CFD Project 02
Report

Submitted to: D.R.Mahdi Pourbagian

Submitted by: Arshia Saffari

Spring of 2023

Problem:

Numerical solution to a Laplacian equation in 2D plane with unity length on each side and boundary condition

$$f(0,y) = f(x,0) = f(1,y) = 0, f(x,1) = 1.$$

Laplace equation:

$$\nabla^2 f = 0$$

In 2D plane this expands to:

$$\frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

Numerical Method:

Discretization:

Using central difference method

$$\frac{\partial^2 f}{\partial x^2} = \frac{f(x+\Delta x,y) - 2f(x,y) + f(x-\Delta x,y)}{\Delta x^2}$$

$$\frac{\partial^2 f}{\partial y^2} = \frac{f(x,y+\Delta y) - 2f(x,y) + f(x,y-\Delta y)}{\Delta y^2}$$

Substituting into the original equation, discretized Laplacian equation with 2nd order accurate central difference method becomes:

$$\frac{f(x+\Delta x,y) - 2f(x,y) + f(x-\Delta x,y)}{\Delta x^2} + \frac{f(x,y+\Delta y) - 2f(x,y) + f(x,y-\Delta y)}{\Delta y^2} = 0$$

Since our nodes at sides have known temperatures, we only write equations for middle nodes. Using naming convention M for unknown temperature nodes in the x direction denoted by index i and N for unknown temperature nodes in the y direction denoted by index j and node count x for total nodes in the x direction and node count y for total nodes in the y direction, we proceed by writing equations.

Knowing that node count x = M + 2 and node count y = N + 2,

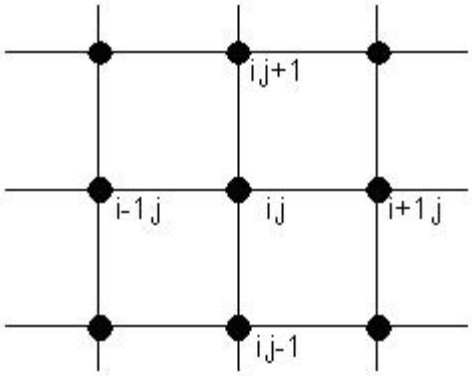
$$\frac{f(x+\Delta x,y) - 2f(x,y) + f(x-\Delta x,y)}{1} + \left(\frac{\Delta x}{\Delta y}\right)^2 \frac{f(x,y+\Delta y) - 2f(x,y) + f(x,y-\Delta y)}{1} = 0$$

$$\Delta x = \frac{L}{(M+2)-1}, \Delta y = \frac{H}{(N+2)-1} \rightarrow \gamma = \frac{L(N+1)}{H(M+1)}$$

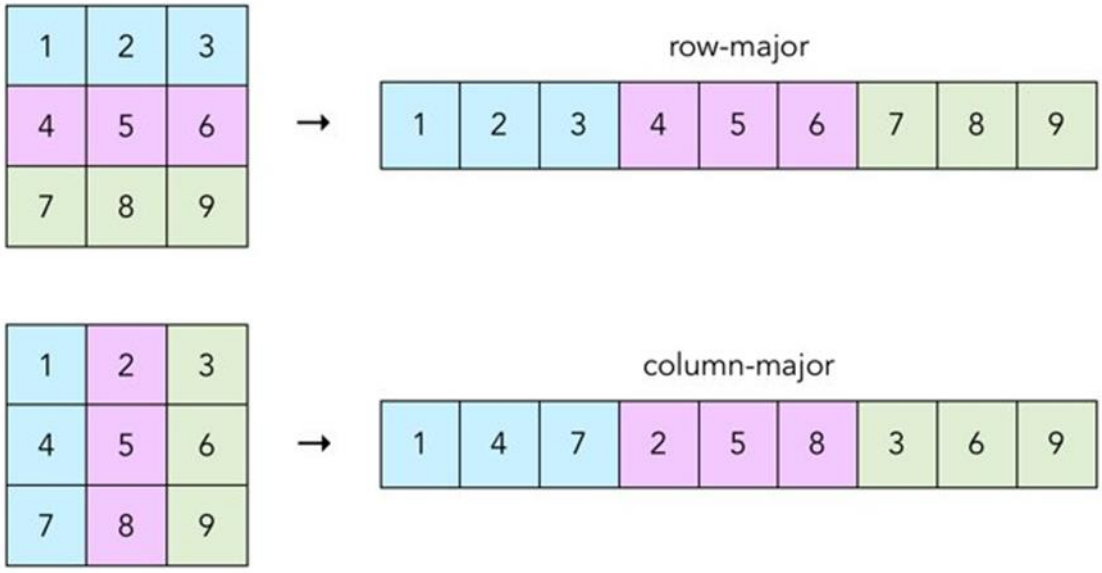
Since in our case $\Delta x = \Delta y \rightarrow T_{i+1,j} + T_{i-1,j} - 4T_{i,j} + T_{i,j+1} + T_{i,j-1} = 0$

But for the general case: $T_{i+1,j} + T_{i-1,j} - 2(1+\gamma^2)T_{i,j} + \gamma^2 T_{i,j+1} + \gamma^2 T_{i,j-1} = 0$

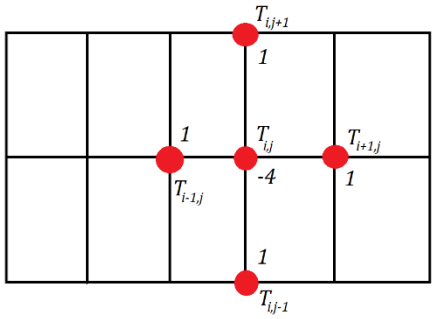
Our nodes are indexed using two indexes i and j. But the unknown vector in equation $A \times x = b$ (as it is a vector,) only needs 1 index. In order to place unknown temperatures in the x vector, the two indexes i and j need to be mapped to a single index.



This process called flattening is almost always done in 2 ways. Row major and column major. Arbitrary using column major otherwise known as Fortran style in programming the relationship between the row and col indexes in the matrix and i index in the vector is $col = \frac{i}{rows}, row = i \,\%\, rows$.



Writing the discretized equation for node located at i, j.



Organizing this into a vector using row major, the equation for node i, j

$$0 + 0 + \cdots + \gamma^2 T_{i,j-1} + \cdots + T_{i-1,j} - 2(1+\gamma^2)T_{i,j} + T_{i+1,j} + \cdots + \gamma^2 T_{i,j+1} + \cdots + 0 + 0 = 0$$

$$\begin{matrix} T_{0,2} & T_{1,2} & T_{2,2} & T_{3,2} \\ T_{0,1} & T_{i,j} & T_{2,1} & T_{3,1} \\ T_{0,0} & T_{1,0} & T_{2,0} & T_{3,0} \end{matrix} \rightarrow \begin{matrix} T_{0,0} & T_{1,0} & T_{2,0} & T_{3,0} & T_{0,1} & T_{i,j} & T_{2,1} \\ 0 & 1 & 2 & 3 & 4 & j \times M + i & 1 \times 4 + 2 = 6 \end{matrix} \cdots$$

Rewriting the equation above using the only one index:

$$0 + 0 + \cdots + T_{(j-1)M+i} + \cdots + T_{jM+i-1} - 4T_{jM+i} + T_{jM+i+1} + \cdots + T_{(j+1)M+i} + \cdots + 0 + 0 = 0$$

$$\text{Equation for node } T_{i,j} = 0 + \cdots + T_{(j-1)M+i} \qquad \begin{matrix} T_{i-1,j} or T_{jM+i-1} \\ \times \quad T_{i,j} \text{ or } T_{jM+i} \quad = \\ T_{2,0} \end{matrix}$$

The temperature at boundaries is known, so this can be written for nodes where neither i or j is 0 nor the maximum value of $node\ count - 1$ in their respective directions, in which case T is moved to the RHS with its coefficient.

The C/C++ code would be

```
void HeatEquationSolver::createEquationMatrix() {
		// 5 elements per row
		for (size_t i = 0; i < m_eqMatDims; i++)
			m_eqKnownVec.push_back(0);
		std::vector<Eigen::Triplet<double> > Vec;
		Vec.reserve(m_eqMatDims * 5);
		size_t N = m_nodeCount.ycount - 2;
		size_t M = m_nodeCount.xcount - 2;
		double L = m_computeDomain.xrange.second -
m_computeDomain.xrange.first;
		double H = m_computeDomain.yrange.second -
m_computeDomain.yrange.first;
		double gamma = L * (N + 1) / (H * (M + 1));
		//for (size_t i = 0; i < M * N; i++)
		//	for (size_t j = 0; j < M * N; j++)
		//		//m_eqMat.A[i][j] = 0;
		for (size_t i = 0; i < M; i++) {
			for (size_t j = 0; j < N; j++) {
```

```
                        // for node i,j:
                        if (j == 0) {
                              //m_eqMat.b[j * M + i] -= 0 * gamma;
                              //m_eqMat.A[j * M + i][(j + 1) * M + i] = 1 *
gamma;
                              m_eqKnownVec[j * M + i] -= 0 * gamma;
                              Vec.push_back(Eigen::Triplet<double>(j * M + i,
(j + 1) * M + i, 1 * gamma));
                        }
                        else if (j == N - 1) {
                              //m_eqMat.b[j * M + i] -= 1 * gamma;
                              //m_eqMat.A[j * M + i][(j - 1) * M + i] = 1 *
gamma;
                              m_eqKnownVec[j * M + i] -= 1 * gamma;
                              Vec.push_back(Eigen::Triplet<double>(j * M + i,
(j - 1) * M + i, 1 * gamma));
                        }
                        else {
                              //m_eqMat.A[j * M + i][(j - 1) * M + i] = 1 *
gamma;
                              //m_eqMat.A[j * M + i][(j + 1) * M + i] = 1 *
gamma;
                              Vec.push_back(Eigen::Triplet<double>(j * M + i,
(j - 1) * M + i, 1 * gamma));
                              Vec.push_back(Eigen::Triplet<double>(j * M + i,
(j + 1) * M + i, 1 * gamma));

                        }
                        if (i == 0) {
                              //m_eqMat.b[j * M + i] -= 0;
                              //m_eqMat.A[j * M + i][j * M + i + 1] = 1;
                              m_eqKnownVec[j * M + i] -= 0;
                              Vec.push_back(Eigen::Triplet<double>(j * M + i, j
* M + i + 1, 1));
                        }
                        else if (i == M - 1) {
                              //m_eqMat.b[j * M + i] -= 0;
                              //m_eqMat.A[j * M + i][j * M + i - 1] = 1;
                              m_eqKnownVec[j * M + i] -= 0;
                              Vec.push_back(Eigen::Triplet<double>(j * M + i, j
* M + i - 1, 1));
                        }
                        else {
                              //m_eqMat.A[j * M + i][j * M + i - 1] = 1;
                              //m_eqMat.A[j * M + i][j * M + i + 1] = 1;
                              Vec.push_back(Eigen::Triplet<double>(j * M + i, j
* M + i - 1, 1));
                              Vec.push_back(Eigen::Triplet<double>(j * M + i, j
* M + i + 1, 1));
                        }
                        //m_eqMat.A[j * M + i][j * M + i] = -4;
                        Vec.push_back(Eigen::Triplet<double>(j * M + i, j * M +
i, -4));
                  }// for j
            }// for i
            m_eqSparseMat.resize(N * M, N * M);
            m_eqSparseMat.setFromTriplets(Vec.begin(), Vec.end());
```

```
        } /* createEquationMatrix */
```

Which after defining some constants, the matrix coefficients are created based on node location.

If node is in the bottom of the computational domain (j==0), there will be a coefficient for the north node as it is unknown temperature node inside the domain but the south node has no temperature in the known vector and it modifies the known vector b.

The same procedure is repeated for top side as well as sides in the x direction.

It might be interesting to note that there are always at most 5 elements per row in the coefficients matrix and there are about 158404 rows and columns in the coefficient matrix in the case of a 400 by 400 grid. This matrix requires $8 \times 158404 \times 158404 = 200{,}734{,}617{,}728$ bytes of memory which is not possible to store in anything other than quite expensive servers. But as mentioned above there are only 5 non-zero elements maximum in each row. If this matrix is stored in a sparse matrix there only needs to be $158404 \times 5 \times 3 \times 8 = 19{,}008{,}480$ bytes which is possible.

1)

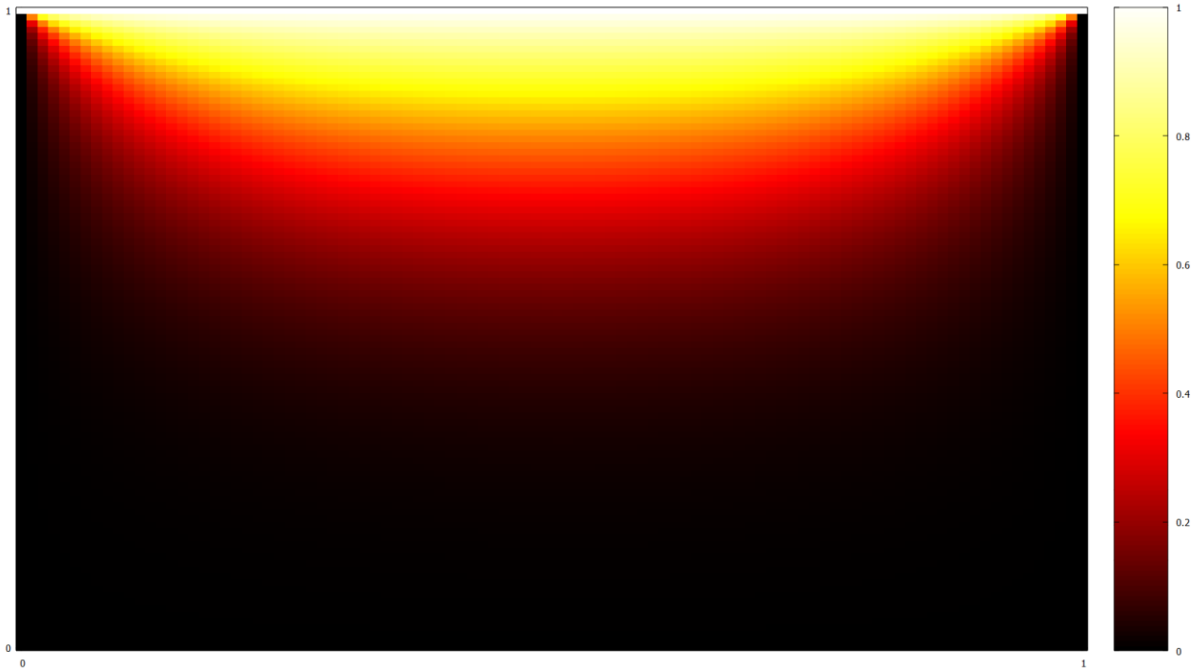
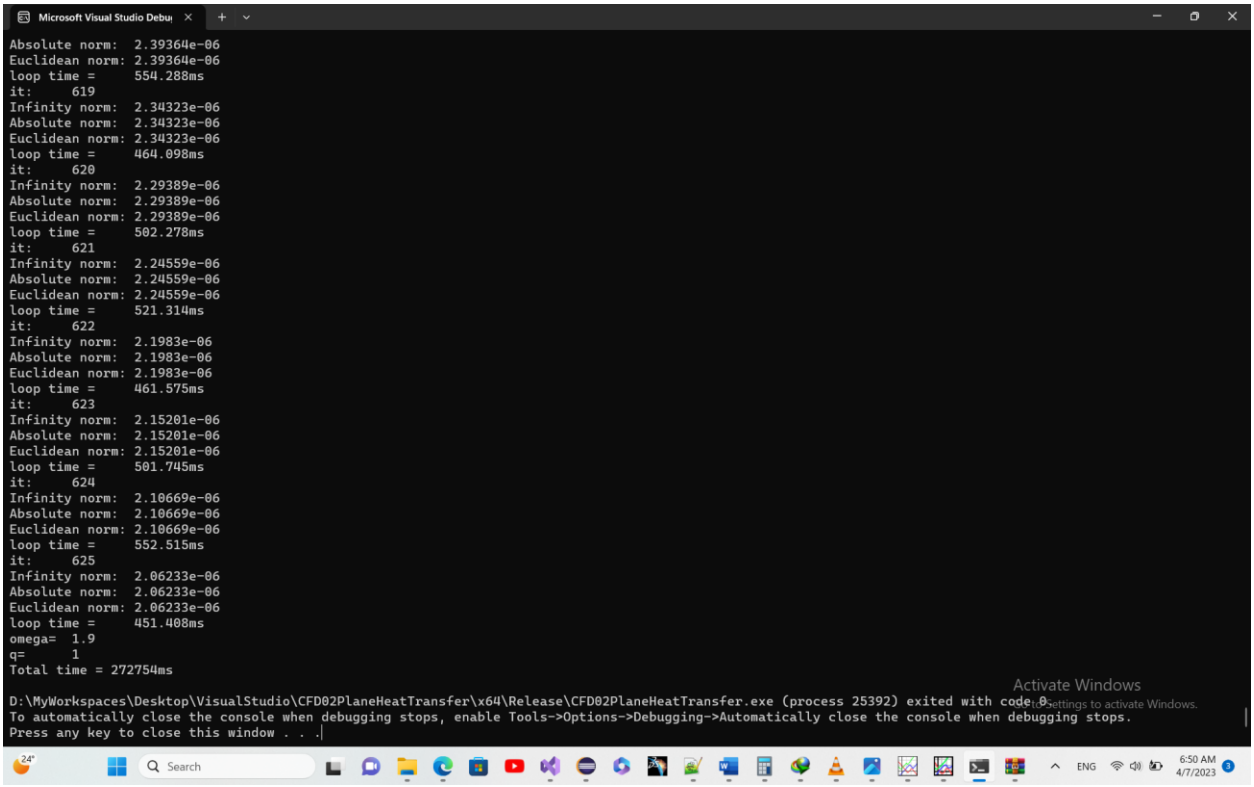
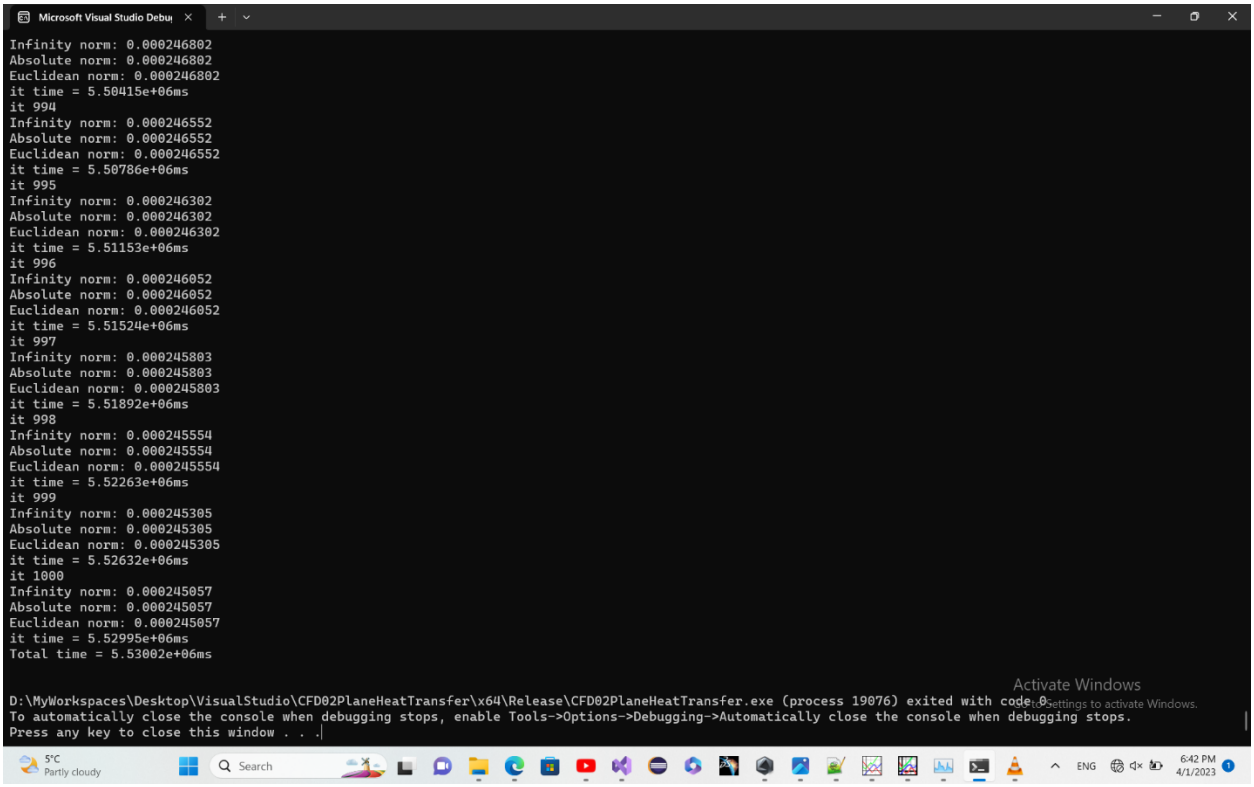


Figure 1. 100 by 100 grid



Took a little more than 5 minutes. 625 iterations and error less than 1e-6.

Figure 2. 200 by 200 grid



Took about 1.5 hours. 1000 iterations and error about 2*1e-3.

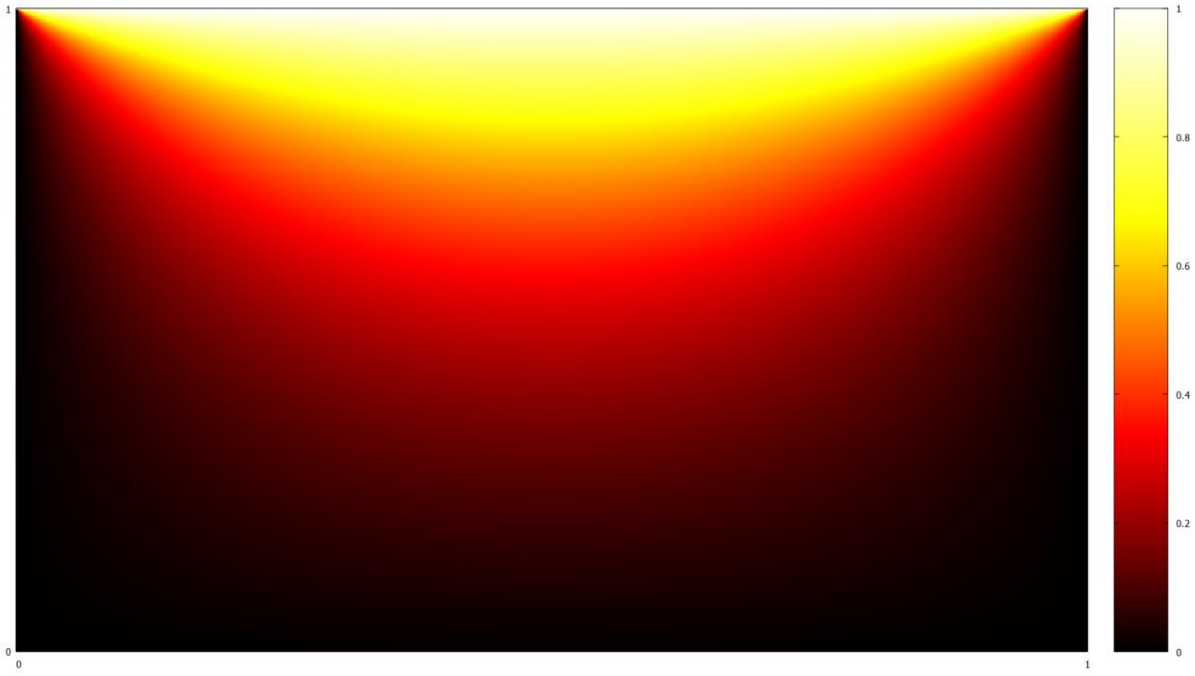


Figure 3. 400 by 400 grid

Run with an altered solver to increase speed. Basically, as it's known which elements are non-zero in the coefficients matrix since we created it. It took about 2ms for each loop and completed in minutes.

The general Gauss-Seidel spent more than 7 hours.

While two girds 100 by 100 and 400 by 400 look very similar, just latter looking sharper, the 200 by 200 grid looks different, as if it transfers less heat. It's the only one where solver was exited by iteration limit and not tolerance limit although the error is not that significant to make difference this large.

2)

Running the code on multiple threads with different over-relaxation and under-relaxation parameters and plotting them yields results below.

Since Gauss-Seidel method takes a long time and there is no emphasize on using the dimensions stated earlier, the results are computed for smaller grids in this section but the general idea seems applicable to other grids.

It took more than 1.5 hours to calculate 1000 iterations on the 200 by 200 grid.

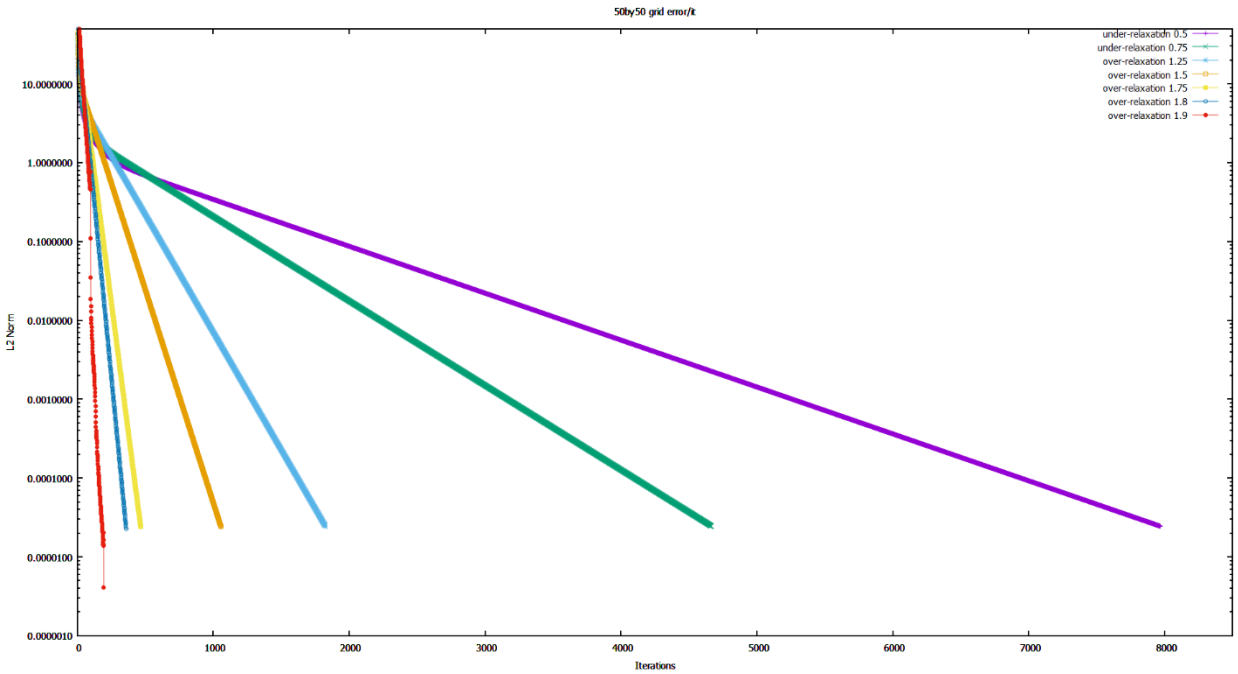
Since in Gauss-Seidel method each iteration depends on the last one, it's not easy to parallelize the program to run on multiple threads. Not sure how can it be improved through code.

```
// CFD02PlaneHeatTransfer.cpp : This file contains the 'main' function. Program
execution begins and ends there.
//

#include <iostream>
#include <HeatEquationSolver.hpp>
#include <chrono>
#include <thread>

int main()
{
    Range2D domain;
    Index2D counts1;
    domain.xrange.first = 0;
    domain.xrange.second = 1;
    domain.yrange.first = 0;
    domain.yrange.second = 1;
    counts1.xcount = 50;
    counts1.ycount = 50;

    ArshiaCFD::HeatEquationSolver P1(domain, counts1, 1);
    ArshiaCFD::HeatEquationSolver P2(domain, counts1, 0.5);
    ArshiaCFD::HeatEquationSolver P3(domain, counts1, 1.5);
    ArshiaCFD::HeatEquationSolver P4(domain, counts1, 0.75);
    ArshiaCFD::HeatEquationSolver P5(domain, counts1, 1.25);

    auto t1 = std::chrono::high_resolution_clock::now();
    std::thread Work1(&ArshiaCFD::HeatEquationSolver::run,P1);
    std::thread Work2(&ArshiaCFD::HeatEquationSolver::run, P2);
```

```
    std::thread Work3(&ArshiaCFD::HeatEquationSolver::run, P3);
    std::thread Work4(&ArshiaCFD::HeatEquationSolver::run, P4);
    std::thread Work5(&ArshiaCFD::HeatEquationSolver::run, P5);
    Work1.join();
    Work2.join();
    Work3.join();
    Work4.join();
    Work5.join();
    auto t2 = std::chrono::high_resolution_clock::now();
    std::chrono::duration<double, std::milli> ms_double = t2 - t1;
    std::cout << "Total time = " << ms_double.count() << "ms\n";
}
```

Using gnuplot

```
reset session

clear
set xrange [0:8500]
set yrange [1e-6:50]

set title '50by50 grid error/it'

set xlabel 'Iterations'
set ylabel 'L2 Norm'

set logscale y

set xtics 1000
set ytics 1e-6,10,50

unset key


set key at screen 0.98,0.95
plot 'File0.500000.txt' w lp ls 1 ti 'under-relaxation 0.5'
set key at screen 0.98,0.93
plot 'File0.750000.txt' w lp ls 2 ti 'under-relaxation 0.75'
set key at screen 0.98,0.91
plot 'File1.250000.txt' w lp ls 3 ti 'over-relaxation 1.25'
set key at screen 0.98,0.89
plot 'File1.500000.txt' w lp ls 4 ti 'over-relaxation 1.5'
set key at screen 0.98,0.87
plot 'File1.750000.txt' w lp ls 5 ti 'over-relaxation 1.75'
set key at screen 0.98,0.85
plot 'File1.800000.txt' w lp ls 6 ti 'over-relaxation 1.8'
set key at screen 0.98,0.83
plot 'File1.900000.txt' w lp ls 7 ti 'over-relaxation 1.9'
```

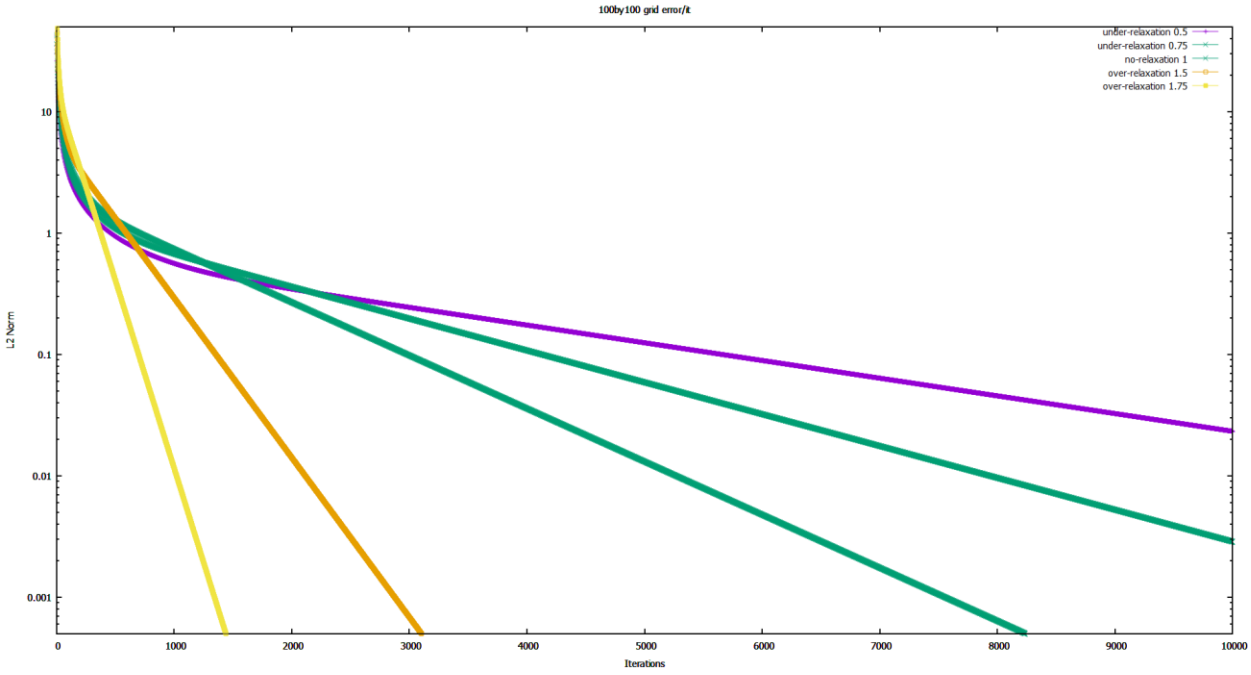


As can be seen at high errors under relaxation actually increases speed at which error is decreasing. But after error reaches around $10^{1.4} \sim 25$ over-relaxation curves intersect with that of under relaxation and after that, the higher over-relaxation, the faster it converges. At about over-relaxation 2 it never converges.

Also, for 100 by 100 gird

```
reset session

clear
set xrange [0:10000]
set yrange [5*1e-4:50]

set title '100by100 grid error/it'

set xlabel 'Iterations'
set ylabel 'L2 Norm'

set logscale y

set xtics 1000
set ytics 1e-6,10,50

unset key


set key at screen 0.98,0.95
```

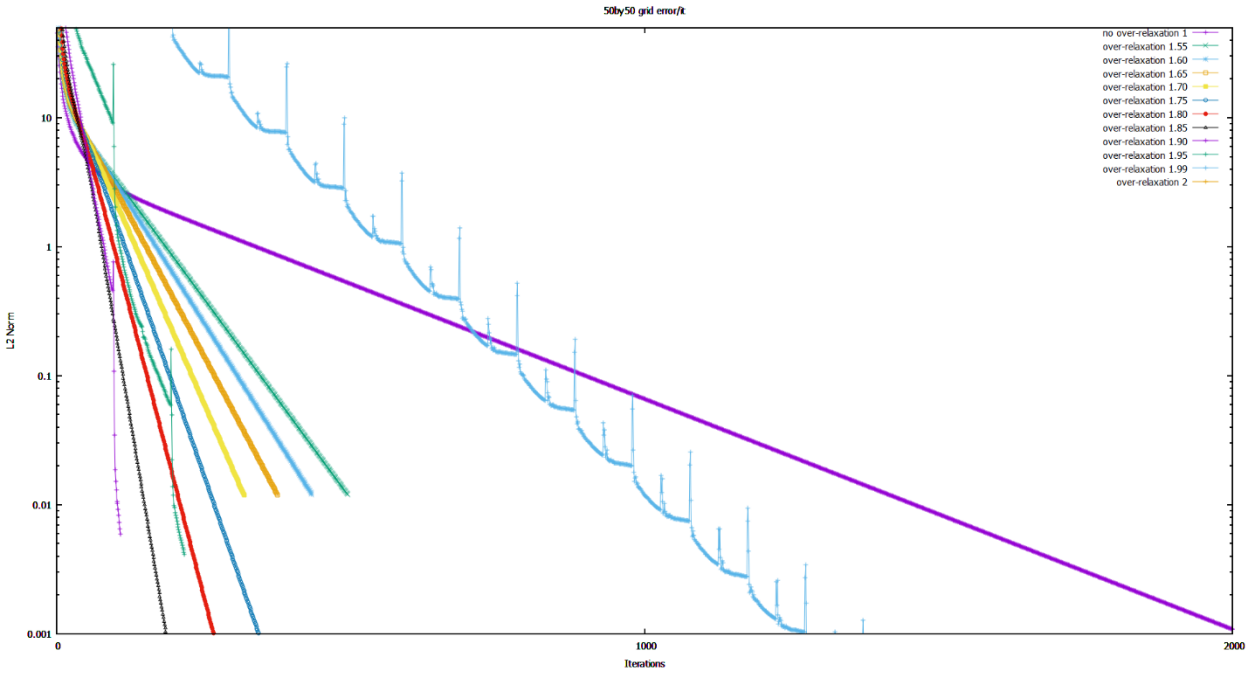
```
plot 'File0.500000.txt' u 1:3 w lp ls 1 ti 'under-relaxation 0.5'
set key at screen 0.98,0.93
plot 'File0.750000.txt' u 1:3 w lp ls 2 ti 'under-relaxation 0.75'
set key at screen 0.98,0.91
plot 'File1.000000.txt' u 1:3 w lp ls 3 ti 'no-relaxation 1'
set key at screen 0.98,0.89
plot 'File1.500000.txt' u 1:3 w lp ls 4 ti 'over-relaxation 1.5'
set key at screen 0.98,0.87
plot 'File1.750000.txt' u 1:3 w lp ls 5 ti 'over-relaxation 1.75'
```



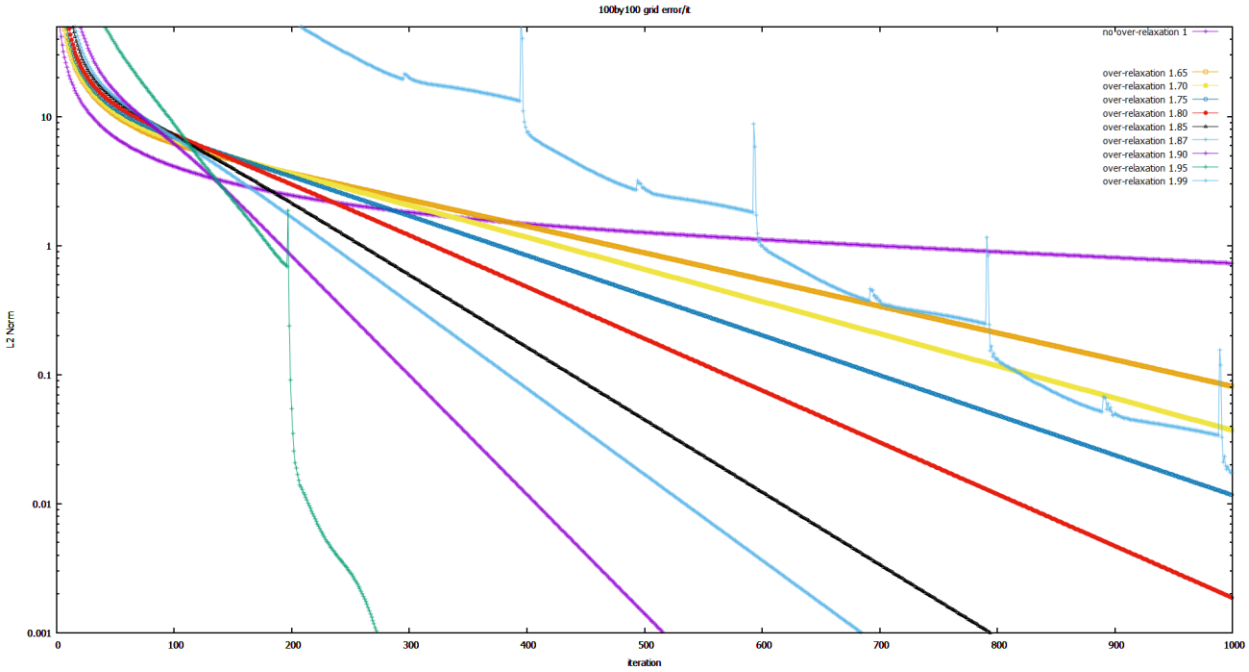
Which is very similar to the last one.

Generally, the higher the over-relaxation, the faster it converges. $\omega = 2$ keeps the error oscillating between about 400 to 200 and doesn't converge. Going a little lower at $\omega = 1.99$ it still converges but not that fast. In fact, it's iteration count is closer to $\omega = 1$ then others.

The best spot however looks to be about 1.9-1.85 for 50 by 50 grid.



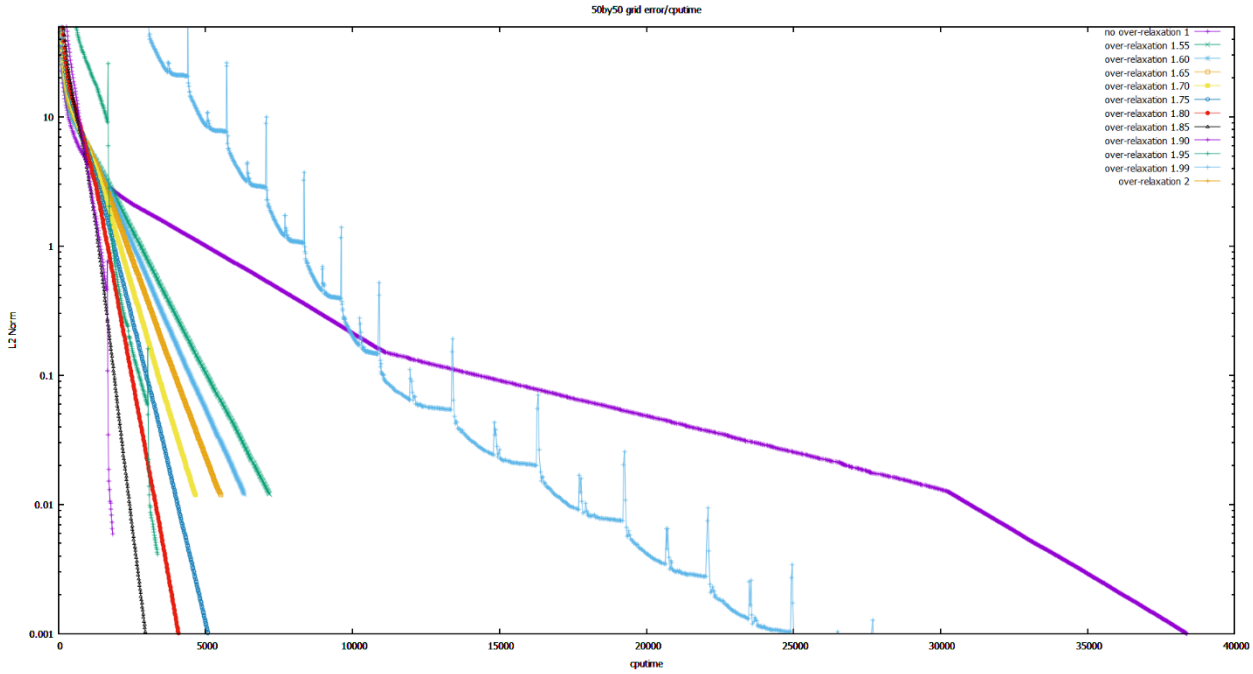
For 100 by 100 grid

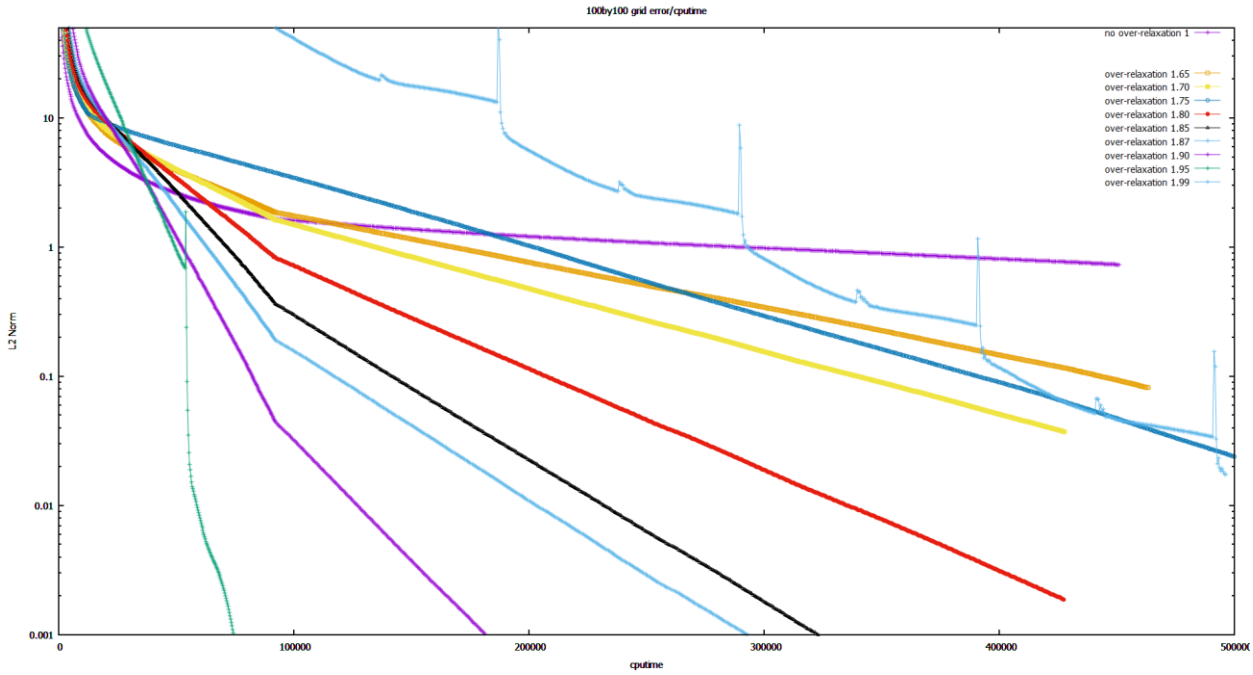


For this gird size, 1.95 error decreases faster but it's not that consistent. The best over-relaxation is about 1.9-1.95.

3)

Cpu time:





Which looks identical to iteration count plot. Just scaled differently.

4) using formula

We can calculate $\frac{x_{k+1}-x_k}{x_k-x_{k-1}}$

And $\frac{x_k-x_{k-1}}{x_{k-1}-x_{k-2}}$

Summation over the domain.

$$q \approx \frac{\log\left|\frac{x_{k+1}-x_k}{x_k-x_{k-1}}\right|}{\log\left|\frac{x_k-x_{k-1}}{x_{k-1}-x_{k-2}}\right|}$$

Using code

```
double nu = 0;
double dn = 0;
for (size_t i = 0; i < n; i++) {
    //Convergnce
    nu += (m_x[i] - oldX[i]) / (oldX[i] - old2X[i]);
    dn += (oldX[i] - old2X[i]) / (old2X[i] - old3X[i]);
}
double q = (log10(nu) / log10(dn));
```

Convergence order is found to be around 1.