# Eulerian Fluid Simulation

## GAMES 201 Lecture 4

Yuanming Hu

MIT CSAIL

June 22, 2020

## Today's topic: Eulerian fluid simulation

- Eulerian representation uses still sensors in space, usually arranged in a regular grid/triangular mesh.
- A little bit of math - but not too much.
- This course: intuitive derivation - instead of finite volume/finite difference.

### Recommended book

A great introduction to Eulerian fluid simulation:
**Fluid simulation for computer graphics**[1] by *Robert Bridson*.

[1]R. Bridson (2015). *Fluid simulation for computer graphics*. CRC press.

## Table of Contents

## Material Derivatives: Lagrangian v.s. Eulerian

$$\frac{\mathrm{D}}{\mathrm{D}t} := \frac{\partial}{\partial t} + \mathbf{u} \cdot \nabla$$

E.g.,

$$\frac{\mathrm{D}T}{\mathrm{D}t} = \frac{\partial T}{\partial t} + \mathbf{u} \cdot \nabla T$$

$$\frac{\mathrm{D}\mathbf{u}_x}{\mathrm{D}t} = \frac{\partial \mathbf{u}_x}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u}_x$$

$\mathbf{u}$: material (fluid) velocity. Many other names: Advective/Lagrangian/particle derivative.

Intuitively, change of physical quantity on a piece of material =

1. change due to time $\frac{\partial}{\partial t}$ (Eulerian).
2. change due to material movement $\mathbf{u} \cdot \nabla$.

## (Incompressible) Navier–Stokes equations

$$\rho \frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = -\nabla p + \mu \nabla^2 \mathbf{u} + \rho \mathbf{g}, \nabla \cdot \mathbf{u} = 0$$

**or**

$$\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{g}, \nabla \cdot \mathbf{u} = 0$$

$\mu$: dynamic viscosity; $\nu = \frac{\mu}{\rho}$: kinematic viscosity.

### Variants of the N-S equations

The Navier-Stokes equations have many variants. Here we show a version that is the most friendly to fluids simulation in computer graphics. In graphics we usually drop **viscosity** except for highly viscous materials (e.g., honey).

## Operator splitting [More details]

$$\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = -\frac{1}{\rho}\nabla p + \mathbf{g} \qquad \nabla \cdot \mathbf{u} = 0$$

Split the equations above into three parts:

$$\begin{aligned}
\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} &= \mathbf{0}, \quad \frac{\mathrm{D}\alpha}{\mathrm{D}t} = \mathbf{0} \quad \textbf{(advection)} && (1)\\
\frac{\partial \mathbf{u}}{\partial t} &= \mathbf{g} \quad \text{(external forces, optional)} && (2)\\
\frac{\partial \mathbf{u}}{\partial t} &= -\frac{1}{\rho}\nabla p \quad \text{s.t.} \quad \nabla \cdot \mathbf{u} = \mathbf{0} \quad \textbf{(projection)} && (3)
\end{aligned}$$

$\alpha$: any physical property (temperature, color, smoke density etc.)

## Eulerian fluid simulation cycle

Time discretization with splitting: for each time step,

1. Advection: "move" the fluid field. Solve $\mathbf{u}^*$ using $\mathbf{u}^t$

$$\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = \mathbf{0}, \quad \frac{\mathrm{D}\alpha}{\mathrm{D}t} = \mathbf{0}$$

2. External forces (optional): evaluate $\mathbf{u}^{**}$ using $\mathbf{u}^*$

$$\frac{\partial \mathbf{u}}{\partial t} = \mathbf{g} \quad \text{(external forces, optional)}$$

3. Projection: make velocity field $\mathbf{u}^{t+1}$ divergence-free based on $\mathbf{u}^{**}$

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{1}{\rho}\nabla p \quad \text{s.t.} \quad \nabla \cdot \mathbf{u}^{t+1} = \mathbf{0}$$

# Table of Contents

## Spatial discretization using *cell-centered* grids

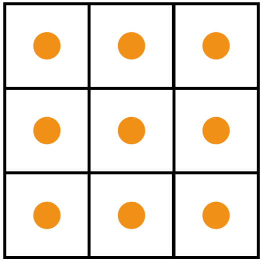


Figure: $\mathbf{u}_x$, $\mathbf{u}_y$, $p$ are all stored at the center (orange) of cells.

```
n, m = 3, 3
u = ti.var(ti.f32, shape=(n, m)) # x-component of velocity
v = ti.var(ti.f32, shape=(n, m)) # y-component of velocity
p = ti.var(ti.f32, shape=(n, m)) # pressure
```

## Spatial discretization using *staggered* grids

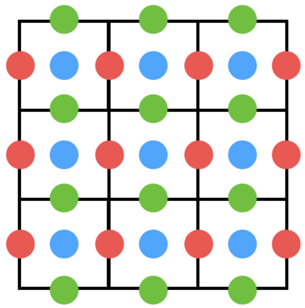


Figure: Red: $\mathbf{u}_x$; Green: $\mathbf{u}_y$; Blue: $p$.

```
n, m = 3, 3
u = ti.var(ti.f32, shape=(n+1, m)) # x-component of velocity
v = ti.var(ti.f32, shape=(n, m+1)) # y-component of velocity
p = ti.var(ti.f32, shape=(n, m)) # pressure
```

## Bilinear interpolation

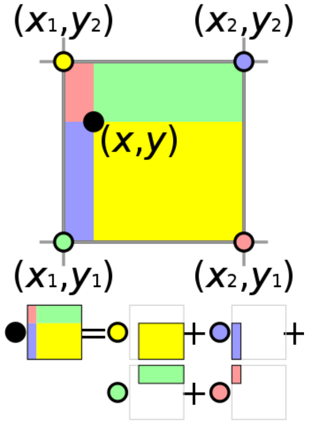


Figure: Bilinear interpolation: value at $(x, y)$ is a weighted average of the four corners. Source: Wikepedia

## Table of Contents

## Advection schemes

A trade-off between numerical viscosity, stability, performance and complexity:

- Semi-Lagrangian advection[2]
- MacCormack/BFECC[3]
- "BiMocq$^2$"[4]
- Particle advection (PIC/FLIP/APIC/PolyPIC, later in this course)
- ...

[2] R. Courant, E. Isaacson, and M. Rees (1952). "On the solution of nonlinear hyperbolic differential equations by finite differences". In: *Communications on pure and applied mathematics* 5.3, pp. 243–255; J. Stam (1999). "Stable fluids". In: *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pp. 121–128.

[3] B. Kim et al. (2005). *Flowfixer: Using BFECC for fluid simulation*. Tech. rep. Georgia Institute of Technology; A. Selle et al. (2008). "An unconditionally stable MacCormack method". In: *Journal of Scientific Computing* 35.2-3, pp. 350–371.

[4] Z. Qu et al. (2019). "Efficient and conservative fluids using bidirectional mapping". In: *ACM Transactions on Graphics (TOG)* 38.4, pp. 1–12.

# Semi-Lagrangian advection

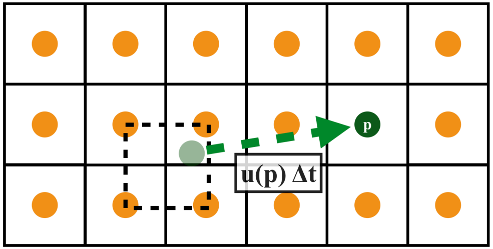


Figure: What should be the field value at $\mathbf{p}$ now based on the field and velocity at the previous time step? Well, just let reverse the simulation...

```
@ti.func
def semi_lagrangian(x, new_x, dt):
    for I in ti.grouped(x):
        new_x[I] = sample_bilinear(x, backtrace(I, dt))
```

## What if ...

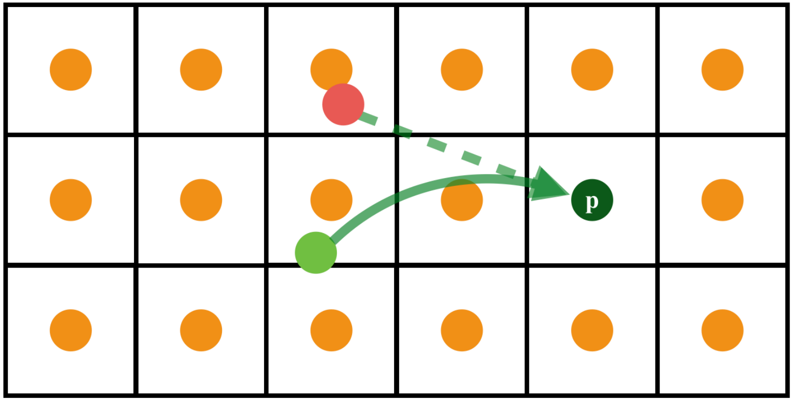


Figure: The real trajectory of material parcels can be complex... Red: a naive estimation of last position; Light gree: the true previous position.

## Going back in time (Demo)

Initial value problem (ODE): simply use explicit time integration schemes, e.g.,

- Forward Euler (“RK1”)

```
p -= dt * velocity(p)
```

- Explicit Midpoint (“RK2”)

```
p_mid = p - 0.5 * dt * velocity(p)
p -= dt * velocity(p_mid)
```

- RK3

```
v1 = velocity(p)
p1 = p - 0.5 * dt * v1
v2 = velocity(p1)
p2 = p - 0.75 * dt * v2
v3 = velocity(p2)
p -= dt * (2 / 9 * v1 + 1 / 3 * v2 + 4 / 9 * v3)
```

## BFECC and MacCormack advection schemes

BFECC: Back and Forth Error Compensation and Correction

- $\mathbf{x}^* = \mathsf{SL}(\mathbf{x}, \Delta t)$
- $\mathbf{x}^{**} = \mathsf{SL}(\mathbf{x}^*, -\Delta t)$
- Estimate the error $\mathbf{x}^{\text{error}} = \frac{1}{2}(\mathbf{x}^{**} - \mathbf{x})$
- Apply the error $x^{\text{final}} = \mathbf{x}^* + \mathbf{x}^{\text{error}}$

Be careful: need to prevent overshooting.

**Demo!**

```
@ti.func
def maccormack(x, dt):
    semi_lagrangian(x, new_x, dt)
    semi_lagrangian(new_x, new_x_aux, -dt)

    for I in ti.grouped(x):
        new_x[I] = new_x[I] + 0.5 * (x[I] - new_x_aux[I])
```

## Table of Contents

## Chorin-style projection

How to ensure the velocity field is divergence free after projection?

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{1}{\rho}\nabla p \quad \text{s.t.} \quad \nabla \cdot \mathbf{u} = \mathbf{0} \quad \textbf{(projection)}$$

Expand (using finite difference in time):

$$\mathbf{u}^* - \mathbf{u} = -\frac{\Delta t}{\rho}\nabla p \quad \text{s.t.} \quad \nabla \cdot \mathbf{u}^* = 0 \tag{4}$$

$$\mathbf{u}^* = \mathbf{u} - \frac{\Delta t}{\rho}\nabla p \quad \text{s.t.} \quad \nabla \cdot \mathbf{u}^* = 0 \tag{5}$$

$$\nabla \cdot \mathbf{u}^* = \nabla \cdot (\mathbf{u} - \frac{\Delta t}{\rho}\nabla p) \tag{6}$$

$$0 = \nabla \cdot \mathbf{u} - \frac{\Delta t}{\rho}\nabla \cdot \nabla p \tag{7}$$

$$\nabla \cdot \nabla p = \frac{\rho}{\Delta t}\nabla \cdot \mathbf{u} \tag{8}$$

## Poisson's equation

(From the previous slide)

$$\nabla \cdot \nabla p = \frac{\rho}{\Delta t} \nabla \cdot \mathbf{u} \tag{9}$$

... which is Poisson's equation

$$\nabla \cdot \nabla p = f \quad \textbf{or} \quad \Delta p = f. \tag{10}$$

$\Delta = \nabla^2 = \nabla \cdot \nabla$ is the **Laplace operator**. If $f = 0$, the equation is called **Laplace's equation**.

## Spatial discretization (2D)

Recall the equation for $p$:

$$\nabla \cdot \nabla p = \frac{\rho}{\Delta t} \nabla \cdot \mathbf{u} \tag{11}$$

Discretize on a 2D grid:

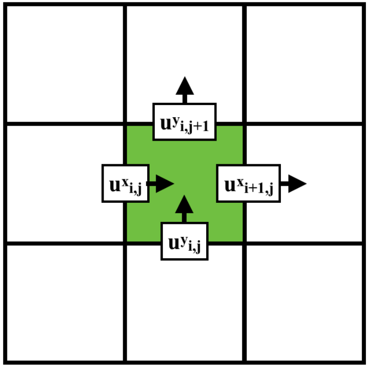
$$\begin{aligned} (\mathbf{A}\mathbf{p})_{i,j} &= (\nabla \cdot \nabla p)_{i,j} = \frac{1}{\Delta x^2}(-4p_{i,j} + p_{i+1,j} + p_{i-1,j} + p_{i,j-1} + p_{i,j+1}) && (12) \\ \mathbf{b}_{i,j} &= \left(\frac{\rho}{\Delta t}\nabla \cdot \mathbf{u}\right)_{i,j} = \frac{\rho}{\Delta t \Delta x}(\mathbf{u}^x_{i+1,j} - \mathbf{u}^x_{i,j} + \mathbf{u}^y_{i,j+1} - \mathbf{u}^y_{i,j}) && (13) \end{aligned}$$

Again, a linear system:

$$\mathbf{A}_{nm \times nm}\mathbf{p}_{nm} = \mathbf{b}_{nm}$$

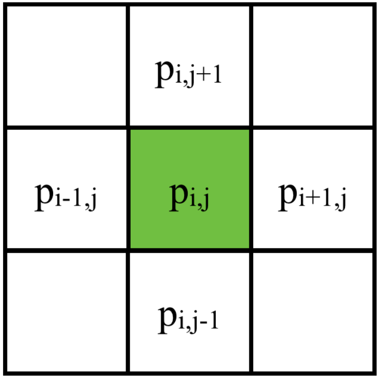
$n, m$: numbers of cells along the $x-$ and $y-$axis.

## Spatial discretization (2D): $\nabla \cdot \mathbf{u}$



$$(\nabla \cdot \mathbf{u})_{i,j} = \frac{1}{\Delta x}(\mathbf{u}^x_{i+1,j} - \mathbf{u}^x_{i,j} + \mathbf{u}^y_{i,j+1} - \mathbf{u}^y_{i,j})$$

## Spatial discretization (2D): $\nabla \cdot \nabla p$



$$(\nabla \cdot \nabla p)_{i,j} = \frac{1}{\Delta x^2}(-4p_{i,j} + p_{i+1,j} + p_{i-1,j} + p_{i,j-1} + p_{i,j+1})$$

Question: How to handle Dirichlet and Neumann boundaries?

## Table of Contents

## The top 10 algorithms from the $20^{\text{th}}$ century

- 1946: The Metropolis Algorithm for Monte Carlo.
- 1947: Simplex Method for Linear Programming.
- 1950: Krylov Subspace Iteration Method.
- 1951: The Decompositional Approach to Matrix Computations.
- 1957: The Fortran Optimizing Compiler.
- 1959: QR Algorithm for Computing Eigenvalues.
- 1962: Quicksort Algorithms for Sorting.
- 1965: Fast Fourier Transform.
- 1977: Integer Relation Detection.
- 1987: Fast Multipole Method.

## The top 10 algorithms from the $20^{\text{th}}$ century

- 1946: The Metropolis Algorithm for Monte Carlo.
- 1947: Simplex Method for Linear Programming.
- **1950: Krylov Subspace Iteration Method.**
- **1951: The Decompositional Approach to Matrix Computations.**
- 1957: The Fortran Optimizing Compiler.
- 1959: QR Algorithm for Computing Eigenvalues.
- 1962: Quicksort Algorithms for Sorting.
- **1965: Fast Fourier Transform.**
- 1977: Integer Relation Detection.
- **1987: Fast Multipole Method.**

## Solving large-scale linear systems

Many physics engines boil down to a (huge) linear system solve:

$$\mathbf{A}\mathbf{x} = \mathbf{b}$$

How to solve it:

- Direct solvers (e.g., PARDISO)
- Iterative solvers:
  - Gauss-Seidel
  - (Damped) Jacobi
  - (Preconditioned) Krylov-subspace solvers (e.g., conjugate gradients)

Good numeric solvers are usually **composed** of different solvers: e.g., *multigrid-preconditioned conjugate gradients with damped Jacobi smoothing and PARDISO at the bottom multigrid level.*

## Matrix storage

What's special about $\mathbf{A}$: often sparse, symmetric & positive-definite (SPD).

How to store $\mathbf{A}$? Options:

1. As a dense matrix (e.g., `float A[1024][1024]` doesn't scale but works)
2. As a sparse matrix (various sparse matrix formats: CSR, COO, ...)
3. Don't store it at all (aka. **Matrix-free**, often the ultimate solution...)

Modern computer architecture: memory bandwidth is expensive but FLOPs are free. So compute matrix entries on-the-fly (instead of fetching values from memory) can sometimes be good to performance.

## Krylov-subspace solvers

Krylov-subspace solvers are among most efficient linear system solvers. The most well-known version: **conjugate gradients (CG)**.
Less frequently used (in graphics):

- Conjugate residuals (CR)
- Generalized minimal residual method (GMRES)
- Biconjugate gradient stabilized (BiCGStab)
- ...

**Recommended book**

An Introduction to the Conjugate Gradient Method Without the Agonizing Pain[5] by Jonathan Richard Shewchuk.

[5] J. R. Shewchuk et al. (1994). *An introduction to the conjugate gradient method without the agonizing pain.*

## Conjugate gradients

```
$\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$
$\mathbf{p}_0 = \mathbf{r}_0$
$k = 0$
while True:
    $\alpha_k = \frac{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}{\mathbf{p}_k^\mathsf{T}\mathbf{A}\mathbf{p}_k}$
    $\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k\mathbf{p}_k$
    $\mathbf{r}_{k+1} = \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$
    if $||\mathbf{r}_{k+1}||$ is sufficiently small, break
    $\beta_k = \frac{\mathbf{r}_{k+1}^\mathsf{T}\mathbf{r}_{k+1}}{\mathbf{r}_k^\mathsf{T}\mathbf{r}_k}$
    $\mathbf{p}_{k+1} = \mathbf{r}_{k+1} + \beta_k\mathbf{p}_k$
    $k = k + 1$
return $\mathbf{x}_{k+1}$
```

### Residual v.s. Error

Small residual $\mathbf{r}$ does not mean small error $\mathbf{e}$, especially when $\mathbf{A}$ is poorly conditioned (i.e., with a huge condition number (next slide)).

## Eigenvalues and condition numbers

Recall that if

$$\mathbf{A}\mathbf{x} = \lambda \mathbf{x},$$

then $\lambda$ is an **eigenvalue** of $\mathbf{A}$ and $\mathbf{x}$ is an **eigenvector** of $\mathbf{A}$.

The **condition number** $\kappa$ of SPD matrix $\mathbf{A}$:

$$\kappa(\mathbf{A}) = \lambda_{\textsf{max}} / \lambda_{\textsf{min}}$$

In general: a smaller condition numbers means faster convergence.
(Note that condition numbers have many different definitions.)

## Iterative solver trick: Warm starting

If you start with an initial guess that is close to the solution, very likely fewer iterations are needed.
"Warm starting": use the $\mathbf{p}$ from the last frame as the initial guess of the current frame.

Online demo

In practice works well for (damped) Jacobi/Gauss-Seidel/CG, but for MGPCG (later in this lecture) it doesn't work well.

## Preconditioning

Find an approximate operator $\mathbf{M}$ that is close to $\mathbf{A}$ but easier to invert. Then,

$$\mathbf{Ax} = \mathbf{b} \iff \mathbf{M}^{-1}\mathbf{Ax} = \mathbf{M}^{-1}\mathbf{b}$$

**Intuition:** $\mathbf{M}^{-1}\mathbf{A}$ may have a smaller condition number (closer to identity) or better eigenvalue clustering than $\mathbf{A}$ itself.

**Question:** why not directly let $\mathbf{M} = \mathbf{A}$?

## Common preconditioners

- Jacobi (diagonal) preconditioner $\mathbf{M} = \mathbf{diag}(\mathbf{A})$
- Poisson preconditioner
- (Incomplete) Cholesky decomposition
- Multigrid: $\mathbf{M}$ = very complex linear operator that almost inverts $\mathbf{A}$...
  - Geometric multigrid
  - Algebraic multigrid
- Fast multipole method (FMM)

# (Geometric) Multigrid methods

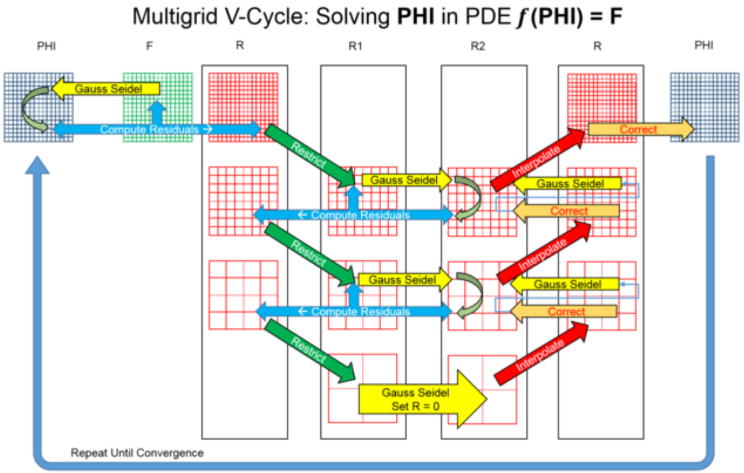


Figure: Multigrid V-cycle (source: Wikipedia)

## The Multigrid design space

1. Restriction/prolongation
2. Cycle (**V**/W/F cycle)
3. Smoothers ((red-black) Gauss-Seidel, Jacobi, damped Jacobi, etc.)
4. Number of levels (e.g., coarsen until the bottom level has $<$ 50K voxels)
5. Bottom level solver (Brute-force Jacobi or direct solvers)
6. Number of pre/post iterations (usually, 2-5)
7. Coarsening and boundary handling (e.g., Galerkin coarsening, semi-algebraic multigrid)
8. ...

## Multigrid preconditioned conjugate gradients (MGPCG)

When solving Poisson's equation in graphics, people ususally use geometric multigrid as the preconditioner for conjugate gradients.

### Recommended reading

If you want to learn more about multigrid (and linear solvers in general):

- **A multigrid tutorial**[6].
- A seminal and easy-to-understand multigrid paper in graphics: **A parallel multigrid Poisson solver for fluids simulation on large grids**[7]

**Taichi demo: 2D/3D multigrd:** `ti example mgpcg_advanced`

[6] W. L. Briggs, V. E. Henson, and S. F. McCormick (2000). *A multigrid tutorial*. SIAM.

[7] A. McAdams, E. Sifakis, and J. Teran (2010). "A Parallel Multigrid Poisson Solver for Fluids Simulation on Large Grids.". In: *Symposium on Computer Animation*, pp. 65–73.

## Summary: Eulerian fluid simulation in graphics

For each time step,

- Advection: "move" the fluid field. Solve for $\mathbf{u}^*$ using $\mathbf{u}^t$

$$\frac{\mathrm{D}\mathbf{u}}{\mathrm{D}t} = \mathbf{0}, \quad \frac{\mathrm{D}\alpha}{\mathrm{D}t} = \mathbf{0}$$

  **Key**: Use a advection scheme with low numerical viscosity (e.g., MacCormack/BFECC/Particle advection)

- Projection: make velocity field $\mathbf{u}^{t+1}$ divergence-free based on $\mathbf{u}^*$

$$\frac{\partial \mathbf{u}}{\partial t} = -\frac{1}{\rho}\nabla p \quad \text{s.t.} \quad \nabla \cdot \mathbf{u}^{t+1} = \mathbf{0}$$

  **Key**: Use a fast linear solver (e.g., MGPCG).

# Combining advection with reflection: IVOCK

## SIGGRAPH 2015

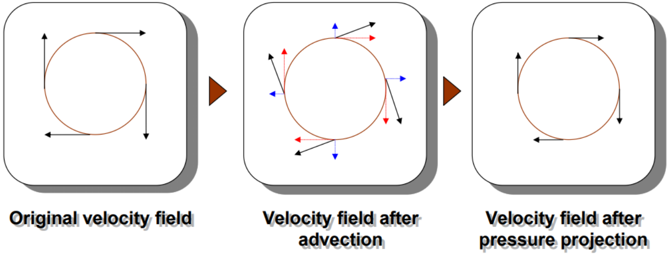

Original velocity field
Velocity field after advection
Velocity field after pressure projection

Figure: IVOCK: Restoring Missing Vortices in Advection-Projection Fluid Solvers[8]

[8]X. Zhang, R. Bridson, and C. Greif (2015). "Restoring the missing vorticity in advection-projection fluid solvers". In: *ACM Transactions on Graphics (TOG)* 34.4, pp. 1–8.

## Advection-Reflection solver

SIGGRAPH 2018

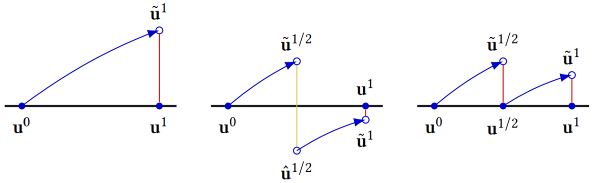


Fig. 2. A geometric interpretation of our method. *Left:* In a standard advection-projection solver, projection to the divergence-free subspace causes kinetic energy loss (red). *Middle:* Our reflection solver uses an energy-preserving *reflection* (yellow) halfway through the advection step, dramatically reducing the energy loss caused by the final projection. Our method has effectively identical computational cost to an advection-projection solver with half the time step *(right)*, but loses less energy.

Figure: Source: An Advection-Reflection Solver for Detail-Preserving Fluid Simulation[9]

[9] J. Zehnder, R. Narain, and B. Thomaszewski (2018). "An advection-reflection solver for detail-preserving fluid simulation". In: *ACM Transactions on Graphics (TOG)* 37.4, pp. 1–8.

## Possible extensions

- Going to 3D
- Accurate boundary conditions and fluid-solid coupling[10]
- Two phase fluid simulation[11]
- Handling free surfaces (level sets)[12]
- Vortex methods[13]

A well-implemented Eulerian fluid solver counts as Homework 1 :-)

[10] C. Batty, F. Bertails, and R. Bridson (2007). "A fast variational framework for accurate solid-fluid coupling". In: *ACM Transactions on Graphics (TOG)* 26.3, 100–es.

[11] R. Ando, N. Thuerey, and C. Wojtan (2015). "A stream function solver for liquid simulations". In: *ACM Transactions on Graphics (TOG)* 34.4, pp. 1–9.

[12] S. Osher, R. Fedkiw, and K Piechor (2004). "Level set methods and dynamic implicit surfaces". In: *Appl. Mech. Rev.* 57.3, B15–B15.

[13] X. Zhang and R. Bridson (2014). "A PPPM fast summation method for fluids and beyond". In: *ACM Transactions on Graphics (TOG)* 33.6, pp. 1–11.