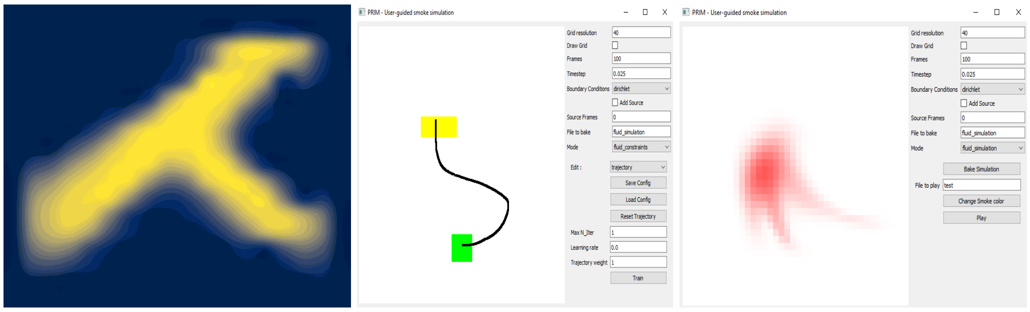
User-guided smoke simulation

SAMMY RASAMIMANANA, Télécom Paris, France



Grid resolution 40
Draw Grid ☐
Frames 100
Timestep 0.025
Boundary Conditions dirichlet
☐ Add Source
Source Frames 0
File to bake fluid_simulation
Mode fluid_constraints
Edit : trajectory
Max N_Iter 1
Learning rate 0.0
Trajectory weight 1

Grid resolution 40
Draw Grid ☐
Frames 100
Timestep 0.025
Boundary Conditions dirichlet
☐ Add Source
Source Frames 0
File to bake fluid_simulation
Mode fluid_simulation
File to play test

Fig. 1. The result of optimizing a velocity field to make a density field matching a target shape *(left)* after a certain number of frame. Our differentiable stable fluid solver allows to compute gradients with respect to velocity field and was incorporate into an optimizer that guides the motion *(middle)* of a morphing fluid *(right)*.

In this project, we present a user-guided interface that uses a differentiable stable fluid solver implemented with TensorFlow *to get an artistic control over the fluid trajectory. After a quick reminder of stable fluid solvers, we describe the architecture of our differentiable solver and show how it can be trained using backpropagation. We then demonstrate the effectiveness of our solver on a matching-shape problem and on a constrained trajectory problem. Our results show that our differentiable stable fluid solver can achieve some state-of-the-art performance while only optimizing the initial velocity field.*

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

1 INTRODUCTION

1.1 Motivation

Fluid simulation is an intriguing and difficult field to master completely, particularly when it comes to accurately modeling natural phenomena such as the flow of water, smoke or fire. On one hand, an artist working in the special effects industry will strive to fully understand the physics of the Navier-Stokes equations to give the fluid a realistic behavior. On the other hand, pragmatism and knowledge of physics may limit a scientist in the process of creating stylized 2D or 3D animations. As most solutions are based on numerical solvers, the advent of machine learning tools [Abadi et al. 2015] that improve differentiable physics [Thuerey et al. 2021] presents a great opportunity to solve constraint-based physics problems. Consequently, this project aims at creating an interface that would enable user-guided control of a smoke simulation. The main objective is to have artistic control over the smoke trajectory while morphing the fluid towards a given target.

1.2 Related work

Previous studies in producing dynamic effects have focused on investigating the effects of injecting physcially-based elements in the design process of stylized animations. [Xing et al. 2016] presents an interactive interface for designing elemental dynamics for animated illustrations. The idea is to use *flow particles* that create local velocity fields which will influence the strokes within its defined range. They implemented different types of flow particles to represent various velocity

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

field for different patterns such as wind, swirl or smoke. These fields can convey both realistic and artistic effects based on user specification. [Holl et al. 2020] has developed a novel hierarchical predictor-corrector machine learning method to understand and control complex nonlinear physical systems over time. Their method requires a differentiable partial differential equations (PDE) solver and relies on subdividing the scheme in half: planning optimal trajectories and controllling the parameters with a control neural network. Our approach uses similar ideas to find an initial velocity field that would allow the smoke to reach a target while following a certain trajectory before turning into that target.

1.3 Notations

For a detailed reference of the mathematical notations employed in this report, kindly refer to the appendix A.1.

2 STABLE FLUID SOLVER

Our 2D implementation is based on [Stam 2001] scheme that goes over the different steps of the Navier-Stokes equation before projecting the velocity field to the divergence-free space :

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

The fluid domain is discretized into a $n \times n$ grid. All the components of a scalar field are stored in a vector of size n^2. Therefore, for a vector field $u = (u_x, u_y)$, we have two huge vectors of size n^2.

Algorithm 1 Stable fluid solver loop for the velocity field

```
1: $u \longleftarrow u + \Delta t f$ //addExternalForces(u,f)
2: $u \longleftarrow u\big(x - \Delta t u_x(x,y), y - \Delta t u_y(x,y)\big)$ //advect(u)
3: $u \longleftarrow (I - \Delta t \nu \nabla^2)^{-1} u$ //diffuse(u), where $I$ is the identity operator
4: $q \longleftarrow$ solution of Poisson equation $\nabla \cdot u = \nabla^2 q$ //solvePressure(u)
5: $u \longleftarrow u - \nabla q$ //project(u)
```

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.

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

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 (2).

$$u = (u_x, u_y) \text{ where } u_x = \begin{pmatrix} u_{x,00} \\ u_{x,10} \\ \vdots \\ u_{x,n0} \\ u_{x,n1} \\ \vdots \\ u_{x,nn} \end{pmatrix} \text{ and } u_y = \begin{pmatrix} u_{y,00} \\ u_{y,10} \\ \vdots \\ u_{y,n0} \\ u_{y,n1} \\ \vdots \\ u_{y,nn} \end{pmatrix} u_x[i + jn] = u_{x,ij} \text{ and } u_y[i + jn] = u_{y,ij}$$

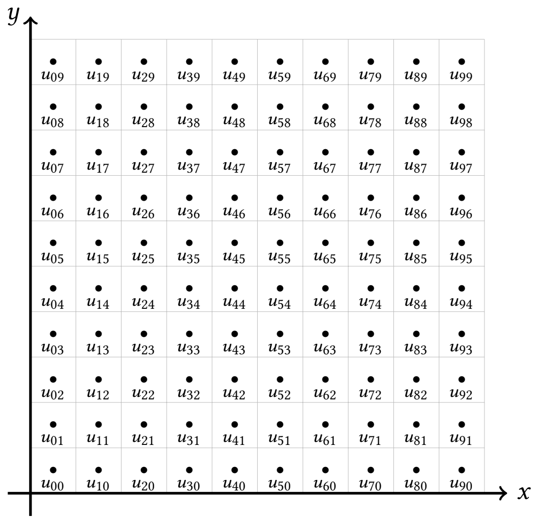

y
x
u09 u19 u29 u39 u49 u59 u69 u79 u89 u99
u08 u18 u28 u38 u48 u58 u68 u78 u88 u98
u07 u17 u27 u37 u47 u57 u67 u77 u87 u97
u06 u16 u26 u36 u46 u56 u66 u76 u86 u96
u05 u15 u25 u35 u45 u55 u65 u75 u85 u95
u04 u14 u24 u34 u44 u54 u64 u74 u84 u94
u03 u13 u23 u33 u43 u53 u63 u73 u83 u93
u02 u12 u22 u32 u42 u52 u62 u72 u82 u92
u01 u11 u21 u31 u41 u51 u61 u71 u81 u91
u00 u10 u20 u30 u40 u50 u60 u70 u80 u90

Fig. 2. Visualisation of the data structure used for the velocity field

Algorithm 2 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)
```

2.1 Centered grid

The first version of our solver uses the centered grid representation, which means every physical quantities is sampled at the center of the grid.

When using this configuration, a Poisson equation $w = \nabla^2 q$ is discretized with the forward derivative and then with the backward derivative:

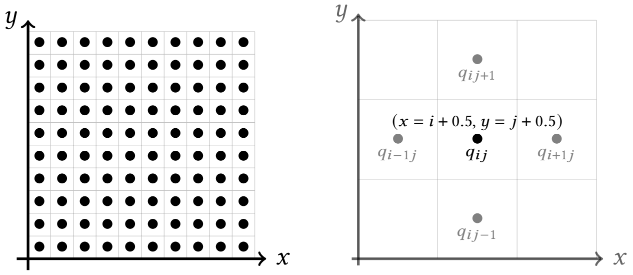


Fig. 3. Our discretization grid (*left*) and physical quantities stored at a cell (i, j). The cells represented here are of size 1×1 for clarity purpose. Hence, for a physical quantity q, its sampled value at cell (i, j) is $q_{ij} = q(x, y) = q(i + 0.5, j + 0.5)$ (*right*).

$$\begin{aligned}\forall i, j \in [\![2, n-1]\!], w_{ij} &= \frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial y^2} = \frac{\left(\frac{\partial q}{\partial x}\right)_{i+1} - \left(\frac{\partial q}{\partial x}\right)_i}{dx} + \frac{\left(\frac{\partial q}{\partial y}\right)_{j+1} - \left(\frac{\partial q}{\partial y}\right)_j}{dy} \\ &= \frac{q_{i+1j} - q_{ij} - (q_{ij} - q_{i-1j})}{(dx)^2} + \frac{q_{ij+1} - q_{ij} - (q_{ij} - q_{ij-1})}{(dy)^2} \\ \text{so } w_{ij} &= \frac{q_{i+1j} + q_{i-1j} + q_{ij+1} + q_{ij-1} - 4q_{ij}}{h^2} \text{ by taking } dx = dy = h \qquad (3)\end{aligned}$$

and the divergence of a vector field $q = (q_x, q_y)$ would be discretized using the average derivative

$$\forall i, j \in [\![2, n-1]\!], (\nabla \cdot q)_{ij} = (\nabla \cdot q)(x, y) = \frac{\partial q_x}{\partial x} + \frac{\partial q_y}{\partial y} = \frac{q_{i+1j} - q_{i-1j}}{2dx} + \frac{q_{ij+1} - q_{ij-1}}{2dy}$$

2.2 Differentiability using 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. 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. From now, we will consider our stable fluid solver without external forces, diffusion and dissipation. Denote by

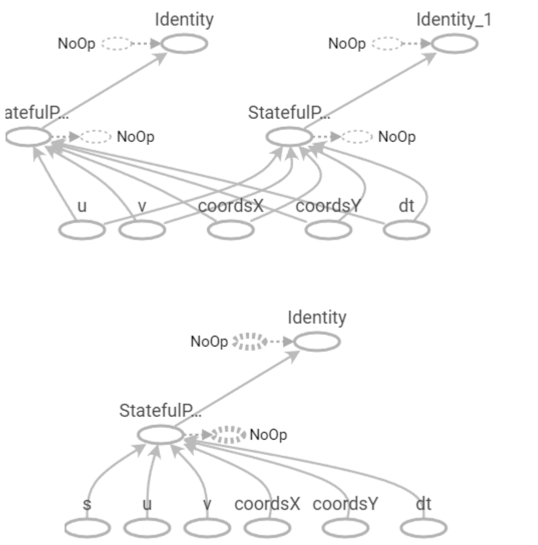
- $\mathbb{F}_1$ the solver for the velocity field u,
- $\mathbb{F}_2$ the solver for a scalar field s,
- $\mathbb{A}_1$ the advection step for the velocity field,
- $\mathbb{A}_2$ the advection step for the scalar field,
- and $\mathbb{P}$ the projection step.

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

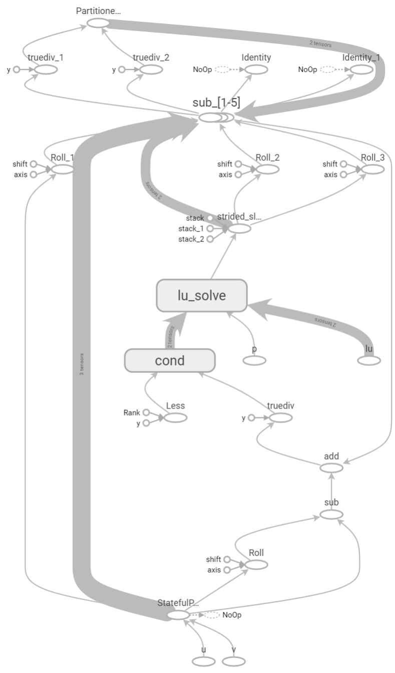
$$\left.\frac{\partial \mathbb{F}_2}{\partial u}\right|_u = \left.\frac{\partial \mathbb{A}_2}{\partial u}\right|_{(s, \mathbb{P}(\mathbb{A}_1(u).))} \times \left.\frac{\partial \mathbb{P}}{\partial u}\right|_{\mathbb{A}_1(u)} \times \left.\frac{\partial \mathbb{A}_1}{\partial u}\right|_u$$

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

```python
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])
```



(a) TensorFlow's graph for the advection of the velocity and the density fields



(b) TensorFlow's graph for the projection of the velocity field

Fig. 4. Computation graph of the different step of our TensorFlow fluid solver. The visual were obtained using TensorBoard and TensorFlow's graph module.

To build the graphs of the different functions in figure 4, we used the decorator `@tf.function` which allows TensorFlow to trace a function and add it in the computation graph. This computation graph represents the operations and dependencies within the function. Combined with the `tf.vectorized_map` function, this decorator also speed up the advection part because all points on the grid are simulatneously bilinearly interpolated since the computation graph is only built once.

2.3 Staggered grid

Due to the numerical dissipation of the density field, we tried to use another configuration to limit the discrete approximation that led to numerical loss for the density field. In staggered grid, all scalar fields are still sampled at the center of the cells, but the velocity field is now sampled at the edges of the grid.

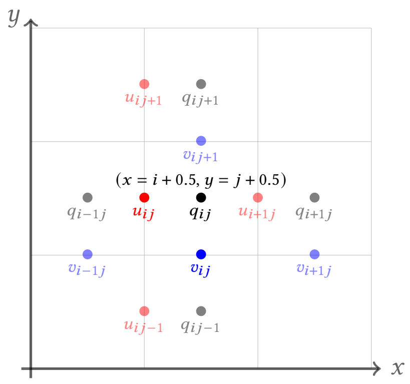


Fig. 5. The cells represented here are of size 1×1 for clarity purpose. A scalar field q is still sampled at the centered of a cell (i, j): $q_{ij} = q(x, y) = q(i + 0.5, j + 0.5)$. The velocity field is sampled at the edges of the cells such that $u_{ij} = u(x - 0.5, y) = u(i, j + 0.5)$ (the x-component, in red) and $v_{ij} = v(x, y - 0.5) = v(i + 0.5, j)$ (the y-component, in blue)

When using this configuration, a Poisson equation $w = \nabla^2 q$ is discretized at the center of the cells with the forward derivative:

$$\begin{aligned}
\forall i, j \in [\![2, n-1]\!], w_{ij} &= \frac{\partial^2 q}{\partial x^2} + \frac{\partial^2 q}{\partial y^2} = \frac{\left(\frac{\partial q}{\partial x}\right)_{i+0.5} - \left(\frac{\partial q}{\partial x}\right)_{i-0.5}}{dx} + \frac{\left(\frac{\partial q}{\partial y}\right)_{j+0.5} - \left(\frac{\partial q}{\partial y}\right)_{j-0.5}}{dy} \\
&= \frac{q_{i+1j} - q_{ij} - (q_{ij} - q_{i-1j})}{(dx)^2} + \frac{q_{ij+1} - q_{ij} - (q_{ij} - q_{ij-1})}{(dy)^2} \\
\text{so } w_{ij} &= \frac{q_{i+1j} + q_{i-1j} + q_{ij+1} + q_{ij-1} - 4q_{ij}}{h^2} \text{ by taking } dx = dy = h \qquad (4)
\end{aligned}$$

The gradient of a scalar field q at the edges and the divergence of the velocity field (u, v) at the center of the cells would be discretized using the forward derivative too:

$$\forall i \in [\![1, n-1]\!] j \in [\![1, n]\!], (\nabla q)_x(x - 0.5, y) = \frac{\partial q}{\partial x} = \frac{q_{i+1j} - q_{ij}}{dx} \qquad (5)$$

$$\forall i \in [\![1, n]\!] j \in [\![2, n]\!], (\nabla q)_y(x, y - 0.5) = \frac{\partial q}{\partial y} = \frac{q_{ij} - q_{ij-1}}{dy} \qquad (6)$$

$$\forall i \in [\![1, n-1]\!] j \in [\![1, n]\!], \left(\nabla \cdot \begin{pmatrix} u \\ v \end{pmatrix}\right)(x, y) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = \frac{u_{i+1j} - u_{ij}}{dx} + \frac{v_{i+1j} - v_{ij}}{dy} \qquad (7)$$

3 CONSTRAINING THE SIMULATION

Being able to compute the gradient of our stable fluid solver is a powerful tool. It allows our solver to be integrated into constraint-based problems. The one we chose to tackle is the following: given an initial shape s_0, a target shape s^* and a trajectory $\gamma(t)$ that ultimately reaches s^*, we want to find an initial velocity field u_0 such that under its influence, s_0 follow the trajectory γ and to turn into the target shape s^* at the end. We now need to find an appropriate loss function $\mathcal{L}(s^*, s_0, \gamma, u_0)$ that describes this problem, so that we can optimize it using gradient descent.

Algorithm 3 Gradient descent

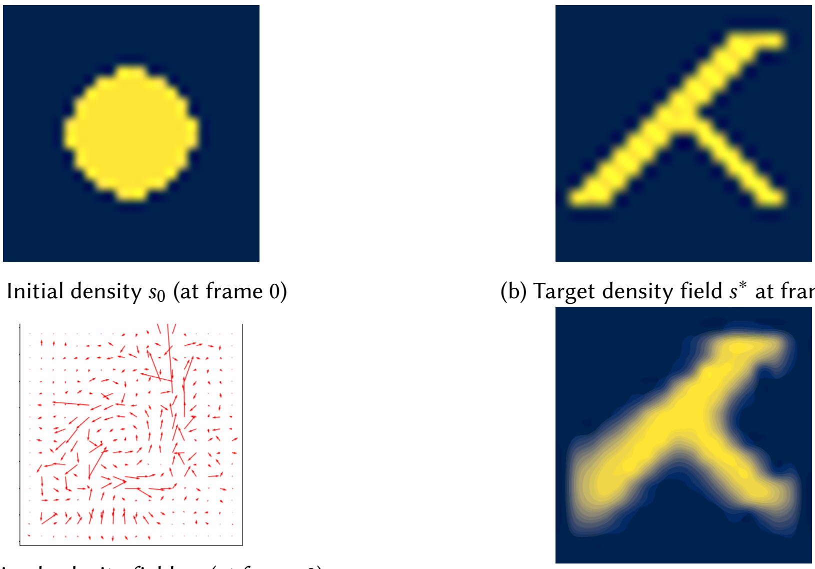
```
1: $u_0 \longleftarrow$ a certain velocity field
2: while $iter <$ MAX_ITER and $\mathcal{L}(s^*, s_0, \gamma, u_0) > \eta$ and $\left\| \frac{\partial \mathcal{L}}{\partial u_0}\Big|_{u_0} \right\| > \varepsilon$ do
3:     Select some learning rate $\alpha$
4:     $u_0 \longleftarrow u_0 - \alpha \frac{\partial \mathcal{L}(s^*, s_0, \gamma, u_0)}{\partial u_0}\Big|_{u_0}$
5:     $iter \longleftarrow iter + 1$
6: end while
```

3.1 Density constaint

The two shapes described earlier are actually representing density fields. Denoting by $\mathbb{F}$ our stable fluid solver, a first component of the loss function is the quadratic loss

$$\mathcal{L}_{density} = \frac{1}{2}||\mathbb{F}(s_0, u_0) - s^*||^2 \geqslant 0$$

that, when being optimized, attempts to match the simulated density $\mathbb{F}(s_0, u_0)$ to s^* .


(a) Initial density s_0 (at frame 0)

(b) Target density field s^* at frame 20

(c) Trained velocity field u_0 (at frame 0)

(d) Simulated density field $\mathbb{F}(s_0, u_0)$ at frame 20

Fig. 6. Results for a training to try getting a mirrored λ in a 20×20 grid. The training was designed so that the trained density would reach the target density at frame 20 without following any particular trajectory.

3.2 Velocity constraint

Now that we can morph an initial density s_0 to a target s^*, we want s_0 to follow a trajectory $\gamma(t)$ while morphing into s^*. To do so, the idea is to make the velocity field following that trajectory: we sample the different cells $\gamma(t_0), \cdots, \gamma(t_N)$ where the trajectory passes by. Then, for a cell $\gamma(t_i)$ we compute its velocity along the curve using finite difference: $u^*(\gamma(t_i)) = \gamma(t_{i+1}) - \gamma(t_{i-1})$. Denoting $\mathbb{F}(u_0, t_i)(\gamma(t_i))$ the simulated velocity field at time t_i and cell $\gamma(t_i)$, a velocity component for the loss function can then be introduced

$$\mathcal{L}_{velocity} = \sum_{i=0}^{N} 1 - \cos\Big(u^*(\gamma(t_i)), \mathbb{F}(u_0, t_i)(\gamma(t_i))\Big) \geqslant 0$$

The choice of cosine similarity allows to align the velocity field with the trajectory without caring about its magnitude, whereas a quadratic loss would have require renormalizing the trajectory.

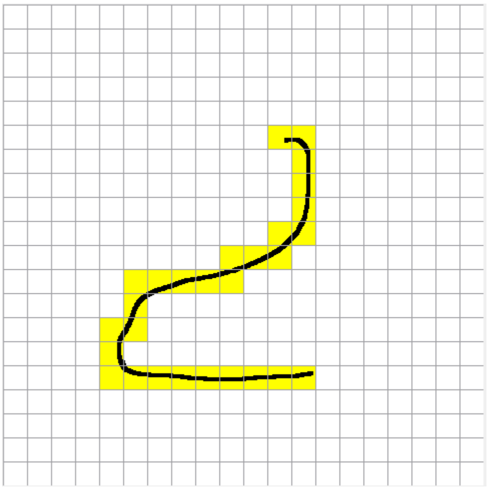


Fig. 7. Sampling cells $\gamma(t_0), \cdots, \gamma(t_N)$ of a trajectory $\gamma(t)$.

3.3 About the initialisation in the gradient descent

Ultimately, our loss function we want to optimize is

$$\mathcal{L}(s^*, s_0, \gamma, u_0) = \mathcal{L}_{density} + \mathcal{L}_{velocity} = \frac{1}{2}||\mathbb{F}(s_0, u_0) - s^*||^2 + \sum_{i=0}^{N} 1 - \cos\Big(u^*(\gamma(t_i)), \mathbb{F}(u_0, t_i)(\gamma(t_i))\Big) \geqslant 0$$

The result of the gradient descent can heavily depends on the initial velocity field u_0. We tried multiple types of initialisation.

3.3.1 *Curl approach.* The fundamental theorem of vector calculus states that any divergence-free vector field u_0 can be expressed as a solenoidal field, meaning that the vector field derives from a vector potential A, such that $\nabla \times A = u_0$. The curl approach consists in training A and then computing u_0 to enforce the divergence-free condition on u_0.

Algorithm 4 Curl approach

```
1: $A \longleftarrow$ a random vector field
2: while $iter$ < MAX_ITER and $\mathcal{L}(s^*, s_0, \gamma, \nabla \times A) > \eta$ and $\left\| \frac{\partial \mathcal{L}}{\partial A}\Big|_A \right\| > \varepsilon$ do
3:     Select some learning rate $\alpha$
4:     $A \longleftarrow A - \alpha \left.\frac{\partial \mathcal{L}(s^*, s_0, \gamma, \nabla \times A)}{\partial A}\right|_A$
5:     $iter \longleftarrow iter + 1$
6: end while
```

3.3.2 Vortex approach. Another idea was to use an embodiment of the divergence-free phenomenon: vortices. In this approach, the training occurs on a set of vortices. More precisely the user would choose a number m of vortices $v_k(p_k, r_k, a_k)$, and then the optimizer calculates their position p_k, their radius r_k and their magnitude a_k. Since a vortex is divergence-free and $\nabla \cdot$ is a linear operator, $u_0 = v_1(p_1, r_1, a_1) + \cdots + v_m(p_m, r_m, a_m)$ is still divergence-free.

Algorithm 5 Vortex approach

```
1: $m \longleftarrow$ a number of vortices
2: $p = (p_1, \cdots, p_m), r = (r_1, \cdots, r_m), a = (a_1, \cdots, a_m) \longleftarrow m$ random positions, radius and
   magnitude
3: while $iter$ < MAX_ITER and $\mathcal{L}(s^*, s_0, \gamma, p, r, a) > \eta$ and $\left\| \left.\frac{\partial \mathcal{L}}{\partial p, r, a}\right|_{p,r,a} \right\| > \varepsilon$ do
4:     Select some learning rate $\alpha$
5:     $p, r, a \longleftarrow p, r, a - \alpha \left.\frac{\partial \mathcal{L}(s^*, s_0, \gamma, p, r, a)}{\partial p, r, a}\right|_{p,r,a}$
6:     $iter \longleftarrow iter + 1$
7: end while
```

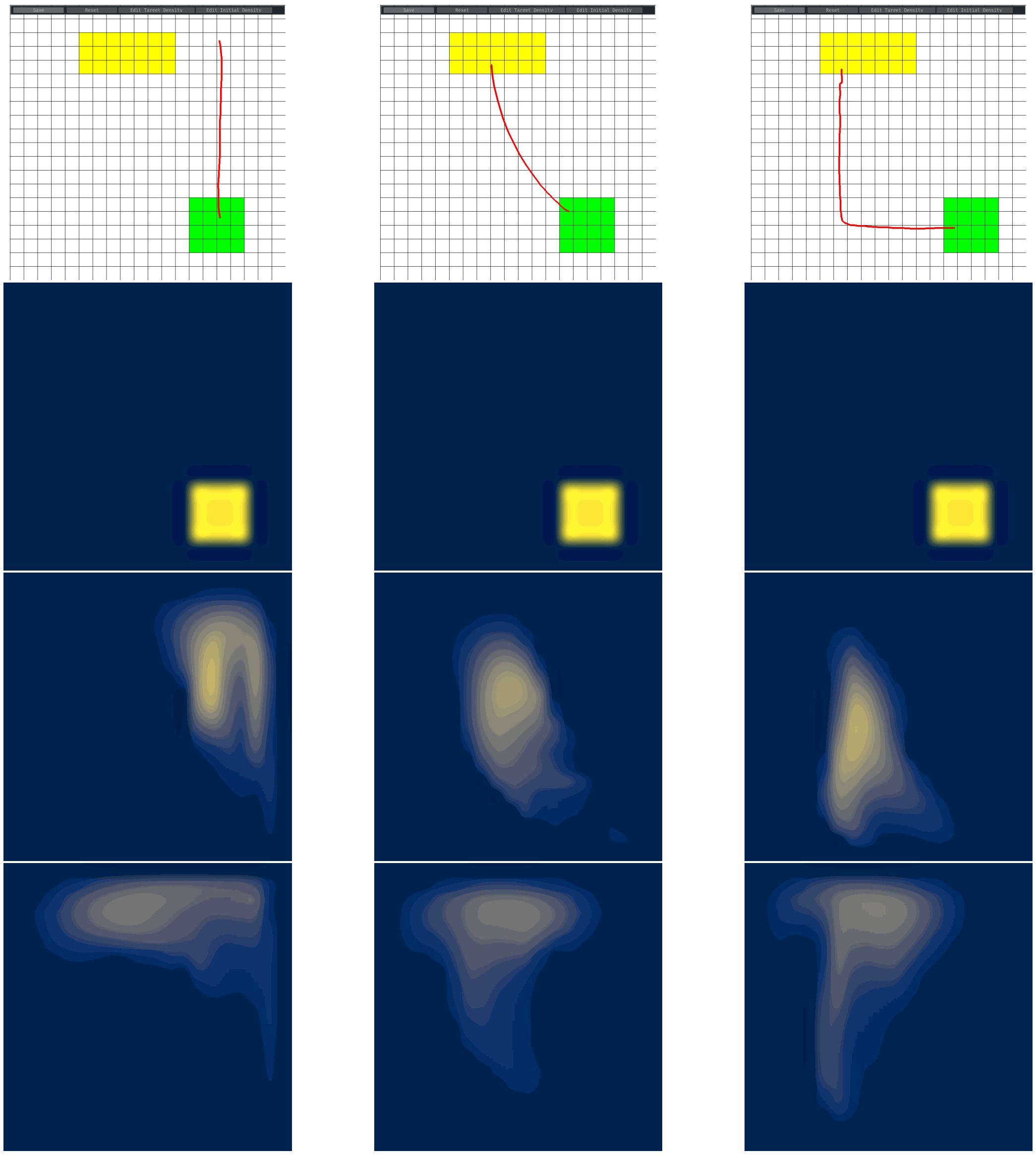
3.3.3 Trajectory approach. The approach ultimately we kept was to initialize u_0 to $u^*(\gamma(t))$ so that the optimiser gets a little help in understanding the velocity constraint. However, this approach alone does not guarantee that u_0 is divergence-free, this is the reason why we project the velocity field after each iteration of the optimizer.

Algorithm 6 Trajectory approach

```
1: $u_0 \longleftarrow u^*(\gamma(t))$
2: $u_0 \longleftarrow$ project($u_0$)
3: while $iter$ < MAX_ITER and $\mathcal{L}(s^*, s_0, \gamma, u_0) > \eta$ and $\left\| \left.\frac{\partial \mathcal{L}}{\partial u_0}\right|_{u_0} \right\| > \varepsilon$ do
4:     Select some learning rate $\alpha$
5:     $u_0 \longleftarrow u_0 - \alpha \left.\frac{\partial \mathcal{L}(s^*, s_0, \gamma, u_0)}{\partial u_0}\right|_{u_0}$
6:     $u_0 \longleftarrow$ project($u_0$)
7:     $iter \longleftarrow iter + 1$
8: end while
```

4 RESULTS AND COMMENTS

All the previous steps were implemented in Python using TensorFlow to make a differentiable stable fluid solver. In this section, we discuss different results we obtained. We mostly worked on simple trajectories and shapes at first, and tried to complexify them over time.



(a) Trajectory 1 (b) Trajectory 2 (c) Trajectory 3

Fig. 8. Three examples of configuration we worked on. *From up to down*: configuration (the target density is drawn in yellow while the initial density is in green), simulation at frame 0, simulation at frame 40, simulation at frame 80.

4.1 Boundary Conditons

The equations presented in section 2 do not incorporate the consideration of boundary conditions. Those conditions must be fulfilled at each step of algorithm 1. In this section, we will discuss the specific boundary conditions that have been investigated and explored during this project.

4.1.1 Reflection. The first boundary condition we tried was to reflect the normal component of the velocity field at the borders of the grid, so that the fluid would be encapsulated inside four walls. This boundary condition was used with the centered grid because of its simplicity.

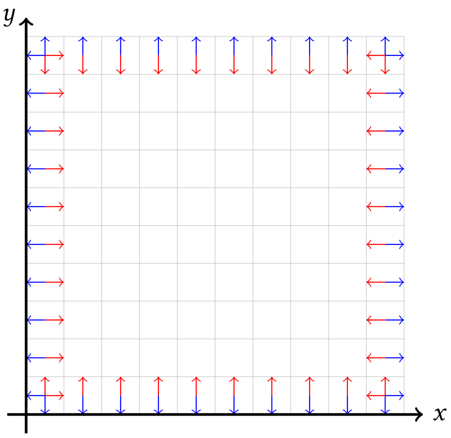


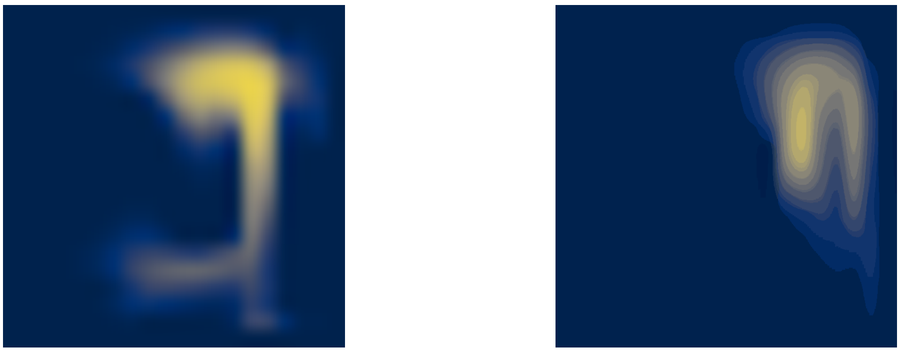
Fig. 9. The four walls of the reflexive boundaries reverse the normal component of the velocity field. This boundary condition changes the blue velocity field into the red one. For the corners, the value after the reflection is the average of its top/bottom and left/right neighbors.

4.1.2 Dirichlet boundary condition. Dirichlet boundary condition is basically setting the pressure q to zero at air (outside the borders of the grid). We chose to use Dirichlet boundary condition with free surface flow, i-e no obstacle but the fluid can be considered surrounded by air. For example, if there is air at cell $(i+1, j)$, then $q_{i+1j} = 0$ in equation (4).

4.1.3 Neumann boundary condition. Neumann boundary condition consists in setting $\frac{\partial q}{\partial x} = 0$ at the border of the grid (that may change equations (5) and (6)). It can be interpretated as an absence of influence from the air pressure on the fluid pressure. We chose to use this boundary condition combined with four walls that are surrounding the fluid. Therefore, if there is a wall at cell $(i+1, j)$, equation (4) becomes $w_{ij} = \frac{q_{i-1j} + q_{ij+1} + q_{ij-1} - 3q_{ij}}{h^2}$. Moreover, to take into account the presence of obstacle at the borders of the grid, we also used the no-slip condition that sets the velocity to zero at those borders.

4.2 Centered grid VS Staggered grid

The centered grid arrangement simplifies the interpolation of values within the simulation domain. However, it can introduce numerical instabilities when dealing with incompressible fluids due to the presence of the checkerboard pattern. On the other hand, the staggered grid overcomes this issue by staggering the fluid properties, with the velocity components defined at the face centers of the grid cells, while pressure is stored at the cell centers.



(a) Simulation in configuration 1 at frame 40, using centered grid

(b) Simulation in configuration 1 at frame 40, using staggered grid

Fig. 10. Centered grid *(left)* vs Staggered grid *(right)*. Due to numerical instabilities, a bottom leftover of density can appear when using centered grid, while the staggered grid reduces this *checkerboard pattern*.

The staggered grid arrangement naturally satisfies the divergence-free constraint and mitigates the checkerboard problem, leading to more stable simulations. Moreover, the Dirichlet and Neumann boundary conditions are easier to write. However, switching to staggered grid did not really solved the numerical dissipation issue.

4.3 Numerical dissipation

Because of the discretization of the fluid domain, there is some numerical dissipation that leads to a loss of density over time. Our guess is that this numerical dissipation sometimes misleads the optimisation process and makes it impossible to solve. One way to limit this misdirection could be to renormalize the density field at each iteration of the fluid solver. However, this would make the simulation losing its physical meaning, therefore we only renormalised the density when drawing the frame in the user interface, so that the numerical dissipation is visually weakened without interfering with the simulation.

5 USER-GUIDED INTERFACE

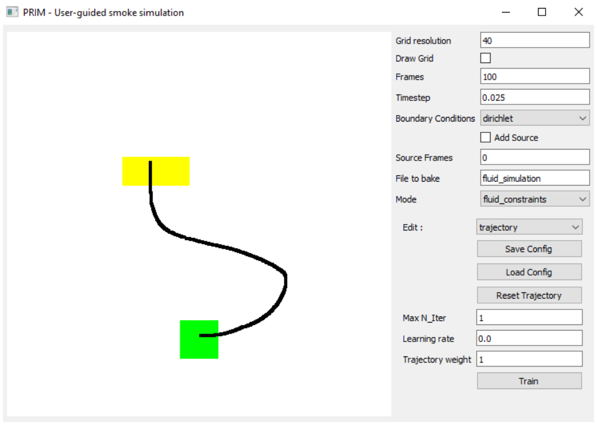
Once the training process was fully implemented, we created, using Python's `PyQt` module, a simple user interface to directly draw the constraints and perform the simulation.

5.1 Constraint layer

The constraint layer allows the user to specify the target density, the inital density and the trajectory to follow. It is also possible to adjust some of the simulation parameters and the optimization parameters.

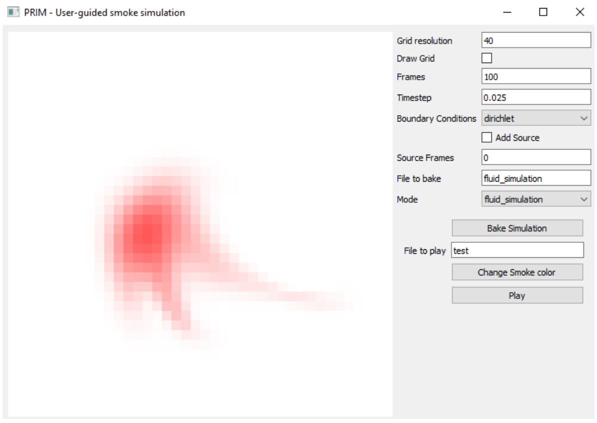
5.2 Simulation layer

The simulation layer is the part of the interface that bakes and draws the fluid simulation. The user can specify the same simulation parameters than in the constraint layer. There is also the possibility to change the color of the smoke.



Grid resolution	40
Draw Grid	☐
Frames	100
Timestep	0.025
Boundary Conditions	dirichlet
Add Source	☐
Source Frames	0
File to bake	fluid_simulation
Mode	fluid_constraints
Edit :	trajectory
Max N_Iter	1
Learning rate	0.0
Trajectory weight	1

Fig. 11. Screenshot of the constraint layer. Once the settings are all defined, the user can click on the `Train` button to launch the optimization (*training*) process. The result will be saved in a `.json` file that can be loaded and played in the simulation layer. It also possible for the user to import or export the configuration of the constraint.



Grid resolution	40
Draw Grid	☐
Frames	100
Timestep	0.025
Boundary Conditions	dirichlet
Add Source	☐
Source Frames	0
File to bake	fluid_simulation
Mode	fluid_simulation
File to play	test

Fig. 12. Screenshot of the simulation layer. Once the settings are all defined, the user can click on the `Bake Simulation` button to launch the simulation. The result will be saved in a `.json` file that can be loaded and played in the same layer right after.

6 DISCUSSIONS AND FUTURE WORK

The previous PRIM version of this project had one major drawback: the computation using TensorFlow was too slow and took about 30 minutes to simulate a 40×40 grid. Fully exploiting the computation graph with the decorator `@tf.function` and using LU-factorization to solve the Poisson equations dramatically reduced the time computation and made us overcome that drawback.

However, with a large resolution $n \times n$ (typically with $n > 128$), the solver and consequently the optimizer still has a long computation time. This is mostly due to the Laplacian matrix of size $n^2 \times n^2$, built to solve the Poisson equation. Compared to a `Numpy` and `Scipy` solver, it cannot being speed up using sparse representation of matrices because TensorFlow does not support linear solving using `SparseMatrix` structure yet. One temporary solution would be optimising on a low resolution and then exporting and extapoling the optimised velocity field to a high resolution solver.

Although the tool works well for simple trajectory like in figure 8, when user draws more complex trajectory like in figure 11, the optimizer is not guaranteed to find a solution. There are several possibilities that may help to overcome this issue. Currently, the velocity constraint linearly samples the cells to define $\mathcal{L}_{velocity}$, but it could be interesting to see the result by changing the loss function with a sampling of cells depending on the curvature of the trajectory and more generally with a non-linear sampling $\gamma(t) \longmapsto \gamma(f(t))$ where f is non-linear. Another approach could be limiting numerical dissipation by using a PIC/FLIP approach to store the density in particles or by using image sharpening. Finally, one can imagine a divide and conquer approach by cutting the trajectory to intermediate trajectories and carrying the density on those intermediate trajectories. The difficulty would be to make a continuous fluid motion with all those intermediate trajectories.

7 CONCLUSION

In conclusion, this project has deepened our understanding of fluid dynamics and has provided us with valuable experience with the TensorFlow library. We managed to built a user-friendly interface that allows to perform basic fluid simulation and optimisation even for someone with little knowledge of fluid dynamics. While some of our results seems to reach some state-of-art-level production with only optimizing the initial velocity field, there is still much work and more complex approaches to explore in order to provide accurate complex physical simulation that can be controlled by an artist.

ACKNOWLEDGMENTS

The author would like to thank Amal Dev Parakkat and Kiwon Um of Télécom Paris as well as Damien Rohmer of École polytechnique 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

Philipp Holl, Vladlen Koltun, and Nils Thuerey. 2020. Learning to Control PDEs with Differentiable Physics. (2020). arXiv:2001.07457

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

Jun Xing, Rubaiat Kazi, Tovi Grossman, Li-Yi Wei, Jos Stam, and George Fitzmaurice. 2016. Energy-Brushes: Interactive Tools for Illustrating Stylized Elemental Dynamics. 755–766. DOI:https://doi.org/10.1145/2984511.2984585

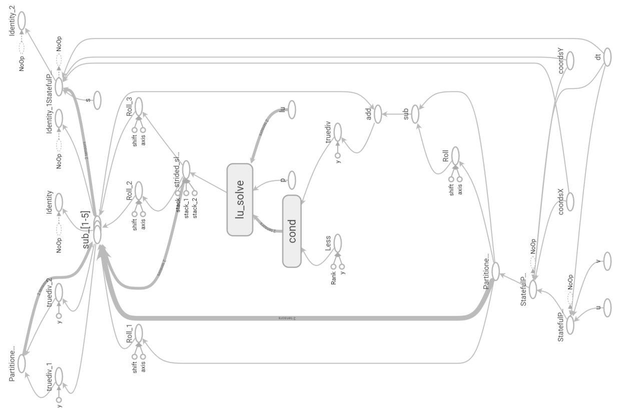
A APPENDIX

A.1 Notations

Our solver is currently in 2D, so if nothing is mentioned, all the specified mathematical 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

A.2 Computational graph of the solver



Identity_2
NoOp
Identity_1
StatefulP...
Identity
sub_[1-5]
truediv_2
truediv_1
Partitione...
Roll_3
Roll_2
Roll_1
shift
axis
strided_sl...
stack
stack_1
stack_2
lu_solve
cond
truediv
Less
Rank
add
sub
Roll
Partitione...
StatefulP...
coordsY
coordsX
dt