

# **Stable Fluids**

by

**Lukas Polthier, Johannes von Lindheim**  
based on Stam, 1999

Scientific Visualization, winter 15/16

Supervisor: Prof. Dr. Konrad Polthier



We hereby declare that this article is based on our work, unless stated otherwise. No other person's work has been used without due acknowledgement in this article. All references and verbatim extracts have been quoted, and all sources of information have been specifically acknowledged.

Berlin, 29th April 2016

Lukas Polthier  
Johannes von Lindheim  
Institut für Mathematik  
Arnimallee 6  
Freie Universität Berlin  
14195 Berlin  
[info@lukas-polthier.de](mailto:info@lukas-polthier.de)  
[jovoli@gmx.de](mailto:jovoli@gmx.de)

# Table of Contents

<b>0 Abstract</b>	<b>1</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Mathematical Modelling</b>	<b>2</b>
2.1 Basic Equations . . . . .	2
2.2 Method of Solution and Discretization . . . . .	3
2.2.1 Add force . . . . .	3
2.2.2 Advect . . . . .	3
2.2.3 Diffuse . . . . .	4
2.2.4 Project . . . . .	5
2.3 Moving Substances through the Fluid . . . . .	5
2.4 Gauß-Seidel . . . . .	6
2.5 Boundary conditions . . . . .	7
2.6 Vorticity Confinement . . . . .	7
2.7 Dependence on the Parameters . . . . .	8
2.8 Colors and Import of Images . . . . .	9
<b>3 Implementation and Framework</b>	<b>10</b>
3.1 Features of our Application . . . . .	10
3.2 Capabilities of JavaView . . . . .	11
3.3 Splines . . . . .	11
3.4 Block Size . . . . .	13
3.5 Code Files . . . . .	13
3.6 Framework Structure . . . . .	13
<b>4 Results</b>	<b>16</b>
<b>5 Possible Extensions</b>	<b>17</b>

# 0 Abstract

This article is a tutorial to implement the “Stable Fluids” solver by Stam [3] in a 2d-implementation. Key features of our fluid simulation are a nice user interface to interactively manipulate the fluid, the ability to control the physical constants of the fluid and the visualization of pictures made by smoke via foto-import. We provide the mathematical idea, numerical computations and the concept of the framework which we implemented in JavaView [7]. The main features of this solver in general are real time computation and being unconditionally stable. Moreover it allows to compute a wide range fluids with i.e. different viscosities and diffusion attributes.

## 1 Introduction

For many years, building realistic fluid solvers has been a challenging problem in the VFX industry.

In computer graphics, in contrast to many physical applications, the visual appearance of the fluid is very important whereas physical accuracy is not essential. In order to animate the fluid, it is required to provide real time computation. Typically this results in lower bounds on physical accuracy.

The Stable Fluids Solver by Stam [3] first gave a method that provides unconditionally stable fluids which are computed in real time. Previous solvers are unstable, i.e. they require small time steps which contradicts real time computation, or they are not based on the full Navier-Stokes equations and thus do not reflect a realistic fluid simulation.

The general problem is to create a realistic looking fluid simulation that can be computed in real time. This yields to constructing a stable solver for the Navier-Stokes equations. Stams method of solution has been implemented in this article in the framework of JavaView [7]. We added the representation of colours, the functionality to import photos and a give the user a nice framework to interactively manipulate the fluid. In the following all steps are described in detail to provide the reader with all necessary information use the solver in applications.

## 2 Mathematical Modelling

### 2.1 Basic Equations

Let  $\Omega \subset \mathbb{R}^2$  be the domain of interest, e.g. in our case we use  $\Omega = (0,1)^2$ . Now let  $u \in C^2(\mathbb{R} \times \Omega, \mathbb{R}^2)$  denote the velocity vector field and  $p \in C^2(\mathbb{R} \times \Omega, \mathbb{R})$  denote the pressure field.  $u$  describes the velocity of the “ambient fluid”, whereas  $p$  denotes the pressure of the fluid. Both fields depend on the time  $t \in \mathbb{R}$  and the position in space  $x \in \Omega$ . The evolution of these fields is given by the Navier-Stokes equations

$$\operatorname{div} u = 0 \quad (2.1)$$

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \frac{1}{\rho} \nabla p + \nu \Delta u + f, \quad (2.2)$$

where  $\nu, \rho$  are constants that determine the viscosity of the fluid and the density respectively. To clarify notation, in our two dimensional environment equation 2.2 can be written explicitly as

$$\begin{pmatrix} \frac{\partial u_1}{\partial t} \\ \frac{\partial u_2}{\partial t} \end{pmatrix} = - \begin{pmatrix} u_1 \frac{\partial u_1}{\partial x_1} + u_2 \frac{\partial u_1}{\partial x_2} \\ u_1 \frac{\partial u_2}{\partial x_1} + u_2 \frac{\partial u_2}{\partial x_2} \end{pmatrix} - \frac{1}{\rho} \begin{pmatrix} \frac{\partial p}{\partial x_1} \\ \frac{\partial p}{\partial x_2} \end{pmatrix} + \nu \begin{pmatrix} \frac{\partial^2 u_1}{\partial x_1^2} + \frac{\partial^2 u_1}{\partial x_2^2} \\ \frac{\partial^2 u_2}{\partial x_1^2} + \frac{\partial^2 u_2}{\partial x_2^2} \end{pmatrix} + \begin{pmatrix} f_1 \\ f_2 \end{pmatrix}.$$

Here,  $u_1, u_2$  denote the first and second component of  $u$  respectively. The pressure field and the velocity field that appear in the Navier-Stokes equations, are related. In fact, by combining equation 2.1 and 2.2 we obtain a single equation as follows.

By the Helmholtz-Hodge decomposition theorem we have that any  $C^2$  vector field  $w = u + \nabla q + \text{res}$  uniquely decomposes into a divergence-free part  $u$ , a gradient field  $\nabla q$  and a residual term depending on the genus of the surface. In our case, the residual term vanishes as we operate on  $\Omega \subset \mathbb{R}^2$  and  $\Omega$  is contractible.

Let  $P : C^2(\Omega, \mathbb{R}^2) \rightarrow \{f \in C^2(\Omega, \mathbb{R}^2), \operatorname{div} f = 0\}$  denote the projection operator onto the divergence free part of the vector field. The operator  $P$  is implicitly defined by

$$\operatorname{div} w = \Delta q. \quad (2.3)$$

With Neumann boundary condition ( $\frac{\partial q}{\partial n} = 0$  on  $\partial\Omega$ ,  $n$  is the outward normal), equation 2.3 is a Poisson equation. Let  $q$  denote a solution, then  $P$  is defined by  $Pw = w - \nabla q$ . If we now apply  $P$  to both sides of 2.2, the Navier-Stokes equation compress into our fundamental equation 2.4. Equation 2.1 is no longer needed, as the projection ensures that the resulting field is divergence free. In particular the pressure field drops out as  $\nabla p$  is a gradient field.

$$\begin{aligned} \frac{\partial u}{\partial t} &= P \left( -(u \cdot \nabla)u - \frac{1}{\rho} \nabla p + \nu \Delta u + f \right) \\ &= P \left( -(u \cdot \nabla)u + \nu \Delta u + f \right) \end{aligned} \quad (2.4)$$

## 2.2 Method of Solution and Discretization

### 2.2 Method of Solution and Discretization

Equation 2.4 consists of four parts, the **add force** term  $f$ , the **advection** term  $(u \cdot \nabla)u$ , the **diffusion** term  $\nu\Delta u$  and the **projection** operator  $P$ . Both time and space are discretized with time step  $\Delta t$  and some equidistant grid points of distance  $h = \frac{1}{n}$ . The equation is solved from an initial state  $u^0 = u(t, x)$  by marching through time with next iterate  $u(t + \Delta t, x)$ . Each of the four terms in equation 2.4 is applied successively to the initial state  $u^0 \in C^2(\Omega, \mathbb{R})$ . The general procedure is

$$u^0 \xrightarrow{\text{add force}} u^1 \xrightarrow{\text{advect}} u^2 \xrightarrow{\text{diffuse}} u^3 \xrightarrow{\text{project}} u^4.$$

The solution at time  $t + \Delta t$  is then given by  $u(x, t + \Delta t) = u^4(x)$ .

#### 2.2.1 Add force

The add force step incorporates additional force by the user and buoyancy force due to uplift of lighter or hotter gases, respectively downlift of heavier or cooler gases.

$$u^1(x) = u^0(x) + \Delta t f(t, x)$$

The buoyancy force is computed using Archimedes' principle. In a simplified approach, heaviness is equal to the density of the smoke. After computing the average temperature of the fluid, the upward force is determined for each pixel separately depending on the difference with respect to the average temperature:

```
foreach Grid cell do
    Cell.weight  $\leftarrow$  Sum of densities in this cell
     $f \leftarrow T_{amb} * \text{Cell.weight} + (1 - T_{amb}) * \text{weight of one block of air}$ 
end
```

#### 2.2.2 Advect

The advect step accounts for the advection or convection of the fluid itself, i.e. this step lets the fluid "flow" or move a little according to its own speed. The advection step is fundamental to this particular fluid solver. The design of the advect method is the reason why this method is called "Stable" Fluids, as this solver will never "blow up", independent of the size of the time step  $\Delta t$ .

The method can be understood intuitively: All particles in the fluid are moved by the velocity of the fluid itself. To obtain the velocity at the point  $x$  at time  $t + \Delta t$  we backtrace the the point  $x$  through the velocity field at time  $t$ . This defines a path  $p : (-\delta, \delta) \times \Omega \rightarrow \Omega$  corresponding to a streamline of the fluid. The velocity  $u^2(x)$  is the set to be the velocity of  $u^1(p(-\Delta t, x))$  at the previous time step:

$$u^2(x) = u^1(p(-\Delta t, x)).$$

## 2.2 Method of Solution and Discretization

Again, this step is unconditionally stable, independent of the step size.

In practice we trace each particle backward in time with some Runge-Kutta scheme. Then we interpolate the vector at  $p(-\Delta t, x)$  linearly from the values of the vector field on the four neighbouring grid points. We obtained good results even with simple Euler methods.

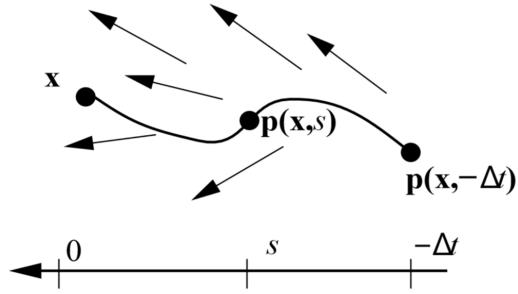


Figure 2.1: Illustration of advection method.

### 2.2.3 Diffuse

This step solves the diffusion of the fluid itself, i.e. the “friction” between parts of the fluid with different velocity. This effect is equivalent to the diffusion equation

$$\frac{\partial u^2}{\partial t} = \nu \Delta u^2. \quad (2.5)$$

The most straightforward way would be to discretize the Laplacian and solve the resulting sparse linear system. However, this approach is unstable when the viscosity is large. For our implicit approach we proceed by approximating  $\frac{\partial u}{\partial t}$  with the backward difference quotient:

$$\frac{u(t + \Delta t, x) - u(t, x)}{\Delta t} = \nu \Delta u(t - \Delta t, x)$$

Finally, this yields

$$(I - \nu \Delta t) u^3(x) = u^2(x). \quad (2.6)$$

We now discretize 2.6 using a finite difference method and obtain

$$\left( \begin{bmatrix} & & & \\ & 1 & & \\ & & \ddots & \\ & & & 1 \end{bmatrix} - \begin{bmatrix} & & & \\ -4 & 1 & & \\ & 1 & \ddots & \\ & & \ddots & 1 \\ 1 & & & \\ & & & 1 & & \\ & & & & 1 & \\ & & & & & -4 \end{bmatrix} \nu \frac{\Delta t}{h^2} \right) u^3 = u^2. \quad (2.7)$$

### 2.3 Moving Substances through the Fluid

The resulting square matrix has  $n \cdot m$  rows and columns, where  $n, m$  denote the number of pixels in the x- and y-axis respectively. Solving such a system can be done efficiently by iterative schemes, e.g. Gauß-Seidel. In particular, due to the symmetric, positive definite nature of the sparse matrix, iterative schemes are very effective.

#### 2.2.4 Project

The last step makes the vectorfield mass preserving, i.e. divergence free. We already discussed in the derivation of our fundamental equation that the projection is obtained by solving

$$\operatorname{div} u = \Delta q.$$

When discretized, this equation becomes

$$\frac{1}{2h} \begin{bmatrix} 0 & 1 \\ 1 & \ddots & \ddots \\ \vdots & \ddots & \ddots & 1 \\ 1 & \ddots & \ddots & \ddots & 1 \\ 1 & 0 & \ddots & \ddots & \ddots & 1 \end{bmatrix} u_1^3 + \frac{1}{2h} \begin{bmatrix} 0 & & 1 \\ & \ddots & & 1 \\ 1 & & \ddots & & 1 \\ & \ddots & & \ddots & & 1 \\ 1 & & 0 & & & \ddots & 1 \\ & & & \ddots & & & \ddots & 1 \\ & & & & \ddots & & & \ddots & 1 \\ & & & & & \ddots & & & -4 \end{bmatrix} u_2^3 = \begin{bmatrix} -4 & 1 & 1 \\ 1 & \ddots & & 1 \\ \vdots & \ddots & \ddots & \ddots & 1 \\ 1 & & \ddots & & \ddots & 1 \\ & \ddots & & \ddots & & \ddots & 1 \\ & & \ddots & & & & 1 \\ & & & \ddots & & & & 1 \\ & & & & \ddots & & & & -4 \end{bmatrix} q. \quad (2.8)$$

This equation is quite similar to 2.7 as the matrices have the same structure. As in equation 2.7 we need to solve a sparse linear system which can be done analogously.

## 2.3 Moving Substances through the Fluid

Our solver enables us to compute the ambient fluid. However, we need to visualize the vector field. A substance, that is injected in the fluid and does not interact with it, will be advected by the vector field and diffuse at the same time. This density will thus visualize the fluid as one can see it in real life when milk is stirred in a coffee or smoke is injected in the air. Let  $d \in C^2(\mathbb{R}^+ \times \Omega, \mathbb{R})$  denote the density field of such a substance. The evolution of this scalar field is given by

$$\frac{\partial d}{\partial t} = -u \cdot \nabla d + \kappa \Delta d - \alpha d + S, \quad (2.9)$$

where  $\kappa$  is the diffusion constant,  $\alpha$  the dissipation term and  $S$  is a source term. The dissipation term, which will be dropped in our modelling, describes the effect that kinetic energy is converted in thermal energy. The diffusion constant determines the effect of diffusion, i.e. the effect that density “interfuses” with nearby density and the overall picture becomes “softer”.

The terms in equation 2.9 are quite similar to the terms in our fundamental equation 2.4. Thus the evolution of the density field can be computed analogously to the computation of the velocity field. In particular we need to perform the advect step and the diffuse step, which

## 2.4 Gauß-Seidel

correspond to the movement of the density along the velocity field and the diffusion of the density itself.

Thus we do not require additional methods and we can reuse our numerical approximation from the previous section.

### 2.4 Gauß-Seidel

Since our square matrices are symmetric, positive-definite, iterative schemes like the Gauß-Seidel-method, which we use, are known to converge and appropriate to solve our system. The technique works by decomposing the linear system of equations  $Ax = b$  into

$$(L + U)x = b \Leftrightarrow Lx = b - Ux, \quad (2.10)$$

where

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \dots & \dots & a_{nn} \end{bmatrix} = \begin{bmatrix} a_{11} & 0 & \dots & 0 \\ a_{21} & a_{22} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & \dots & \dots & a_{nn} \end{bmatrix} + \begin{bmatrix} 0 & a_{12} & \dots & a_{1n} \\ 0 & 0 & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \dots & \dots & 0 \end{bmatrix} = L + U, \quad (2.11)$$

i.e.  $L$  is the lower triangular component and  $U$  is the strict upper component of  $A$ . The Gauß-Seidel technique now solves the left hand side of 2.10 using a previous solution of  $x$  for the right hand side, i.e.

$$x^{(k+1)} = L^{-1}(b - Ux^{(k)}). \quad (2.12)$$

Since  $L$  is in triangular form, this can be done sequentially (and therefore faster than computing the inverse) by using forward substitution:

$$x_i^{(k+1)} = \frac{1}{a_{ii}} \left( b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(k+1)} - \sum_{j=i+1}^n a_{ij} x_j^{(k)} \right) \quad (2.13)$$

Considering the special case of our matrices, for instance for the diffusion step, we can write  $u$  as a column vector with  $n \cdot m$  entries and boil this down to

$$u_{(i,j)}^{(k+1)} = \frac{1}{1 + 4c} \left( u_{(i,j)}^{(k+1)} + c \left( u_{(i-1,j)}^{(k+1)} + u_{(i,j-1)}^{(k+1)} + u_{(i+1,j)}^{(k)} + u_{(i,j+1)}^{(k)} \right) \right), \quad (2.14)$$

where  $c = \nu \frac{\Delta t}{h^2}$  and  $u_{(i,j)}^{(k)}$  gives the  $k$ -th iterate of the velocity vector at grid point  $(i, j)$  which corresponds to the entry  $i + j \cdot n$  in the column vector representation of  $u$ .

In contrast to similar methods like the Jacobi method, this cannot be performed parallel, since in this sequential approach, all values  $u_1^{(k+1)}, \dots, u_i^{(k+1)}$  are needed to compute  $u_{i+1}^{(k+1)}$ . So unfortunately, starting multiple threads to speed up the calculation is not an option.

## 2.5 Boundary conditions

### 2.5 Boundary conditions

To make sure, that nothing leaves the finite grid, we put a border with a thickness of one cell all around the grid, which is acting as a “wall”. After all calculations performed, these grid cells are given the velocity vector that would emerge if the vector right next to them (in the inner grid) would be “bounced off this wall”.

### 2.6 Vorticity Confinement

For a sufficient number of grid points and infinite computational power, the solver described in the current chapter 2 gives the fluid a realistic behaviour, quite similar to the real life experience. In practice, computational power is limited and we have to use a relatively small number of grid points to provide real-time computation. It turns out that small scale details, in particular rotational turbulences are lost due to numerical dissipation on the coarse grid.

The key idea of Vorticity Confinement is to add these details artificially in a way that the properties of the fluid are preserved and the effect of numerical dissipation is damped. In a follow-up paper [2], Stam proposes this idea based on previous work of Steinhoff [4].

The vorticity of a 3d vector field  $u$  is defined as

$$\omega = \nabla \times u,$$

i.e. in our two dimensional case, we have

$$\omega = \begin{pmatrix} \frac{\partial}{\partial x_1} \\ \frac{\partial}{\partial x_2} \\ \frac{\partial}{\partial x_3} \end{pmatrix} \times \begin{pmatrix} u_1 \\ u_2 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \frac{\partial}{\partial x_1} u_2 - \frac{\partial}{\partial x_2} u_1 \end{pmatrix}.$$

We identify  $\omega$  with a 1d scalar field on  $\Omega$ . The vorticity measures the amount of small scale detail of the fluid. The normalized gradient of the absolute vorticity

$$N = \frac{\nabla |\omega|}{|\nabla |\omega||}$$

is a vector field on  $\Omega$  that points from regions of low vorticity to regions of high vorticity.

Finally the small scale detail is added as follows

$$f = \varepsilon h(N \times \omega) = \begin{pmatrix} N_1 \\ N_2 \\ 0 \end{pmatrix} \times \begin{pmatrix} 0 \\ 0 \\ \omega \end{pmatrix} = \begin{pmatrix} N_2 \omega \\ -N_1 \omega \\ 0 \end{pmatrix}. \quad (2.15)$$

Again in equation 2.15 we identify 2d vector fields with 3d vector fields that are constant in the with respect to the direction of the  $x_3$ -axis. Here  $\varepsilon > 0$  is a factor that determines the effect of vorticity confinement. The dependence on  $h$  gives that small scale detail is added

## 2.7 Dependence on the Parameters

proportional to the grid size, i.e. to the numerical dissipation. Hence the convergence of the solver is preserved.

In fact the construction provides that small scale detail is added, exactly where it is needed. The reader can convince himself of the effectiveness in figure 2.2.

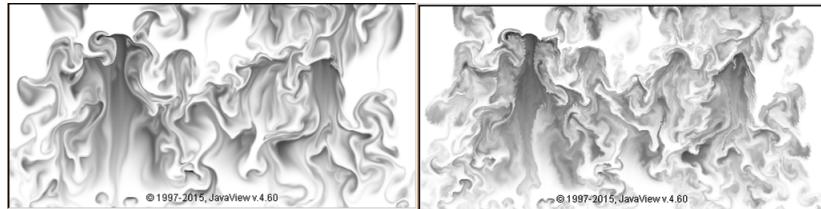


Figure 2.2: The left picture shows the fluid solver without vorticity confinement, whereas the right picture shows the same setting with medium vorticity confinement. Both pictures were computed with the same grid size and initial configuration. One can see clearly how the left picture misses small scale rotational turbulences. The right picture provides a more gaseous behaviour of the fluid.

## 2.7 Dependence on the Parameters

The solver is compatible with a wide range of parameters. Thus we can simulate different kinds of fluids, depending on viscosity, buoyancy force, diffusion, etc.

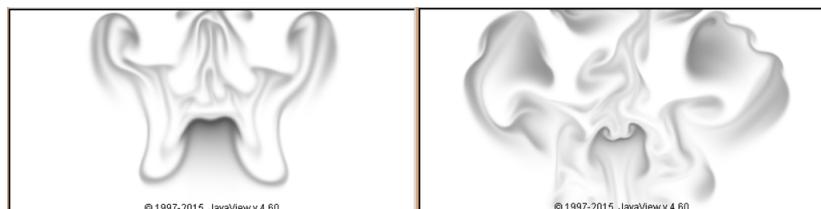


Figure 2.3: The left picture shows a fluid with high viscosity, e.g. honey or very thick water. The right picture shows a more liquid fluid like thin water or some gas. Both pictures were created with identical initial condition and vary only in the viscosity constant. In both pictures there is no vorticity confinement to emphasize the effect of viscosity.

## 2.8 Colors and Import of Images



Figure 2.4: In the left picture there is no diffusion, whereas in the right picture we can clearly see how diffusion influences the nature of the fluid. There are less details and the picture looks overall smoother. Again both simulations vary only in the effect of diffusion and are computed with the same resolution.

## 2.8 Colors and Import of Images

There are several ways to represent colors. One of the most common principles are RGB or CMY color schemes. In the black and white fluid solver described so far there is only one density array which evolves according to the velocity field. To represent colors, we use three separate density fields, each one of them representing either cyan, magenta or yellow and evolving according to the same velocity field, i.e. the same “ambient fluid”.

We are now able to import pictures, which behave like liquid or smoke, i.e. the picture can be manipulated interactively and behaves like the ambient fluid.

One can experiment with different kinds of buoyancy forces, e.g. only specific colors generate an uplift where some colors are heavier than others.

# 3 Implementation and Framework

We implemented our whole framework in the environment of JavaView. In the following, we want to show some features of our application, how they work and why we think they make sense for visualizing smoke.

## 3.1 Features of our Application

First, we want to mention and describe all the (non-default-JavaView) features in the application, i.e. what all the buttons and sliders are for to give the reader a better understanding or motivation of the following chapters.

- Sliders:
  - BlockSize: Edge length of one “pixel block” (see 3.4) in pixels.
  - Density radius: Radius of the “pencil” drawing density into the display (see 3.3).
  - Density constant: Controlling, how much smoke is inserted.
  - Force radius: Radius of the “force pencil”, i.e. controlling, how big the region is, which is pulled by inserting force with the mouse.
  - Force constant: Controlling, how much force is inserted.
  - Buoyancy force: Controlling, how much buoyancy force is inserted (see 2.2.1). Can be set to negative values, which makes hot (i.e. dense) smoke sink instead of rise.
  - Diffusion: Controlling, how much the fluid diffuses (see 2.2.3).
  - Viscosity: Controlling viscosity parameter of the fluid (see also 2.7).
  - Vorticity: Controlling, how much force is inserted, to preserve details lost by vorticity confinement (see 2.6).
  - Projection Accuracy: Takes only 0 and 1. If set to 1, the solver produces more accurate results (performing multiple projection steps between all other steps) at the cost of longer computation times (see also 2.2.4).
- Buttons:
  - Reset: Resets application to starting conditions (without affecting the size of the application).
  - Clear Canvas: Clears the canvas from all smoke and sets the fluid to stand still, but keeps all slider values etc. the same.
  - Flip Color: If color mode is one, this button selects the next color for drawing density.
  - Import Image: Imports the image “image.png” from the same directory as the executable into the canvas. The pixels of the image are treated as colored smoke (see 2.8).
  - Freeze: Pauses the fluid to move. Clicking once more wakes it up again.

### 3.2 Capabilities of JavaView

- Color On/Off: As default, the application starts with greyscale smoke. This button turns on color mode, where you can draw smoke of multiple colors into the display. See also the button “Flip Color”.

## 3.2 Capabilities of JavaView

We want to give a brief overview over the features of JavaView that were used in the development of the project and also how they were used.

- To be able to show an image visualizing the smoke in the application, we use the common `PvDisplayIf` class. One can fix the camera over an x-y-coordinate system by selecting the correct camera: `m_disp.selectCamera(PvCameraIf.CAMERA_ORTHO_XY)`. Then one can set the image one is manipulating as the background image by `m_disp.setBackgroundImage(m_image)`.
- For handling mouse input, there are the two functions `pickInitial(PvPickEvent pickEvent)` and `dragCamera(PvCameraEvent cameraEvent)` which are invoked by JavaView, when the left resp. right mouse button is clicked. These methods were overwritten to add smoke resp. force. The mouse coordinates can be obtained from the corresponding events in the parameter list.
- The parameters of the fluid solver (the “mathematics object”) and block size (see 3.4) are controlled by sliders in the application. One is able to add these for types like `PuInteger` or `PuDouble` by for instance adding `m_Project.m_vorticityConf` to some `PsPanel` in the application.
- Time can be handled by an animation, i.e. an instance of the class `PsAnimation`. Registering this animation in the project class’ superclass makes JavaView invoke the method `setTime(PsTimeEvent timeEvent)` which was overwritten to call our `computeImage()`-function. To overcome the problem, that every animation has a maximum time, we are increasing the maximum by one timestep for every call of `setTime`.
- When the user drags though the display with the left mouse button, the corresponding `PvPickEvents` yield some loose points in the canvas. The class `PgBezierCurve` is used to connect them with spline curves, such that there is not only some bumps, but a whole continuous trace of smoke added in the canvas, like one would expect for instance from a pen. For details, see 3.3.

## 3.3 Splines

As mentioned above, the user is able to “draw” smoke with the left mouse button like with a pen, even though JavaView just provides a handful of input points. If we just would add smoke there, just some single pixels would be provided with some smoke.

### 3.3 Splines



Figure 3.1: Some smoke drawn into the display by the user with the mouse.

To be able to achieve this, we connected the mouse input points from between two frames with a spline curve consisting of mostly cubic bezier curves or lower degree at the start or end where less information is available about the continuation of the curve. The resulting spline curve is  $C^1$ -continuous.

We also show the formula for the control points in the general case of cubic curves.

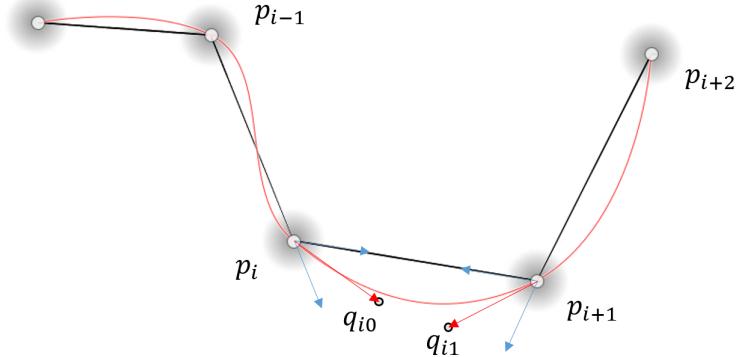


Figure 3.2: Scheme how control points are computed.

Then we compute the control points of the connecting cubic bezier curve to be

$$q_{i0} = p_i + \alpha |p_i - p_{i+1}| \cdot ((p_i - p_{i-1})^* + (p_{i+1} - p_i)^*)^*$$

$$q_{i1} = p_{i+1} + \alpha |p_i - p_{i+1}| \cdot ((p_i - p_{i+1})^* + (p_{i+1} - p_{i+2})^*)^*$$

where  $p^*$  is a notation for  $\frac{p}{|p|}$ . In our case we chose  $\alpha = \frac{2}{5}$ .

### 3.4 Block Size

## 3.4 Block Size

Since for a large enough canvas sizes interaction with the smoke in real time is not possible because of the large amount of computations, the user should be able to choose lower resolutions on the same canvas size to make the application run faster. The corresponding (larger) pixels are called *blocks*.

If the user wants to change the size of a block, the color value of the new blocks are taken to be the average of the previous color of all pixels, that lie in that block.

## 3.5 Code Files

The program consists of four code files:

- App.java: This is the actual application and contains the entry point (main). It is loading and starting the project inside a window of sizes and with canvas dimensions which are specified here.
- FluidSolver.java: The `FluidSolver` object does all the mathematics of the program, i.e. it holds the vector field for the fluid and one scalar field for each density array and calculates all four steps of the method of solution of the fluid respectively the three steps for all density arrays. The user (i.e. the `Painter` class) just needs to call `velocitySolver()` and `densitySolver()`.
- Painter.java: This class contains almost all the framework. It is calling the `FluidSolver`'s solution methods, handling the mouse input from the user, drawing into the display and computing colors out of the calculation results from the `FluidSolver`.
- Painter\_IP.java: This class contains the design of the info panel and connection of the GUI elements to the `Painter` class.

## 3.6 Framework Structure

We now want to give a (brief and simplified) overview over the structure of the implementation of the framework contained in the `Painter` class.

Every time when `setTime()` is invoked (when the animation has moved forward in time), the thread dives into the heart of the program. The structure there is as follows:

### 3.6 Framework Structure

```

setTime()
  computeImage()
    Add Input Force
    Add Input Density
    if isFrozen == false then
      | Solve ambient fluid
      | Solve densities
    end
    while incorrecSizes() do
      | wait()
    end
    Copy solution into density array of canvas
    computeColors()
  end
end

```

Since JavaView runs multi-threaded, the user could possibly change the BlockSize or the size of the canvas, while a calculation of the Fluid Solver is running. But then the dimensions of the Fluid Solver and the display's density array do not match, so as seen above, the thread first has to `wait()`.

Eventually a `componentResized`-event or the `update()`-method of the `blockSize`-variable is triggered. In the first case, the Painter's procedure `resizeImage()` is invoked or in the latter case, `update()` in turn calls `changeBlockSize()`. Both adjust the dimensions of the Fluid Solver (see below).

Only now can we go on in `computeImage()` and copy the solution of the differential equations into the own density array of the size of the canvas. The two adjustment functions look like this:

```

resizeImage()
  if this.size == display.size then
    this.size ← display.size
    resolution ← this.size/blockSize
    Resize FluidSolver
    Resize density arrays
    Resize images
    Initialize and start animation
  end
  notifyAll()
end

```

Note the very important call of `notifyAll()`, which is the sign for the possible other thread sitting in `computeImage()` to stop `waiting` and further process the calculation results of the Fluid Solver.

The `changeBlockSize()`-function also works rather straightforward:

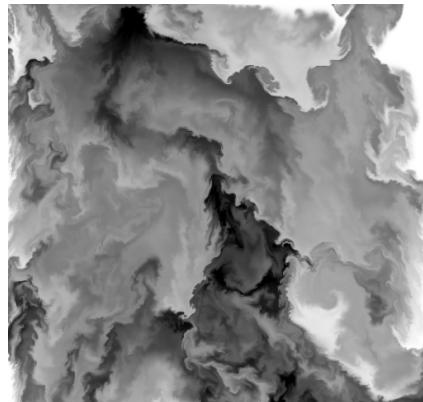
### 3.6 Framework Structure

```
changeBlockSize()
| Copy current FluidSolver
| Resize FluidSolver
foreach Block of FluidSolver do
|   Store average of pixels lying in this Block
end
notifyAll()
end
```

Also note that `computeImage()`, `resizeImage()` and `changeBlockSize()`, as well as some other minor methods, are declared as `synchronized` to prevent multiple threads (for instance the thread handling the animation time) from writing into the data arrays at the same time, which would cause race conditions.

## 4 Results

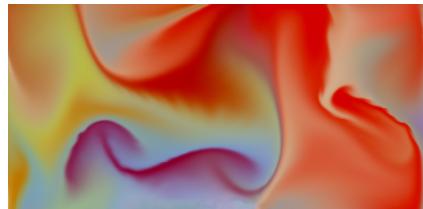
Our solver could be used for many purposes requiring smoke-like motions, e.g. in graphic design or computer games. It can also be used to create purely artistic effects. Some possibilities generated with the solver are shown in the following.



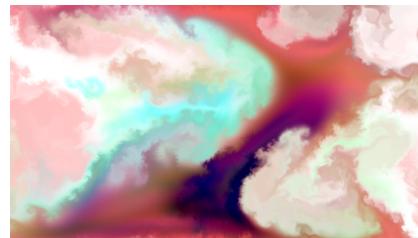
(a) Smoke generated with a single density array.



(b) Colored smoke generated with three density arrays for the CMY-colors



(c) Fluid with high viscosity.



(d) Still standing fluid, which has been diffusing for a while.



(e) Distorted Marilyn Monroe.



(f) Distorted pineapple.

## 5 Possible Extensions

There are several positive and negative aspects about our project:

- Positive:

- The resulting smoke looks detailed and realistic.
- One can produce artistic effects with multiple colors or imported images.
- The interactive manipulation works really fine, especially with the Bézier-curve functionality.

- Negative:

- The speed of our application on grid sizes starting from about  $256 \times 512$  would need further enhancement. The BlockSize-feature is not really solving the problem, if one wants these “high” resolutions for the simulation.
- The smoke looks like it is moving out of the display despite our boundary conditions handling and mass preserving projection method. This undesirable effect would need further study.

All in all the fluid solver provides a very realistic simulation which can be used to simulate various phenomena and fluids. In particular the extension with Vorticity Confinement results in an overall realistic and intuitive behaviour of the fluid.

Nevertheless, there are further extensions or improvements of our application one could think of. As mentioned in chapter 2, the solver can be implemented in three dimensions without further adaption of the mathematical model. This gives interesting possible extensions e.g. light sources which makes it possible to animate more complex phenomena such as clouds that are based mainly on the appearance of light and darkness. However due to the increase in grid points and more entries in the sparse matrices, we are not able to compute a 3D version in real time. Also a 3D implementation requires a renderer and the interactive user interface becomes more complicated. In this 3D environment, it would also be interesting to put objects inside the domain, e.g. a car or an airplane and blow some smoke over it like in a wind channel. One could thereby explore and visualize the aerodynamic properties of these objects.

Furthermore, our implementation uses Gauss-Seidel to solve the sparse linear systems. While this is easy to implement and relatively fast for the sparse matrices we consider, it does not reach the speed of multigrid methods.

Lastly, again coming from an implementation specific point of view, it would be convenient to be able to export videos of the rendered images for fast replay, e.g. for presentation purposes.

# Reference List

- [1] H. Alt: “Lineare Funktionalanalysis”, fifth edition, Springer, 2006.
- [2] R. Fedkiw, J. Stam, H. W. Jensen: Visual Simulation of Smoke, Siggraph, 2001.
- [3] J. Stam: “Stable Fluids”, Siggraph, 1999.
- [4] J. Steinhoff, D. Underhill. Modification of the euler equations for “vorticity confinement”: Application to the computation of interacting vortex rings. Physics of Fluids, 6(8):2738– 2744, 1994.
- [5] J. Stoer: “Numerische Mathematik 1”, ninth edition, Springer, 2005.
- [6] J. Stoer, R. Bulirsch: “Numerische Mathematik 2”, fifth edition, Springer, 2005.
- [7] K. Polthier: JavaView, program refer to javaview.de.