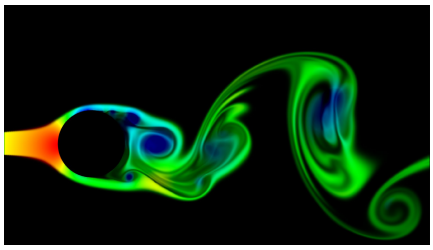


How to write an Eulerian Fluid Simulator

with 200 lines of JavaScript code

Matthias Müller, Ten Minute Physics

For the code and the demo see:

www.matthiasmueller.info/tenMinutePhysics

TypeScript port: https://github.com/p-sun/typescript-fluid-simulator

a few remarks...

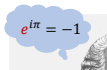
Fluid =

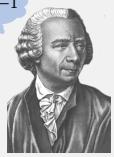
or

liquid gas

Similar physical structures → same simulation method

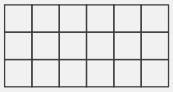
Eulerian





Swiss

Leonhard Euler (1707-1783)



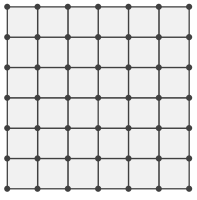
grid-based

Italian

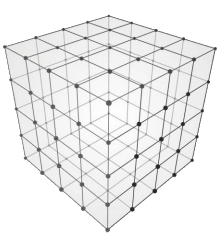
Joseph-Louis Lagrange (1736-1813)

grid-free (e. g. particles)

From 2d to 3d

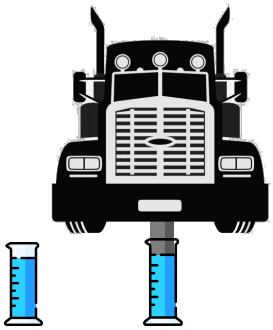


straight forward - left as an exercise



Incompressible Fluid

- We assume incompressibility
- Water is very close to being incompressible
  10,000 kg / cm² → 3% compression!
- Reasonable assumption for free gas as well



→ compressible simulation in upcoming tutorial

Inviscid Fluid

- We assume zero viscosity
- Good approximation for gas and water

inviscid

viscous

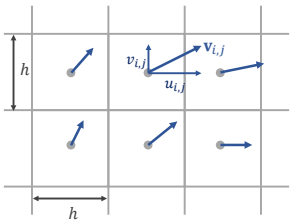
→ viscous simulation in upcoming tutorial

the method…

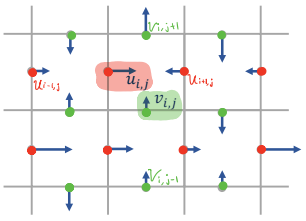
Fluid as a Velocity Field on a Grid

u = horizontal vector
v = vertical vector

- Velocity is a 2d vector $\mathbf{v} = \begin{bmatrix} u \\ v \end{bmatrix}$ (vectors are written in boldface)



collocated grid



staggered grid

h = grid height

Fluid Simulation

Step 1
Modify velocity values (e.g. add gravity)

step 2
Make the fluid incompressible (projection)

step 3
Move the velocity field (advection)

Step 1:
Update Velocity

for all i, j

$$v_{i,j} \leftarrow v_{i,j} + \Delta t \cdot g$$

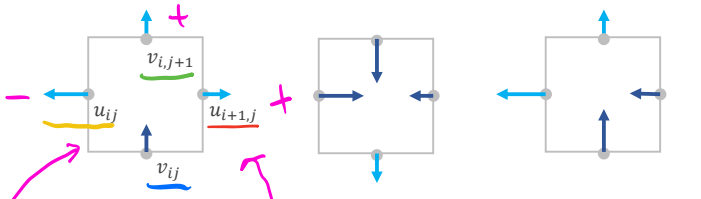
Just add gravity

- Gravity g: $-9.81\ m/s^2$
- Timestep Δt: (e. g. $\frac{1}{30} s$)

```
integrate(dt: number, gravity: number) {
  let n = this.numY;
  for (let i = 1; i < this.numX; i++) {
    for (let j = 1; j < this.numY - 1; j++) {
      if (this.s[i * n + j] != 0.0 && this.s[i * n + j - 1] != 0.0)
        this.v[i * n + j] += gravity * dt;
    }
  }
}
```

Divergence (Total Outflow)

$$d \leftarrow u_{i+1,j} - u_{i,j} + v_{i,j+1} - v_{i,j}$$



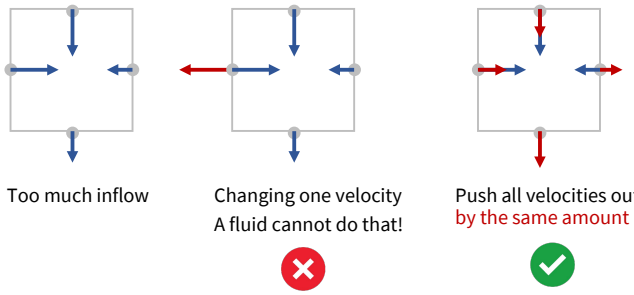
- Positive
- Too much outflow

- Negative
- Too much inflow

- Zero
- Incompressible

Note the signs are opposite on each axis.

Forcing Incompressibility

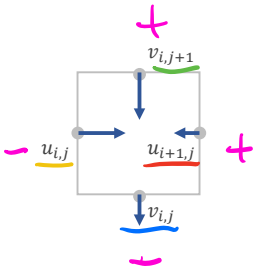


Too much inflow

Changing one velocity
A fluid cannot do that!

Push all velocities outward
by the same amount

Forcing Incompressibility

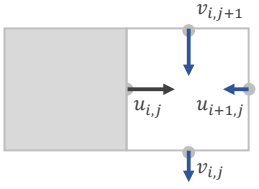


$$d \leftarrow u_{i+1,j} - u_{i,j} + v_{i,j+1} - v_{i,j}$$

$$\begin{aligned} u_{i,j} &\leftarrow u_{i,j} + d/4 \\ u_{i+1,j} &\leftarrow u_{i+1,j} - d/4 \\ v_{i,j} &\leftarrow v_{i,j} + d/4 \\ v_{i,j+1} &\leftarrow v_{i,j+1} - d/4 \end{aligned}$$

Note signs

Obstacles / Walls



$$d \leftarrow u_{i+1,j} - u_{i,j} + v_{i,j+1} - v_{i,j}$$

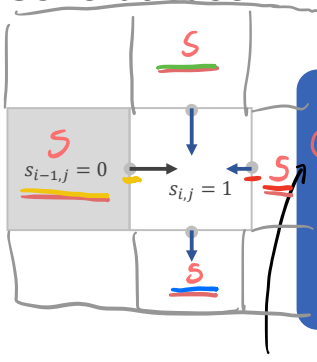
$$\begin{aligned} \sout{u_{i,j}} &\sout{\leftarrow u_{i,j} + d/4} \\ u_{i+1,j} &\leftarrow u_{i+1,j} - d/3 \\ v_{i,j} &\leftarrow v_{i,j} + d/3 \\ v_{i,j+1} &\leftarrow v_{i,j+1} - d/3 \end{aligned}$$

$u_{i,j} \neq 0$

- Moving object
- Turbine pushing fluid (e. g. wind tunnel)

General Case

Scalar s in a cell is...
0 for obstacles 1 for fluid



$$d \leftarrow u_{i+1,j} - u_{i,j} + v_{i,j+1} - v_{i,j}$$

$$s \leftarrow s_{i+1,j} + s_{i-1,j} + s_{i,j+1} + s_{i,j-1}$$

$$\begin{aligned} u_{i,j} &\leftarrow u_{i,j} + d\, s_{i-1,j}/s \\ u_{i+1,j} &\leftarrow u_{i+1,j} - d\, s_{i+1,j}/s \\ v_{i,j} &\leftarrow v_{i,j} + d\, s_{i,j-1}/s \\ v_{i,j+1} &\leftarrow v_{i,j+1} - d\, s_{i,j+1}/s \end{aligned}$$

Look at its NSEW neighbours to get total # of obstacles

Step 2: Make fluid incompressible

Solving the Grid

for n iterations
  for all i, j

$$d \leftarrow u_{i+1,j} - u_{i,j} + v_{i,j+1} - v_{i,j}$$

$$s \leftarrow s_{i+1,j} + s_{i-1,j} + s_{i,j+1} + s_{i,j-1}$$

$$\begin{aligned} u_{i,j} &\leftarrow u_{i,j} + d\, s_{i-1,j}/s \\ u_{i+1,j} &\leftarrow u_{i+1,j} - d\, s_{i+1,j}/s \\ v_{i,j} &\leftarrow v_{i,j} + d\, s_{i,j-1}/s \\ v_{i,j+1} &\leftarrow v_{i,j+1} - d\, s_{i,j+1}/s \end{aligned}$$

- Gauss-Seidel method
- On the boundary we access cells outside of the grid!

So we can:

- Add border cells
- Set $s_{i,j} = 0$ for walls
- or copy neighbor values

```
integrate(dt: number, gravity: number) {
  let n = this.numY;
  for (let i = 1; i < this.numX; i++) {
    for (let j = 1; j < this.numY - 1; j++) {
      if (this.s[i * n + j] != 0.0 && this.s[i * n + j - 1] != 0.0)
        this.v[i * n + j] += gravity * dt;
    }
  }
}
```

gravity

$$V_{ij} \leftarrow V_{ij} + a\Delta t$$

```
solveIncompressibility(scene: Scene, numIters: number, dt: number) {
  const n = this.numY;
  const cp = (this.density * this.h) / dt;

  for (let iter = 0; iter < numIters; iter++) {
    for (let i = 1; i < this.numX - 1; i++) {
      for (let j = 1; j < this.numY - 1; j++) {
        if (this.s[i * n + j] == 0.0) continue;

        // s - # of NSEW obstacles
        // s_ij is 0 for obstacle, 1 for fluid
        const sx0 = this.s[(i - 1) * n + j];
        const sx1 = this.s[(i + 1) * n + j];
        const sy0 = this.s[i * n + j - 1];
        const sy1 = this.s[i * n + j + 1];
        const s = sx0 + sx1 + sy0 + sy1;
        if (s == 0.0) continue;

        // div - Total divergence of NSEW velocities
        const div =
          this.u[(i + 1) * n + j] -
          this.u[i * n + j] +
          this.v[i * n + j + 1] -
          this.v[i * n + j];

        // dDiv - Average divergence of non-obstacle NESW cells
        const dDiv = (-div / s) * scene.overRelaxation;

        // u, v - Velocity
        // Note that if s_ij is an obstacle, its velocity doesn't change.
        this.u[i * n + j] -= sx0 * dDiv;
        this.u[(i + 1) * n + j] += sx1 * dDiv;
        this.v[i * n + j] -= sy0 * dDiv;
        this.v[i * n + j + 1] += sy1 * dDiv;

        // p - Pressure. Not needed for simulation.
        this.p[i * n + j] += cp * dDiv;
      }
    }
  }
}
```

$\frac{ph}{\Delta t}$

$-\frac{Od}{s}$

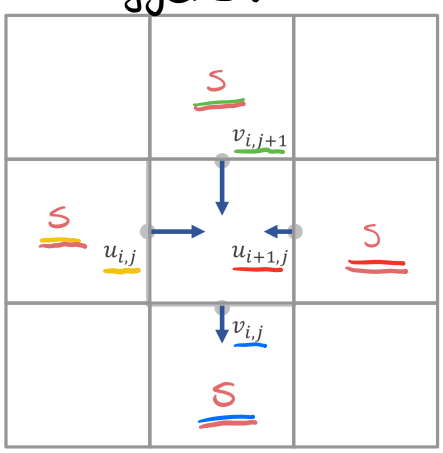
for n iterations

for all i, j overrelax *

$d \leftarrow u_{i+1,j} - u_{i,j} + v_{i,j+1} - v_{i,j}$

$s \leftarrow s_{i+1,j} + s_{i-1,j} + s_{i,j+1} + s_{i,j-1}$

Staggered Grid



$$u_{i,j} \leftarrow u_{i,j} + d\, s_{i-1,j}/s$$
$$u_{i+1,j} \leftarrow u_{i+1,j} - d\, s_{i+1,j}/s$$
$$v_{i,j} \leftarrow v_{i,i} + d\, s_{i,j-1}/s$$
$$v_{i,j+1} \leftarrow v_{i,j+1} - d\, s_{i,j+1}/s$$

$$P_{ij} \leftarrow P_{ij} - \frac{Od}{s} \cdot \frac{ph}{\Delta t}$$

(Optional)

Measuring Pressure

for all i, j
  $p_{i,j} \leftarrow 0$
for n iterations
  for all i, j
    $d \leftarrow u_{i+1,j} - u_{i,j} + v_{i,j+1} - v_{i,j}$
    $s \leftarrow s_{i+1,j} + s_{i-1,j} + s_{i,j+1} + s_{i,j-1}$
    ...
    $p_{i,j} \leftarrow p_{i,j} + \frac{d}{s} \cdot \frac{\rho h}{\Delta t}$

Should be − sign b/c we want to reduce divergence.

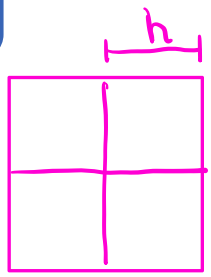
$p_{i,j}$ is physical pressure

ρ density ← always 1000 in the demo

h grid spacing

Δt time step

Not necessary for simulation



Overrelaxation

makes Gauss-Seidel faster to converge!

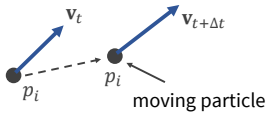
- Gauss-Seidel is very simple to implement
- Needs more iterations than global methods
- Acceleration: Overrelaxation – Magic!!

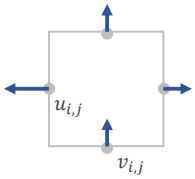
$$d \leftarrow o(u_{i+1,j} - u_{i,j} + v_{i,j+1} - v_{i,j})$$

- Multiply the divergence by a scalar $1 < o < 2$
- I use $o = 1.9$ in the code
- Increases convergence dramatically
- The computed pressure value is still correct!

Step 3: Advection

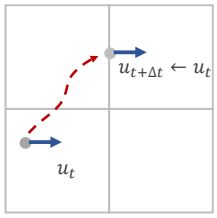
Last step: Advection





- In a fluid the velocity state is carried by the particles
- Particles move, grid cells are static
- We need to move the velocity values in the grid!

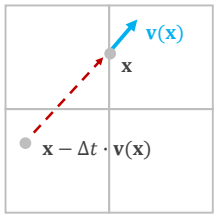
Semi-Lagrangian Advection



- Which fluid "particle" moved to the location where u is stored?
- Set the new velocity $u_{t+\Delta t}$ to the velocity u_t at the previous position

Step 3A - Advect velocity

Semi-Lagrangian Advection

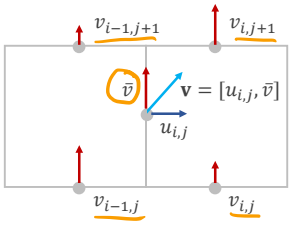


Approximate prev position

$$X_0 = X - \Delta t \cdot V(X)$$

- Compute $\mathbf{v}$ at position $\mathbf{x}$ where u is stored
- The previous location can be approximated as $\mathbf{x} - \Delta t \cdot \mathbf{v}(\mathbf{x})$
- Assuming a straight path introduces viscosity!
  Can be reduced with vorticity confinement

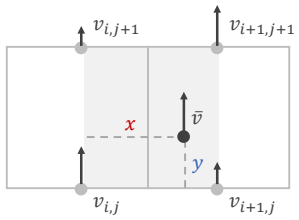
Get the 2d Velocity



- Average surrounding vertical velocities

$$\bar{v} = (v_{i,j} + v_{i,j+1} + v_{i-1,j} + v_{i-1,j+1})/4$$

General Grid Interpolation



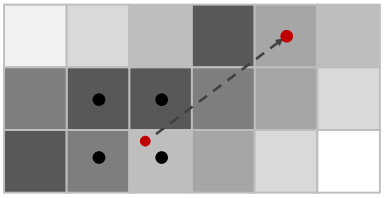
$w_{00} = 1 - x/h \quad w_{01} = x/h$
$w_{10} = 1 - y/h \quad w_{11} = y/h$

$$\bar{v} = w_{00}w_{10}v_{i,j} + w_{01}w_{10}v_{i+1,j} + w_{01}w_{11}v_{i,j+1} + w_{00}w_{11}v_{i+1,j+1}$$

Weighted average

Step 3B - Advect smoke

Smoke Advection

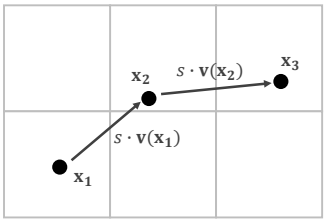


- Store density value at the center of each cell
- Advect it like the velocity components

Streamlines

$\mathbf{x} \leftarrow$ start position
$s \leftarrow$ step size

for n steps
  $\mathbf{v} \leftarrow$ sampleV($\mathbf{x}$)
  $\mathbf{x} \leftarrow \mathbf{x} + s\mathbf{v}$



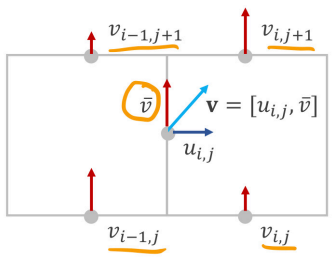
Let's look into the code...

```
extrapolate() {
  let n = this.numY;
  for (let i = 0; i < this.numX; i++) {
    // 0th col = 1st col
    this.u[i * n + 0] = this.u[i * n + 1];
    // last col = 2nd last col
    this.u[i * n + this.numY - 1] = this.u[i * n + this.numY - 2];
  }
  for (let j = 0; j < this.numY; j++) {
    // 0th row = 1st row
    this.v[0 * n + j] = this.v[1 * n + j];
    // last row = 2nd last row
    this.v[(this.numX - 1) * n + j] = this.v[(this.numX - 2) * n + j];
  }
}
```

```
simulate(scene: Scene, dt: number, gravity: number, numIters: number) {
  this.integrate(dt, gravity);  // Add gravity  ω  v' = v + aΔt

  this.p.fill(0.0);
  this.solveIncompressibility(scene, numIters, dt);

  this.extrapolate();
  this.advectVel(dt);
  this.advectSmoke(dt);
}
```

```ts
// Set velocity to be the predicted velocity of a particle dt time
// in the past, calculated by averaging neighbouring velocities.
advectVel(dt: number) {
  this.newU.set(this.u);
  this.newV.set(this.v);

  const n = this.numY;
  const h = this.h;
  const h2 = 0.5 * h;

  for (let i = 1; i < this.numX; i++) {
    for (let j = 1; j < this.numY; j++) {
      // u component (horizontal) ------------
      if (
        this.s[i * n + j] != 0.0 &&
        this.s[(i - 1) * n + j] != 0.0 &&
        j < this.numY - 1
      ) {
        // position at index i,j
        const x = i * h;
        const y = j * h + h2;

        // velocity at index i,j
        const u = this.u[i * n + j];
        const v = this.avgV(i, j); // similar to this.sampleField(x, y, 'V_FIELD');

        // previous position, dt time ago
        const x0 = x - dt * u;
        const y0 = y - dt * v;

        //  weighted average of velocity at prev position
        const u0 = this.sampleField(x0, y0, 'U_FIELD');
        this.newU[i * n + j] = u0;
      }
      // v component (vertical) ------------
      // Same as above, but for v instead of u
      if (
        this.s[i * n + j] != 0.0 &&
        this.s[i * n + j - 1] != 0.0 &&
        i < this.numX - 1
      ) {
        const x = i * h + h2;
        const y = j * h;
        const u = this.avgU(i, j); // similar to this.sampleField(x, y, 'U_FIELD');
        const v = this.v[i * n + j];
        const x0 = x - dt * u;
        const y0 = y - dt * v;
        const v0 = this.sampleField(x0, y0, 'V_FIELD');
        this.newV[i * n + j] = v0;
      }
    }
  }

  this.u.set(this.newU);
  this.v.set(this.newV);
}
```

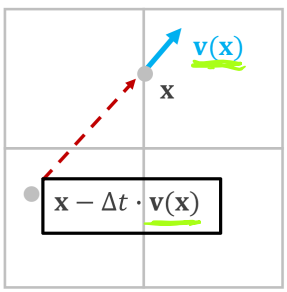
Get the 2d Velocity



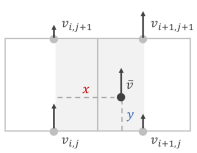
- Average surrounding vertical velocities

$$\bar{v} = (v_{i,j} + v_{i,j+1} + v_{i-1,j} + v_{i-1,j+1})/4$$

Approximate prev position



General Grid Interpolation



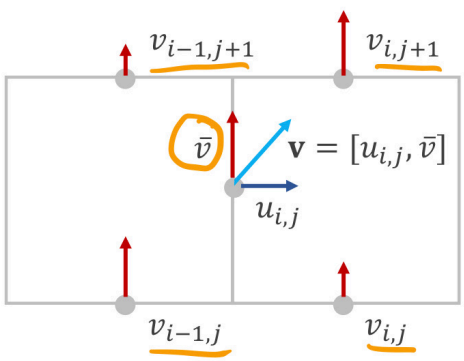
$w_{00} = 1 - x/h$ $w_{01} = x/h$
$w_{10} = 1 - y/h$ $w_{11} = y/h$

$$\bar{v} = w_{00}w_{10}v_{i,j} + w_{01}w_{10}v_{i+1,j} + w_{01}w_{11}v_{i,j+1} + w_{00}w_{11}v_{i+1,j+1}$$

```ts
// Similar to advectVel, but for smoke density.
// Averages neighbouring smoke densities to get new density.
advectSmoke(dt: number) {
  this.newM.set(this.m);

  const n = this.numY;
  const h = this.h;
  const h2 = 0.5 * h;

  for (let i = 1; i < this.numX - 1; i++) {
    for (let j = 1; j < this.numY - 1; j++) {
      if (this.s[i * n + j] != 0.0) {
        const u = (this.u[i * n + j] + this.u[(i + 1) * n + j]) * 0.5;
        const v = (this.v[i * n + j] + this.v[i * n + j + 1]) * 0.5;
        const x = i * h + h2 - dt * u;
        const y = j * h + h2 - dt * v;

        this.newM[i * n + j] = this.sampleField(x, y, 'S_FIELD');
      }
    }
  }
  this.m.set(this.newM);
}
```

```
// Average horizontal velocity at index (i, j)
avgU(i: number, j: number) {
  const n = this.numY;
  const u =
    (this.u[i * n + j - 1] +
      this.u[i * n + j] +
      this.u[(i + 1) * n + j - 1] +
      this.u[(i + 1) * n + j]) *
    0.25;
  return u;
}

// Average vertical velocity at index (i, j)
avgV(i: number, j: number) {
  const n = this.numY;
  const v =
    (this.v[(i - 1) * n + j] +
      this.v[i * n + j] +
      this.v[(i - 1) * n + j + 1] +
      this.v[i * n + j + 1]) *
    0.25;
  return v;
}
```

Get the 2d Velocity

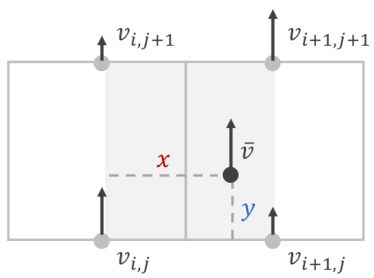


- Average surrounding vertical velocities

$$\bar{v} = (v_{i,j} + v_{i,j+1} + v_{i-1,j} + v_{i-1,j+1})/4$$

```
// Average weighted velocity at position (x, y)
sampleField(x: number, y: number, field: Field) {
  const n = this.numY;
  const h = this.h;
  const h1 = 1.0 / h;
  const h2 = 0.5 * h;

  x = Math.max(Math.min(x, this.numX * h), h);
  y = Math.max(Math.min(y, this.numY * h), h);

  let dx = 0.0;
  let dy = 0.0;

  let f;
  switch (field) {
    case 'U_FIELD':
      f = this.u;
      dy = h2;
      break;
    case 'V_FIELD':
      f = this.v;
      dx = h2;
      break;
    case 'S_FIELD':
      f = this.m;
      dx = h2;
      dy = h2;
      break;
  }

  const x0 = Math.min(Math.floor((x - dx) * h1), this.numX - 1);
  const tx = (x - dx - x0 * h) * h1;
  const x1 = Math.min(x0 + 1, this.numX - 1);

  const y0 = Math.min(Math.floor((y - dy) * h1), this.numY - 1);
  const ty = (y - dy - y0 * h) * h1;
  const y1 = Math.min(y0 + 1, this.numY - 1);

  const sx = 1.0 - tx;
  const sy = 1.0 - ty;

  const val =
    sx * sy * f[x0 * n + y0] +
    tx * sy * f[x1 * n + y0] +
    tx * ty * f[x1 * n + y1] +
    sx * ty * f[x0 * n + y1];

  return val;
}
```

General Grid Interpolation



$$w_{00} = 1 - x/h \qquad w_{01} = x/h$$
$$w_{10} = 1 - y/h \qquad w_{11} = y/h$$

$$\bar{v} = w_{00}w_{10}v_{i,j} + w_{01}w_{10}v_{i+1,j} + w_{01}w_{11}v_{i,j+1} + w_{00}w_{11}v_{i+1,j+1}$$