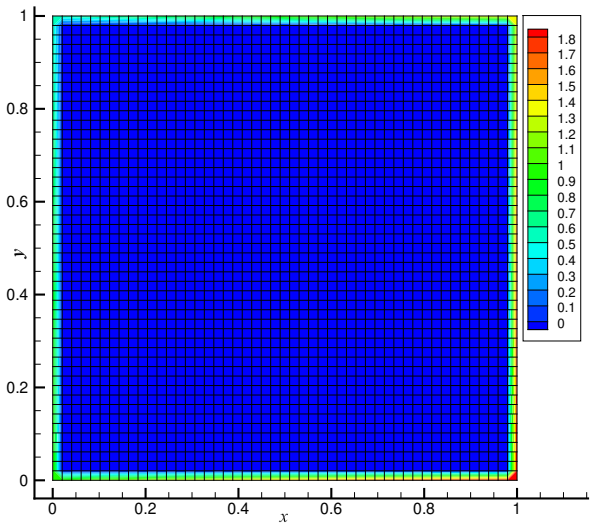
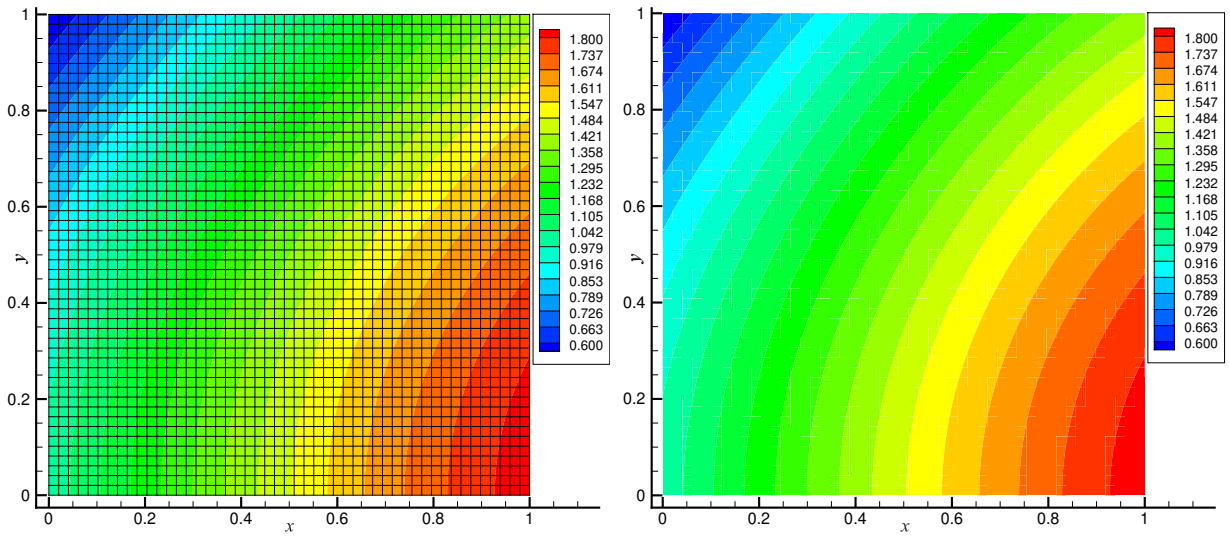


Figure 4: Contour of initial and boundary values.



(a) Numerical solution.

(b) The analytical solution based on Eq. 2.

Figure 5: Analytical and numerical solution for a grid of size (50×50).

To verify the implemented code, a set of test cases have been done in this project. As an example the results for a grid of size (50×50) is depicted in Fig. 5a. As shown in this picture the domain is limited to $x \in [0, 1]$ and $y \in [0, 1]$. But, to have a quantitative comparison, the numerical solution should be compared to the analytical solution. Here, the numerical and analytical solutions are obtained at the centerline of the domain at $x = 0.5$. As can be seen in Fig. 6 there is a good agreement to the analytical solution.

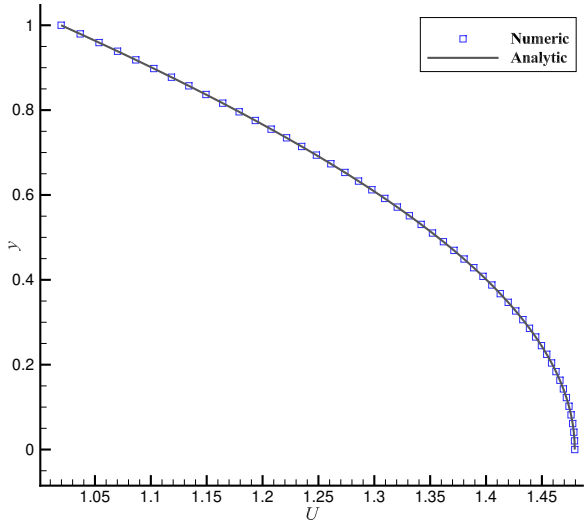


Figure 6: Comparison of analytical and numerical solution profile at $x = 0.5$.

However for further investigations a set of test cases provided in Table 1 are used. In the next

section these tests are used for studying the order of accuracy.

Table 1: Parameters of test cases.

Cases	n_x	n_y	x_{min}	x_{max}	y_{min}	y_{max}
1	5	5	0	1	0	1
2	10	10	0	1	0	1
3	20	20	0	1	0	1
4	40	40	0	1	0	1
5	80	80	0	1	0	1
5	7	6	-3	3π	3	4π
6	14	12	-3	3π	3	4π
7	28	24	-3	3π	3	4π
8	56	48	-3	3π	3	4π
9	112	96	-3	3π	3	4π
10	5	9	$-\pi$	2	-5π	3π
11	10	18	$-\pi$	2	-5π	3π
12	20	36	$-\pi$	2	-5π	3π
13	40	72	$-\pi$	2	-5π	3π
14	80	144	$-\pi$	2	-5π	3π

5.2 Order of accuracy

After verification, it is required to verify the order of accuracy of the method of discretization. To measure the error between the analytical and numerical implementations, the L2 norm is using according to the Eq. 9:

$$\text{L2error} \equiv ||u^n - u^a||_{L_2} = \sqrt{\frac{1}{Nx \times Ny} \sum_{i=0}^{Nx-1} \sum_{j=0}^{Ny-1} \left(u^n_{i,j} - u^a_{i,j}\right)^2} \tag{9}$$

where u^a and u^n represent analytical and numerical solution respectively. Also Nx and Ny are the number of nodes in $x-$ and $y-$direction.

As in this project, a "centered" approach has been used for spatial discretization, it is expected that the implemented code be second-order accuracy. Fig. 7 shows the trend of error as a function of grid resolution. Based on this table it is clear the order of accuracy is 2 as expected. As mentioned before, to verify the Jacobi solver, just the error compared to the dense solver. As shown in Fig. 7b the error is the same as the dense solver. For visualization, the order of accuracy is plotted in Fig. 8. In this figure, a nominal second order line is plotted for a better understanding of the order of accuracy.

Poisson 2D							Centered Scheme	
Cases	nx	ny	xmin	xmax	ymin	ymax	Error L2	Order
1	5	5	0	1	0	1	2.16E-04	
2	10	10	0	1	0	1	5.03E-05	2.11
3	20	20	0	1	0	1	1.20E-05	2.07
4	40	40	0	1	0	1	2.93E-06	2.03
5	80	80	0	1	0	1	7.20E-07	2.02
6	7	6	-3	3 pi	3	4 pi	3.67E-01	
7	14	12	-3	3 pi	3	4 pi	7.17E-02	2.36
8	28	24	-3	3 pi	3	4 pi	1.66E-02	2.11
9	56	48	-3	3 pi	3	4 pi	4.05E-03	2.04
10	112	96	-3	3 pi	3	4 pi	9.99E-04	2.02
11	5	9	-pi	2	-5 pi	3 pi	5.94E-01	
12	10	18	-pi	2	-5 pi	3 pi	8.38E-02	2.83
13	20	36	-pi	2	-5 pi	3 pi	1.95E-02	2.1
14	40	72	-pi	2	-5 pi	3 pi	4.76E-03	2.04
15	80	144	-pi	2	-5 pi	3 pi	1.18E-03	2.01

(a) Jacobi solver.

Poisson 2D							Centered Scheme	
Cases	nx	ny	xmin	xmax	ymin	ymax	Error L2	Order
1	5	5	0	1	0	1	2.16E-04	
2	10	10	0	1	0	1	5.03E-05	2.11
3	20	20	0	1	0	1	1.20E-05	2.07
4	40	40	0	1	0	1	2.93E-06	2.03
5	80	80	0	1	0	1	7.20E-07	2.02
6	7	6	-3	3 pi	3	4 pi	3.67E-01	
7	14	12	-3	3 pi	3	4 pi	7.17E-02	2.36
8	28	24	-3	3 pi	3	4 pi	1.66E-02	2.11
9	56	48	-3	3 pi	3	4 pi	4.05E-03	2.04
10	112	96	-3	3 pi	3	4 pi	9.99E-04	2.02
11	5	9	-pi	2	-5 pi	3 pi	5.94E-01	
12	10	18	-pi	2	-5 pi	3 pi	8.38E-02	2.83
13	20	36	-pi	2	-5 pi	3 pi	1.95E-02	2.1
14	40	72	-pi	2	-5 pi	3 pi	4.76E-03	2.04
15	80	144	-pi	2	-5 pi	3 pi	1.18E-03	2.01

(b) Dense solver.

Figure 7: L2 error obtained according to Eq. 9 for different test cases.

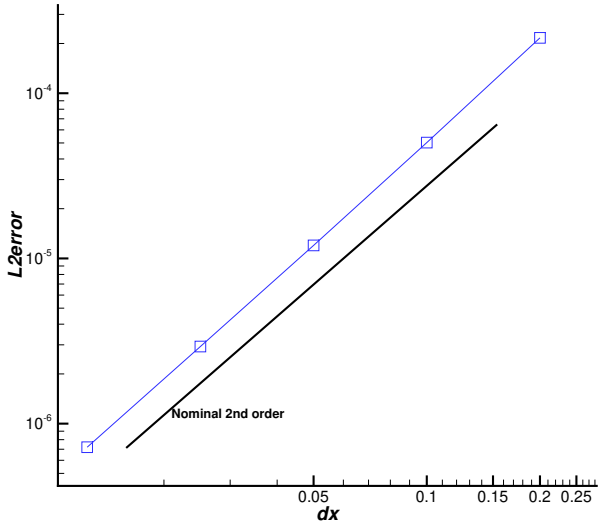


Figure 8: Plot of L2error for different grid size at $t = 0.01$.

5.3 Performance analysis

In this section, the execution time of different parts of the code is investigated. After that, the overall time to solve the Poison equation is studied.

As mentioned before, in the parallel programming field the first step is to find the bottleneck of the code. To do that there are plenty of tools, such as Vtune, to help programmers. Although

in this project I used Vtune, it is impossible to analyze the code as it is impossible to access the source code of ArrayFire in Vtune. So, I decided just to print the execution time for any parts of the program. As could be seen in this Table 2 the execution time for solving the Sys. of the equation for CUDA and OpenCL backend are considerably high compared to CPU backend.

Table 2: Execution time of different parts of the Dense solver. Time is in second.

Programming approach	**CPU**	**OpenCL**	**CUDA**
Creating right hand side of Eq. in Naive C++	0.0011968	—	—
Creating coefficient matrix in Naive C++	0.0007243	—	—
Transferring coefficient matrix to ArrayFire	0.0002931	0.0040418	0.0019466
Transferring right hand side to ArrayFire	0.182144	0.0005502	0.0003008
Transferring results to host	2.11079	0.0004493	0.0003437
Creating dense matrix from sparse on AF	0.0007495	0.0110339	0.550372
Solving dense Sys. of Eq. on ArrayFire	0.0008474	10.9655	11.2908

The overall execution time of the code is shown in Table 3. On the system with 6GB of graphic card, it is possible to solve a system of a maximum of 150 nodes by using *DensSolver*. As could be seen the execution time for OpenCL and CUDA is more than CPU for this solver. But for *JacabiSolver* as it is using the sparse function of ArrayFire, it is possible to solve large-scale systems. Based on the elapsed time provided in Table 3 the execution time for the OpenCL backend is much less than CPU. It should mention in Jacobi just 100 iterations have been done to provide the below table.

Table 3: Execution time of *DenseSolver* and *JacobiSolver*.

Solver	n_x	n_y	CPU	OpenCL	CUDA
DenseSolver	100	100	7.90132	15.9207	12.1161
DenseSolver	150	150	74.8714	didn't-run	didn't-run
JacobiSolver	100	100	1.26465	0.979789	—
JacobiSolver	150	150	2.77076	1.00735	—
JacobiSolver	200	200	1.36894	0.972371	—
JacobiSolver	400	400	8.43189	1.56236	—
JacobiSolver	1000	1000	44.0959	5.21885	—
JacobiSolver	2000	2000	175.531	18.5337	—
JacobiSolver	5000	5000	717.379	117.879	—

5.4 Memory usage analysis

After investigating the execution time, the memory usage should be studied. As mentioned before, the maximum grid which could be solved on the computer, used for these tests, is $n_x = n_y = 150$. So, we don't investigate this approach in terms of memory usage. As could be seen in Table 4 the memory consumption of the *JacabiSolver* is much better as it is using the sparse functions of ArrayFire.

Table 4: Memory usage of *JacabiSolver* for different grid sizes.

n_x	n_y	Memory usage (MB)
100	100	1.4
150	150	3.2
200	200	5.8
400	400	23.5
1000	1000	148.0
2000	2000	593.7
5000	5000	3715.9
10000	10000	14870.5

6 Conclusion

In this project, we use the ArrayFire capabilities to solve a system of equations. To use memory efficiently, the coefficient matrix is stored in sparse format. As already ArrayFire doesn't support the sparse matrix for solving Sys. of equations, a Jacobi solver developed in the project. Although the Jacobi method is efficient in terms of memory usage, it depends on the initial condition.

References

[1] P. Yalamanchili, U. Arshad, Z. Mohammed, P. Garigipati, P. Entschev, B. Kloppenborg, J. Malcolm, J. Melonakos, ArrayFire - A high performance software library for parallel computing with an easy-to-use API (2015).
URL https://github.com/arrayfire/arrayfire

[2] P. J. Roache, Code Verification by the Method of Manufactured Solutions , Journal of Fluids Engineering 124 (1) (2001) 4–10. arXiv:https://asmedigitalcollection.asme.org/fluidsengineering/article-pdf/124/1/4/5901562/4_1.pdf, doi:10.1115/1.1436090.
URL https://doi.org/10.1115/1.1436090

[3] G. H. Golub, J. M. Ortega, Chapter 9 - the curse of dimensionality, in: G. H. Golub, J. M. Ortega (Eds.), Scientific Computing and Differential Equations, Academic Press, Boston, 1992, pp. 273–307. doi:https://doi.org/10.1016/B978-0-08-051669-1.50013-7.
URL https://www.sciencedirect.com/science/article/pii/B9780080516691500137

[4]

[5] Wikipedia, https://en.wikipedia.org/wiki/jacobi_method .

[6] stattrek, https://stattrek.com/matrix-algebra/find-matrix-inverse.aspx .