Differentiable Stable Fluid Solver using TensorFlow

SAMMY RASAMIMANANA, Telecom Paris, France

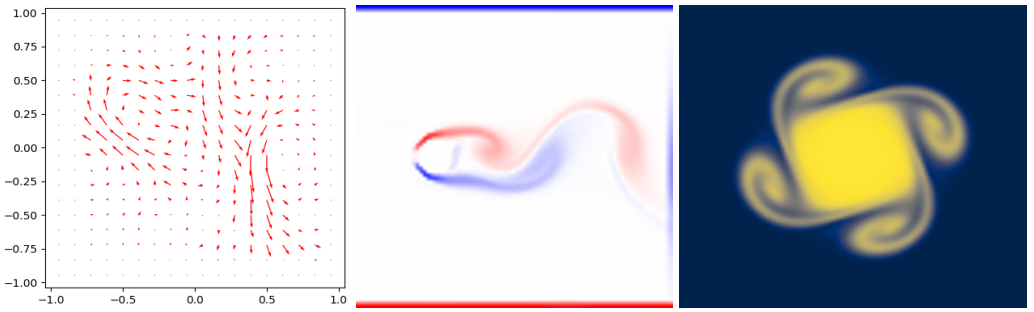


Fig. 1. Density field of a vortex simulation (*right*) and vorticity field of a Karman vortex (*middle*) simulation. Our differentiable stable fluid solver allows to compute gradients of scalar fields with respect to velocity field. It can then be incorporate into training models that would find velocity fields (*left*) that make a density field match a certain target shape.

In this project, we present a differentiable stable fluid solver implemented using TensorFlow. We start by introducing the basics of fluid mechanics and stable fluid solvers. We then describe the architecture of our differentiable solver and show how it can be trained using backpropagation. We demonstrate the effectiveness of our solver on various fluid simulation tasks. Our results show that our differentiable stable fluid solver can achieve state-of-the-art performance while being differentiable, making it suitable for potential integration into deep learning pipelines. Our work opens up different possibilities for using fluid simulations in machine learning applications such as generative models and data augmentation.

Additional Key Words and Phrases: fluid simulation, optimisation, TensorFlow, gradient, differentiable solver

1 INTRODUCTION

1.1 Motivation

Whether in the special effects industry or in engineering, fluid simulation is an intriguing field of study. A reliable fluid solver is crucial in numerous applications, as it enables the accurate modeling of natural phenomena such as the flow of water, smoke, or fire. However, the complete resolution of the equations governing fluid behavior remains an open problem. Most solutions are based on numerical solvers. Therefore, the advent of machine learning presents an opportunity to refine these techniques. Machine learning offers methods that leverage previous fluid simulation data to accelerate the resolution process or enhance its precision. The objective of this project is to develop a stable fluid solver using the TensorFlow machine learning library. The solver should be differentiable and able to integrate into training processes.

1.2 Related work

Stable Fluid Solver has been an active area of research in computational fluid dynamics (CFD) for several decades [Bridson 2015]. The seminal work on the subject is attributed to Jos Stam's paper "Stable Fluids" in 1999 [Stam 2001]. In this paper, Stam presented a novel method for solving the Navier-Stokes equations for fluid simulation, which introduced a new numerical method known as the "Stam's method." Since then, many researchers have built on this work, and a variety of

This report is submitted as a part of an IGR *Projet Recherche et Innovation Master* (PRIM) project, Telecom Paris.
Author's address: Sammy Rasamimanana, Telecom Paris, France.

stable fluid solvers have been proposed. One approach to stable fluid solvers is based on smoothed particle hydrodynamics (SPH), which simulates fluid flow using particles. SPH is particularly suited for simulating complex free surface phenomena. Recently, differentiable stable fluid solvers have gained significant attention due to their ability to be integrated into differentiable programming frameworks. These solvers allow the gradients of the fluid simulation to be computed efficiently, making them useful for applications such as optimization and machine learning. Differentiable fluid solvers have been implemented using various deep learning frameworks, including TensorFlow, PyTorch, and JAX.

1.3 Notations

For a detailed reference of the mathematical notations employed in this report, kindly refer to the appendix.

2 NAVIER-STOKES EQUATIONS

2.1 Basic equations

A fluid whose density ρ and temperature are nearly constant is described by a velocity field $u : \mathbb{R}^N \longrightarrow \mathbb{R}^N$ and a pressure field $p : \mathbb{R}^N \longrightarrow \mathbb{R}$, where $N = 2$ or 3. First, the law of conservation of matter should be fulfilled. According to this law, the mass of an object or collection of objects never changes over time, no matter how the constituent parts rearrange themselves. In fluid dynamics, this means that the amount of fluid entering a given region must be equal to the amount of fluid leaving that region, and consequently leads to a divergence-free condition when applied to a fluid with a constant density:

$$\nabla \cdot u = 0 \tag{1}$$

Then, the conservation of the momentum should also be ensured.

$$\frac{\partial u}{\partial t} = -(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \nu \nabla^2 u + f \tag{2}$$

Both those equations use spatial derivative, and therefore requires boundary conditions. In this work, we will consider only *fixed* boundary conditions, which can be interpreted as walls.

2.2 A projection operator

As mentionned in section 1.1, equations (1) and (2) have not yet been entirely solved analytically. One technique to get a good approximative solution is to march through each step of (2) with implicit equations $\frac{\partial u}{\partial t} = g(u)$ and ultimately ensure (1):

$$\begin{aligned}
u_1 &= u(t) + \Delta t f \quad \texttt{(external forces)} \\
u_2 &= u_1 - \Delta t (u_2 \cdot \nabla) u_2 \quad \texttt{(advection)} \\
u_3 &= u_2 + \Delta t \nu \nabla^2 u_3 \quad \texttt{(diffusion)} \\
u_4 &= u_3 - \frac{\Delta t}{\rho} \nabla p(t) \quad \texttt{(pressure)} \\
u_5 &= P(u_4) \text{ such that } \nabla \cdot u_5 = 0 \quad \texttt{(divergence-free)} \\
u(t + \Delta t) &= u_5
\end{aligned}$$

However, the difficulty lies in finding a function P that would make $\nabla \cdot u_5 = 0$. Some of the solvers directly use the pressure term $-\frac{\Delta t}{\rho}\nabla p(t)$ to make the velocity field divergence-free, hence combining the pressure step and the divergence-free step. [Stam 2001] introduces a slightly different approach that relies on the Helmholtz-Hodge decomposition.

Theorem 1 (Helmholtz-Hodge decomposition). Any vector field $w : \mathbb{R}^3 \longrightarrow \mathbb{R}^3$ can be decomposed into

$$w = u + \nabla q$$

where $u : \mathbb{R}^3 \longrightarrow \mathbb{R}^3, \nabla \cdot u = 0$ and $q : \mathbb{R}^3 \longrightarrow \mathbb{R}$

The idea is then define a projector $\mathbb{P}$ and to apply it to (2).

Definition 1 (Projection operator). Let $\mathbb{P}$ the operator which projects any vector field w onto its divergence free part.

$$\mathbb{P} : \mathbb{R}^3 \longrightarrow \mathbb{R}^3$$
$$w = u + \nabla q \longmapsto \mathbb{P}(w) = u = w - \nabla q$$

Remark 1. An important remark is that the scalar field q can be recovered with a Poisson equation.

$$\nabla \cdot w = \underbrace{\nabla \cdot u}_{0} + \nabla \cdot \nabla q \iff \nabla \cdot w = \nabla^2 q$$

Remark 2. Some notable properties are

(i) $\forall u : \mathbb{R}^3 \to \mathbb{R}^3, \nabla \cdot u = 0 \Longrightarrow \mathbb{P}(u) = u$
(ii) $\forall q : \mathbb{R}^3 \to \mathbb{R}, \mathbb{P}(\nabla q) = 0$

By applying $\mathbb{P}$ to the Navier-Stokes equation (2), we keep $\nabla \cdot u = 0$ implicit.

$$\mathbb{P}\left(\frac{\partial u}{\partial t}\right) = \mathbb{P}\left(-(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \nu \nabla^2 u + f\right)$$
$$\frac{\partial u}{\partial t} = \mathbb{P}\left(-(u \cdot \nabla)u\right) - \underbrace{\mathbb{P}\left(\frac{1}{\rho}\nabla p\right)}_{0} + \mathbb{P}\left(\nu \nabla^2 u\right) + \mathbb{P}\left(f\right)$$
$$\frac{\partial u}{\partial t} = \mathbb{P}\left(-(u \cdot \nabla)u + \nu \nabla^2 u + f\right) \tag{3}$$

Therefore, our stable fluid solver will follow this algorithm:

Algorithm 1 Stable fluid solver loop for the velocity field

1: $u \longleftarrow u + \Delta t f$ `//addExternalForces(u,f)`
2: $u \longleftarrow$ `advect(u)`
3: $u \longleftarrow$ `diffuse(u)`
4: $q \longleftarrow$ solution of Poisson equation $\nabla \cdot u = \nabla^2 q$ `//solvePressure(u)`
5: $u \longleftarrow u - \nabla q$ `//project(u)`

3 TECHNICAL DETAILS

3.1 Model

For our implementation, we will work exclusively in 2D. The fluid domain is discretized into a $n \times n$ grid where each physical quantity related to the fluid is stored and evaluated at the center of each cell.

Since we want our solver to be compatible with TensorFlow, we first implemented the solver in `Python`, using the `Numpy` library. For any vector field, we tried two different data structures.

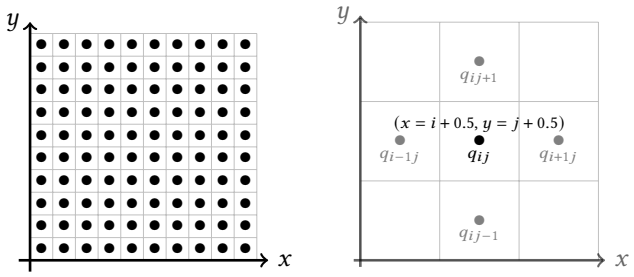


Fig. 2. Our discretization grid (*left*) and physical quantities stored at a cell (i, j). The cells represented here are of size 1×1 for clarity purpose. Hence, for a physical quantity q, its sampled value at cell (i, j) is $q_{ij} = q(x, y) = q(i + 0.5, j + 0.5)$ (*right*).

Data structure 1: All the vectors are stored inside a matrix of size $n \times n$ that would represent our grid with a reverted y axis, thus leading to an array of size $n \times n \times 2$ since we are in 2D.

Data structure 2: All the components of the vector field are stored in a vector of size n^2. Therefore, for a vector field $q = (q_x, q_y)$, we have two huge vectors of size n^2.

$$u = \begin{pmatrix} u_{00} & u_{10} & \cdots & u_{n0} \\ u_{01} & u_{11} & \cdots & u_{n1} \\ \vdots & \vdots & \ddots & \vdots \\ u_{0n} & u_{1n} & \cdots & u_{nn} \end{pmatrix} \qquad u = (u_x, u_y) \text{ where } u_x = \begin{pmatrix} u_{x,00} \\ u_{x,10} \\ \vdots \\ u_{x,n0} \\ u_{x,n1} \\ \vdots \\ u_{x,nn} \end{pmatrix} \text{ and } u_y = \begin{pmatrix} u_{y,00} \\ u_{y,10} \\ \vdots \\ u_{y,n0} \\ u_{y,n1} \\ \vdots \\ u_{y,nn} \end{pmatrix}$$

$u[i, j] = u_{ji} = (u_{x,ji}, u_{y,ji})$
Data structure 1

$u_x[i + jn] = u_{x,ij}$ and $u_y[i + jn] = u_{y,ij}$
Data structure 2

Fig. 3. Visualisation of the data structures used for the velocity field

3.2 Advection

Let $u_1 = (u_{1,x}, u_{1,y})$ be the velocity field after external forces were applied. The advection step `advect(u)` treats the term $-(u \cdot \nabla)u$ in (3), which makes the Navier-Stokes equations non-linear. The idea used by [Stam 2001] is based on the *method of characteristics*. It consists in backtracing the point (x, y) through the velocity field u such that $u(x, y, t + \Delta t) = u(x - \Delta t u_x(x, y, t), y - \Delta t u_y(x, y, t), t)$. Thus the implicit advection solver defines the new velocity field u_2 by

$$u_2(x, y) = u_1(x - \Delta t u_{1,x}(x, y), y - \Delta t u_{1,y}(x, y))$$

However, when backtracing a center (x, y) of our cells, the point $(x - \Delta t u_x, y - \Delta t u_y)$ will not necessarely be the center of one of our grid cells. Therefore, the value at that point is unknown, so we bilinearly interpolate it. Moreover, as mentionned in section 2.1, we are using fixed boundary conditions. In other words, we clamp the value of $(x - \Delta t u_x, y - \Delta t u_y)$ so that it stays inside our grid.

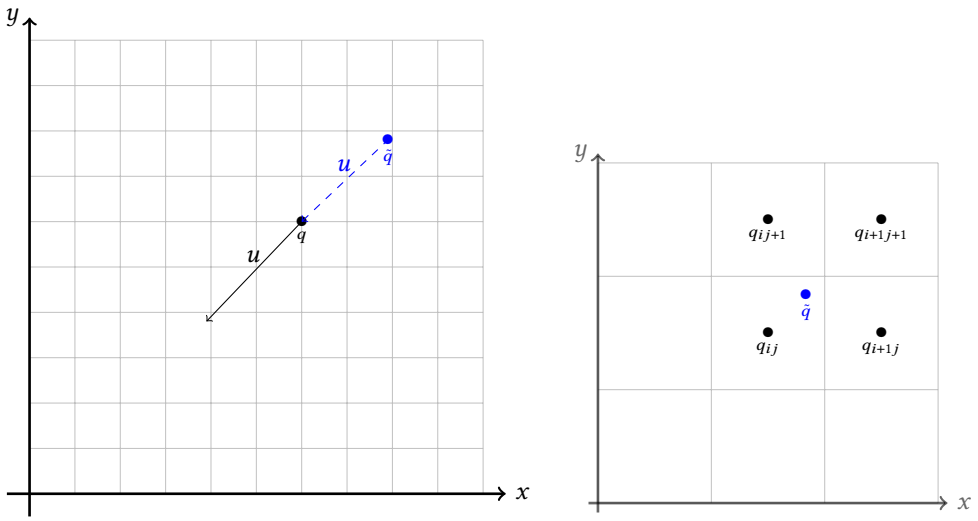


Fig. 4. Point q is backtraced to $\tilde{q}$ using the current velocity field u (*left*). The value at q is bilinearly interpolated (*right*) using the stencil $(q_{ij}, q_{i+1j}, q_{ij+1}, q_{i+1j+1})$

3.3 Poisson solver

Next two steps will involve the operator ∇^2, which implies to solve a Poisson equation $w = \nabla^2 q$, where q is the unknown and w a given scalar field. First, we discretize the Poisson equation with the forward derivative and then with the backward derivative:

$$\begin{aligned}
\forall i, j \in [\![2, n-1]\!], w_{ij} = \frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial y^2} &= \frac{\left(\frac{\partial q}{\partial x}\right)_{i+1} - \left(\frac{\partial q}{\partial x}\right)_{i}}{dx} + \frac{\left(\frac{\partial q}{\partial y}\right)_{j+1} - \left(\frac{\partial q}{\partial y}\right)_{j}}{dy} \\
&= \frac{q_{i+1j} - q_{ij} - (q_{ij} - q_{i-1j})}{(dx)^2} + \frac{q_{ij+1} - q_{ij} - (q_{ij} - q_{ij-1})}{(dy)^2} \\
\text{so } w_{ij} &= \frac{q_{i+1j} + q_{i-1j} + q_{ij+1} + q_{ij-1} - 4q_{ij}}{h^2} \text{ by taking } dx = dy = h \qquad (4)
\end{aligned}$$

This discrezaition did not consider the boundary conditions. To take theme into account, (4) is slightly modified for $i, j \in \{0, n\}$. With the fixed boundary counditions we are using, it means that $q_{i\pm1 j\pm1} = 0$ when $i \pm 1, j \pm 1 \in \{-1, n+1\}$. Then, we found two ways to build our Poisson solver, both different than the FISHPAK routines used by [Stam 2001]. Those Poisson solvers are both based on the data structures mentioned in section 3.1.

- The first way is to build a matrix A such that (4) is equivalent to $Aq = w$, where q and w follow the data structure 2. A is called the *Laplacian matrix*.
- The second way is to use a Gauss-Seidel like process: we iteratively update q_{ij} using the values of the stencil, until a certain number of iterations.

Algorithm 2 Poisson solver based on Laplacian Matrix

$A \longleftarrow$ zero matrix of size $n^2 \times n^2$
for $it \in [\![0, n^2 - 1]\!]$ **do**
  $A[it, it] \longleftarrow \dfrac{-4}{h^2}$
  $i \longleftarrow it\%n$
  $j \longleftarrow it//n$
  if $i > 1$ **then**
    $A[it, it-1] \longleftarrow \dfrac{1}{h^2}$
  end if
  if $i < n - 1$ **then**
    $A[it, it+1] \longleftarrow \dfrac{1}{h^2}$
  end if
  if $j > 1$ **then**
    $A[it, it-n] \longleftarrow \dfrac{1}{h^2}$
  end if
  if $j < n - 1$ **then**
    $A[it, it+n] \longleftarrow \dfrac{1}{h^2}$
  end if
end for
return $q = A^{-1}w$

Algorithm 3 Poisson solver based on Gauss-Seidel process

$q \longleftarrow$ zero matrix of size $n \times n$
while $iter < N_{iter}$ **do**
  for $i \in [\![1, n]\!]$ **do**
    for $j \in [\![1, n]\!]$ **do**
      $q[i, j] \longleftarrow \dfrac{1}{4}\left(h^2 w[i, j] - (q[i+1, j] + q[i-1, j] + q[i, j+1] + q[i, j-1])\right)$
    end for
  end for
  $iter \longleftarrow iter + 1$
end while
return q

3.4 Diffusion

The diffusion step `diffuse(u)` updates the velocity field u such that $\dfrac{\partial u}{\partial t} = \nu\nabla^2 u$. Denoting u_2 the velocity field after the advection step, an implicit solver is used to find u_3 the velocity field after diffusion: $u_3 = u_2 + \Delta t \nu \nabla^2 u_3$. This can be written $(I - \Delta t \nu \nabla^2)u_3 = u_2$ where I is the identity operator, leading to a solvable matrix equation. Using the same discretization process than in section 3.3, it possible to use the same Gauss-Seidel or Laplacian matrix algorithm with the changes $\dfrac{-4}{h^2} \longrightarrow \dfrac{1 + 4\nu\Delta t}{h^2}$ and $\dfrac{1}{h^2} \longrightarrow \dfrac{-\nu\Delta t}{h^2}$.

3.5 Projection

Once we're done with advection and diffusion, we need to project the velocity u_3 we obtained, so that $u(t + \Delta t) = \mathbb{P}(u_3)$ is divergence-free. To do so, we need to find q using the remark 2. This step requires to solve the Poisson equation $\nabla \cdot u_3 = \nabla^2 q$, using one of the algorithm described in section 3.3 with $w = \nabla \cdot u_3$.

3.6 Moving a substance through the fluid

When a non-reactive substance s moves through a fluid, it experiences a variety of physical effects that can be modeled using fluid dynamics. One way to represent the motion of a non-reactive substance is through the use of a density field, which is a scalar field that describes the distribution of the substance within the fluid, such as milk stirred in coffee or smoke rising from a cigarette. The motion of the substance can then be described by the advection of the density field, which is governed by the Navier-Stokes equations as well.

$$\frac{\partial s}{\partial t} = -u \cdot \nabla s + \kappa \nabla^2 s - \alpha s + f_s \tag{5}$$

As the substance s moves through the fluid, it can also undergo diffusion and mixing with the surrounding fluid, which can be captured through the additional diffusion term $-\alpha s$ in the equation (5).

Algorithm 4 Stable fluid solver loop for a substance s moving in fluid

1: $u \longleftarrow$ Stable fluid solver loop for the velocity field
2: $s \longleftarrow s + \Delta t f_s$ `//addExternalForces(s,f)`
3: $s \longleftarrow s(x - \Delta t u_x(x, y), y - \Delta t u_y(x, y))$ `//advect(s,u)`
4: $s \longleftarrow (I - \Delta t \kappa \nabla^2)^{-1} s$ `//diffuse(s)`
5: $s \longleftarrow \frac{s}{1 + \Delta t \alpha}$ `//dissipate(s)`

The dynamics of the density field can be visualized using various techniques, such as particle tracing or contour plotting.

4 RESULTS AND COMMENTS

All the previous steps were implemented in `Python` using `Numpy` to make a stable fluid solver. In this section, we discuss different results we obtained.

4.1 Settings

Our fluid is characterized by the following parameters :

- `SIZE`: a integer corresponding to n the size of the axis in the grid
- `TIMESTEP`: a float corresponding to Δt the time step
- `N_FRAMES`: the number of frame to render
- `GRID_MIN`: the bottom left corner of the grid will be at (`GRID_MIN`, `GRID_MIN`)
- `GRID_MAX`: the upper right corner of the grid will be at (`GRID_MAX`, `GRID_MAX`)
- `VISC`: the viscosity ν of the fluid
- `K_DIFF`: the diffusion constant κ for the fluid's density
- `ALPHA`: the dissipation rate α for the fluid's density
- `u_init`: the initial velocity field
- `density_init`: the initial density field

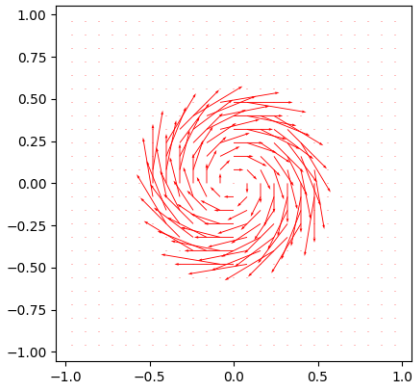
4.2 Vortex

In this example, our initial velocity field is a vortex defined by

$$u(x,y) = \begin{cases} \begin{pmatrix} -a(x-c_x) \\ a(y-c_y) \end{pmatrix} & \text{if } (x-c_x)^2 + (y-c_y)^2 \leqslant r^2 \\ 0 & \text{otherwise} \end{cases}$$

where $c = \begin{pmatrix} c_x \\ c_y \end{pmatrix}$ is the center of the vortex, a its magnitude and r its radius given by the user. The viscosity is set to $\nu = 0$. For the density field, we also started with a sphere with the same center c than the vortex, and remove the diffusion and disspation phenomena ($\kappa = \alpha = 0$):

$$s(x,y) = \begin{cases} 1 & \text{if } (x-c_x)^2 + (y-c_y)^2 \leqslant r^2 \\ 0 & \text{otherwise} \end{cases}$$

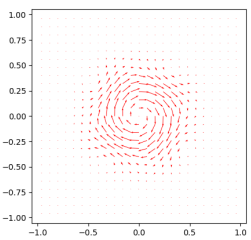


(a) Initial velocity field (grid $[-1,1]$ of size 25×25)

(b) Initial density field (grid $[-1,1]$ of size 99×99)

Fig. 5. Initial vortex field. As the number of arrows in the velocity field increases, displaying it in a readable manner becomes challenging. To address this issue, we have included lower resolution visuals in this report, which allow for a more accessible interpretation of the velocity field.

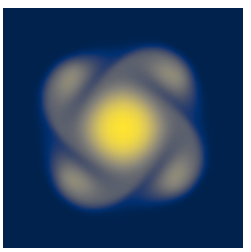
We then simulated the fluid's motion over 300 frames with a timestep $\Delta t = 0.025$s.



(a) Velocity at frame 20

(b) Density at frame 20

(c) Density at frame 50

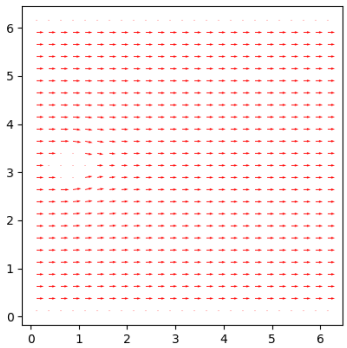


(d) Density at frame 170

Fig. 6. Evolution of a vortex field. As the number of arrows in the velocity field increases, displaying it in a readable manner becomes challenging. To address this issue, we have included lower resolution visuals in this report, which allow for a more accessible interpretation of the velocity field.

4.3 Karman vortex street

An unsteady fluid behavior is the Karman vortex street, which occurs when a fluid flows past a bluff body, such as a cylinder or sphere. This results in the formation of alternating vortices in the wake of the body, which are shed downstream in a periodic manner. This phenomenon has important practical applications in fields such as aerospace engineering, where it can affect the stability of aircraft and rockets. To visualize it, we represented the vorticity, which is defined by the curl of the velocity field: $\nabla \times u$. In 2D, it corresponds to a scalar field equal to $\frac{\partial u_y}{\partial x} - \frac{\partial u_x}{\partial y}$. Note that the boundary conditions are slightly changed: instead of fixing 4 walls, we open up the right-side.

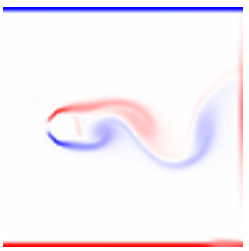


(a) Initial velocity field (grid $[-1, 1]$ of size 25×25)

(b) Initial vorticity field (grid $[-1, 1]$ of size 90×90)

Fig. 7. Initial velocity field and vorticity field.

(a) Vorticity at frame 70



(b) Vorticity at frame 441

(c) Vorticity at frame 580

Fig. 8. Evolution of a Karman vortex street.

4.4 About the data structure

We simulated the vortex in section 4.2 with both data structures described in 3.1. While the exceution time was roughly the same, differences started to appear at some points. For instance, when increasing the resolution of the grid or the magnitude of the vortex, the data structure 1 started to present some artifacts: the density field kept increasing its size, while the velocity field remained the same as in data structure 2. At the time this report is written, the cause of this artifact is still unknown to us. It might be because something was wrong in our implementation or either because the Gauss-Seidel process did not converge well, but the fact that the velocity field is the same than the one obtained with data structure 2, plus the fact that dissipation rate and diffusion rate were

set to 0 for the density field, make it hard to explain.

Moreover, when using `TensorFlow`, we cannot use `for` loops to do operations on the velocity field u (but it is still possible to use them to build variables), otherwise the software is not able to keep trace of the velocity field and consequently the gradient cannot be computed. Both those reasons led us to prefer using data structure 2 over data structure 1 in this work.

4.5 TensorFlow

TensorFlow is a popular open-source software library for machine learning and artificial intelligence that has become increasingly popular in recent years [Abadi et al. 2015]. One of the most exciting areas of research that TensorFlow has enabled is in differentiable physics. By using TensorFlow to develop differentiable solvers for physical systems, it is possible to simulate complex systems with more accuracy and efficiency than traditional numerical methods. This is because the differentiable solvers can be trained using machine learning techniques, allowing them to learn from data and refine their models over time. The ability to simulate physical systems in this way has many practical applications, such as in the design of new materials or the optimization of energy systems.

4.6 Differentiable physics

Differentiable physics is a set of methods that aims at incorporating differentiable numerical simulations into the learning process [Thuerey et al. 2021]. In TensorFlow, it is ensured by keeping trace of the variables so that the gradient can be numericallly computed using the chain rule for all operations used in a given simulation. For example, consider our stable fluid solver without external forces, diffusion and dissipation. Denote by

- $\mathbb{F}_1$ the solver for the velocity field u,
- $\mathbb{F}_2$ the solver for a scalar field s,
- $\mathbb{A}_1$ the advection step for the velocity field,
- $\mathbb{A}_2$ the advection step for the scalar field,
- and $\mathbb{P}$ the projection step.

Then $\mathbb{F}_2(s) = \mathbb{A}_2(s, \mathbb{F}_1(u)) = \mathbb{A}_2(s, \mathbb{P} \circ \mathbb{A}_1(u))$, and its gradient with respect to u is the Jacobian matrix given by

$$\left.\frac{\partial \mathbb{F}_2}{\partial u}\right|_u = \left.\frac{\partial \mathbb{A}_2}{\partial u}\right|_{(s,\mathbb{P}(\mathbb{A}_1(u).))} \times \left.\frac{\partial \mathbb{P}}{\partial u}\right|_{\mathbb{A}_1(u)} \times \left.\frac{\partial \mathbb{A}_1}{\partial u}\right|_u$$

Then, using backpropagation, gradient information flow from a simulator into an neural network and vice versa. In TensorFlow, the syntax would be the following:

```
import tensorflow as tf
u_init = ...#Initialize u_init
density_init = ...#Initialize density_init
u_init = tf.convert_to_tensor(u_init)
density_init = tf.convert_to_tensor(density_init)
with tf.GradientTape() as tape:
    velocity_field = tf.Variable(u_init)
    density_field = solveDens(density_init, solveVel(velocity_field))
grad = tape.gradient([density_field], [velocity_field])
```

4.7 An application to a matching shape problem

Being able to compute the gradient of our stable fluid solver is a powerful tool. One of the application can be the following: imagine you have a certain pattern s_0 created by milk stirred in your coffee. Then, by changing in a certain way the velocity field of your coffee, you can make this milk taking a desired shape. Formally, let t be the density field we are looking for. Using the same notations as in section 4.6, we introduce the quadratic loss function $\mathcal{L}(s,u) = ||\mathbb{F}_2(s) - t||^2$. Starting with the circle density field s_0, the idea is to minimize $\mathcal{L}$ by performing a gradient descent:

$$\underset{u}{\operatorname{argmin}}\, \mathcal{L}(s_0,u) = \underset{u}{\operatorname{argmin}}\, ||\mathbb{A}_2(s_0, \mathbb{F}_1(u)) - t||^2 \tag{6}$$

The minimization of (6) is done with the sequence $u_{n+1} = u_n - \alpha_n \left.\frac{\partial \mathcal{L}(s_0,u_n)}{\partial u}\right|_{u_n}$ where $\alpha_n \in \mathbb{R}$ is the learning rate.

Algorithm 5 Gradient descent

1: $u \longleftarrow$ a certain velocity field
2: **while** $iter <$ `MAX_ITER` **and** $\mathcal{L}(s_0,u) > \eta$ **and** $\left\| \left.\frac{\partial \mathcal{L}}{\partial u}\right|_u \right\| > \varepsilon$ **do**
3: Select some learning rate α
4: $u \longleftarrow u - \alpha \left.\frac{\partial \mathcal{L}(s_0,u)}{\partial u}\right|_u$
5: $iter \longleftarrow iter + 1$
6: **end while**

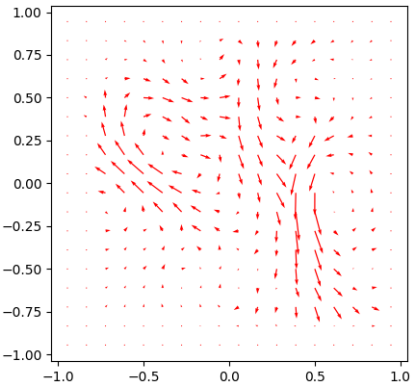
5 DISCUSSIONS AND FUTURE WORK

During our implementation phase, one of the hurdle was to understand the TensorFlow framework. Our first implementation was using data structure 1 (cf. section 3.1) relying on the `Numpy` library, but when we tried to make it compatible with TensorFlow, the solver was not differentiable. Our guess is that the different `for` loops made the framework losing trace of the velocity field, which resulted as a `None` gradient. Indeed, TensorFlow requires every operation applied on the variables to be differentiable, which is not the case for `for` loops. We tried to flatten the matrice of data structure 1 but wrapping and unwrapping it made the time cost too expensive, se we instead went for data structure 2 directly.

Moreover, due to the backpropagation of the variables, our TensorFlow version of the solver takes much more time to execute. This issue was a delicate point to deal with when we trained our model. Indeed, when doing a gradient descent, depending of the nature of the involved functions, we may not find a global optimum, or even worse, we may diverge. Combining this remark with the fact that the execution of our fluid solver using TensorFlow makes the gradient descent delicate to cope with. This is the reason why we trained our solver over only 20 frames and clamped the number of iterations to 100 so that it does not take forever. One way to remove this problem could be parallelization of the process, but this require more investigation about TensorFlow's workspace because no `for` loops are involved in the operations. We also tried to take variable learning rate $\alpha_n = \frac{\alpha}{\sqrt{n}}$: the idea was to globally localize the optimum and then refine the convergence at each step by taking a smaller rate to avoid oscillation or divergence.

(a) Starting density (at frame 0)

(b) Target density field at frame 20



(c) Trained velocity field at frame 5

(d) Obtained density field at frame 20

Fig. 9. Results for a training to trying to get the kanji symbol for *human beings* in japanese/chinese in a 18×18 grid. The training was designed so that the trained density would reach the target density at frame 20. The density images are reversed for clarity purpose (they were at first oriented like the velocity field but it turns out the images were upside down versions of what we wanted). The training reached the maximum number of 100 iterations but the loss function $\mathcal{L}$ was still decreasing so a higher maximum number of iterations would likely give a better result

Additionnaly, having a differentiable solver at disposition opens a bunch of perspective for fluid simulation. With higher power of calculation, it could be possible to try the matching shape problem in higher resolution or to extend the solver in 3D. Since our solver is able to compute the gradients of the fluid simulation with respect to arbitrary parameters, using it during constraint-based fluid simulation would also be a very interesting field to further investigate.

6 CONCLUSION

In conclusion, this project has provided us with valuable experience in implementing a differentiable stable fluid solver using TensorFlow. While the results are promising and show the potential of this approach for fluid simulation and control, there is still much work to be done in optimizing the performance and exploring its applications in more complex scenarios. Overall, this project has deepened out understanding of fluid dynamics and machine learning, and we look forward to continuing to explore this exciting intersection of fields.

A APPENDIX

Our solver is currently in 2D, so if nothing is mentioned, all the specified mathemical notations are taken in $\mathbb{R}^2$.

$[\![a, b]\!]$ refers to all the integers between $a \in \mathbb{Z}$ and $b \in \mathbb{Z}$ (included)
Δt is the time step of the simulation
$u = \begin{pmatrix} u_x \\ u_y \end{pmatrix} \in \mathbb{R}^2$ is a column-vector
if $u \in \mathbb{R}^n$ is a column vector, then its size is denoted n, and its i^{th} component is u_i
if $M \in \mathcal{M}_{n,m}(\mathbb{R})$ is a matrix, then its size is denoted by $n \times m$ and its elements by $m_{ij} \in \mathbb{R}$
if $M \in \mathcal{M}_{n,m}(\mathbb{R}^d)$ is a tensor, then its size is denoted by $n \times m \times d$ and its elements by $m_{ij} \in \mathbb{R}^d$
if $u : \mathbb{R}^n \longrightarrow \mathbb{R}$, then $\left.\frac{\partial u}{\partial x_n}\right|_x$ denotes its partial derivative with respect to the n^{th} variable, evaluated at point x
if $u : \mathbb{R}^n \longrightarrow \mathbb{R}$, and $v \in \mathbb{R}^n$, then $\left.\frac{\partial u}{\partial x}\right|_v = \left(\left.\frac{\partial u}{\partial x_1}\right|_v, \cdots, \left.\frac{\partial u}{\partial x_n}\right|_v \right)^T$ denotes its partial derivative with respect to the x variable, evaluated at point v
if $U : \mathbb{R}^n \longrightarrow \mathbb{R}^m$, then its Jacobian matrix is $\left(\frac{\partial U}{\partial x}\right)_{i \in [\![1,m]\!], j \in [\![1,n]\!]} = \left(\frac{\partial U_i}{\partial x_j}\right)_{i \in [\![1,m]\!], j \in [\![1,n]\!]}$
$\cdot$ is the standard scalar product
if $u, v \in \mathbb{R}^2$, then $u \times v = u_x v_y - u_y v_x$
∇ is the nabla operator and in 2D, $\nabla = \begin{pmatrix} \frac{\partial}{\partial x} \\ \frac{\partial}{\partial y} \end{pmatrix}$
∇u is the gradient of u
$\nabla \cdot u$ is the divergence of u
$\nabla \times u$ is the curl of u
$\nabla^2 = \nabla \cdot \nabla$ is the divergence of the gradient and is called the *Laplacian operator*
I is the identity operator
$|| \cdot ||$ is the Euclidean norm

ACKNOWLEDGMENTS

The author would like to thank Mr. Kiwon Um of Telecom Paris for monitoring this project.

REFERENCES

Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

R. Bridson. 2015. *Fluid Simulation for Computer Graphics, Second Edition.* Taylor & Francis. https://books.google.fr/books?id=7MySoAEACAAJ

Jos Stam. 2001. Stable Fluids. *ACM SIGGRAPH 99* 1999 (11 2001). DOI: https://doi.org/10.1145/311535.311548

Nils Thuerey, Philipp Holl, Maximilian Mueller, Patrick Schnell, Felix Trost, and Kiwon Um. 2021. *Physics-based Deep Learning.* WWW. https://physicsbaseddeeplearning.org