Finite-Time Singularities

Hanno Rein

St John's College, University of Cambridge

I declare that this essay is work done as part of the Part III Examination. It is the result of my own work, and except where stated otherwise, includes nothing which was performed in collaboration. No part of this essay has been submitted for a degree or any such qualification.

Signed ....H.a.n.n.o...R.e.i.n.....

Contact address:

Hanno Rein
Kieferweg 10
72810 Gomaringen
Germany

http://hanno-rein.de

Contents

1 Introduction

In this essay I am solving an asymptotic approximation of the Euler equations for an inviscid and incompressible perfect fluid in two dimensions. The aim is to start with a smooth initial solution, propagate it through time and find a singularity within finite time. The appearance of a singularity is contrary to the believe that a smooth initial solution of the Navier-Stokes equations remains smooth for all times. Thus a study of the singularity can provide an insight into the nature of the breakdown of the asymptotic approximation I am using.
In the following I say that a singularity is forming if a quantity in the simulation or their derivatives is growing very fast (exponentially) and without any bound. On might think that the breakdown is due to viscosity but the same behaviour was found in a three dimensional inviscid flow [see Souza and Cowley, 2006]. I try to find a similar behaviour in a two dimensional system.
First of all this will be of pure theoretical interest. However, from a physical point of view one can understand the breakdown as a stall. Consequently this research could also become important in a practical environment like aircraft construction.

In section one I derive the differential equations including the corresponding boundary conditions that I solve during the essay later on.
The methods I use to solve the equations numerically are explained in section two. This includes a spectral method and a finite difference scheme. I also explain in detail how I evolve the equations through time.
Section three contains my results. First of all I check that my scheme is actually solving the differential equation correctly to high order. Secondly I display the results for two different initial conditions where singularities form and work out different parameters of the singularity such as speed and time of formation.
The results are finally discussed in section four.
The appendix consists of additional colour plots and my complete C++ source code.

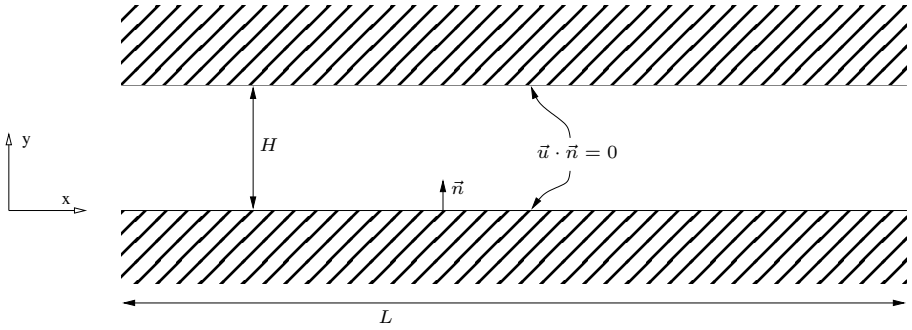


Figure 1: Long pipe setup

1.1 Derivation of differential equations

Let ρ be the density, p the pressure and $\vec{u} = (u, v)^T$ the velocity for an inviscous fluid in two dimensions. Then the Navier-Stokes equations are

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \rho \vec{u} &= 0 \\ \rho \left(\frac{\partial}{\partial t} + \vec{u} \cdot \nabla \right) \vec{u} &= 0. \end{aligned}$$

If we set ρ constant and rescale p they simplify to

$$\begin{aligned} \nabla \cdot \vec{u} &= 0 \\ \frac{\partial u}{\partial t} + (\vec{u} \cdot \nabla) u &= -\frac{\partial p}{\partial x} \\ \frac{\partial v}{\partial t} + (\vec{u} \cdot \nabla) v &= -\frac{\partial p}{\partial y}. \end{aligned}$$

We are in particular interested in a flow through a pipe where the height H is much smaller than the length L (see figure 1). Therefore we make a coordinate transformation and rescale the x-coordinate

$$x \to x' = \epsilon x \tag{1}$$

where ϵ is small because x is large compared to y. After this transformation the x and y coordinates have the same order of magnitude. The other variables transform like

$$\begin{aligned} t \to t' &= \epsilon t \\ y \to y' &= y \\ v \to v' &= \frac{\partial y'}{\partial t'} = \frac{1}{\epsilon} \frac{\partial y}{\partial t} = \frac{1}{\epsilon} v \\ u \to u' &= u \\ p \to p' &= p. \end{aligned} \tag{2}$$

After this transformation the equations we have to solve are (we drop the $'$)

$$\begin{aligned} \not{\epsilon} \frac{\partial u}{\partial x} + \not{\epsilon} \frac{\partial v}{\partial y} &= 0 \\ \not{\epsilon} \frac{\partial u}{\partial t} + \not{\epsilon} u \frac{\partial u}{\partial x} + \not{\epsilon} v \frac{\partial u}{\partial y} &= -\not{\epsilon} \frac{\partial p}{\partial x} \\ \epsilon^2 \frac{\partial v}{\partial t} + \epsilon^2 u \frac{\partial v}{\partial x} + \epsilon^2 v \frac{\partial v}{\partial y} &= -\frac{\partial p}{\partial y}. \end{aligned}$$

Now let $\epsilon \to 0$. We finally get

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{3}$$

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} + v \frac{\partial u}{\partial y} = -g \tag{4}$$

$$0 = -\frac{\partial p}{\partial y} \tag{5}$$

where we have defined

$$g := \frac{\partial p}{\partial x}$$

because the constant term of p is not of interest in the further discussion.

1.2 Boundary conditions

The boundary conditions on the boundaries $y = 0$ and $y = Y(x)$ are

$$\begin{aligned} v\big|_{y=0} &= 0 \\ \vec{u}\cdot\vec{n}\big|_{y=Y(x)} &= 0, \end{aligned}$$

where $\vec{n} = \left(-\frac{\partial Y(x)}{\partial x}, 1\right)^T$. After the transformations (1) and (2) these conditions remain unchanged. I will not use the most general conditions derived here but set $Y(x) = 1 = const$ during all the computations. Thus $\vec{n} = (0, 1)$ which simply implies $v = 0$ on the surfaces $y = 0$ and $y = 1$. The boundary conditions on the surfaces $x = 0$ and $x = L$ are discussed in the next section.

2 Numerical methods

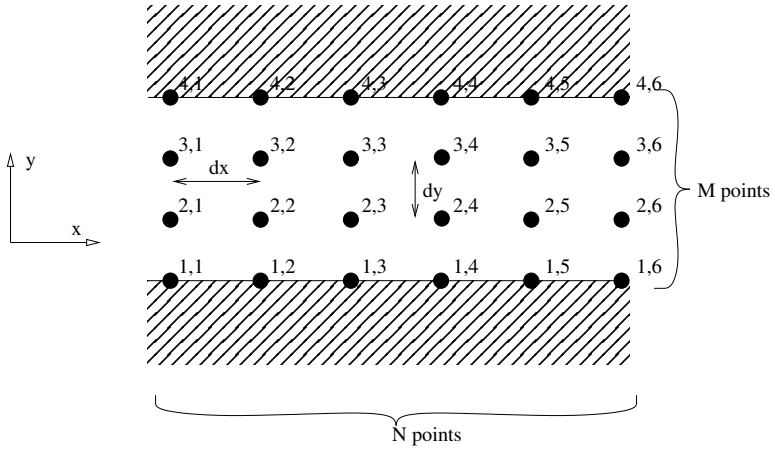


Figure 2: Grid Point Setup.

We assume a periodic boundary in the x-direction. That gives us the change to use a high accuracy spectral method to calculate derivatives in this direction. In the y-direction we work with a centered second order finite difference scheme. The grid point setup is shown in figure 2. I put grid points on the boundaries to simplify the numerical schemes and minimise errors.

2.1 Spectral methods

The derivatives in the x-direction which we have to calculate are $\frac{\partial u}{\partial x}$, $\frac{\partial p}{\partial x}$. This is done with a Fourier series to get a high order accuracy. I use the FFTW package [see Frigo and Johnson, 2006] for all Fourier transformations. One transformation needs $O(n \log n)$ operations. To minimize numerical errors I make it possible to set all modes explicitly to zero which have a norm less then a given value δ. It turns out that these unphysical modes grow exponentially and create large oscillations that break down the simulation even before the "physical" singularity appears. In practice this noise appears at an order of magnitude of about $10^{-11} - 10^{-13}$. Thus $\delta \approx 10^{-10}$ works well in most cases.

2.2 Finite difference scheme

In the y-direction a standard second oder finite difference scheme has been used to approximate $\frac{\partial u}{\partial y}$.

```
u_y[i][j] = (u[i][j+1]-u[i][j-1]) / (2*dy)
```

We don't have to worry about a single-sided finite difference scheme at the boundary because $v|_{\text{boundary}} = 0$ and so we do not need to calculate the derivative $\frac{\partial u}{\partial y}$ at all to evaluate (4).

2.3 Integration

The integrations in the y-direction are calculated with the standard Euler rule to ensure second order accuracy. It is essential that the boundary terms are weighted correctly. Otherwise we loose an order in the accuracy.

2.4 One time step iteration

This section shows how I perform a single timestep for a given initial solution. Suppose we have the values u_0, v_0, g_0 at a time $t = 0$.

- We first calculate

$$f_0(u, v) \quad := \quad -u\frac{\partial u}{\partial x} - v\frac{\partial v}{\partial y}$$

using the methods discussed above. If we calculated f_{-1} one timestep earlier we could predict the value of $f_{\frac{1}{2}}$ at $t = \frac{1}{2}dt$ to second order accuracy in time using the formula

$$\begin{aligned} f_{\frac{1}{2}} &= f_0 + \frac{1}{2}dt\frac{\partial f}{\partial t} \\ &= f_0 + \frac{1}{2}dt\frac{f_0 - f_{-1}}{dt} \\ &= \frac{3}{2}f_0 - \frac{1}{2}f_{-1}. \end{aligned}$$

This formula will be used after we have done the first timestep and f_{-1} is available. Thus the first timestep is still of first order. However, it turns out that this does not change the overall order. Otherwise we could solve this problem with a smaller timestep during the first few steps.

- Integrating equation (3) from 0 to the top $Y(x)$, evaluated at the new timestep $t = dt$ and using the definition of $f_{\frac{1}{2}}$ gives

$$\begin{aligned} &\int_0^{Y(x)} \frac{\partial}{\partial x}\left(u_0 + dt\left(f_{\frac{1}{2}} - \frac{1}{2}g_0 - \frac{1}{2}g_1\right)\right)dy \\ &= -\int_0^{Y(x)} \frac{\partial v}{\partial y}dy = -v\left(Y(x)\right) + v(0) = 0. \end{aligned}$$

This equation is of second order accuracy in time. Every quantity except g_1 is known. Thus this can be used to calculate g_1 except its constant term which is unavailable because of the x-derivative. During this calculation we do not change the constant term of g.

- Now we can easily calculate the new values of u and v at $t = dt$:

$$u_1(x,y) = u_0 + dt \cdot \left(f_{\frac{1}{2}} - \frac{1}{2}g_0 - \frac{1}{2}g_1 \right) \quad (6)$$

$$v_1(x,y) = -\int_0^y \frac{\partial u_1}{\partial x} dy. \quad (7)$$

3 Results

3.1 Order tests

During this section I use the following initial conditions for u at $t = 0$:

$$u[i][j] = \sin\left(\frac{i}{L} \cdot 2\pi\right) \cdot \sin\left(\frac{j}{H} \cdot 2\pi\right) \quad (8)$$

where L and H are the simulation's length and height (see figure 1). To be consistent with the boundary conditions I use equation (7) to get $v|x][y]$ at $t = 0$. The pressure is set to an (arbitrary) constant value.
I choose these conditions because on the one hand they are smooth at the beginning ($t < 1$) but on the other hand they also lead to a singularity for large t ($t \approx 1.6$). This enables me to do order tests and singularity exploration with the same data.
To get an estimate error I do one run with a high accuracy and calculate the difference between the final timesteps of the best and all the other runs using a 2-norm

$$er_\Sigma(\text{run}) \approx \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{M} \left(u_{\text{run}}[i][j] - u_{\text{best}}[i][j]\right)^2}.$$

Note that if I vary the number of grid points I can only sum over those grid points that are at the same position during every run to get a meaningful error estimation.

3.1.1 Order in Δt

For the order test in Δt I use a 100 times 100 grid and a final time $t_{end} = 0.2$. The timesteps computed are $\Delta t = 0.0001$, 0.0002, 0.0004, 0.0008, 0.001, 0.002, 0.004, 0.00625, 0.008, 0.01. The square-root of the error is plotted as a function of Δt in figure 3. As one can see the scheme is second order in time. However, it should be pointed out that the accuracy of the scheme decreases when I use a timestep much smaller than 0.00001. This is expected and due to the accumulation of numerical errors which inevitably occur.

3.1.2 Order in dy

Again I use a final time $t_{end} = 0.2$ and $N = 100$ grid points in the x-direction. The timestep is fixed at $\Delta t = 0.001$. I vary the number of grid points in the y-direction from $M = 51, 61, 71, 81, 91, 101, 151, 201, 301, 401, 501, 601, 701$ to 801. The odd numbers ensure that I have grid points in the middle of the y-direction. I need them to calculate the difference between grid points from various runs at the same position. Again the scheme is second order, as one can see in figure 4.

3.1.3 Order in dx

The order in dx is not as easy to determine as before due to the high accuracy that the scheme has. The accumulated error is plotted in figure 5 using $t_{end} = 0.2$, $\Delta t = 0.001$ and $M = 100$. It vanishes in the limit $dx \to 0$ as expected. However, one might think that it is not even first order by looking at the plot. But one has to keep in mind that this is the accumulated error er_Σ. For example, the error at a single grid point at $dx = 0.002$ is approximately

$$er_\Sigma \cdot \frac{1}{M} \cdot \frac{1}{N} \approx 8 \cdot 10^{-8} \cdot \frac{1}{100} \cdot \frac{1}{500} = 1.6 \cdot 10^{-12}$$

But this is already the scale where computational errors have to be taken into account and thus what is plottet in figure 5 is a sum of the computational and schematic errors.

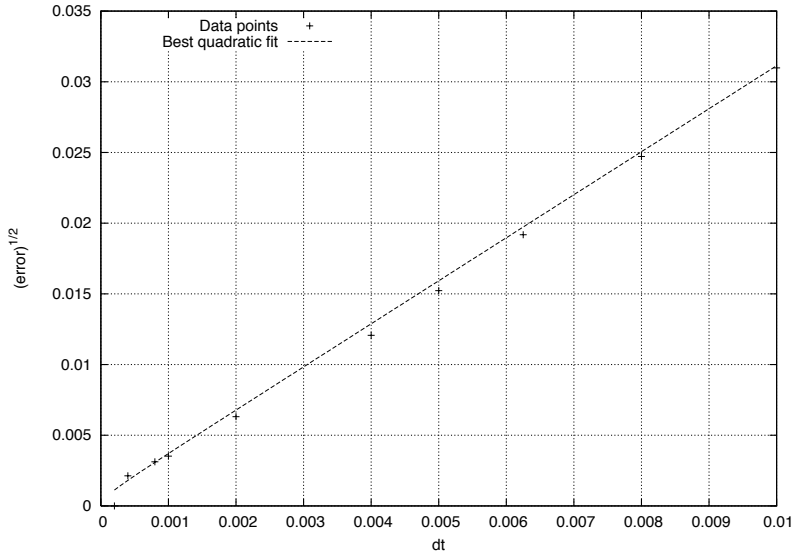


Figure 3: Plot of the square root of the accumulated error er_Σ as a function of the timestep Δt.

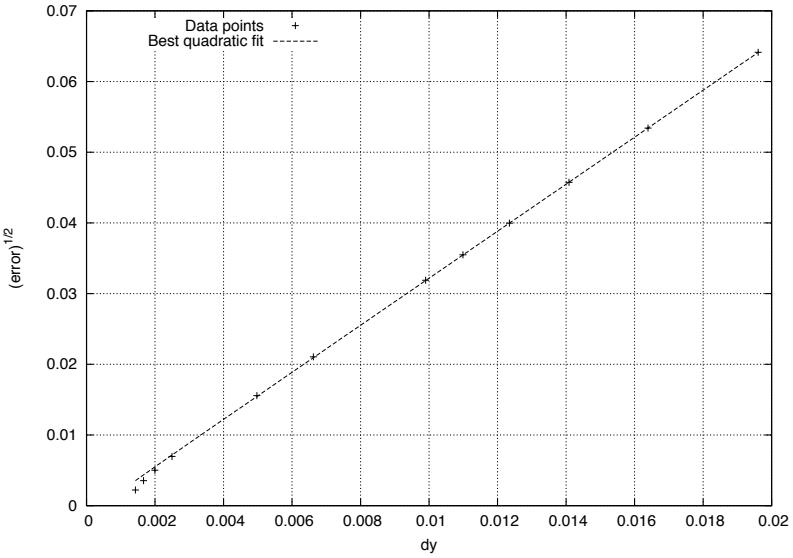


Figure 4: Plot of the square root of the accumulated error er_Σ as a function of the grid-width dy.

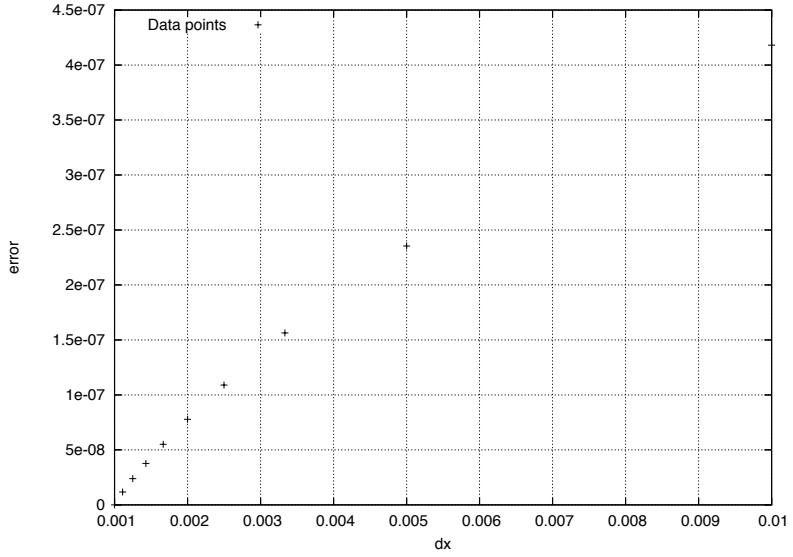


Figure 5: Plot of the accumulated error er_Σ as a function of the grid-width dx.

3.2 Singularities

3.2.1 Most symmetric initial condition

The most symmetric non trivial initial condition is given by equation (8). This corresponds to two contrariwise spinning vertices as one can see in figure 6. The length and direction of each arrow corresponds to the velocity at this point. See also appendix B for colour plots.

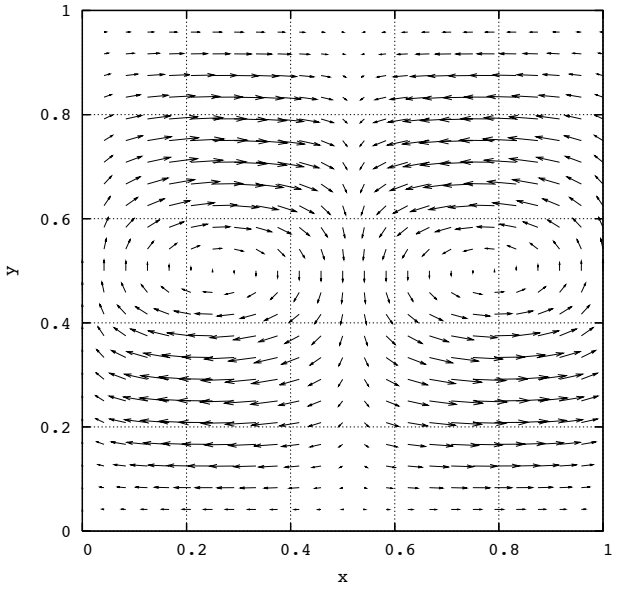


Figure 6: Velocity arrow plot of the initial conditions given by equation (8).

It turns out that a singularity forms at a time $t_{crit} \approx 1.6$. Just before the numerical scheme breaks down because the spectral methods cannot handle a step function. In figure 7 the value

$$\Theta \quad = \quad \frac{1}{\max_{i,j} \left| \frac{\partial u[i][j]}{\partial x} \right|} = \min_{i,j} \left| \frac{1}{\frac{\partial u[i][j]}{\partial x}} \right|$$

is plotted as a function of time for various parameters. In the limit $N \to \infty$, $M \to \infty$ and $\Delta t \to 0$ one can see that this goes to zero as $t \to t_{crit}$, so $\partial u / \partial x \to \infty$. This coincides with my definition of a singularity given earlier. The best linear fit in the range $[1.2 : 1.5]$ for the high accuracy run with $N = 400$, $M = 400$, $\Delta t = 0.00025$ gives $t_{crit} = (1.5650 \pm 0.0004)$.
The velocity in the x-direction at a fixed value of y is $u(x, y = \frac{1}{3}H, t) =: u_{1/3}(x, t)$ and is plotted for various times in figure 8. The profile steepens as $t \to t_{crit}$. This finally leads to the singularity. The Fourier mode coefficients of $u_{1/3}(x, t)$ are plotted in figure 9 for $t = 0.1, 0.3, 0.5, 0.7, 0.9, 1.1, 1.3, 1.5$. The simulation uses $\Delta t = 0.0001$, $M = 40$ and $N = 500$. The numerical noise at large k does not interfere the simulation, so I do not need to suppress it yet. However, at

$t = 1.5$ the plot shows clearly some numerical oscillations. One can see that the spectrum flattens as $t \to t_{crit}$. This spectrum can be approximated by

$$u_{1/3}(k,t) = \tilde{u}(t) \cdot e^{-\alpha(t)k}. \tag{9}$$

Where it is important that the coefficient $\alpha(t)$ is dependent on time. A small value of α corresponds to a flat spectrum. So we expect $\alpha \to 0$ as $t \to t_{crit}$. Thus the time dependence can be used to study the formation time of the singularity and is plotted in figure 10. Therefore I fit the spectrum in each case manually only in the range where it is linear in figure 9. Then I use a linear and a quadratic function as a first approximation to the behavior of α near t_{crit}. This gives the new values of $t_{crit} = (1.46 \pm 0.15)$ in the linear and $t_{crit} = (1.51 \pm 0.13)$ in the quadratic case. Note that these numbers have an error of about 10%. This is mainly due to the difficulty of fitting the behaviour of $u_{1/3}(k,t)$ because it is not exactly the one used in equation (9). Also the real behaviour α is neither linear nor quadratic.

It is also possible to do this discussion with the pressure gradient $g(x)$ instead of $u(x, y = H/3)$ as in the next section. However, it turns out that the pressure gradient is very small ($\approx 10^{-6}$) at the beginning, so the numerical error is larger. Another property of the singularity is its speed at the moment of formation. I therefore calculate the location of the maximum slope of $u_{1/3}(x,t)$ and call this quantity say $x_m(t)$. In figure 11 the positions of the global and local maxima are plotted as a function of time. The singularity forms at $x = 0.5$. One can see that with this initial configuration it's location is not changing at all. Thus $\frac{dx_m}{dt} = 0$. The other maxima at $x \approx 0.2$ and $x \approx 0.8$ do not form singularities. We will see a more interesting behaviour of this quantity in the next section.

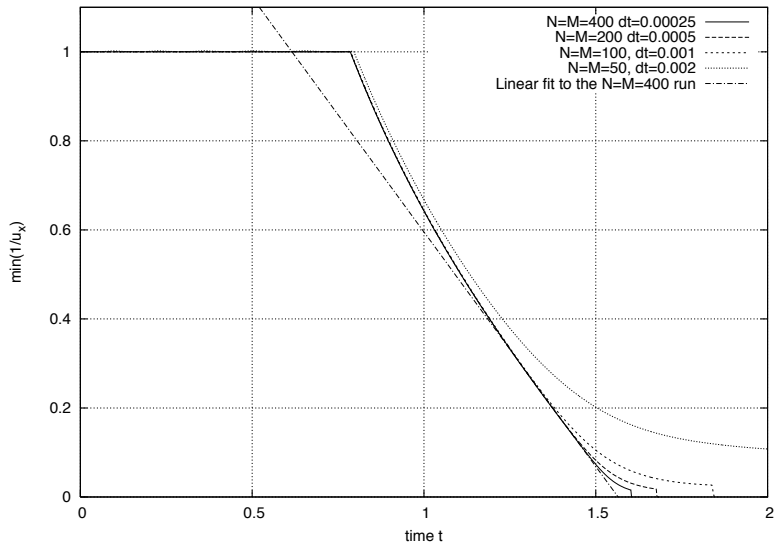


Figure 7: Plot of $\Theta = \min_{i,j} |1/\frac{\partial u[i][j]}{\partial x}|$. The best linear fit in the range $[1.2 : 1.5]$ gives $t_{crit} = (1.5650 \pm 0.0004)$.

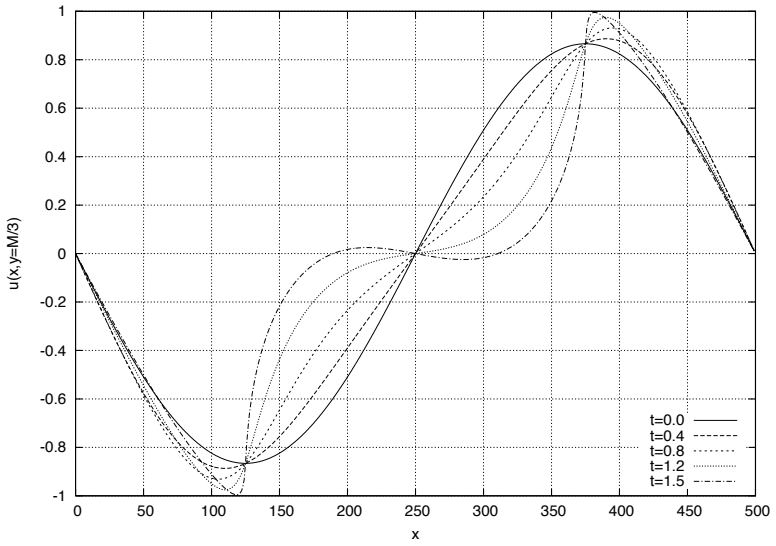


Figure 8: The solution for $u_{1/3}(x,t) = u(x, y = 1/3 H, t)$ at various times. This run uses $\Delta t = 0.0001$, $M = 40$ and $N = 500$.

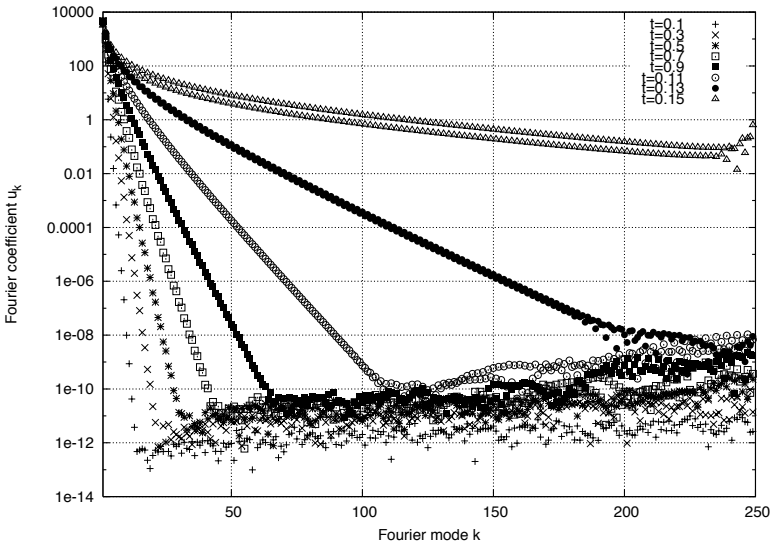


Figure 9: Plot of the Fourier coefficients of u at $y = \frac{1}{3}M$. No cut-off δ was used in this run. One can see the numerical noise growing at large k and late times.

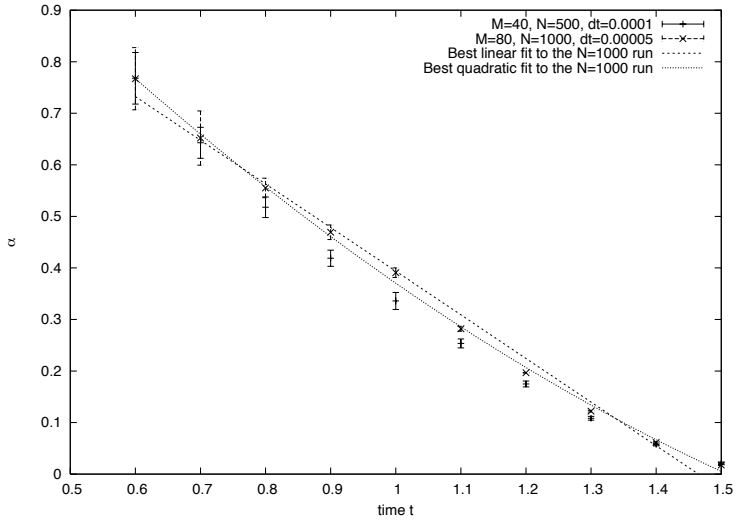


Figure 10: Plot of coefficient α. The errorbars represent the residual error of the fit.

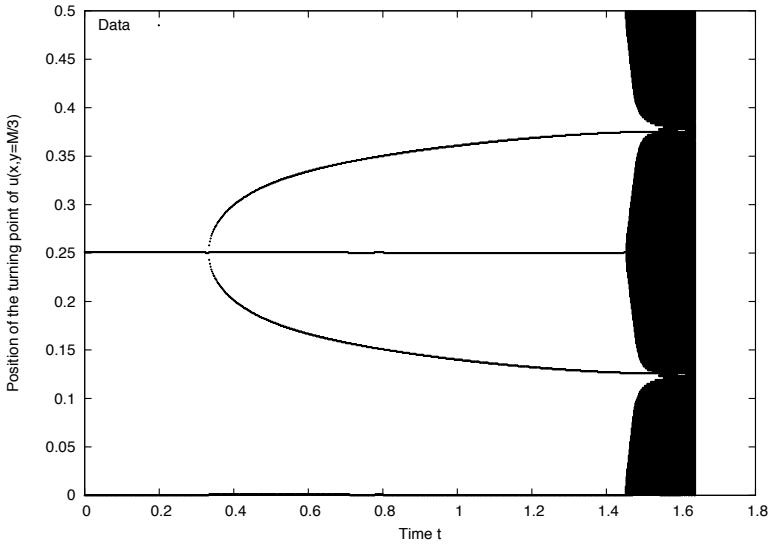


Figure 11: Positions where the slope of $u_{1/3}(x,t)$ is a local maximum. At times $t > 1.4$ oscillations occur. The detection of the maximal slope is no longer possible.

3.2.2 x^4 initial condition

Now I use these initial conditions for u at $t = 0$

$$u[i][j] = \sin\left(\frac{i}{L}\cdot 2\pi\right)\cdot\left[-10\left(\frac{j}{H}\right)^4 + 1.8\left(\frac{j}{H}\right)^2 - 0.025\right].$$

Again I use equation (7) to get $v[i][j]$ at $t = 0$. The special parameters -10, 1.8, -0.025 ensure that the boundary conditions are satisfied. As in the previous example a singularity forms. This time the scheme breaks down at $t_{BD} \approx 1.5$. The behavior of

$$\Theta = \min_{i,j}\left|\frac{1}{\frac{\partial u[i][j]}{\partial x}}\right|$$

is plotted in figure 12 as a function of time. One can see that the result depends very strongly on the grid-width dx. This is due to the fact that at times $t \approx t_{crit}$ the velocity profile v has structures of size dx (see also pictures in appendix C). At this point the scheme is in a quasi stable configuration which results in the plateau shown in figure 12. However, sooner or later oscillations break down the scheme completely. I plotted the height of this plateau $h_\theta(dx)$ as a function of the grid-width in figure 13. On can see that $h_\theta(dx) \to 0$ as $dx \to 0$. Thus $u[i][j] \to \infty$ as $t \to t_{crit}$. This shows that we are looking at a real and not just a computational singularity.
The best linear fit to Θ in the high accuracy run ($N = M = 400$, $\Delta t = 0.00025$) in the range $[1.0 : 1.1]$ gives a singularity forming time $t_{crit} = (1.1590 \pm 0.0002)$. Note that this is much smaller than the time when the scheme break down ($t \approx 1.5$).
Notice also the kink in figure 12 at $t \approx 0.9$. Although this looks very dramatic nothing special happens. The kink appears because the position of the maximum of $\frac{\partial u[i][j]}{\partial x}$ changes. A local maximum becomes the new global one.
In the previous section I discussed the behaviour of $u_{1/3}(x,t)$. This time I concentrate on the pressure gradient $g(x,t)$. This quantity is already independent of y. We do not need to specify a privileged line $y = const$. However, there is no systematic difference between the two. The Fourier coefficients of $g(x,t)$ are shown in figure 14 at times $t = 0.05$, 0.25, 0.50, 0.75, 1.00 and 1.25. Again, we see that the spectrum flattens as $t \to t_{crit}$. As before, it can be approximated by equation (9) in an appropriate interval. The result, the time-dependence of the parameter α, is plotted for two different runs in figure 15. At late times ($t > 0.25$) the behaviour is linear. The best linear fit in this range gives a new singularity formation time $t_{crit} = (1.20 \pm 0.01)$.
The last thing to look at is the singularity speed. I calculate the location of the maximum slope $x_m(t)$ as in the previous section. Figure 16 shows this quantity as a function of time. At the beginning we have the maximum constant at $x = 0.5$. Then two new maxima appear and move away from $x = 0.5$. Unlike in the first example, they stop, turn around and move back towards $x = 0.5$. Surprisingly they seem to arrive there exactly at the time t_{crit}. To verify this, I plot the interesting region of figure 16 in figure 17 again and use a linear fit to extrapolate the behaviour close to t_{crit}. I get $t_{crit} = (1.152 \pm 0.002)$ if I assume that $x_m(t) = 0.5$ at t_{crit}.

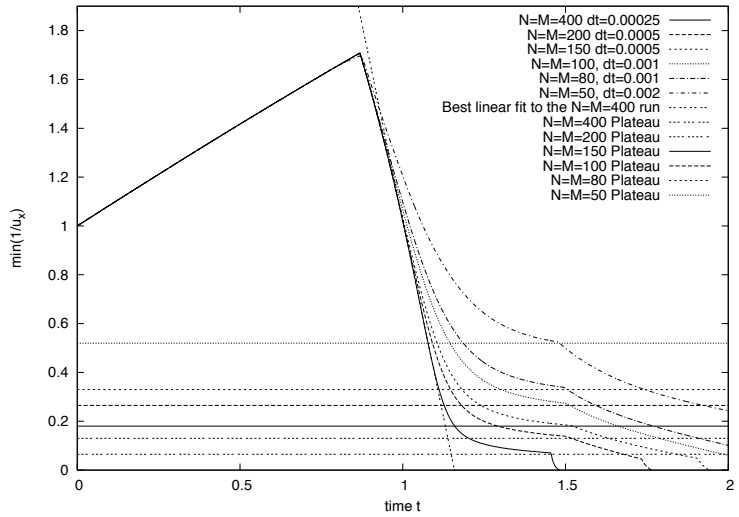


Figure 12: Plot of $min_{i,j} |1/\frac{\partial u[i][j]}{\partial x}|$ with various accuracy. The plot also includes the plateaus $h_\theta(dx)$ at $t \to 1.5$. At $t \approx 1.5$ the numerical scheme breaks down. The data afterwards can be ignored. The best linear fit in the range $[1.0 : 1.1]$ to the high accuracy run gives $t_{crit} = (1.1590 \pm 0.0002)$.

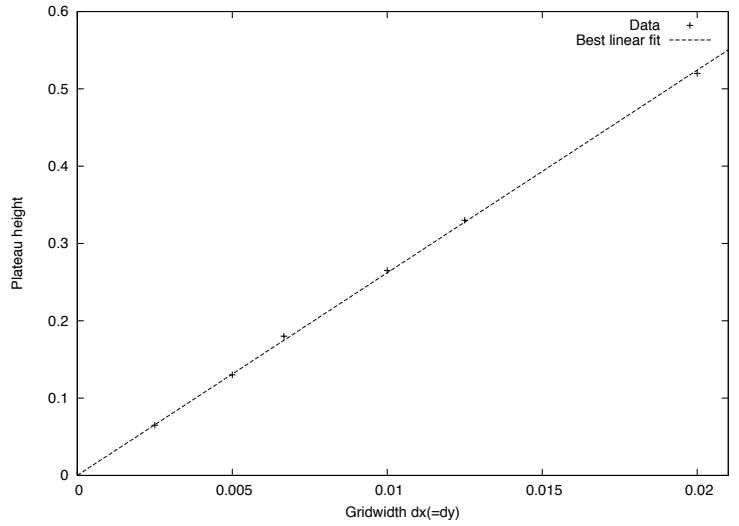


Figure 13: Plot of the plateau-height $h_\theta(dx)$ as a function of grid-width and the best linear fit.

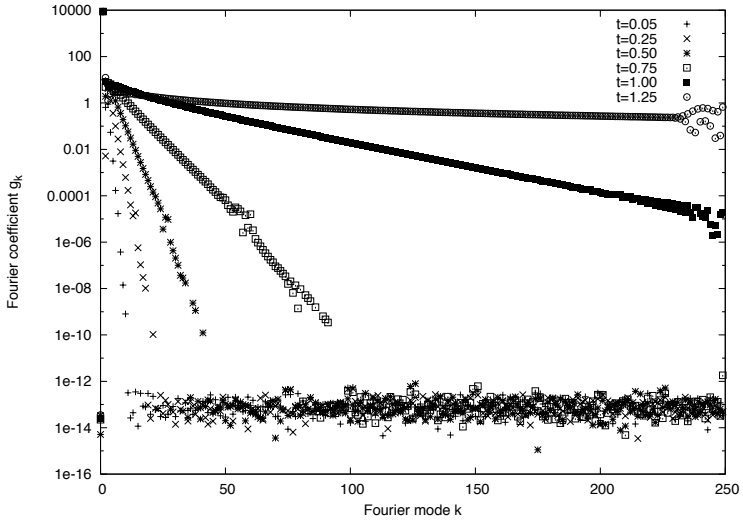


Figure 14: Plot of the Fourier coeffecients $g(k,t)$ at various timesteps with $M = 40$, $N = 1000$, $\Delta t = 0.0001$ and $\delta = 10^{-10}$

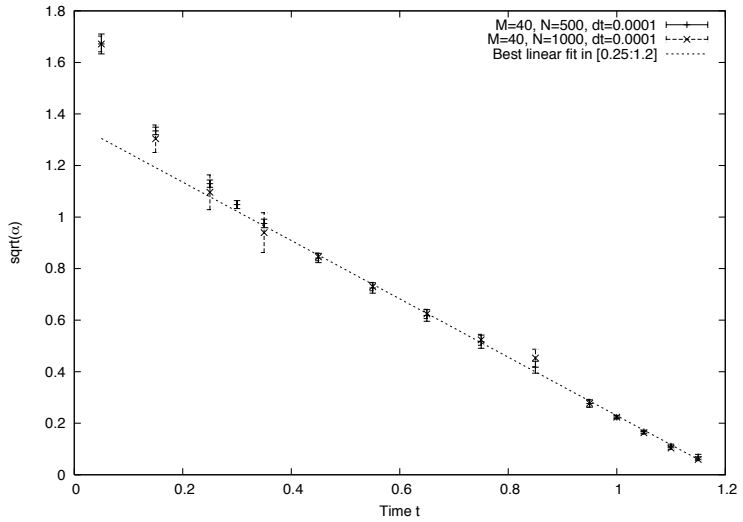


Figure 15: Plot of coeffecient $\sqrt{\alpha}$. The errorbars represent the residual error of the fit. The best linear fit gives $t_{crit} = (1.20 \pm 0.01)$.

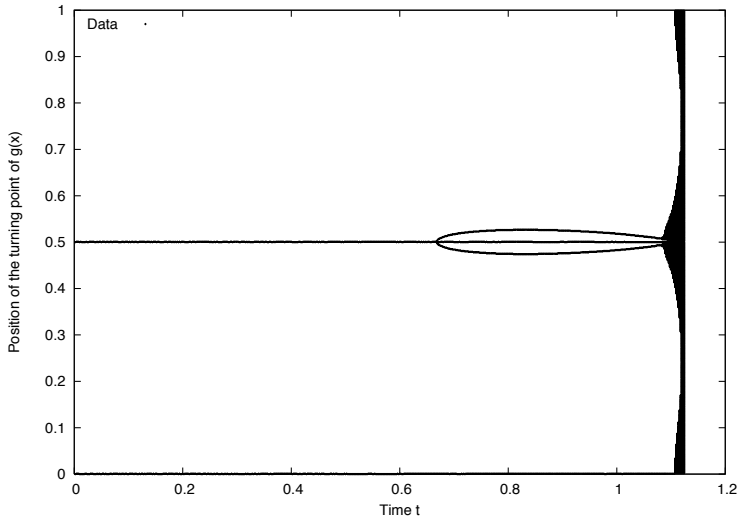


Figure 16: Positions where the slope of $g(x,t)$ is a local maximum. I use $M = 40$, $N = 1000$, $\Delta t = 0.0001$ and $\delta = 10^{-10}$ during this run.

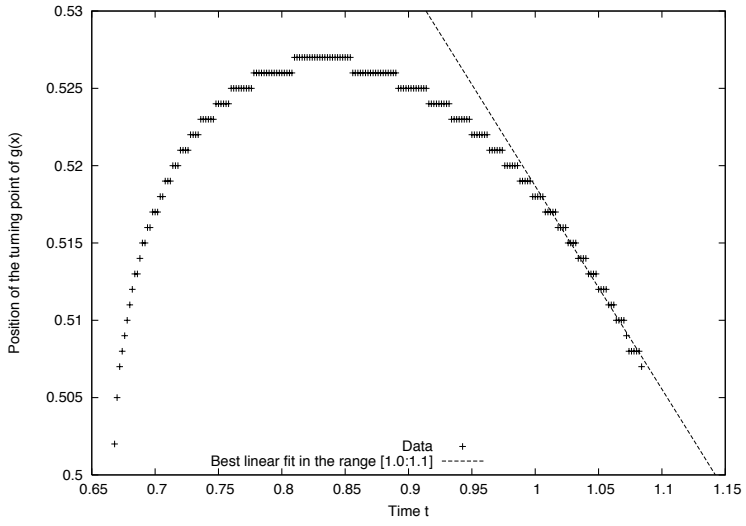


Figure 17: Positions where the slope of $g(x,t)$ is a local maximum. This is a closeup of figure 16. The best linear fit in the range [1.0:1.1] gives $t_{crit} = (1.142 \pm 0.002)$.

4 Conclusions

Let me point out again that the order tests in section 3.1 show clearly that my code is solving the right equations to a high accuracy. During all the calculations the required accuracy did not blow up the runtime beyond a few hours, thanks to the second order schemes used. However, a possible extension to this project would probably include a higher order scheme.
It is easy so find initial conditions that form a singularity sooner or later. Actually it is quite hard to find a non trivial one that does not. I discussed only two of them in this essay because it would go beyond the scope of this essay otherwise.
The results in section 3.2.1 show clearly that a singularity forms if I start with the smooth inital conditions in equation (8). The breakdown of the numerical scheme is due to the appearance of small structures or large k (see also figures in appendix B). This is not only the breakdown of the scheme but also the breakdown of the asymptotic assumption (with $\epsilon \to 0$) that is no longer valid. The different methods of estimating the forming time t_{crit} give rise to quite different values in the range $1.46 - 1.56$. The large interval is due to several reasons. Firstly, one could of course improve the accuracy by using more grid points and smaller timesteps. However this would result in very long runtimes and would also not improve every measured quantity. For example the coefficient α cannot be approximated better if I simply add more grid-points because the small oscillations (large k) are already below the machine accuracy (at early times). Currently I use double precision variables. So an improvement would cause a lot of trouble and a lot of additional computing time. Secondly the large interval is due to the fact that it is very hard to fit a linear curve to the power-spectrum in figure 9. I selected a convenient interval for the fit by hand. This is a large source of error but no automated method worked good enough for me.
All this also applies to the singularity in section 3.2.2. This time the interval of t_{crit} is a bit smaller: $1.15 - 1.20$. Again the method which uses the coefficient α gives a value that seems a bit to large ($t_{crit} = 1.20$). The times that arise from the other methods using the singularity speed and the maximum of u agree with each other within the fit errors. Of course it could be just a coincidence that these two times agree. However they do so well that it appears to be more systematic.
There are several options for further work:

- It would be interesting if slightly different initial conditions to those used in section 3.2.2 still provide the exciting feature of the match of the singularity forming times when using the singularity speed method.
- One could also try to find totally different initial conditions that show the same or maybe a new behaviour.
- I used the special case $Y(x) = const.$ One could do similar calculations with a more general boundary condition.
- It should also be pointed out that one could improve the presented results by using more computing time.

Unfortunately all this goes beyond the limited scope of this essay.

A References

Paul Blackburn and Miguel Gea Milvaques. PNGwriter. *Open source software package*, 2006. URL `http://pngwriter.sourceforge.net/`.

Matteo Frigo and Steven G. Johnson. FFTW. *Open source software package*, 2006. URL `http://fftw.org/`.

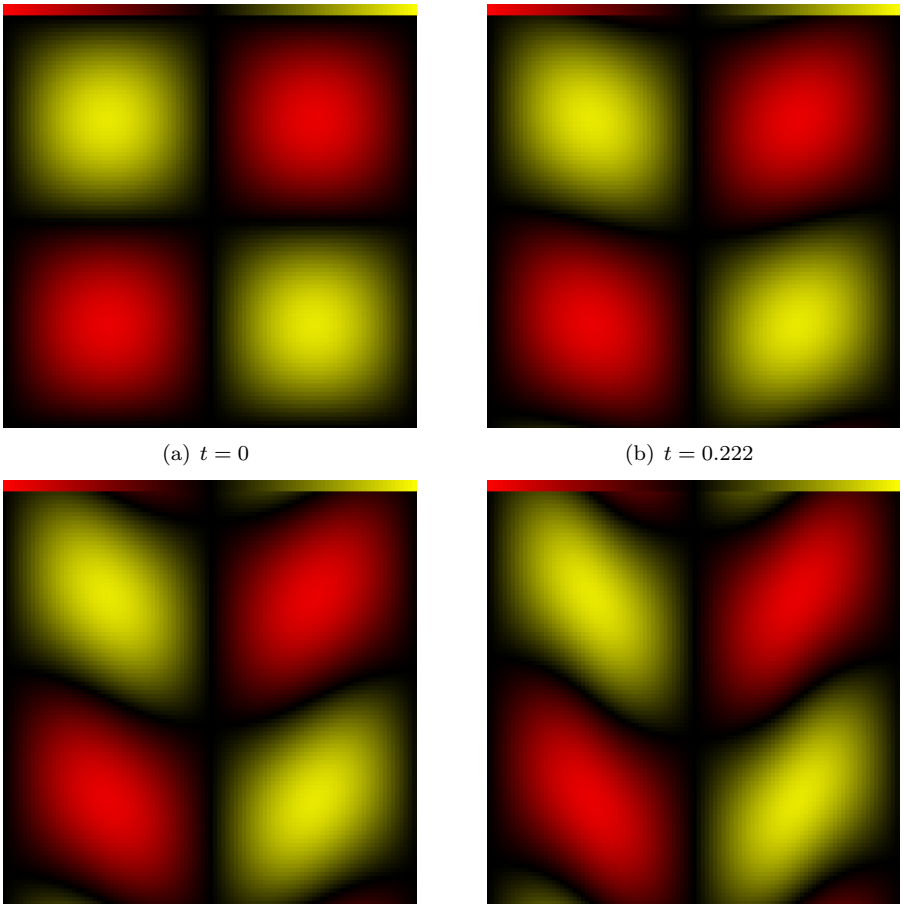
Wilhelm Kley. Skript zur Vorlesung Numerische Hydrodynamik. *Lecture notes*, 2006. URL `http://www.tat.physik.uni-tuebingen.de/~kley/lehre/numhydro/`.

Culbert B. Laney. *Computational Gas Dynamics*. Cambridge University Press, 1998.

Max O. Souza and Stephen J. Cowley. On incipient vortex breakdown. *J. Fluid Mech.*, preprint, 2006. URL `http://damtp.cam.ac.uk/people/cowley/papers/incipient_vbd.ps`.

B Colour plots of the simulation 3.2.1

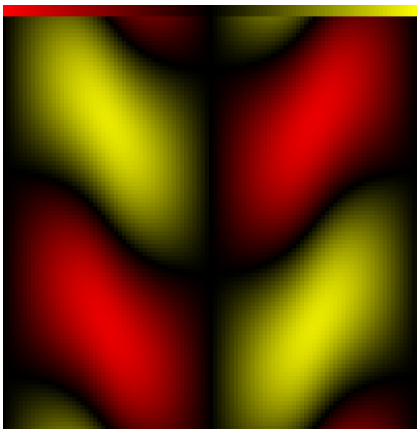
I wrote the routine that creates the following colour plots. PNGwriter [Blackburn and Milvaques, 2006] implements the export as a PNG image. The key to the colours is plotted on the top of each image. Red corresponds to a value of -1.1, yellow to a value of 1.1. All the plots are made with a timestep $\Delta t = 0.001$ and a 100 times 100 grid.



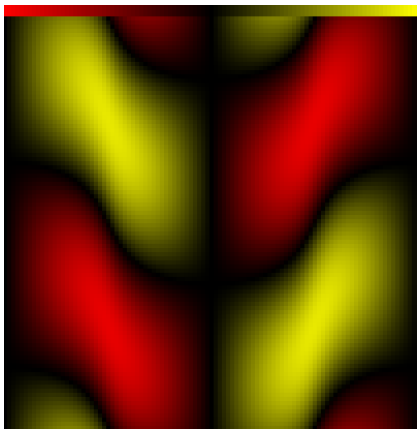
(a) $t = 0$ (b) $t = 0.222$

(c) $t = 0.444$ (d) $t = 0.666$

Figure 18: Colour plots of the velocity in the x-direction u. Red corresponds to a value of -1.1, black to 0 and yellow to 1.1.

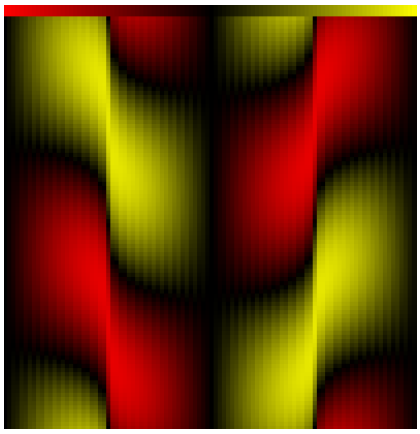


(e) $t = 0.888$

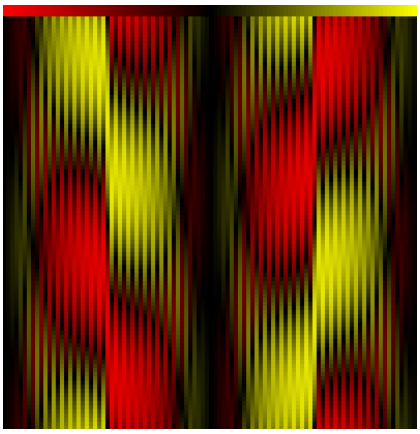


(f) $t = 1.110$

(g) $t = 1.320$

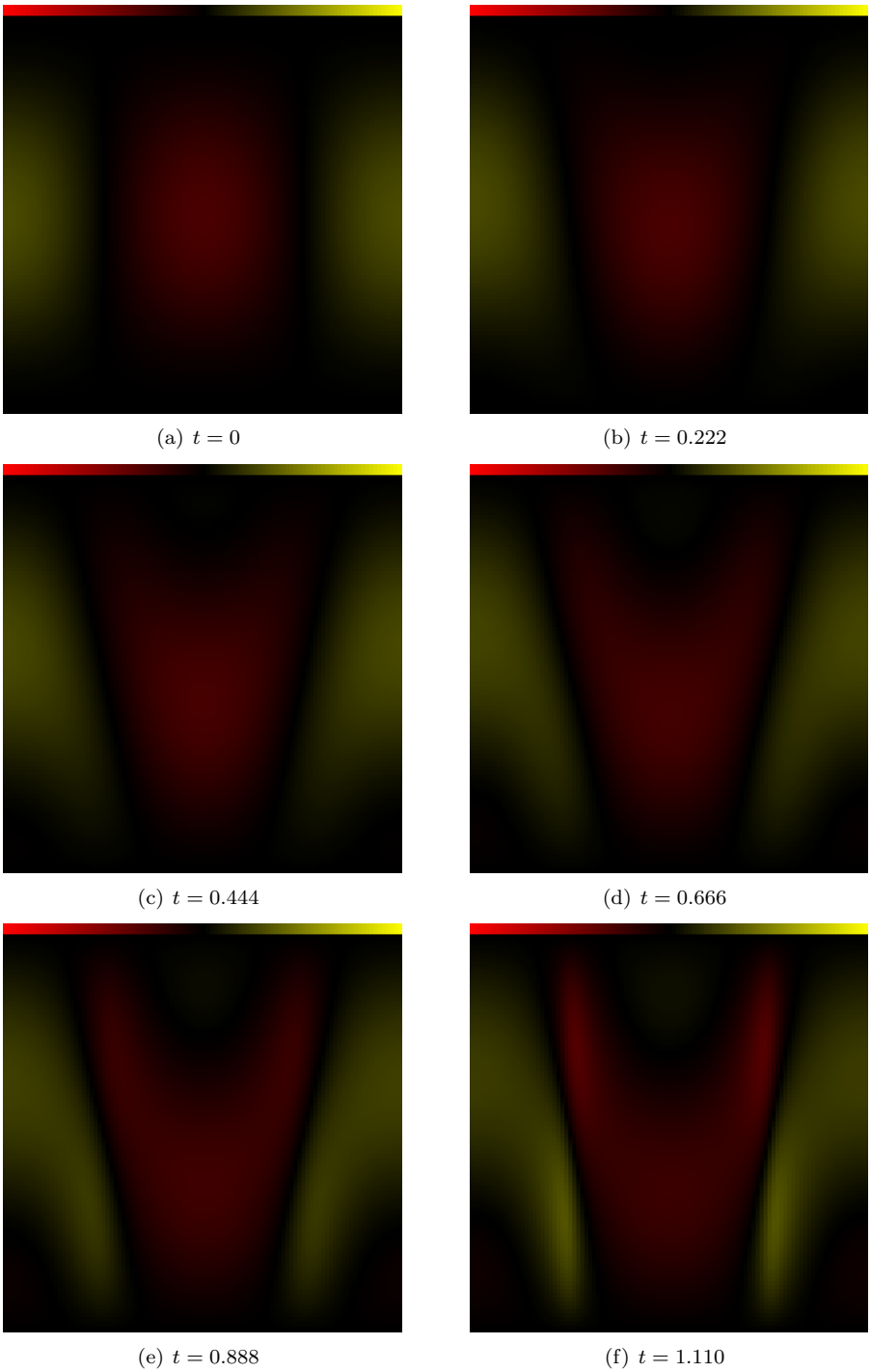


(h) $t = 1.554$



(i) $t = 1.776$

Figure 18: Continued.

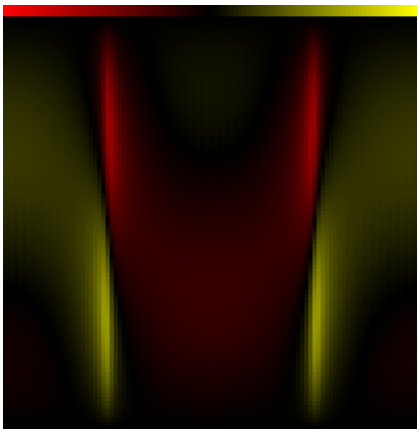


(a) $t = 0$ (b) $t = 0.222$

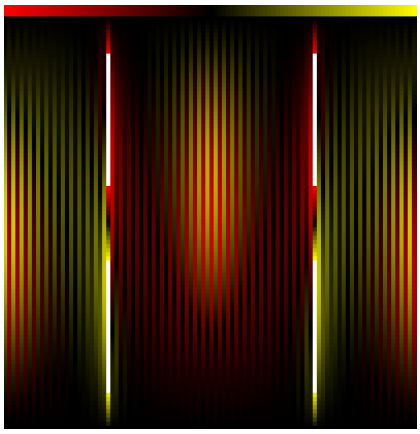
(c) $t = 0.444$ (d) $t = 0.666$

(e) $t = 0.888$ (f) $t = 1.110$

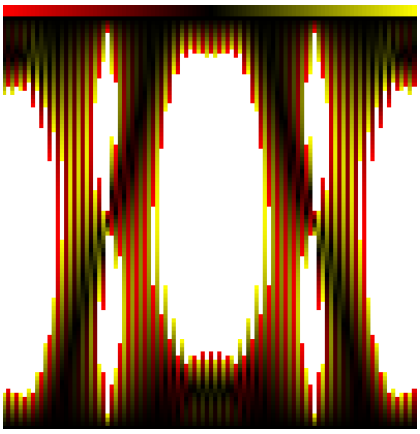
Figure 19: Colour plots of the velocity in the y-direction v. Red corresponds to a value of -1.1, black to 0 and yellow to 1.1. A white colour is plotted if the value is out of range.



(g) $t = 1.332$



(h) $t = 1.554$

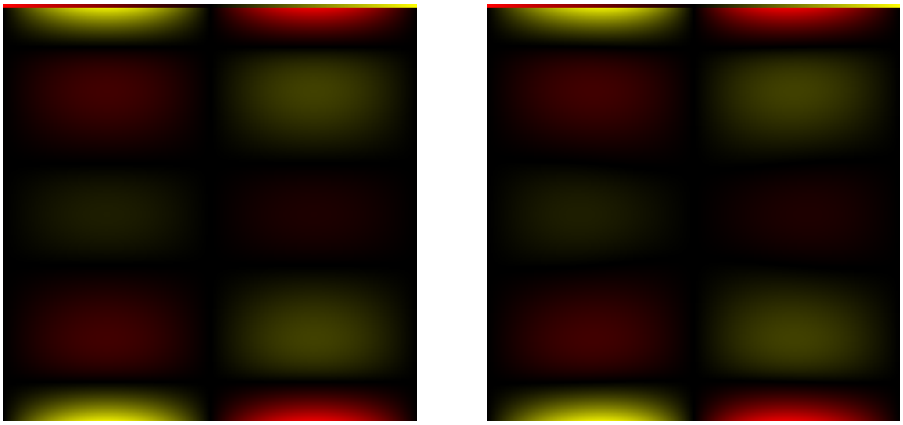


(i) $t = 1.776$

Figure 19: Continued.

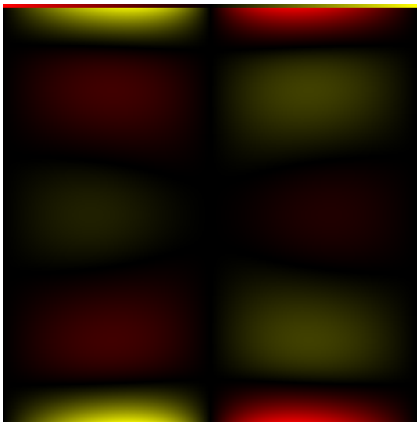
C Colour plots of the simulation 3.2.2

All the plots in this section are made with a timestep $\Delta t = 0.0001$ and a 200 times 200 grid. Note especially the small structures of the size of a grid point that form in the middle at late times.

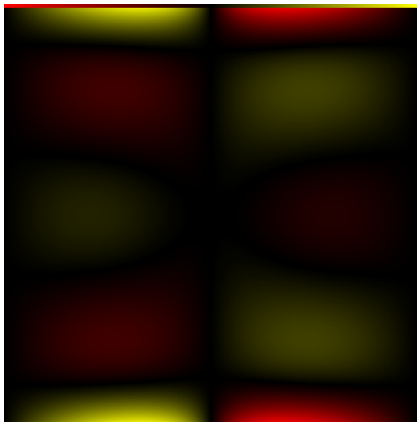


(a) $t = 0$

(b) $t = 0.1875$

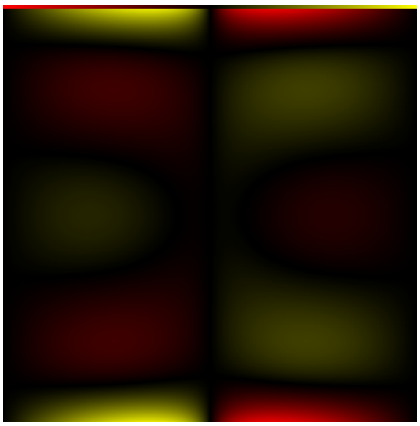


(c) $t = 0.3750$

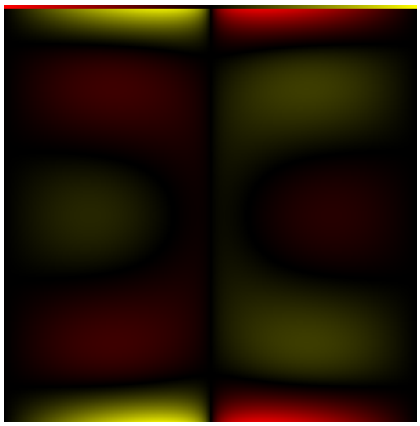


(d) $t = 0.5625$

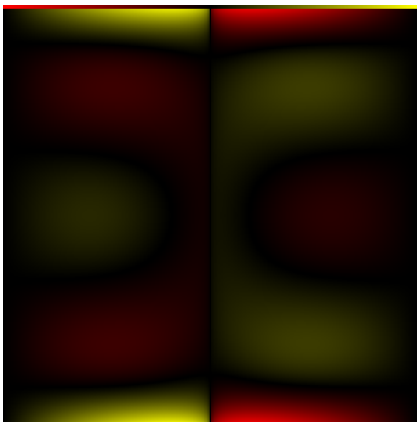
Figure 20: Colour plots of the velocity in the x-direction u. Red corresponds to a value of -1.1, black to 0 and yellow to 1.1.



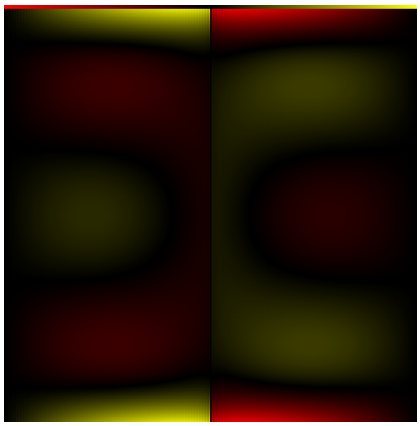
(e) $t = 0.7500$



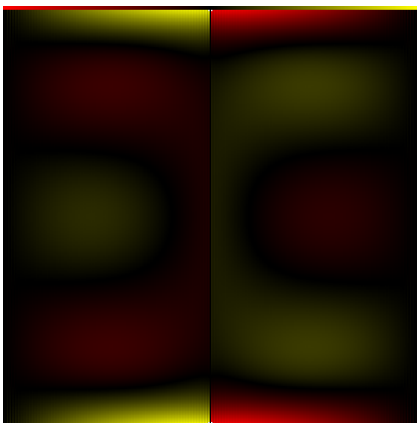
(f) $t = 0.9375$



(g) $t = 1.1250$

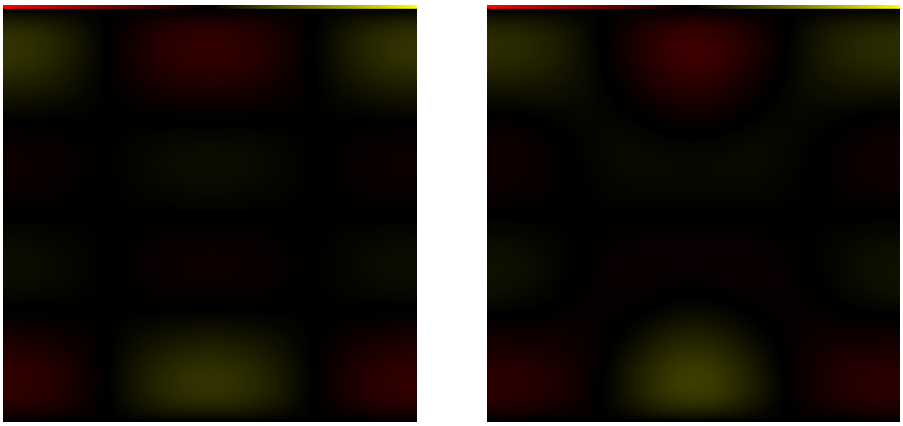


(h) $t = 1.3125$



(i) $t = 1.5000$

Figure 20: Continued.

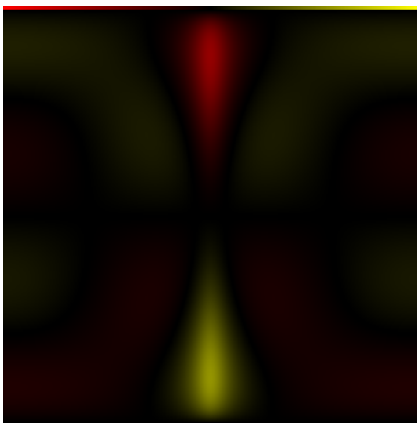


(a) $t = 0$

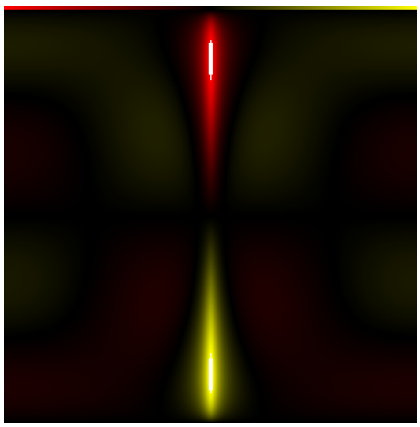
(b) $t = 0.1875$

(c) $t = 0.3750$

(d) $t = 0.5625$

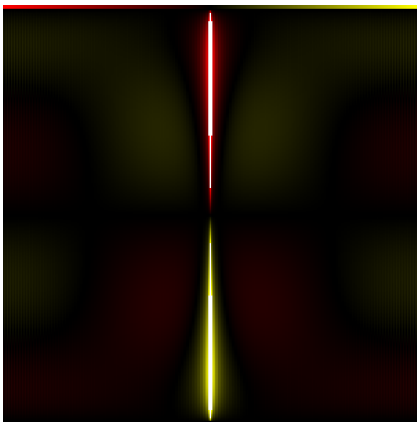


(e) $t = 0.7500$

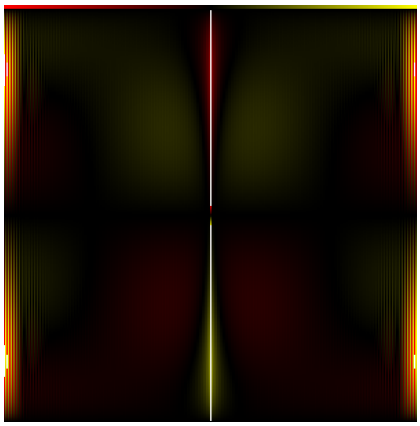


(f) $t = 0.9375$

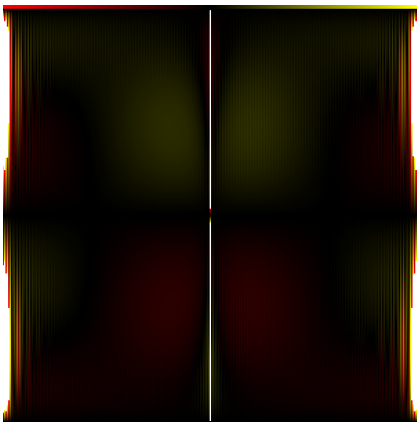
Figure 21: Colour plots of the velocity in the y-direction v. Red corresponds to a value of -0.2, black to 0 and yellow to 0.2.



(e) $t = 1.1250$



(f) $t = 1.3125$



(g) $t = 1.5000$

Figure 21: Continued.

D Source Code Listings

This section includes all the source code I wrote for this essay. It can be compiled using any C++ compiler and nearly every operating system. I tested it under MacOSX 10.4 and Linux. Please note that the FFTW package has to be installed and the libraries need to be linked. I did not find a good plotting program to plot my two dimensional colour data. Therefore I wrote my own plotting routine which can be found in the file `picture.cpp`.

Listing 1: main.h

```
/*
 *  main.h
 *  Main header file. Defines often used functions.
 *  Created by Hanno Rein.
 */

#include <complex.h>
#include <math.h>

//Functions

void iterate(double);
double Y(double);
double Y(int);
void haveugetv();
void setinitialconditions();
void predictfab();
void predictfab_better();
void getg1();
void diffusionu();
void uj_derivative();
double max(double**);
double max(double*);

// Constants

complex<double> I(0,1.);

// Variables
```

Listing 2: main.cpp

```
/*
 *  main.cpp
 *  Main File, includes all mathematical functions.
 *  Created by Hanno Rein.
 *
 */

#include <iostream>
#include <fstream>
#include <complex>
#include <fftw3.h>                  // Fourier transform
#include <stdio.h>
```

```
13 #include <math.h>
14 #include "picture.h"     // Output pictures
15 #include "main.h"
16
17 using namespace std;
18
19 // Precompiler instructions on what to print
20
21 //#define OUTPUT_PICTURES 1
22 //#define OUTPUT_TXT 1
23 //#define OUTPUT_TXT_G_COEFF 1
24 //#define OUTPUT_TXT_U13_COEFF 1
25 //#define OUTPUT_TXT_G_MAXSLOPE 1
26 //#define OUTPUT_TXT_U13_MAXSLOPE 1
27 //#define OUTPUT_TXT_MAXUX 1
28
29
30 // Precompiler instruction on the physical parameters
31 #define N                       1000            // Points in x
       -direction
32 #define M                       40                      //
       Points in y-direction
33 #define MISSPRINTX      1                       // Do print
       every MISSPRINTX grid point in the x-direction
34 #define MISSPRINTY      1                       // Do print
       every MISSPRINTY grid point in the y-direction
35 //#define ONLYPRINTY    M-(M-1)/2-1
36 //#define ONLYPRINTY    M/3
37 #define dt                      0.0001          // Timestep
38 #define sizeY           1.0
39 #define dy                      (sizeY/(double)(M-1))
40 #define dx                      (1./(double)(N))
41
42 #define T                       2.                      // End
       time
43 #define eps                     1E-10           // Cutoff
       value (delta)
44
45 #define pictures        1000            // Number of pictures/
       outputs generated
46 #define picturestart 0
47
48 int picslip =(int)((T-picturestart)/(double)pictures/dt);
49
50
51 // Grid variables
52 double** u              = new double*[N];
53 double* uj              = new double[N]; // temp for fftw
54 std::complex<double>*  c=new std::complex<double> [N/2+1]; //
       temp for fftw
55 double** v              = new double*[N];
56 double*  g              = new double[N]; // p is only x
       dependent
57 double*  g1             = new double[N];
```

```
58 double** s                = new double*[N];
59 double**  fab    = new double*[N];
60 double**  fab_old  = new double*[N];
61 double**  fab_old2 = new double*[N];
62
63 // Variables for outputs of various kinds
64 fstream o0, o4, o6;
65
66 //FFTW plans - p1 is to transform into momentum space, p2
       transforms back.
67 fftw_plan p1,p2;
68
69 //Tools for calculation of max_ij(*)
70 double max(double* var){
71         double temp = var[0];
72         for(int i=0;i<N;i++)
73                 if (temp<var[i]) temp=var[i];
74         return temp;
75 }
76 double max(double** var){
77         double temp = var[0][0];
78         for(int i=0;i<N;i++)
79                 for(int j=0;j<M;j++)
80                         if (temp<var[i][j]) temp=var[i][j];
81         return temp;
82 }
83
84 // Output/picture function
85 void picture(int num,double t){
86         char ret[64];
87 #ifdef OUTPUT_PICTURES
88         sprintf(ret,"./png/u_%d.png",num);
89         picture(ret,u,N,M);
90         sprintf(ret,"./png/v_%d.png",num);
91         picture(ret,v,N,M);
92 #endif
93 #ifdef OUTPUT_TXT
94         sprintf(ret,"./txt/%d.txt",num);
95         fstream o1;
96         o1.open(ret, ios::out);
97         for(int i=0;i<N;i++)
98                 for(int j=0;j<M;j++)
99                 #ifdef MISSPRINTY
100                         if (i%MISSPRINTX==0&&j%MISSPRINTY==0)
                            o1 << i <<"\t"<< j <<"\t"<< u[i][j
                            ] <<"\t"<< v[i][j] << endl;
101                 #endif
102                 #ifdef ONLYPRINTY
103                         if (i%MISSPRINTX==0&&j==ONLYPRINTY) o1
                             << i <<"\t"<< j <<"\t"<< u[i][j]
                            <<"\t"<< v[i][j] << endl;
104                 #endif
105         o1.close();
106         sprintf(ret,"./txt/g%d.txt",num);
```

```
107         fstream o2;
108         o2.open(ret, ios::out);
109         for(int i=0;i<N;i++)
110                 o2 << i << "\t" << g[i] << endl;
111         o2.close();
112 #endif
113
114 #ifdef OUTPUT_TXT_G_COEFF
115         sprintf(ret,"./txt/g_coeff_%d.txt",num);
116         fstream o3;
117         o3.open(ret, ios::out);
118         for(int i=0;i<N;i++)
119                 uj[i]=g1[i];
120                 fftw_execute(p1);
121         for(int i=0;i<N/2;i++)
122                 o3 << i << "\t" << abs(c[i]) << endl;
123         o3.close();
124 #endif
125 #ifdef OUTPUT_TXT_U13_COEFF
126         sprintf(ret,"./txt/u13_coeff_%d.txt",num);
127         fstream o5;
128         o5.open(ret, ios::out);
129         for(int i=0;i<N;i++)
130                 uj[i]=u[i][M/3];
131                 fftw_execute(p1);
132         for(int i=0;i<N/2;i++)
133                 o5 << i << "\t" << abs(c[i])<< endl;
134         o5.close();
135 #endif
136 #ifdef OUTPUT_TXT_MAXUX
137         double temp = 0.;
138         for(int j=0;j<M;j++){
139                 for(int i=0;i<N;i++)
140                         uj[i]=u[i][j];
141                 uj_derivative();
142                 for(int i=0;i<N;i++)
143                         if (temp<uj[i]) temp=uj[i];
144         }
145         o0 << t << "\t" << temp << endl;
146 #endif
147 #ifdef OUTPUT_TXT_G_MAXSLOPE
148         double temp=-1.;
149         for(int i=0;i<N;i++)
150                 uj[i]=g[i];
151         uj_derivative();
152         uj_derivative();
153         for(int i=0;i<N;i++){
154                 if (uj[i]*temp<=0) o4 << t << "\t" << i <<
                        endl;
155                 temp=uj[i];
156         }
157 #endif
158 #ifdef OUTPUT_TXT_U13_MAXSLOPE
159         double temp=-1.;
```

```
160         for(int i=0;i<N;i++)
161                 uj[i]=u[i][M/3];
162         uj_derivative();
163         uj_derivative();
164         for(int i=0;i<N;i++){
165                 if (uj[i]*temp<=0) o6 << t << "\t" << i <<
                        endl;
166                 temp=uj[i];
167         }
168 #endif
169 }
170
171
172 // Main function
173 int main (int argc, char * const argv[]) {
174 //Init outputs
175 #ifdef OUTPUT_TXT_G_MAXSLOPE
176         char ret[64];
177         sprintf(ret,"./txt/gmaxslope.txt");
178         o4.open(ret, ios::out);
179 #endif
180 #ifdef OUTPUT_TXT_U13_MAXSLOPE
181         char ret[64];
182         sprintf(ret,"./txt/u13maxslope.txt");
183         o6.open(ret, ios::out);
184 #endif
185 #ifdef OUTPUT_TXT_MAXUX
186         char ret[64];
187         sprintf(ret,"./txt/maxuxx.txt");
188         o0.open(ret, ios::out);
189 #endif
190
191         //Generate FFTW Plans
192         p1  = fftw_plan_dft_r2c_1d(N, uj, reinterpret_cast<
                fftw_complex*>(c), FFTW_ESTIMATE);
193         p2  = fftw_plan_dft_c2r_1d(N, reinterpret_cast<
                fftw_complex*>(c), uj, FFTW_ESTIMATE);
194
195         //Initialise Variables and set inital conditions
196         setinitialconditions();
197
198         // Main Iteration loop
199         cout << "Starting_Iteration\n";
200         for (int i=0;i*dt<=T;i++){
201
202                 if (i%picslip==0){
203                         cout << "t=" << i*dt << "\ti=" << i;
204                         if(i*dt>=picturestart){
205                                 picture((i-(int)(picturestart/
                                        dt))/picslip,i*dt);
206                                 cout << "\tpic=" << (i-(int)(
                                        picturestart/dt))/picslip;
207                         }
208                         cout << endl;
```

```
209                 }
210                 // Call main iteration loop each timestep
211                 iterate((double)i*dt);
212         }
213     return 0;
214 }
215
216 // Initial conditions 2
217 double hgr(double x){
218         double a=-10.;
219         double b=1.8;
220         double c=-0.025;
221
222         return    a*x*x*x*x*x
223                         +b*x*x
224                         +c;
225 }
226
227 // Init arrays and choose inital conditions
228 void setinitialconditions(){
229         for(int i=0;i<N;i++){
230                 u[i]            = new double[M];
231                 v[i]            = new double[M];
232                 s[i]            = new double[M];
233                 fab[i]          = new double[M];
234                 fab_old[i]      = new double[M];
235                 fab_old2[i]     = new double[M];
236
237                 //Generate initial condition
238                 for(int j=0;j<M;j++){
239                         // Initial conditions 1
240                         // u[i][j] = sin(((double)j*dy/sizeY)
                                *2.*M_PI)*(cos(((double)i*dx+0.25)
                                *2.*M_PI));
241                         // Initial conditions 2
242                         u[i][j] = 5.*hgr((double)j*dy/sizeY
                                -0.5)*(cos(((double)i*dx+0.25)*2.*
                                M_PI));
243                 }
244                 // Set pressure gradient to 0
245                 g[i]=0.;
246         }
247         // Calculate v
248         haveugetv();
249 }
250
251 // Calculate derivative
252 void uj_derivative(){
253         fftw_execute(p1);
254         //derivative
255         for (int i=0;i<N/2+1;i++)
256                 c[i]*=I*(double)i;
257         fftw_execute(p2);
258         for (int i=0;i<N;i++)
```

```
259         // renormalize
260                 uj[i]/=(double)N;
261 }
262
263 // Cut off small values if eps is defined
264 #ifdef eps
265 void cutoffc(){
266         for(int i=0;i<N/2+1;i++)
267                 if (abs(c[i])<eps) c[i]=0.;
268 }
269 #endif
270
271 void remove_oszillations(double** d){
272         #ifdef eps
273         double renorm = sqrt((double)N);
274         for(int j=0;j<M;j++){
275                 for(int i=0;i<N;i++)
276                         uj[i]=d[i][j];
277                 // renormalize
278                 for(int i=0;i<N;i++)
279                         uj[i]/=renorm;
280                 fftw_execute(p1);
281                 cutoffc();
282                 fftw_execute(p2);
283                 for (int i=0;i<N;i++)
284                 // renormalize
285                         uj[i]/=renorm;
286                 for(int i=0;i<N;i++)
287                         d[i][j]=uj[i];
288         }
289         #endif
290 }
291
292 void remove_oszillations(){
293 #ifdef eps
294         remove_oszillations(u);
295         remove_oszillations(v);
296 #endif
297 }
298
299 // Main iteration function - called every timestep once
300 void iterate(double t){
301         if (t==0){
302                 // Do the first timestep in first order
303                 predictfab();
304                 for(int i=0;i<N;i++)
305                         for(int j=0;j<M;j++)
306                                 fab_old[i][j] = fab[i][j];
307         }else{
308                 // Then switch to second order
309                 predictfab_better();
310         }
311         // Get the new pressure
312         getg1();
```

```
313         for(int i=0;i<N;i++)
314                 g[i]=g1[i];
315
316         // Get the new values for u and v
317         for(int i=0;i<N;i++)
318                 for(int j=0;j<M;j++)
319                         u[i][j] = s[i][j]-dt*0.5*g1[i];
320
321         haveugetv();
322         remove_oszillations();
323 }
324
325 // Calculates the pressure and uses fab
326 void getg1(){
327         for(int j=0;j<M;j++)
328                 for(int i=0;i<N;i++)
329                         s[i][j]=u[i][j]+dt*fab[i][j]-0.5*dt*g[
                                i];
330
331         for(int i=0;i<N;i++)
332                 uj[i]=0;
333
334         //Eulers rule (is very simple here)
335         for(int i=0;i<N;i++){
336                 uj[i] += 0.5* (2./dt) *dy *s[i][0];
337                 uj[i] += 0.5* (2./dt) *dy *s[i][M-1];
338                 for(int j=1;j<M-1;j++){
339                         uj[i]+=   (2./dt) *dy *s[i][j];
340                 }
341
342         }
343
344         fftw_execute(p1);
345         c[0]=0.;
346         #ifdef eps
347         cutoffc();
348         #endif
349         fftw_execute(p2);
350         for (int i=0;i<N;i++)
351                 // renormalize
352                 uj[i]/=(double)N;
353         for (int i=0;i<N;i++)
354                 g1[i]= uj[i];
355 }
356
357 // Predictions of fab - second order
358 void predictfab_better(){
359         predictfab();
360         for(int i=0;i<N;i++)
361                 for(int j=0;j<M;j++)
362                         fab_old2[i][j] = fab[i][j];
363         for(int i=0;i<N;i++)
364                 for(int j=0;j<M;j++)
```

```
365                         fab[i][j] = fab[i][j]*3./2.-fab_old[i
                                ][j]*1./2.;
366
367         for(int i=0;i<N;i++)
368                 for(int j=0;j<M;j++)
369                         fab_old[i][j] = fab_old2[i][j];
370 }
371
372 // Predictions of fab - first order
373 void predictfab(){
374         for(int j=0;j<M;j++){
375                 for(int i=0;i<N;i++)
376                         uj[i]=u[i][j];
377                 uj_derivative();
378
379                 for (int i=0;i<N;i++){
380                         fab[i][j] = -u[i][j]*uj[i];
381                         switch(j){
382                         case 0:
383                                 //one side only (not used)
384                                 fab[i][j] -=
385                                         v[i][j]/(2.*dy)*
                                                (4.*u[i][j
                                            +1]-u[i][j+2]-3.*u
                                            [i][j])
386                                         ;
387                                 break;
388                         case M-1:
389                                 //one side only (not used)
390                                 fab[i][j] -=
391                                         v[i][j]/(2.*dy)*
                                                (-4.*u[i][
                                            j-1]+u[i][j-2]+3.*
                                            u[i][j])
392                                         ;
393                                 break;
394                         default:
395                                 // Centered scheme:
396                                 fab[i][j] -=
397                                         v[i][j]/(2.*dy)*
                                                (u[i][j
                                            +1]-u[i][j-1])
398                                         ;
399                                 //Lax Wendroff (not used)
400                         /*      fab[i][j] -= -0.5*(
401                                 -v[i][j]/dy*
                                                        (u
                                            [i][j+1]-u[i][j-1])
402                                 +dt/(dy*dy)*v[i][j]*v[i][j]*
                                            (u[i][j+1]-2.*u[i][j]+
                                        u[i][j-1])
403                                 );*/
404                         }
405                 }
```

```
406         }
407 }
408
409 // Calculate u out of v
410 void haveugetv(){
411         for(int j=0;j<M;j++){
412                 for(int i=0;i<N;i++)
413                         uj[i]=u[i][j];
414                 uj_derivative();
415                 for(int i=0;i<N;i++)
416                         v[i][j]= -uj[i]*dy;
417         }
418
419         double temp,temp2=0.;
420
421         for(int i=0;i<N;i++){
422                 temp = v[i][0];
423                 v[i][0] = 0;
424                 v[i][0] *=0.5;
425                 for(int j=1;j<M;j++){
426                         temp2 = v[i][j];
427                         v[i][j] *= 0.5;
428                         v[i][j] += 0.5*temp + v[i][j-1];
429                         temp=temp2;
430                 }
431                 //cout << v[i][M-1] << endl;
432                 //v[i][M-1]=0;
433         }
434 }
435
436 double Y(int i){
437         return Y((double)i/(double)N);
438 }
439 double Y(double x){
440         return 1.;
441 }
```

Listing 3: picture.h

```
1  /*
2   *  picture.h
3   *  Header File.
4   *  Created by Hanno Rein.
5   *
6   */
7
8  double colormap_r(double);
9  double colormap_g(double);
10 double colormap_b(double);
11 void picture(const char * ,double** ,int ,int );
12
13 #define outputzoomx 4
14 #define outputzoomy 4
15 #define max_color  (.2)
16 #define min_color  (-.2)
```

Listing 4: picture.cpp

```
/*
 *  picture.cpp
 *  Outputs pretty pictures.
 *  Created by Hanno Rein.
 */

#include "picture.h"
#include "pngwriter.h"

// COLORMAP
double colormap_r(double i){
        if(i>1.||i<0.) return 1.;
//      if(2.*i<1.) return 1.;
        if(i<0.5) return 1.-2.*i;
        if(i<1.0) return 2.*i-1.;
        return 0.;
}
double colormap_g(double i){
        if(i>1.||i<0.) return 1.;
//      if(2.*i<1.) return 1.;
//      if(2.*i<2.) return 2.-2.*i;
        if(i<0.5) return 0.;
        if(i<1.0) return 2.*i-1.;

        return 1.;
}
double colormap_b(double i){
        if(i>1.||i<0.) return 1.;
        //if(2.*i<1.) return 1.-2.*i;
        return 0.;
}

// Squares

void picture(const char * name,double** v,int xm,int ym){
        pngwriter png(outputzoomx*xm,outputzoomy*ym+10,0,name)
            ;
        for(int i=0;i<xm;i++)
                for(int j=0;j<ym;j++)
                        png.filledsquare(i*outputzoomx,j*
                            outputzoomy,(i+1)*outputzoomx,(j
                            +1)*outputzoomy,colormap_r((v[i][j
                            ]-min_color)/(max_color-min_color)
                            ),colormap_g((v[i][j]-min_color)/(
                            max_color-min_color)),colormap_b((
                            v[i][j]-min_color)/(max_color-
                            min_color)));
        for(int i=0;i<xm;i++)
                        png.filledsquare(i*outputzoomx,(ym)*
                            outputzoomy,(i+1)*outputzoomx,(ym)
                            *outputzoomy+10,colormap_r((double
                            )i/(double)xm),colormap_g((double)
                            i/(double)xm),colormap_b((double)i
```

```
                                        /(double)xm));
42              png.close();
43      }
```