Simulationsmethoden 1 January 8, 2011

Tutorial 4: *The Finite Difference method*

Michael Kopp (2439093)

1 Parabolic PDE: Diffusion Equation

Idea The Diffusion-Equation

$$D\,\frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t} \tag{1}$$

with $T = T(x,t)$ is discretized using the most simple approach: A Taylor Expansion where terms of order 1 or 2 are omitted. This yields the *Euler Scheme* to numerically solve the equation. A time step is

$$T(i,j+1) = T(i,j) + r\,[T(i+1,j) - 2\,T(i,j) + T(i-1,j)] \tag{2}$$

while the discretization of time into time steps of length Δt and of space in steps of Δx are hidden in $r = D\,\Delta t/\Delta x^2$ and $i \in \mathbb{N}$ means $x = i \cdot \Delta x$ and $j \in \mathbb{N}$ means $t = j \cdot \Delta t$.

Debugging The first error in `example-finite-differences-1.c` was a "–" in line 54: The initial temperature was set to a negative Temperature which does not make sense...

The other error is in line 47: The divisor should be DX∗DX.

A third "error" might be the value of alpha as the problem sheet[1] says $\alpha = 1.172e-5$ – I use this value.

Optimisation To optimize the code, in line 84 ff only the pointers to the arrays should be interchanged – this requires only one triangular change, not nX ones.

The many **if**s in lines 94 ff are each executed each timestep – using **else** would reduce that. Additionally setting the flag could be avoided by simply using a function which writes to files, or one could use a single **if**-statement with a OR-conjunction between the different arguments.

[1] and wikipedia

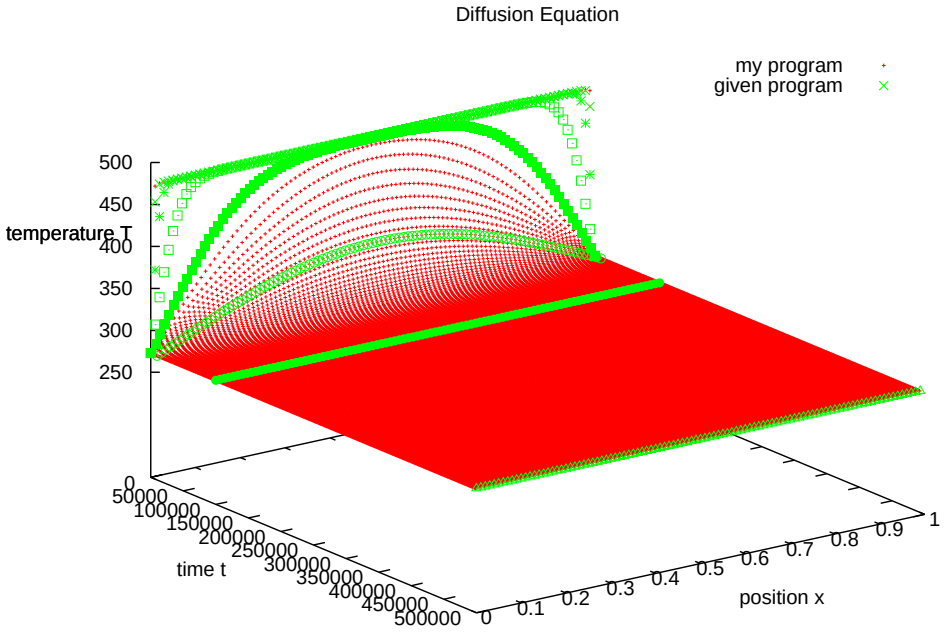


Figure 1: Temperature diffusion on a one dimensional rod: Comparison between the given program an my rewritten one – obviously the programs provide the same values.

Rewritten I rewrote the code completely; in my opinion it's more easy to understand and obviously faster: For the same task, my program took $6.372s$ and the presented one took $8.683s$ – both compiled with the same gcc, both with optimisation level 0: My program even gave more verbose output: Every 500th step was written to a file; these are about 100 times more output...

In figure 1 the two programs are compared and they yield the same results – thus the further results will be computed using my program.

In my version, the diffusion constant is called D instead of α.

Examples In figure 1 you can see that the temperature of the rod will become homogeneous – the whole rod will after some time adopt the temperature of its ends.

In fig. 2(a) you can see the same rod with now two different temperatures at its ends; the temperature profile after some time is a linear function of the

rod position.

Values for r In figure 3 values dx $= 0.01$ and dt $= 5$ were chosen – thus $r = 0.568$. The same boundary conditions as in the simulation of figure 2(a) were chosen but obviously the results differ strongly. Looking at the temperature at magnitudes 10^{63} I'm tempted to regard these results unrealistic...

Since the large value of dt is responsible for the "crash" r must be *smaller* than 0.568; to check this, the same simulation with dx $= 0.1$ was conducted; the result in fig. 2(b) compared to the results in 2(a) show, that now – for $r = 0.568 \cdot 10^{-2}$ everything is fine again.

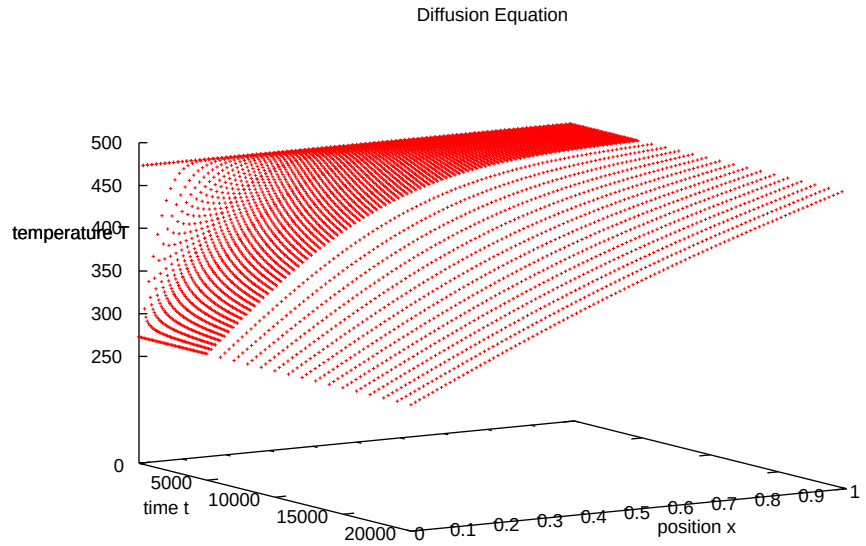
If r is too large – $r \geq 0.5$ –, the "derivation" part in equation 2 (in square brackets) will be "stronger" than the $T(i, j)$ part. The large temperatures escalate each other higher and higher.

That's why the "destructive" effect – the unreasonably high temperatures – occur only in the area of the rod, where the temperature changes over time. To verify this thesis cf figure 3(d): Here the problem is similar to the one in figure 1 and thus on both sides of the rod the temperature changes and thus on both sides these high temperatures occur.

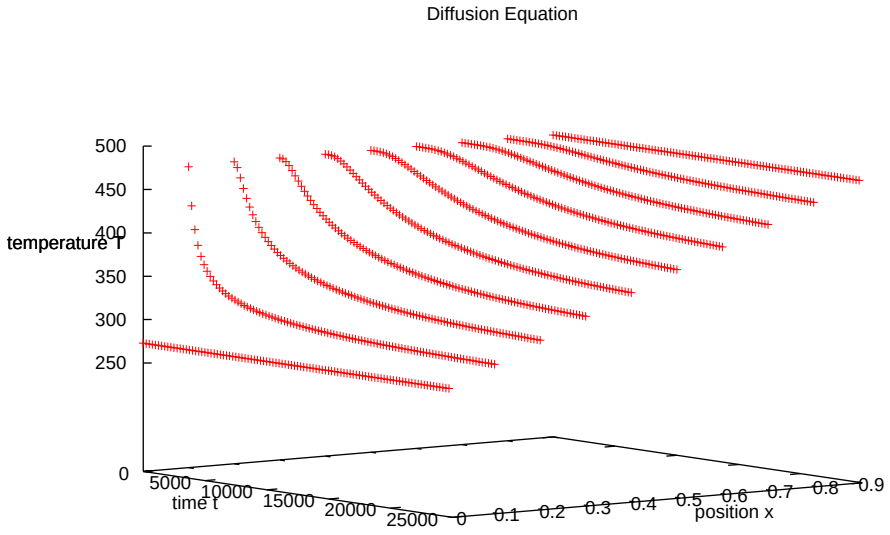
Analytical solution The analytical solution of the symmetric problem was computed for 4 times more points and both results are compared in figure 4. The crucial part was small times: If one used too few addends, the solution does not make sense. In `analytical.cpp` I used an adaptive summation scheme.

In figure 4(c) the analytic and the numerical solution are plotted – you can observe that they yield identical results.[2]

[2]Funnily this is rather check for my summation scheme than for the computed simulation as I modified the summation scheme so that the results fit to simulated ones ;-)

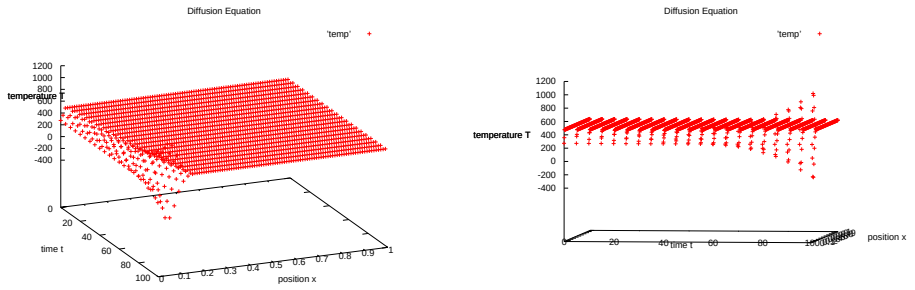


(a) In the beginning of the simulation more time steps were pictured.



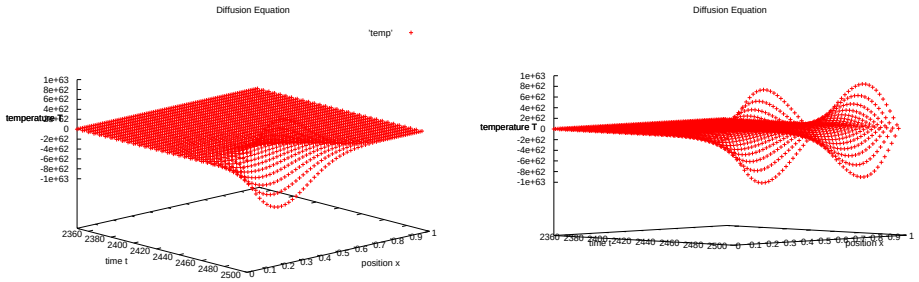
(b) The rod for dx = 0.1 and dt = 5 – obviously the results are identical.

Figure 2: The rod with different temperatures at its ends.



(a) small times

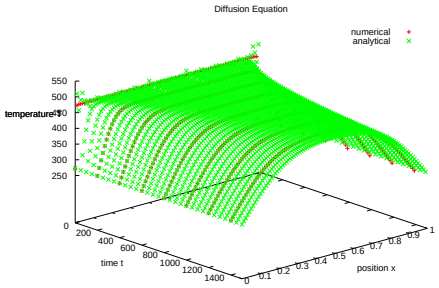
(b) small times – different perspective



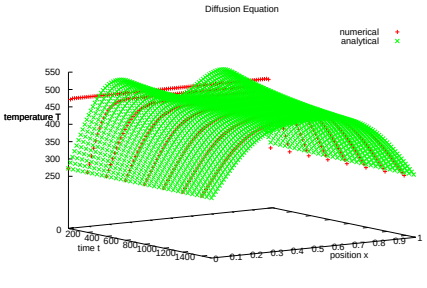
(c) larger times

(d) larger times – problem similar to fig 1

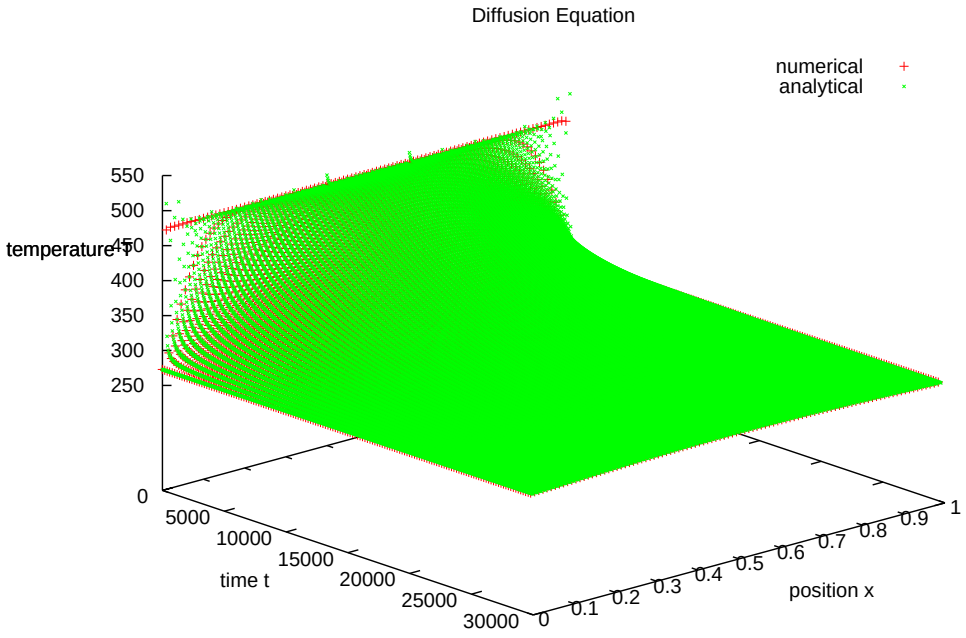
Figure 3: The rod for dx = 0.01 and dt = 5 – for (a) to (c) the same initial conditions as for the simulation in figure 2(a) were used.



(a) small times – good summation



(b) small times – bad summation



(c) all times

Figure 4: Comparison of analytical and numerical solution.

2 Hyperbolic PDE: Wave equation

Idea We now consider the wave equation

$$\frac{\partial^2\phi}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2\phi}{\partial t^2} \tag{3}$$

where ϕ denotes $\phi(x,t)$ for $x \in]0,1[$ and $t \geq 0$ with initial conditions

$$\phi(x,0) = \sin(\pi x) \quad \text{and} \quad \frac{\partial\phi}{\partial t}(x,0) = 0\ . \tag{4}$$

By using the centered second derivative (as we did in the previous chapter) we get the discretized equation

$$\frac{\phi(i+1,j) - 2\phi(i,j) + \phi(i-1,j)}{\Delta x^2} = \frac{1}{c^2}\frac{\phi(i,j+1) - 2\phi(i,j) + \phi(i,j-1)}{\Delta t^2}$$

which we can easily write as

$$\phi(i,j+1) = r\cdot[\phi(i+1,j) + \phi(i-1,j)] + [1-r]\cdot 2\,\phi(i,j) - \phi(i,j-1)\ , \tag{5}$$

with the abbreviation $r = (c\Delta t/\Delta x)^2$. The boundary conditions are

$$\phi(0,t) = \phi(1,t) = 0 \quad \text{resp} \quad \phi(0,j) = \phi(i_{\max},j) = 0\ . \tag{6}$$

Initial Conditions The initial condition (4) can be "translated" into the discrete world using several methods. The most simple one is to use a forward scheme to discretize the derivate $\partial\phi/\partial t$; this yields

$$\phi(i,0) = \phi(i,1) = \sin(\pi\Delta x\, i)\ . \tag{7}$$

A more sophisticated method is to use the centered difference method for the first derivative as well. That way we'll find out that

$$\phi(i,-1) = \phi(i,1)\ ,$$

but this information is not really helpful as for the scheme in equation (5) we'll need both $\phi(i,0)$ and $\phi(i,1)$. So to find out $\phi(i,1)$ we can insert $\phi(i,-1)$ and $\phi(i,1)$ into (5) for $j=0$ and get

$$\begin{aligned}\phi(i,1) &= r\cdot[\phi(i+1,0) + \phi(i-1,0)] + [1-r]\cdot 2\,\phi(i,0) - \phi(i,-1)\ ,\\ 2\,\phi(i,1) &= r\cdot[\phi(i+1,0) + \phi(i-1,0)] + [1-r]\cdot 2\,\phi(i,0)\ ,\end{aligned}$$

and thus finally we get the desired

$$\phi(i,1) = \frac{r}{2}\cdot[\phi(i+1,0) + \phi(i-1,0)] + [1-r]\cdot\phi(i,0)\ . \tag{8}$$

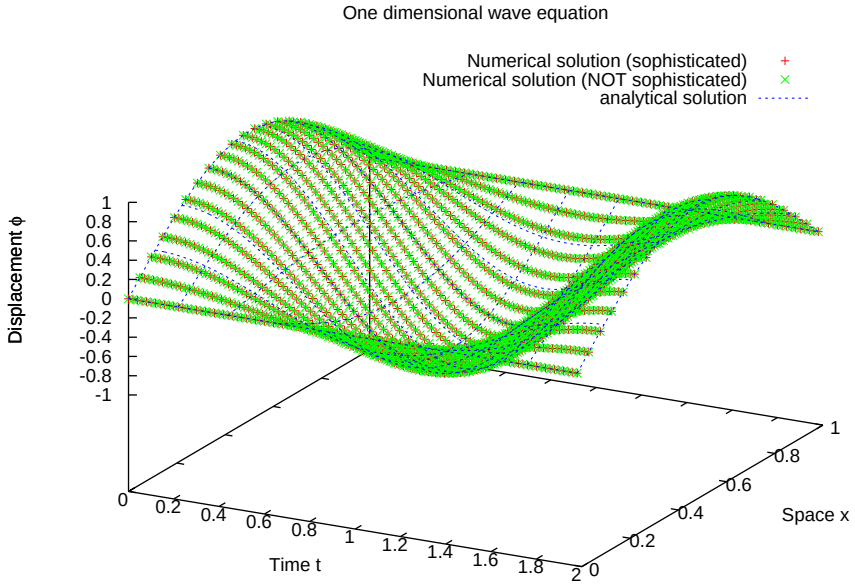


Figure 5: Numerical and analytical solution of the wave equation.

Implementation I could use the construct of the former program `diff.cpp` with some minor adaptions; the most striking one is that the program requires the storage of three time steps (and thus three arrays and a more sophisticated swapping mechanism) and an additional initial routine for the second time step.

Comparison to analytical solution The analytical solution

$$\phi(x,t) = \sin(\pi x)\,\cos(c\pi t) \tag{9}$$

is compared to the numerical one in fig 5.

Here you can observe that both initial conditions (the more and the less sophisticated one [eq. (7) resp. (8)]) provide – practically – identical solutions. And both match the analytical solution very well.

Interestingly you can for some initial conditions see a little disturbance propagating from the "right" ($x = 1$) end towards the "left" end. This is due to the fact that the x-discretization was chosen imperfect: $\Delta x \cdot i^{\max}$ is condemned to be equal 0 all the time but compared to the analytical solution there should

not be a 0 – in this particular case $\Delta x \cdot i^{\max} \neq 1$ whereas only at $0, 1, 2, \ldots$ the analytical solution[3] "allows" $\phi = 0$.

To visualize this better, I composed the little film `wave_disturb.mp4` with the conditions

```
x <= 5.00000
dx = 0.25000
n_x =       20
t <= 30.00000
dt = 0.00050
n_t =   60000
c = 1.00000
```

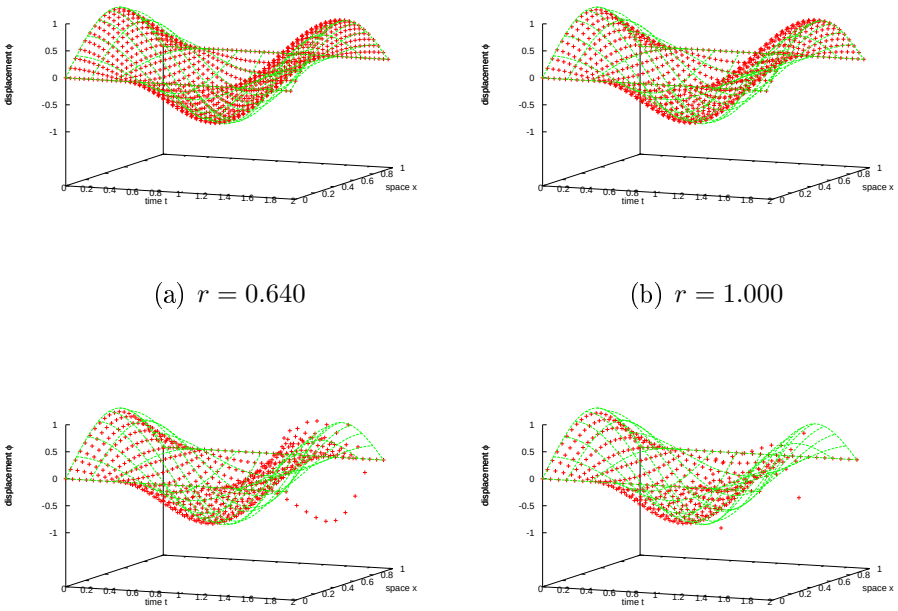
800 time steps were printed out and each one was processed with gnuplot; for a better image the data points were smoothed using method `csplines`. In this film you can see the disturbance wander from right to left, reflect on the left, wander back etc. And you can see clearly that the righter most fixed (at $\phi = 0$) point is at $x \approx 4.75$, not as allowed at $x = 5$. To prove that, watch the video `wave_nodist.mp4` where I made some effort to stop this disturbance.

Stability We find that only simulations with $r \leq 1$ are stable – compare figure 6. For larger r the solution will diverge for larger times. If you take a look at equation (5) you can guess why: for $r > 1$ the second square bracket becomes negative, so the "previous" wave at the considered point not only is not important any more but worse: It "counteracts" against the new wave.

Play Instinct and Beauty in Physics Finally in figure 7(a) you can see the spreading of a δ-peak formed "deformation". You can see how the wave front propagates with $c = 1$: In $t = 0.5$ it travels half the rod length – that's $x = l/2 = 0.5$.

In Figure 7(b) you can find the reflection of a small wave packet – compare also the video `small.mp4`.

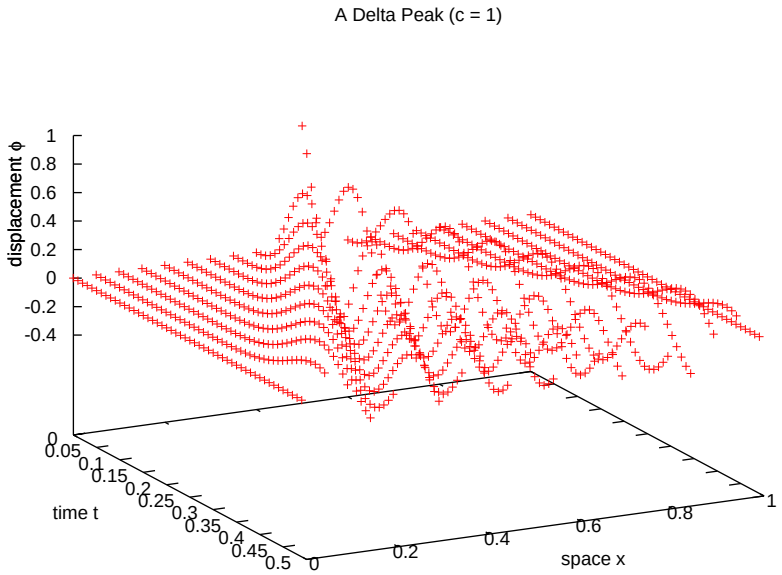
[3] or better: The underlying problem to the analytical solution



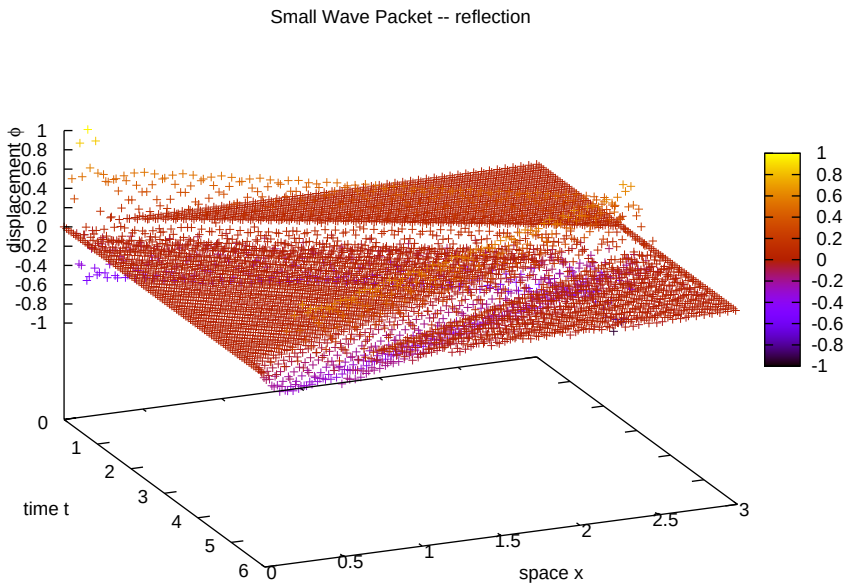
(a) $r = 0.640$ (b) $r = 1.000$

(c) $r = 1.440$ (d) $r = 1.960$

Figure 6: Stability: Numerical and analytical solution for different r.



(a) Spreading of a Delta Peak.



(b) Reflection of a small wave packet.

Figure 7: Some more interesting initial conditions.

3 Elliptic PDE: Poisson Equation

Now we want to solve the Poisson equation

$$\Delta\phi = g$$

in two dimensions – which is

$$\frac{\partial^2\phi}{\partial x^2} + \frac{\partial^2\phi}{\partial y^2} = g \quad \text{with} \quad \phi = \phi(x,y) \text{ and } g = g(x,y) \tag{10}$$

with the boundary conditions

$$\phi|_{\text{boundary}} \equiv 0 \; . \tag{11}$$

Using the centered difference scheme for the second derivative and $\Delta x = \Delta y =: h$ (10) becomes

$$[\phi(i-1,j) - 2\phi(i,j) + \phi(i+1,j)] + [\phi(i,j-1) - 2\phi(i,j) + \phi(i,j+1)] = h^2\, g(i,j) \; ,$$

where $x = i \cdot h$ and $y = j \cdot h$. From this we can simply derive

$$\phi(i,j) = \frac{1}{4}\left[\phi(i-1,j) + \phi(i+1,j) + \phi(i,j-1) + \phi(i,j+1) - h^2 g(i,j)\right] \; . \tag{12}$$

Implementation First I chose the simple *Jacobi Relaxation Method* ("JRM") for a first implementation – use the `-j` flag in `poisson.cpp`. In each of the following algorithms at the beginning a solution for the problem (10) is guessed (this guess is quite rough and must only satisfy the boundary conditions (11)) – and each time in my implementations it's simply $\phi \equiv 0$.

In JRM in each step a new solution ϕ' is computed from the old one (ϕ) according to formula (12): Left hand side is the new and right hand side the old solution.

After each step the new solution is compared to the old one using the metric

$$d(\phi', \phi) := \max_{i,j} \left(|\phi'(i,j) - \phi(i,j)|\right) \tag{13}$$

and if it's smaller than a given ε the loop ends.

The next algorithm I used was *Gauß Seidel Relaxation* ("GSR") – use the `-g` flag. The algorithm is altered only slightly: Instead of using a second array

ϕ', the old one, ϕ, is used to store the values both before and after the step. So during one step the old values of ϕ are overwritten with the new ones.[4]

The third algorithm I used is called *Successive Overrelaxation* ("SOM") – use the `-s` flag. If we name the left hand side of (12) R then this algorithm is

$$(1-\omega)\phi + \omega R \mapsto \phi \; .$$

As you can see only one array[5], ϕ is used like in GSR. In fact for $\omega = 1$ we get back to GSR. The parameter ω (it's called relaxation parameter) must be determined by trial and error. It can be set using the `-w <value>` flag.

Results To check, that all schemes work equally well, cf figure 8. Here all three schemes were used to determine the solution of the problem $g = -1$ for (x, y) within a circle of radius $L/20$ around $(1/2, 1/2)$ and $g = 0$ everywhere else. As it is very hard to see in 8(a) that the values are indeed equal, I separated them in 8(b). But that's not astonishing as I used the same metric (13) for each scheme and each time with the same ε. It might be more interesting, that my program yields the same results as the given one.

Convergence with different meshes I ran the programs (the different algorithms) several times – each time with a different number of nodes. Each time the used CPU-time (using linux' `time` command) was used and the number of iteration steps were counted. Apart from the algorithms each problem was identical, for the SOM the value $\omega = 1.98$ was used; it's similar to the program of the previous paragraph except for the circle's radius is now $L/4$.

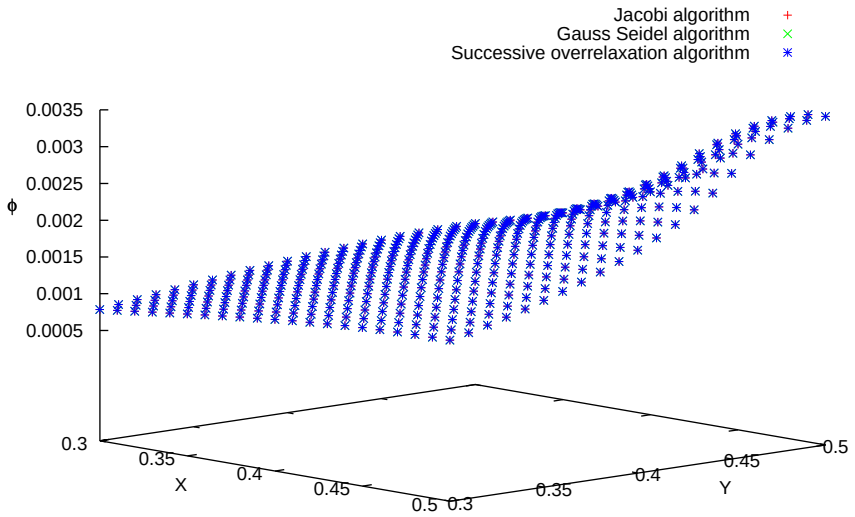
The results are presented in figure 9. You can see clearly that for each scheme the convergence time is a power function of N:[6] $t \propto N^{\alpha}$. The lower α is, the more efficient is the algorithm. In this case, α is the smallest for the SOM – it converges amazingly fast. The αs for JRM and GSR are almost equal but the GSR has a smaller proportional constant, so the GSR is some times faster, but the SOM is some orders of magnitude faster.

You can see, too, that the number of iterations k needed to converge is for SOM almost constant, while for JRM and GSR the same dependency $k \propto N^{\alpha'}$ holds.

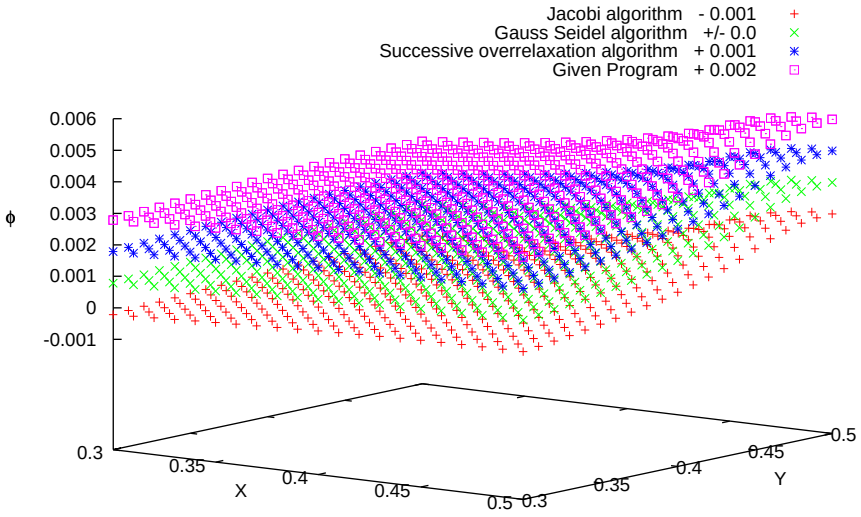
[4] Unfortunately this makes it necessary to copy every single value of the ϕ-array so that one can use the "old" ϕ-values for the comparison with the metric $d(\cdot,\cdot)$ instead of just switching the pointers. This slows the program down slightly.

[5] And thus we have the same problems as in GSR – see footnote 4.

[6] The dependency in the log-log plot is linear: $\log t = \alpha \cdot \log N + \beta$, using $\exp(\cdot)$ on both sides yields $t = \mathrm{e}^{\beta} \cdot N^{\alpha}$.

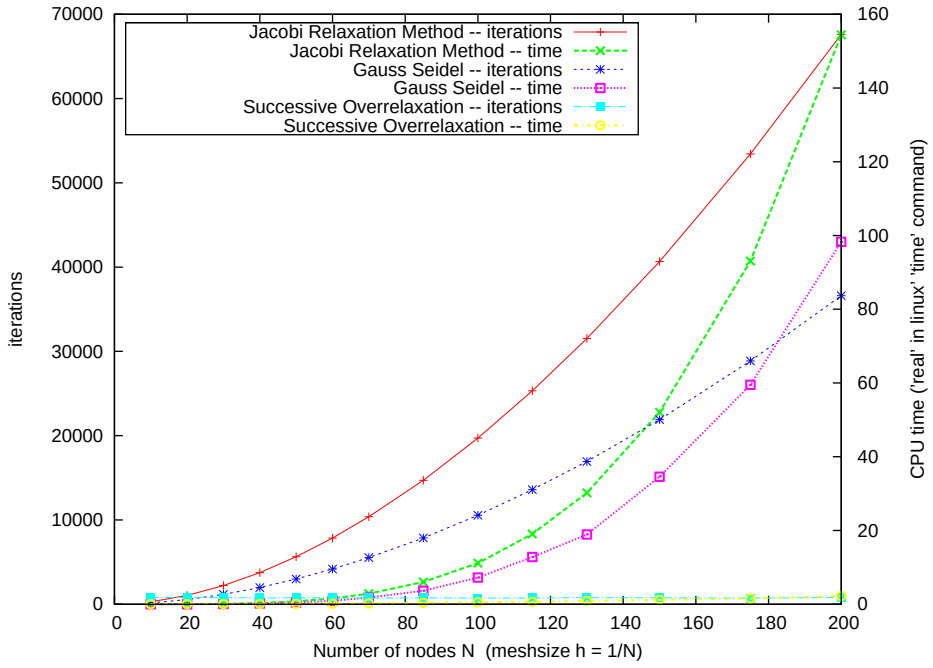


(a) hard to see but identical

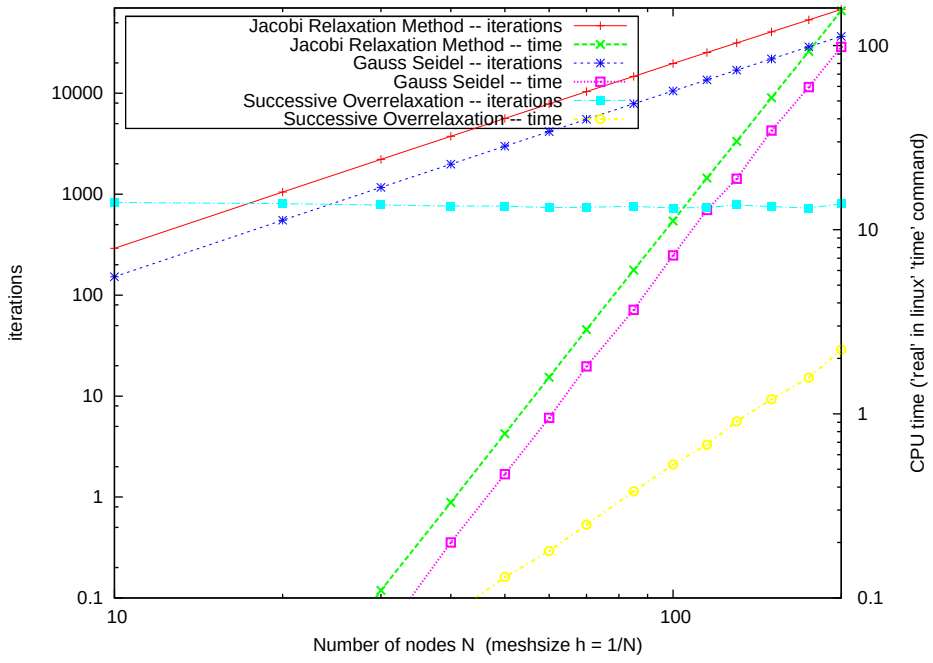


(b) possibly less hard to see that identical

Figure 8: Comparison of the different integration schemes.



(a) lin-lin



(b) log-log

Figure 9: Iterations (thin lines) and time (thick lines) to conversion.

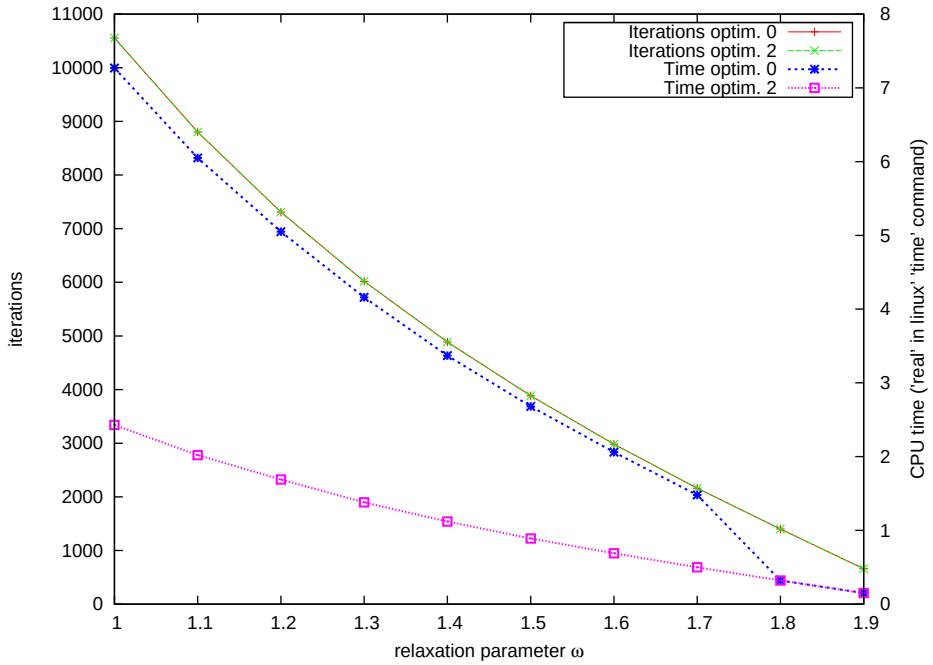


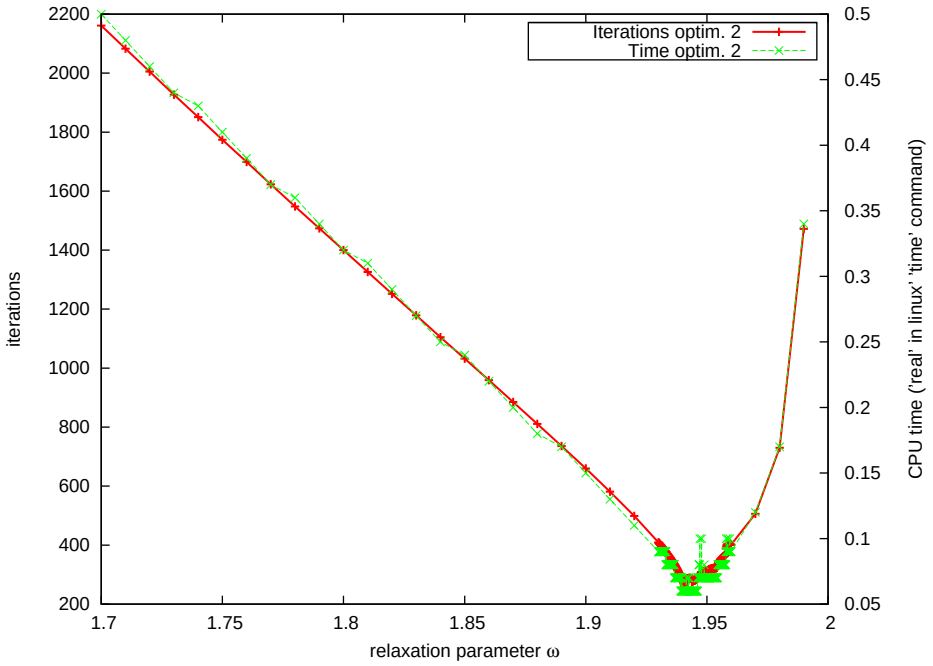
Figure 10: Different values for ω for different compiler optimisation levels (0) and (2)

Best ω parameter In order to optimize the relaxation parameter, I ran the simulation several times for different ω and compared the number of iterations. But before that, take a look at figure 10: Here you can see that using a different optimisation level (here 2 instead of 0) speeds up the computations several times, but not equally for all ω. What I want to show with image 10 is, that it does not necessarily make sense to optimize ω regarding the computation time but that it is probably better to optimize it regarding the number of iterations – and this is what I did.

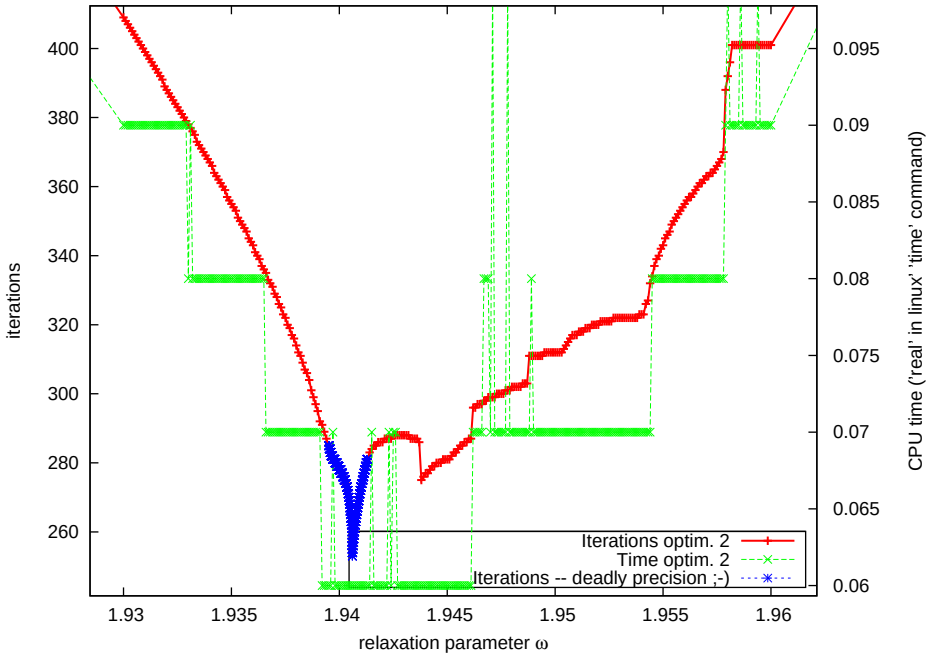
The results are presented in figure 11.[7] As you can see there, the "perfect" omega is $\omega_{\text{optim}} = 1.9406$. So the initial guess 1.98 was quite good ...

Films In order to illustrate how the three different algorithms work, I composed some videos – mostly the same algorithms as in the previous section were used: `jacobi.mp4` illustrates how the JRM works, `gauss.mp4` shows, how the GSR works – these two look quite alike – and finally `succ_palette.mp4` shows how the SOM works (in my opinion this one is the

[7] You can see here, too, that the measured time does not have high enough resolution to determine the optimal ω and that it twitches "strongly".



(a) overview



(b) interesting area

Figure 11: Convergence for different omega.

most interesting one). While viewing, keep the scales of the colour palette in mind – it changes so that the highest point is always yellow.