Universiteit van Amsterdam

Report

Scientific Computing exercise set 1

April 25, 2018

Students:

Sam Ferwerda
10446982

Daan van Ingen
10345078

Teacher:
dr. J.A. Kaandorp

Course:
Scientific Computing

Code:
5284SCCO6Y

1.1 Tools

Our favourite (free) programming tools are:

- StackOverflow (Do I have to explain this?)
- Atom text editor (Nice language support and customise options)
- Python and all its beautiful packages (Man's best friend, dogs come second)
- The terminal (Who needs UI??)
- CSS autoprefixer (Compile your CSS to browser compatible CSS so things you develop in chrome will also work on IE (just kidding, nothing works on IE), but seriously, great tool)

How do I become a better programmer? Embrace new things, be critical on yourself, keep digging, keep learning and take the time to look at other people's code.

1.2 Vibrating string

The wave equation for the vibrating string is given by

$$\frac{\delta^2\Psi}{\delta t^2} = c^2 \cdot \frac{\delta^2\Psi}{\delta x^2}$$

Where the function $\Psi(x,t)$ is the vibration amplitude of the string given at time t. We introduce the following boundary conditions of the string:

$$\Psi(x=0,t) = \Psi(x=L,t) = 0$$

Where we define L as the length of the string. In this report we will use a fixed $L = 1$ and $c = 1$. We can convert this to a discrete time model by dividing L into N equal intervals: $\Delta x = \frac{L}{N}$

$$\frac{\delta\Psi}{\delta x} = \frac{u_{i+1,j} - u_{i,j}}{\Delta x} \tag{1}$$

$$\frac{\delta\Psi}{\delta x} = \frac{u_{i,j} - u_{i-1,j}}{\Delta x} \tag{2}$$

$$\frac{\delta\Psi}{\delta t} = \frac{u_{i+1,j} - u_{i,j}}{\Delta t} \tag{3}$$

$$\frac{\delta\Psi}{\delta t} = \frac{u_{i,j} - u_{i-1,j}}{\Delta t} \tag{4}$$

The second order differences are the differences of equations 1 & 2 and 3 & 4.

$$\frac{\delta^2\Psi}{\delta x^2} = \frac{u_{i+1,j} + u_{i-1,j} - 2u_{i,j}}{\Delta x^2} \frac{\delta^2\Psi}{\delta t^2} = \frac{u_{i,j+1} + u_{i,j-1} - 2u_{i,j}}{\Delta t^2} \tag{5}$$

Combining the second order differences with the wave equation and rewriting for $u_{i,j+1}$ gives:

$$u_{i,j+1} = \left(\frac{c\Delta t}{\Delta x}\right)^2 [u_{i+1,j} + u_{i-1,j} - 2u_{i,j}] - u_{i,j-1} + 2u_{i,j} \tag{6}$$

Now that we have discretized the function, we will study what will happen with the string at different time points and different initial conditions. We will use the following three initial functions:

1. $f(x) = \Psi(x, t = 0) = \sin(2\pi x)$
2. $g(x) = \Psi(x, t = 0) = \sin(5\pi x)$
3. $h(x) = \Psi(x, t = 0) = \sin(5\pi x) \cdot 1_{1/5<x<2/5}$

These function result into the following strings

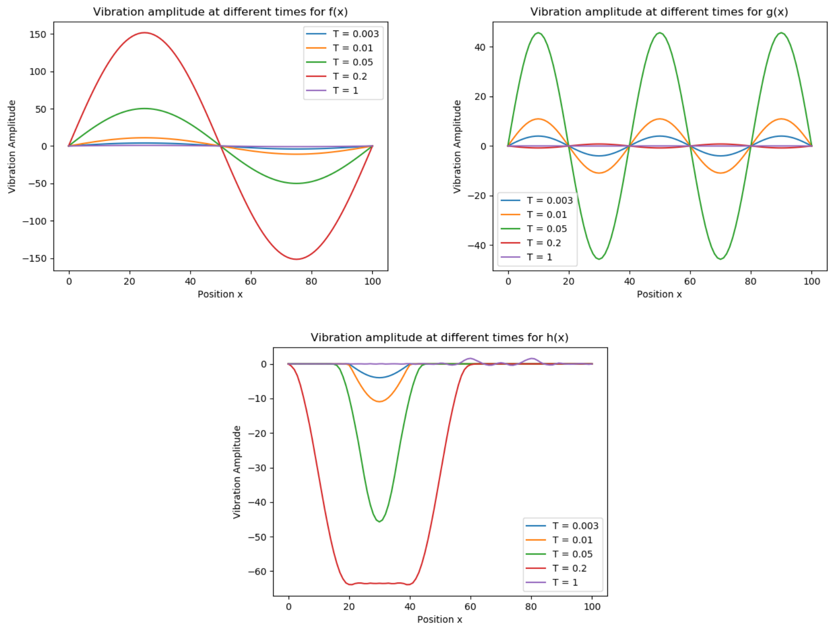

Figure 1: A vibrating string for different initial conditions

1.3 Time dependent diffusion equation

In the next exercise we consider the diffusion equation given by

$$\frac{\partial c}{\partial t} = D\nabla^2 c \text{ , assuming } D = 1.$$

We are using a square domain with $0 \leq x, y \leq 1$. The formula can be discretized such that we get

$$c_{i,j}^{k+1} = c_{i,j}^{k} + \frac{\delta t D}{\delta x^2}(c_{i+1,j}^{k} + c_{i-1,j}^{k} + c_{i,j+1}^{k} + c_{i,j-1}^{k} - 4c_{i,j}^{k}).$$

However, this has left us with the question what happens in the boundaries of this domain. Due the fact that the boundaries in x are periodic, we just have to replace $N+1 \to 1$ and $-1 \to N-1$, assuming we have sliced the domain in N pieces both vertically and horizontally. Using this will result to the following boundary functions given in the table below.

x	y	formula
x	y	$c_{i,j}^{k+1} = c_{i,j}^{k} + \frac{\delta t D}{\delta x^2}(c_{i+1,j}^{k} + c_{i-1,j}^{k} + c_{i,j+1}^{k} + c_{i,j-1}^{k} - 4c_{i,j}^{k})$
0	y	$c_{0,j}^{k+1} = c_{0,j}^{k} + \frac{\delta t D}{\delta x^2}(c_{1,j}^{k} + c_{N-1,j}^{k} + c_{0,j+1}^{k} + c_{0,j-1}^{k} - 4c_{0,j}^{k})$
x	0	$c_{i,0}^{k+1} = 0$
1	y	$c_{N,j}^{k+1} = c_{0,j}^{k+1}$
x	1	$c_{i,N}^{k+1} = 1$
0	0	$c_{0,0}^{k+1} = 0$
1	0	$c_{N,0}^{k+1} = 0$
0	1	$c_{0,N}^{k+1} = 1$
1	1	$c_{N,N}^{k+1} = 1$

Table 1: Boundary functions

We use the formulas from table 1 to write a program to simulate the time dependent diffusion equation. We choose a grid of 40x40 points, time steps of 0.0001 seconds and D = 1. With these values our scheme is stable: $\frac{4\delta t D}{\delta x^2} = 0.16 \leq 1$. Because the scheme only depends on local interaction there is an opportunity for parallel computation. We found that computation time was not too long to make it worthwhile to implement a parallel computation scheme.

To determine the correctness of the numerical result we compared the outcome to the analytical solution:

$$c(x,t) = \sum_{i=0}^{\infty} \left[\operatorname{erfc}\left(\frac{1-x+2i}{2\sqrt{Dt}}\right) - \operatorname{erfc}\left(\frac{1+x+2i}{2\sqrt{Dt}}\right) \right] \tag{7}$$

We limited the summation to 50 times. We used 50 because we found it to be a nice balance between precision and computation time. Figure 2 shows the comparison between the numerical solution and the analytical solution for different times in the diffusion simulation. Visually, we can see that these two figures match closely, the absolute error between the two results is shown in figure 3. This figure shows that the maximum observed error is small ($\sim$ 0.01) and thus the numerical approximation is close to the analytical result. The absolute error is largest at the point where the substance concentration decreases the most. This has probably to do with the discretization applied in the numerical scheme. In the analytical solution the diffusion through the medium is seen as one flowing process, while in the numerical solution some fields at the beginning of the simulation will be 0. This causes a small error that is relatively large for the area in the medium where the decrease in concentration from top to bottom is relatively large.

Figure 4 shows the concentration in the medium at different points in time. The figure corresponds with the results shown in figure 2. In the beginning of the simulation almost all the source is at the top of the medium and slowly this spreads and eventually there will be a linear distribution of the concentration within the medium.

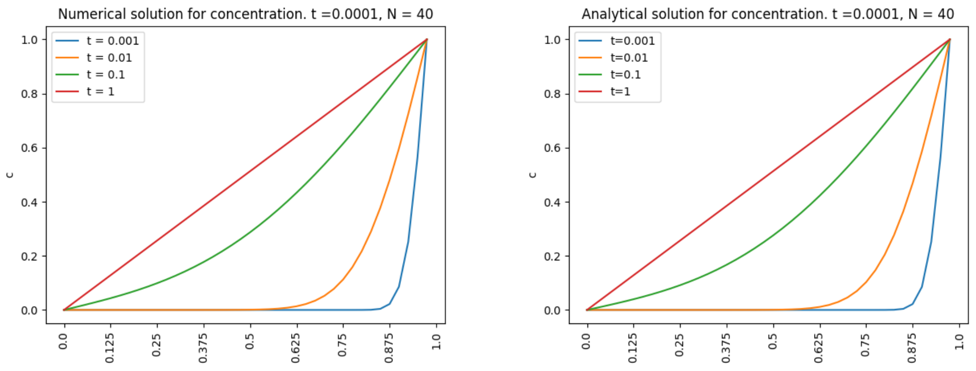

Figure 2: Numerical solution vs the analytical solution for δt = 0.0001 and N = 40

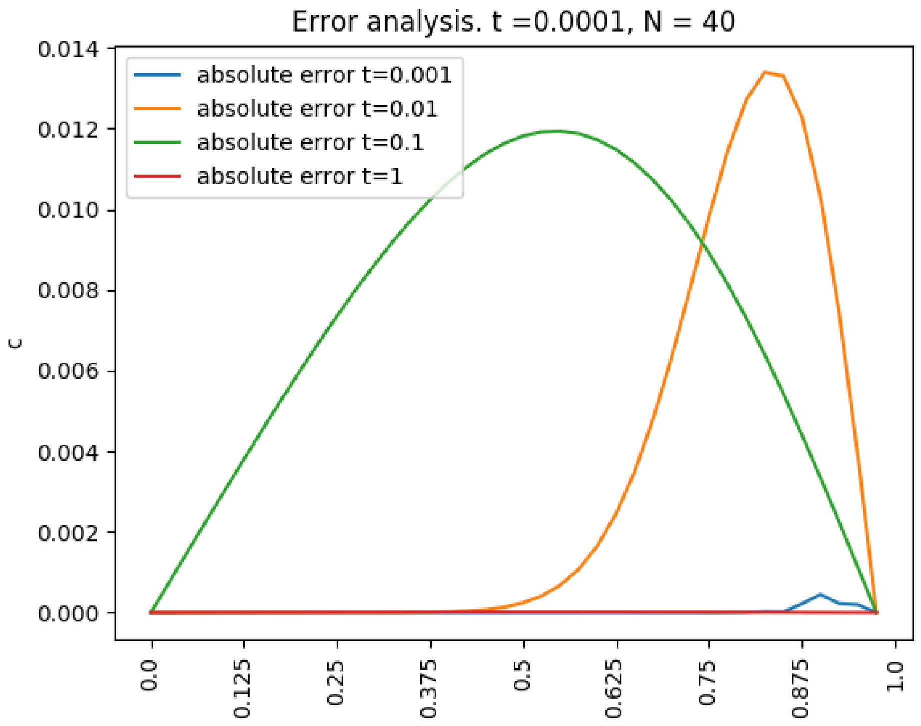

Figure 3: Absolute error between the numerical solution and the analytical solution for δt = 0.0001 and N = 40

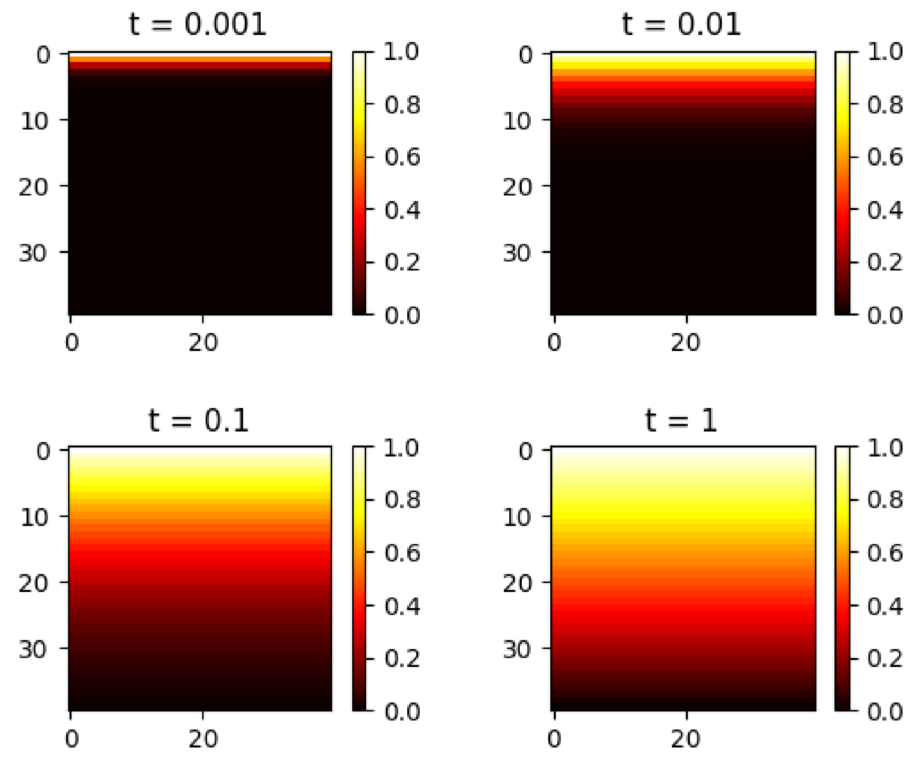

Figure 4: Visualisation of the concentration at different time steps for, δt = 0.0001 and N = 40

1.4 Time independent diffusion equation

If we want the diffusion equation to be independent of time, we have to introduce the Jacobi iteration, the Gauss iteration and the Successive Over Relaxation iteration. They are all similar yet SOR is a little more efficient than Gauss and so is Gauss a little more efficient than Jacobi. The Jacobi iteration is given by

$$c_{i,j}^{k+1} = \frac{1}{4}(c_{i+1,j}^{k} + c_{i-1,j}^{k} + c_{i,j+1}^{k} + c_{i,j-1}^{k})$$

where k represents the k-th iteration. Now as we can see this is independent of time since t is no longer in the equation. However we can increase the efficiency by using the Gauss iteration given by

$$c_{i,j}^{k+1} = \frac{1}{4}(c_{i+1,j}^{k} + c_{i-1,j}^{k+1} + c_{i,j+1}^{k} + c_{i,j-1}^{k+1}).$$

The only difference here are the terms that now have $k+1$ as subscript, meaning that we do not look at the old value of the neighbour. Instead we look at the current, just updated, value of the neighbour. In fact, this way of calculating the concentration is a little bit better then the Jacobi iteration that we introduced before. We will compare them later, after we introduce the last iteration method. The SOR iteration method is given by

$$c_{i,j}^{k+1} = \frac{w}{4}(c_{i+1,j}^{k} + c_{i-1,j}^{k+1} + c_{i,j+1}^{k} + c_{i,j-1}^{k+1}) + (1-w)c_{i,j}^{k}$$

and it is easily noted that it is a slight change of the Gauss iteration. We can change the weight w between 0 and 2, however the most efficient w seems to be somewhere between 1.7 and 2.0. For now just let us take a look at the 3 different iteration methods. See the next figure, were we have taken 2 different values for w.

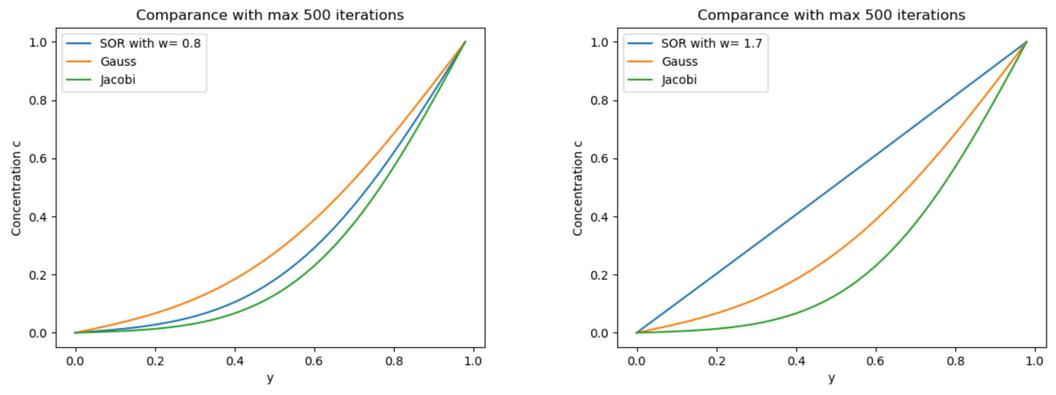

Figure 5: Comparance of different methods

In this figure is clearly visible that Gauss is better than Jacobi, however SOR seems to be only better than these two if we have used the correct w. For example $w = 1.7$ gives a way better result than $w = 0.8$.

Convergence

It would be perfect if we could just let the algorithm do infinite many iterations since that would give the best outcome, probably even the same as the analytical solution. However due to the lack of time we want to find the most efficient way of finding a decent solution. We are therefore interested in the convergence of every method. The convergence is measured by the delta given as

$$\delta = \max_{i,j} |c_{i,j}^{k+1} - c_{i,j}^{k}|$$

We have studied this delta for every iteration method, in the next figure you can see the convergence of the delta when increasing the number of iterations k.

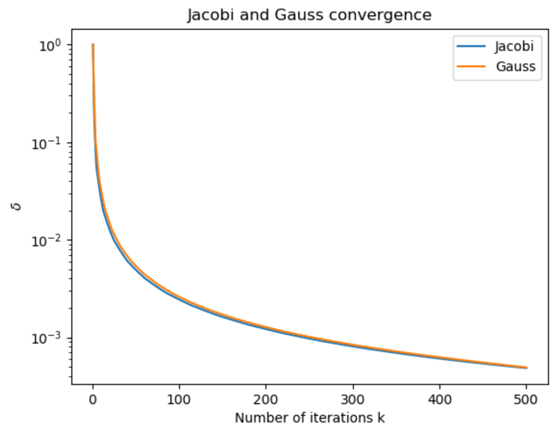

Figure 6: Convergence parameter δ over iterations k

We can see these are very similar, almost identical. For the SOR iteration method we have taken 3 different values of w, namely $w = 0.5, 1, 1.5$. This results into the following 3 graphs.

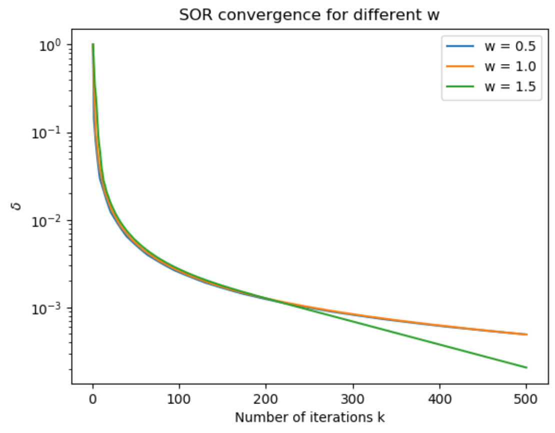

Figure 7: Convergence parameter δ over iterations k for different values of w.

One might ask why the δ, in other words maximum change, is smaller for higher w. A higher w seems to have a higher convergence rate, as we will see in the next subsection, and this means that the SOR with $w = 1.5$ is already near the actual value when iterated 500 times, while the SOR with $w = 1$ might need more than 1500 to achieve the same results.

Optimal w in the SOR method

We have discussed earlier that the SOR method is not efficient for all w and that makes us interested in the optimal w such that we have quick convergence. First let us study the error using $N = 50$ and after we have done that we will check whether or not the error is dependent on the grid size. Now, we are most interested in the error for $1.7 \leq w \leq 2.0$ and so we calculated the errors calculating the error for $\Delta w = 0.01$. We received the following behaviour of the error.

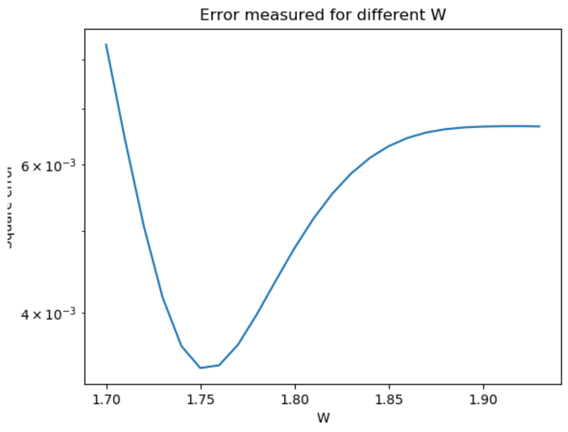

Figure 8: Error for w with $N = 50$

It came out to be the smallest error for $w = 1.75$, however would this w be dependent on the grid size N? It most certainly does, as we can see in the next two figures where we increased the grid size to $N = 60$ and $N = 70$. As we can see the best w shifts towards $w = 2.0$ if we increase

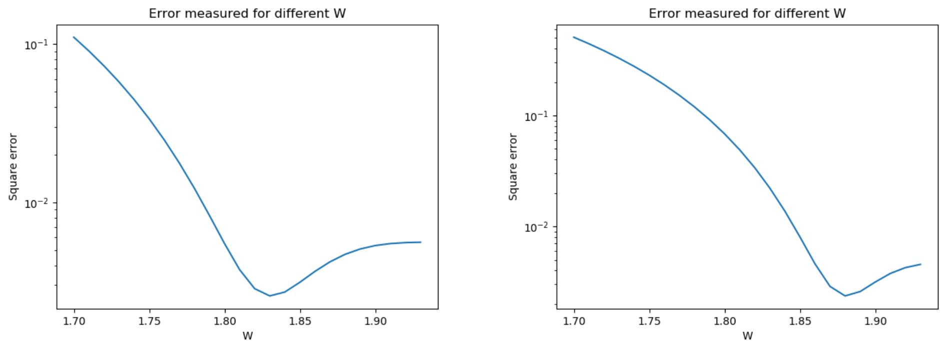

Figure 9: Error for w with $N = 60$ and $N = 70$ resp.

the value of N. Therefore we assume that if we increase N, w will converge to 2.0 and stay there because it is defined to not be higher than 2.

Effects of objects in the concentration

We might want to know what happens to the concentration levels when we add objects to the domain. Assuming that at the place of the object, the concentration is equal to zero. This means that it is harder for the concentration to spread near the object since many of there neighbours are equal to 0 and therefore do not change much. Before we go straight to the practical figures and graphs, lets sketch what we think would happen. If there is no object in the domain, all columns act similar and after many iterations we achieve a straight line. However if we add fixed 0 to the center for example, then all points near that center lost a neighbour and therefore grows less easily. However, due to the lack of growth in for example column 7, column 6 is influenced by column 7 and also loses growth speed even though it is not directly connected with the object. What we see here is a chain reaction leading to more iterations needed to spread the concentration. Of course, increasing the size of the object or the number of objects, increases the minimum iterations needed. The following figure shows us how it looks like with a rectangle of size 10 and size 20 inside the domain.

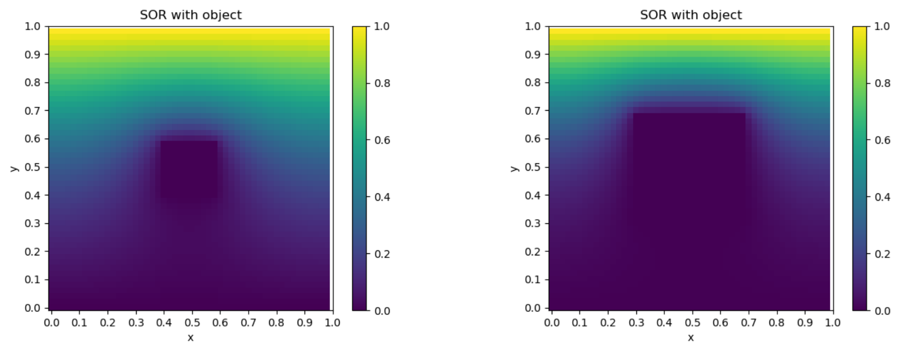

Figure 10: Concentration after 500 iterations

As we can see in figure 10, the concentration in the right figure is having a way more difficult time than the concentration in the left figure. This has to do with the size of the rectangle. Now we are also interested in the optimal w, would it have changed since the last time we checked? Remember that we have found a $w = 1.75$ for a grid of $N = 50$. Suprisingly, we obtained the following results:

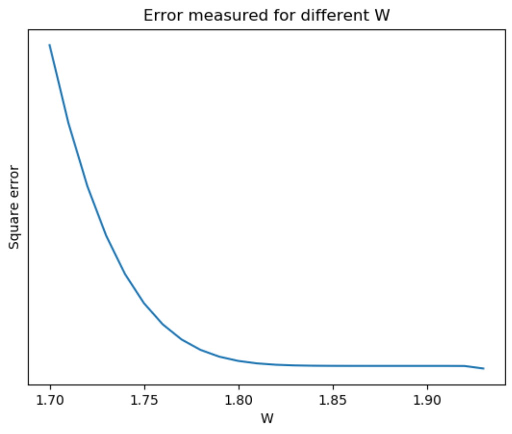

Figure 11: Error for w values with an object in the center

This means that if we add an object to the domain, we want to increase our w as much as possible, remarkable!