FLUID SIMULATION

Kristofer Schlachter

The Equations

- Incompressible Navier-Stokes:

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

  - "Incompressibility condition"

$$\nabla \cdot u = 0$$

Breakdown

$\frac{\partial u}{\partial t}$	**The derivative of velocity with respect to time**. Calculated at each grid point each time step.
$-(\nabla \cdot u)u$	**The convection term**. This is the self advection term. In the code we will use the backward particle trace for this term.
$v\nabla^2 u$ or $v\nabla \cdot \nabla u$	**The viscosity term**. We are actually going to ignore this term. When you do that you are actually using the Euler Equations.
F	**External force**. Any external forces including gravity.

Breakdown

$-\frac{1}{\rho}\nabla p$

The pressure term. ρ is the density of the fluid [a] and p is the pressure. p is whatever it takes to make the velocity field divergence free. The simulator will solve for a pressure that makes our fluid incompressible at each time step.

---

[a] density of water $\rho \approx 1000 \frac{kg}{m^2}$

Incompressible Euler Equations

If you drop the viscosity term from the incompressible Navier Stokes equations we get:

$$\frac{\partial u}{\partial t} + (\nabla \cdot u)u + \frac{1}{\rho}\nabla p = F$$

$$\nabla \cdot u = 0$$

Such an ideal fluid with no viscosity is called *inviscid*. These are the equations we are going to use.

How do we discretize the equations?

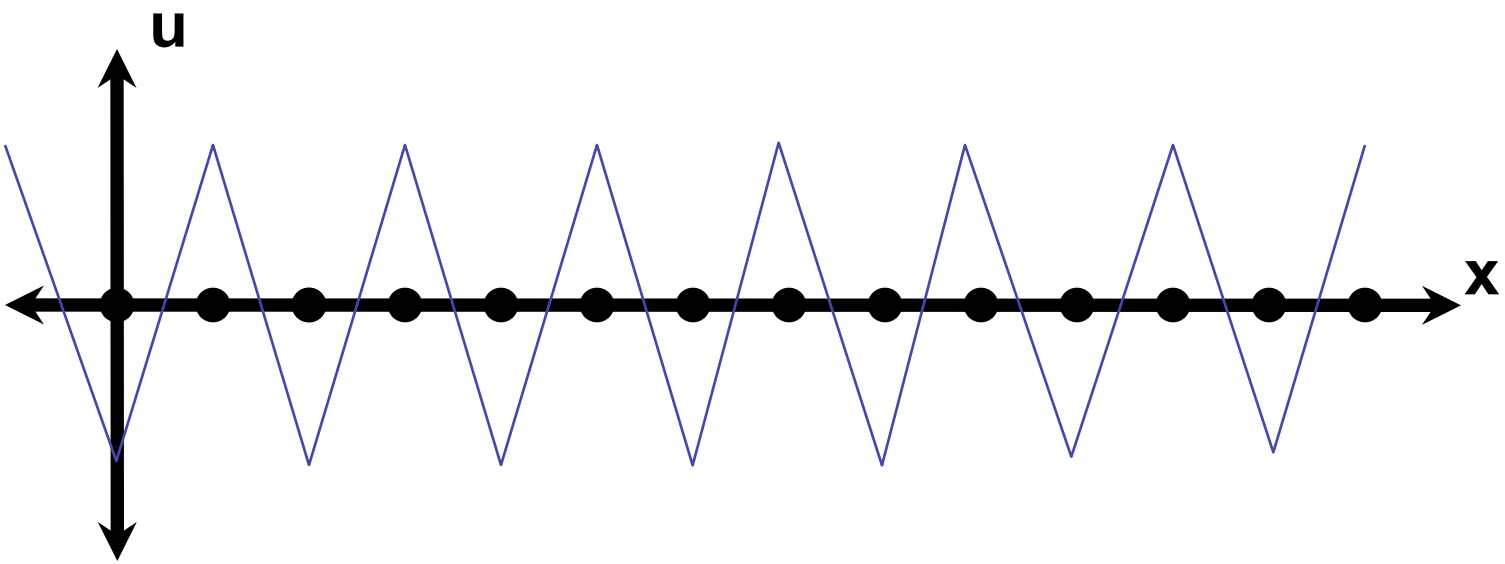
A Simple Grid

- We could put all our fluid variables at the nodes of a regular grid
- But this causes some major problems
- In 1D: incompressibility means $\frac{\partial u}{\partial x} = 0$
- Approximate at a grid point: $\frac{u_{i+1} - u_{i-1}}{2\Delta x} = 0$
- Note the velocity at the grid point isn't involved!

[Bridson 07]

A Simple Grid Disaster

- The only solutions to $\frac{\partial u}{\partial x} = 0$ are u=constant
- But our numerical version has other solutions:



[Bridson 07]

Staggered Grids

- Problem is solved if we don't skip over grid points
- To make it unbiased, we **stagger** the grid: put velocities halfway between grid points
- In 1D, we estimate divergence at a grid point as:

$$\frac{\partial u}{\partial x}(x_i) \approx \frac{u_{i+\frac{1}{2}} - u_{i-\frac{1}{2}}}{\Delta x}$$

- Problem solved!

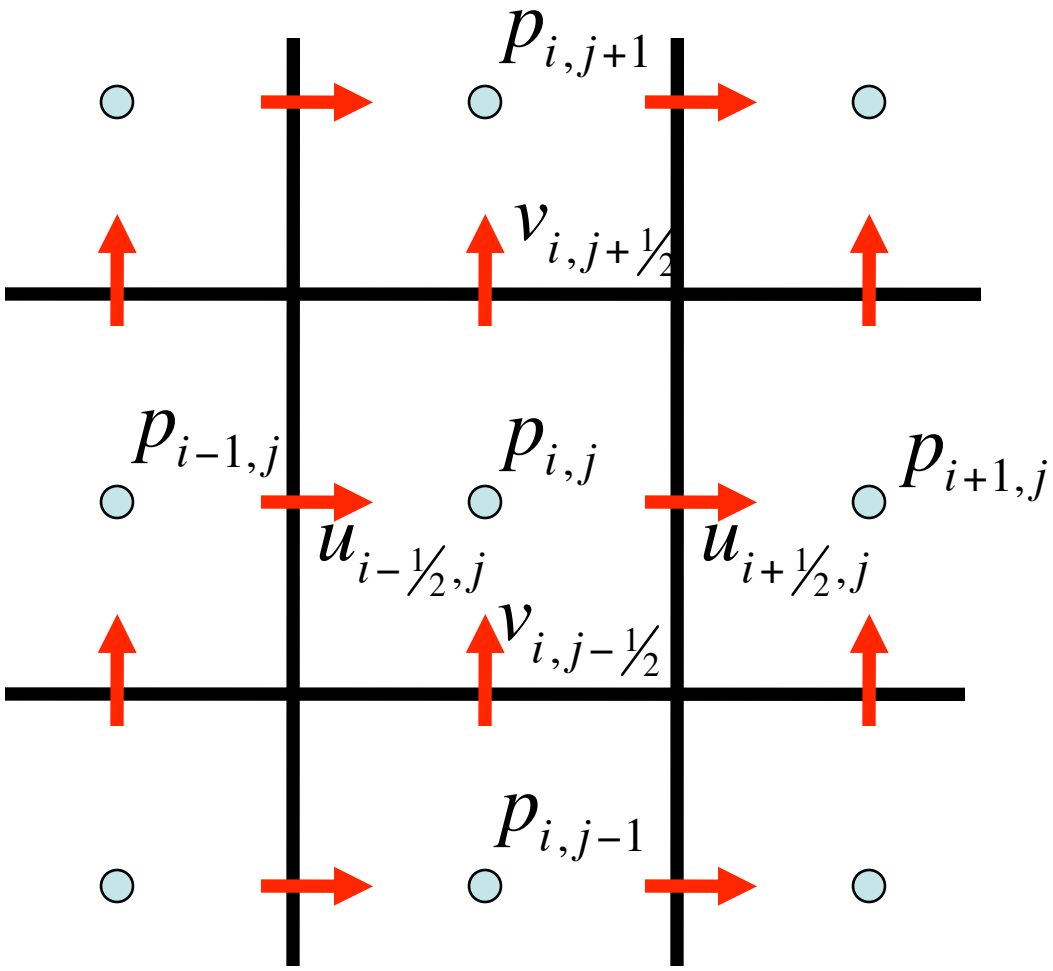
[Bridson 07]

The MAC Grid

- From the Marker-and-Cell (MAC) method [Harlow&Welch'65]
- A particular staggering of variables in 2D/3D that works well for incompressible fluids:
  - Grid cell (i,j,k) has pressure $\mathbf{p}_{i,j,k}$ at its center
  - x-part of velocity $\mathbf{u}_{i+1/2,jk}$ in middle of x-face between grid cells (i,j,k) and (i+1,j,k)
  - y-part of velocity $\mathbf{v}_{i,j+1/2,k}$ in middle of y-face
  - z-part of velocity $\mathbf{w}_{i,j,k+1/2}$ in middle of z-face

[Bridson 07]

MAC Grid in 2D



[Bridson 07]

Math on a MAC Grid

The partial derivative (∂).

$$\frac{\partial f(x,y,z)}{\partial y} = f(x,y+1,z) - f(x,y,z)$$

The gradient operator (∇)

$$\nabla f(x,y,z) = \begin{pmatrix} f(x+1,y,z) - f(x,y,z), \\ f(x,y+1,z) - f(x,y,z), \\ f(x,y,z+1) - f(x,y,z) \end{pmatrix}$$

Math on a MAC Grid

The divergence of a vector field $(\nabla \cdot)$

$$\begin{aligned}\nabla \cdot u(x,y,z) =& (u_x(x+1,y,z) - u_x(x,y,z)) + \\ & (u_y(x,y+1,z) - u_y(x,y,z)) + \\ & (u_z(x,y,z+1) - u_z(x,y,z))\end{aligned}$$

The Laplacian operator (∇^2) or $(\nabla \cdot \nabla)$

$$\begin{aligned}\nabla^2 f(x,y,z) =& f(x+1,y,z) + f(x-1,y,z) + \\ & f(x,y+1,z) + f(x,y-1,z) + \\ & f(x,y,z+1) + f(x,y,z-1) - 6f(x,y,z)\end{aligned}$$

**There might be a one over h squared term missing*

Simulation

- Set $u^A = \texttt{advect}(u^n, \Delta t, u^n)$
- Add $u^B = u^A + \Delta t F$
- Set $u^{n+1} = \texttt{project}(\Delta t, u^B)$

*I am ignoring choosing a time step in this presentation

Advection

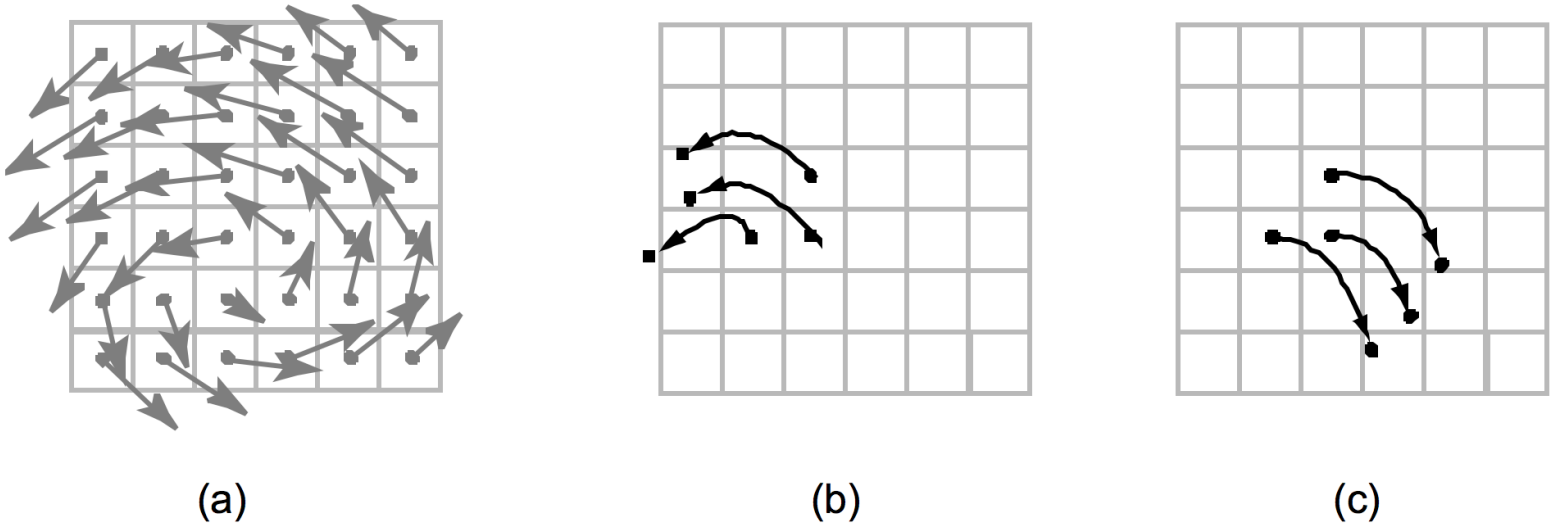


Figure 3: *Basic idea behind the advection step. Instead of moving the cell centers forward in time (b) through the velocity field shown in (a), we look for the particles which end up exactly at the cell centers by tracing backwards in time from the cell centers (c).*
[Stam 2003]

Advection

$$x_{mid} = x_G - \frac{1}{2}\Delta t u\left(x_G\right)$$

$$x_p = x_G - \Delta t u\left(x_{mid}\right)$$

$$q_G^{n+1} = \texttt{interpolate}\left(q^n, x_p\right)$$

```
// Trace a particle from point (x, y, z) for t time using RK2.
Point traceParticle(float x, float y, float z, float t)
    Vector V = getVelocity(x, y, z);
    V = getVelocity(x+0.5*t*V.x,  y+0.5*t*V.y,  z+0.5*t*V.z);
    return Point(x, y, z) + t*V;

// Get the interpolated velocity at a point in space.
Vector getVelocity(float x, float y, float z)
    Vector V;
    V.x = getInterpolatedValue(x/h, y/h-0.5, z/h-0.5, 0);
    V.y = getInterpolatedValue(x/h-0.5, y/h, z/h-0.5, 1);
    V.z = getInterpolatedValue(x/h-0.5, y/h-0.5, z/h, 2);
    return V;

// Get an interpolated data value from the grid.
float getInterpolatedValue(float x, float y, float z, int index)
    int i = floor(x);
    int j = floor(y);
    int k = floor(z);
    return (i+1-x) * (j+1-y) * (k+1-z) * cell(i, j, k).u[index] +
           (x-i) * (j+1-y) * (k+1-z) * cell(i+1, j, k).u[index] +
           (i+1-x) * (y-j) * (k+1-z) * cell(i, j+1, k).u[index] +
           (x-i) * (y-j) * (k+1-z) * cell(i+1, j+1, k).u[index] +
           (i+1-x) * (j+1-y) * (z-k) * cell(i, j, k+1).u[index] +
           (x-i) * (j+1-y) * (z-k) * cell(i+1, j, k+1).u[index] +
           (i+1-x) * (y-j) * (z-k) * cell(i, j+1, k+1).u[index] +
           (x-i) * (y-j) * (z-k) * cell(i+1, j+1, k+1).u[index];
```

[David Cline]

Projection

The `project`$(\Delta t, u)$ routine does the following:

- Calculate the negative divergence b (the right-hand side)
- Set the entries of A
- If using CG - Construct the MIC(0) preconditioner.
- Solve $Ap = b$ with a linear solver. If using CG then solve with MICCG(0), i.e., the PCG algorithm with MIC(0) as preconditioner.
- Compute the new velocities n^{n+1}according to the pressure-gradient update to u.

Projection

Setting up the divergence vector *b* (the right hand side)

$$\begin{aligned}\nabla \cdot u(x,y,z) =& (u_x(x+1,y,z) - u_x(x,y,z)) + \\ & (u_y(x,y+1,z) - u_y(x,y,z)) + \\ & (u_z(x,y,z+1) - u_z(x,y,z))\end{aligned}$$

Projection

Setting up the matrix

$$\begin{bmatrix} -\Omega_1 & \beta_{1,2} & \cdots & \beta_{1,n} \\ \beta_{2,1} & -\Omega_2 & & \vdots \\ \vdots & & -\ddots & \beta_{n-1,n-1} \\ \beta_{n,1} & \cdots & \beta_{n,n-1} & -\Omega_n \end{bmatrix} \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_n \end{bmatrix} = \begin{bmatrix} -D_1 \\ -D_2 \\ \vdots \\ -D_n \end{bmatrix}$$

Where D*i* corresponds to the divergences through cell i. Ω*i* is the number of non-solid neighbors of cell i, and $\beta i,j$ takes values based upon the equation:

$$\beta_{i,j} = \begin{cases} 1 & \text{if cell } i \text{ is a neighbor of cell } j \\ 0 & \text{otherwise} \end{cases}$$

This matrix is well known. It is called a *7 Point Laplacian Matrix*

Projection

Calculate the pressure gradient and subtract it from the velocity field to ensure it is divergence free:

$$u^{n+1}_{i+\frac{1}{2},j,k} = u^{n}_{i+\frac{1}{2},j,k} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x}$$

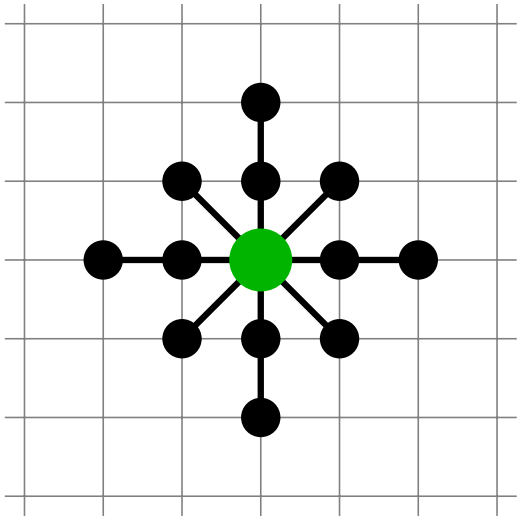
$$v^{n+1}_{i,j+\frac{1}{2},k} = v^{n}_{i,j+\frac{1}{2},k} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x}$$

$$w^{n+1}_{i,j,k+\frac{1}{2}} = w^{n}_{i,j,k+\frac{1}{2}} - \Delta t \frac{1}{\rho} \frac{p_{i,j,k+} - p_{i,j,k}}{\Delta x}$$

Demo

Implementation

Mapping to Mechanisms: Stencils



Common example ("5-point stencil"):

$$u_{i,j}^{n+1} = \frac{1}{h^2}(-4u_{i,j}^n + u_{i-1,j}^n + u_{i+1,j}^n + u_{i,j-1}^n + u_{i,j+1}^n)$$

- Sequential
- OpenMP?
- MPI?
- GPU — 2D?
- GPU — 3D?

Remember this slide from the last lecture?

Implementation

Thoughts on optimization for parallel execution

The most time consuming part of the sim is the pressure solve so the optimization should start with solving it quickly.

Optimizations for serial & CPU:

Blocking: Cache coherent.
Can send blocks to different cores

Optimizations for GPU:

Do blocking again using local memory to store each block.
For 3D use slicing, 3 Slices at a time putting middle slice (block) in local memory.