

# Computational Fluid Dynamics With a Paper Airplane

Tasada, Daniel              Tse, Nathan

February 2, 2025

## **Abstract**

In this paper, we investigate the relationship between an airplane's shape and its performance. Our results show that ...

# **1 Preface**

Nam quis enim. Quisque ornare dui a tortor. Fusce consequat lacus pellentesque metus. Duis euismod. Duis non quam. Maecenas vitae dolor in ipsum auctor vehicula. Vivamus nec nibh eget wisi varius pulvinar. Cras a lacus. Etiam et massa. Donec in nisl sit amet dui imperdiet vestibulum. Duis porttitor nibh id eros.

# Contents

<b>1 Preface</b>	<b>1</b>
<b>2 Nomenclature</b>	<b>3</b>
<b>3 Introduction</b>	<b>4</b>
<b>4 Preliminary Research</b>	<b>5</b>
4.1 Navier-Stokes equations . . . . .	5
4.2 Eulerian vs Lagrangian . . . . .	5
4.3 Genetic Algorithm . . . . .	6
<b>5 First Prototype: Lagrangian Fluid Simulation</b>	<b>7</b>
<b>6 Execution: Eulerian Fluid Simulation</b>	<b>8</b>
6.1 The Math . . . . .	8
6.2 Density Solver . . . . .	9
6.2.1 Adding forces . . . . .	9
6.2.2 Diffusion . . . . .	9
6.2.3 Advection . . . . .	10
6.3 Velocity Solver . . . . .	11
6.3.1 Projection . . . . .	11
6.4 The Implementation . . . . .	11
6.4.1 Introduction to the code . . . . .	11
6.4.2 Voxelizing Geometry . . . . .	13
6.4.3 Implementing obstacle collision . . . . .	14
<b>7 Machine Learning Theory</b>	<b>15</b>
7.1 Fitness Calculation . . . . .	15
7.2 Genetic Algorithm . . . . .	16
<b>8 Conclusion</b>	<b>17</b>
<b>9 Discussion: Challenges, Failures &amp; Future Work</b>	<b>18</b>
9.1 Grid Collision . . . . .	18
9.2 Performance . . . . .	18
9.3 Future Work . . . . .	19
<b>10 Bibliography</b>	<b>20</b>

## 2 Nomenclature

*Acronyms:*

CFD	Computational Fluid Dynamics
ML	Machine Learning
NEAT	NeuroEvolution of Augmenting Topologies
BVH	Bounding Volume Hierarchy
FCL	Flexible Collision Library
AMR	Adaptive Mesh Refinement

*Variables:*

$L$	Angular momentum of a given particle
$I$	Inertia tensor of a given particle
$\omega$	Angular velocity of a given particle
$n$	Collision normal of a given particle
$R_i$	Orientation of a given particle
$\lambda$	Constant used in Lagrangian collision resolution
$u$	Velocity
$\rho$	Density
$\nu$	Viscosity
$\kappa$	Density diffusivity
$f$	External forces
$S$	Density source
$\tau$	Shear stress
$P$	Pressure

### 3 Introduction

This thesis covers the simulation of the aerodynamics of an airplane, using our own Computational Fluid Dynamics model (CFD).

The goal is to simulate the airflow around an airplane's body. CFD is a branch of fluid mechanics that uses numerical methods and algorithms to solve and analyze problems that involve fluid flows. It is used in many fields, including aerospace engineering, automotive engineering, and meteorology.

The application of CFD to an airplane is important because it allows testing of a model's aerodynamic performance without building a physical model, or have to set up a wind tunnel. The practical alternative is much more expensive and time-consuming.

The final aim is to determine how the shape of an airplane's body affects its performance. Part of the project is to dynamically generate 3D models of airplanes, and simulate the airflow around them. The intention is to use machine learning to optimize the shape of the airplane's body to maximize performance.

CFD is challenging in the sense that it requires a good understanding of fluid dynamics, as well as a good understanding of the math involved. CFD is also very expensive from a computational perspective, so code optimization is important.

The thesis questions are the following:

- How does the math behind CFD work?
- How is a CFD implemented?
- How do obstacles work in a CFD?
- How is the performance of an airplane calculated?
- How can machine learning be used to optimize an airplane's shape?

## 4 Preliminary Research

In this paper we aim to create our own CFD model and use machine learning to research the aerodynamics of an airplane. Before we start creating our own model, an understanding of how CFD works and the different types of models and methods that can be used is needed. Most CFD models are based on the Navier-Stokes equations. Because the Navier-Stokes equations are very complex we will not be covering them in depth, but we will explain what they are and how they are used in CFD models. This will be covered in section 4.1. In section 4.2 we will be looking at two different types of methods that are used to simulate fluid dynamics. For the machine learning part, we will be using a genetic algorithm to optimize the shape of the airplane. What genetic algorithms are and how they work will be covered in section 4.3.

### 4.1 Navier-Stokes equations

The base of almost all CFD models are the [Navier-Stokes equations](#). These are partial differential equations that describe fluid flows. We will not go into too much detail about these equations, but the gist of it that these equations return the flow velocity of the fluid. This flow velocity is a vector field that assigns a vector to every point in the fluid at a given time. Using this velocity field, the behavior of the fluid can be simulated. ([Wikipedia, 2024e](#))

### 4.2 Eulerian vs Lagrangian

There are two main methods to simulate fluid dynamics: the Lagrangian method and the Eulerian method.

#### Lagrangian

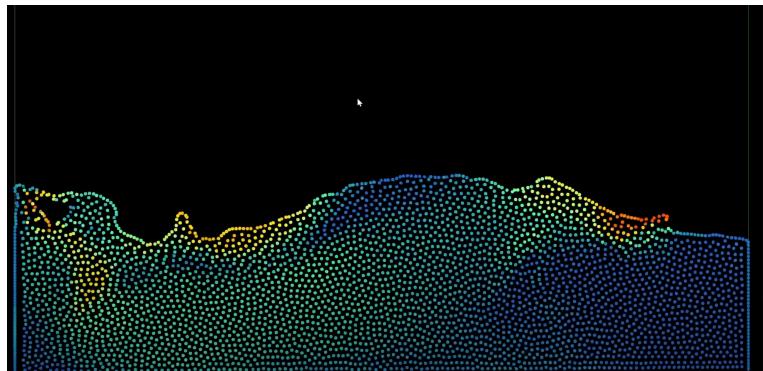


Figure 1: Lagrangian fluid simulation from ([Lague, 2023](#))

The Langrangian method models the fluid as a particle collision system, where the air is represented by particles that follow the flow velocity and interact with each other to emulate a fluid. This method is mostly useful when trying to analyze situations where single fluid particles play an important role.

## Eulerian

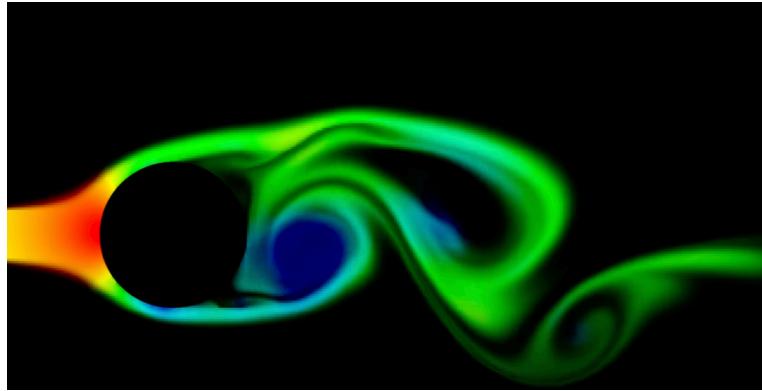


Figure 2: Eulerian fluid simulation from (Müller, 2022)

The other method is the Eulerian method. This method models the fluid as a cellular automata. The fluid is represented by a grid of cells, each of which has its own properties and interacts with its neighbor cells, creating a continuous field. This method is mostly used when trying to analyze the fluid as a whole. It can provide information about the fluid's flow patterns, turbulence, and pressure distribution.

### 4.3 Genetic Algorithm

In machine learning, genetic algorithms are models that mimic the process of natural selection. In a genetic algorithm a population of solutions to an optimization problem is evolved over time towards better solutions. The evolution usually starts from a population of randomly generated solutions. It is an iterative process, where the population in each iteration is called a generation. A fitness function is used every generation to evaluate the fitness of each solution in the population. Fitness is a measure of how well a solution fits the problem. With that in mind an iteration will typically look like this:

1. The fitness of every solution in the population is evaluated.
2. A percentage of the more fit solution are selected from the current population.
3. Each selected solution is randomly modified.
4. These fit solutions and the new mutated solutions are added to the new generation.
5. The new generation of candidate solutions is then used in the next iteration of the algorithm.

(Wikipedia, 2024d)

## 5 First Prototype: Lagrangian Fluid Simulation

We first tried to implement a Lagrangian fluid simulation. Our implementation is based on [Rigid Body Collision Resolution](#) (Hakenberg, 2005). We used this paper as a guide for all the math involved.

The math relies on the momentum, inertia, and velocity of the particles to calculate the collision normal and point of contact. The collision normal is the direction in which the particles are moving away from each other, and the point of contact is the point at which the particles collide.

The following variables are necessary to perform the calculations:

Angular momentum $L$ ( $kg \cdot m^2/s$ ) :	$L = mvr;$
Inertia tensor $I$ ( $kg \cdot m^2$ ) :	$I = \frac{L}{\omega};$
Angular velocity $\omega$ ( $rad/s$ ) :	$\omega = \frac{\Delta\theta}{\Delta t};$

Collision normal( $n \in \mathbb{R}^3$ ) in world coordinates away from body;

Point of contact ( $r_i \in \mathbb{R}^3$ ) in world coordinates with respect to  $p_i$ ;

Orientation ( $R_i \in SO(3)$ ) transforming from object to world coordinates;

Where  $i$  represents one of two particles in a given collision:

Velocity after collision	$\tilde{v}_i,$
Angular velocity after collision	$\tilde{\omega}_i,$
Constant	$\lambda,$

The following formulas represent the relation between particles:

$$\begin{aligned}\tilde{v}_1 &= v_1 - \frac{\lambda}{m_1} n; \\ \tilde{v}_2 &= v_2 + \frac{\lambda}{m_2} n; \\ \tilde{\omega}_1 &= \omega_1 - \Delta q_1; \\ \tilde{\omega}_2 &= \omega_2 + \Delta q_2; \\ \text{where } q_i &:= I_i^{-1} \cdot R_i^{-1} \cdot (r_i \times n), \\ \text{and } \lambda &= 2 \frac{n v_1 - n v_2 + \omega_1 I_1 q_1 - \omega_2 I_2 q_2}{(\frac{1}{m_1} + \frac{1}{m_2}) n^2 + q_1 I_1 q_1 + q_2 I_2}\end{aligned}$$

This was implemented using Go and raylib, and the code is available at [github.com/dtasada/paper](https://github.com/dtasada/paper) at the `lagrangian-go` branch.

When dealing with a lot of particles, the simulation stops performing as well, because the number of calculations and iterations of the formulas listed above has a time complexity of  $O(n^2)$ , where  $n$  is the number of particles. This is because every particle handles collisions with every other particle every single frame. This can be elegantly mitigated by implementing a three-dimensional grid system, where every particle is assigned to a cell in the grid. This way, particles only need to check for collisions with other particles in the same cell or in any of its 26 neighboring cells, assuming that the particles are evenly distributed within the container, each particle only needs to handle collisions with particles in its own cell, and its 26 neighboring cells. This reduces the time complexity to  $O(n)$ .

The simulation was unstable at higher densities, where particles would phase into each other instead of cleanly bouncing. The bug is likely quite simple to fix, but we got distracted by the Eulerian method. Despite the optimizations we made, Lagrangian simulation is still quite computationally expensive, and Go wasn't doing it any favors, because it's not a very memory efficient language. In the end it was put aside, and the Lagrangian implementation was never completed.

## 6 Execution: Eulerian Fluid Simulation

Our next attempt was to try and implement a Eulerian fluid simulation. Our fluid simulation involves a three-dimensional grid of cells, each of which have velocity and density fields. The density fields are purely for the purpose of visualization. In the CFD frontend, we render translucent colored cubes in each cell according to its density field. These translucent cubes represent dye getting dropped into the container. When we are talking about ‘density’, you can image it as the density of dye in the fluid. Each frame, the cells interact with each other according to the [Navier-Stokes equations](#).

The specific math involved in the base equations for the fluid simulation is quite complex, and we couldn’t have derived it on our own, so we used Mike Ash’s implementation [Fluid Simulation for Dummies](#) (Ash, 2006) as the bedrock of our program. Mike Ash’s code is in turn based on Jos Stam’s brilliant work on [Real-Time Fluid Dynamics for Games](#) (Stam, 2003). This math will be explained in section 6.1. How we implemented this math into the density solver of the fluid simulation is explained in section 6.2. The way it is implemented in the velocity solver is explained in section 6.3. We will explain the concepts in 2D, so it is easier to visualize and understand. Lastly, section 6.4 explains how to implement the fluid simulation, which includes the code’s structure and how we implemented the collision detection.

### 6.1 The Math

In Eulerian fluid dynamics we represent fluids with a velocity vector field. This means we assign a velocity vector to every point in space. The Navier-Stokes equations show us how these velocity vectors evolve over time with an infinitesimal timestep.

$$\text{velocity equation: } \frac{\partial u}{\partial t} = -(u \cdot \nabla)u + \nu \nabla^2 u + f$$

$$\text{density equation: } \frac{\partial \rho}{\partial t} = -(u \cdot \nabla)\rho + k \nabla^2 \rho + S$$

The first equation returns the change in velocity in a vector notation. Unlike in Lagrangian fluid simulation, in Eulerian fluid simulation the fluid is not represented by individual particles. That’s why fluid density is used instead, which tells us the amount of particles present in a point in space. The second equation shows us the change in density. Despite that it isn’t needed to fully understand these equations to understand the math behind the simulation, it should be noted that the two equations above look a lot like each other, as this was helpful in developing the simulation. (Stam, 2003)



Figure 3: Dye diffusing in water

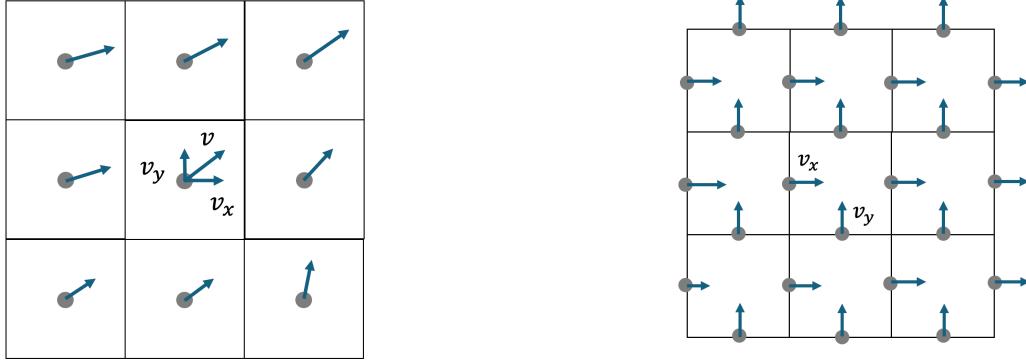


Figure 4: Computational grids

There are two different types of velocity vector fields we can use: collocated and staggered. In the collocated grid all the variables are stored in the center of the cell, whereas in the staggered grid the scalar variables are stored at the center but the velocity variable is stored at the cell face. Usually the use of a staggered grid is preferred, because it's easier to see how much fluid flows from one cell to another (Müller, 2022). However, because Mike Ash and Jos Stam used a collocated grid in their works we decided to use the same grid for the sake of simplicity.

## 6.2 Density Solver

First, let's look at the density solver. The density solver follows three steps: adding forces, then applying diffusion and lastly moving the densities. If we look at the density equation in section 6.1, we see that the change in density in a single time step is influenced by three terms. The first term states that the density should follow the velocity field, the second term represents the density's rate of diffusion and the third term says that the density can increase due to external sources. The density solver will apply these terms in reverse order. So, as stated before, first the forces are added, then the diffusion is applied and finally the densities are moved.

### 6.2.1 Adding forces

The first step is adding forces from a source. In our case forces will be added if the user clicks on the screen. If that happens we will simply add a constant amount of density to the initial density. So:

$$\rho_{t+dt} = \rho_t + S \cdot dt$$

### 6.2.2 Diffusion

The second step is diffusion. Diffusion is the process where the fluid spreads from high density cells to low density cells. The diffusion happens at a constant rate that we will call  $D$ . Now we can simply calculate the diffusion rate  $a$  by multiplying it with the time step:

$$a = dt \cdot D$$

If we take a look at a single cell, we'll see that it exchanges densities with its four neighbors (in 2D, the same applies for a 3D cell's six neighbors). The total difference of the densities

between the cell and its neighbors can be calculated by the following formula, where  $\rho_{x,y}$  is the current density of the cell at position  $(x, y)$ :

$$\rho_{x+1,y} + \rho_{x-1,y} + \rho_{x,y+1} + \rho_{x,y-1} - 4 \cdot \rho_{x,y}$$

By multiplying the difference in densities with the diffusion rate  $a$  and adding it to the current density, we calculate new density of the cell. This can be seen in the following formula, where  $P_{x,y}$  is equal to the density of cell at position  $(x, y)$  after the time step:

$$P_{x,y} = \rho_{x,y} + a \cdot (\rho_{x+1,y} + \rho_{x-1,y} + \rho_{x,y+1} + \rho_{x,y-1} - 4 \cdot \rho_{x,y})$$

But here we run into a problem. If the diffusion rate is big enough, the density in a cell might end up becoming negative. Imagine, for example, a situation were the diffusion rate is equal to 1, the density of a cell is 1 and the densities of its neighbors are all 0. This means that the density of the cell will become -3. This of course is not possible. The solution to this problem is to find the density of the cell when being diffused backwards in time. Which results in the following equation:

$$\rho_{x,y} = P_{x,y} + a \cdot (P_{x+1,y} + P_{x-1,y} + P_{x,y+1} + P_{x,y-1} - 4 \cdot P_{x,y})$$

To solve this system of linear equations for the unknowns  $P_{x,y}$  the [Gauss-Seidel method](#) is used. This is an iterative method to solve a system of linear equations. In short, we apply the following formula an  $N$  amount of times for every cell in the grid:

$$P_{x,y} = \frac{\rho_{x,y} + a \cdot (P_{x+1,y} + P_{x-1,y} + P_{x,y+1} + P_{x,y-1})}{1 + 4a}$$

$N$  is a constant that determines how many times the formula is applied. The higher  $N$  is, the more accurate the result will be. ([Stam, 2003](#))

### 6.2.3 Advection

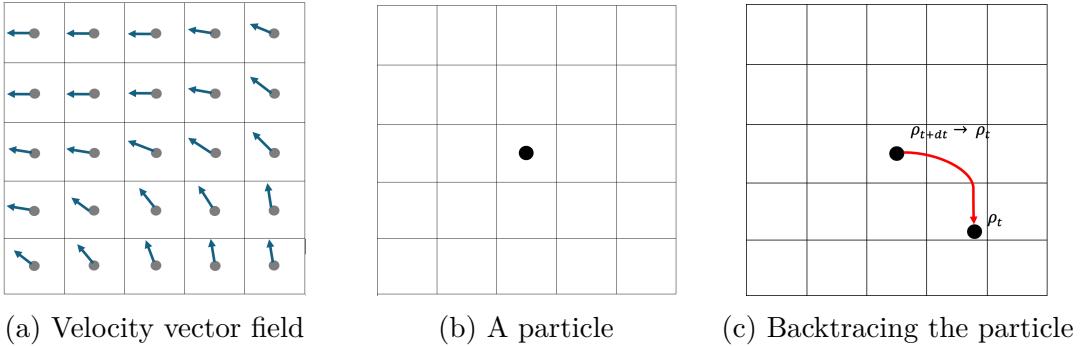


Figure 5: Advection method

The final step of the density solver is moving the densities following the velocity field. Because you can't really move grid cells, we will use a technique where we represent a density as a set of particles. This technique is called semi-lagrangian advection. An obvious way to move the densities is to act like the center of a cell is a particle and move it through the velocity field. But because we have to convert the particles back to a grid, we will use a different method. We will first look at the particles that end up at the center of a cell, as shown in figure 5b. Then we use the velocity of the cell to trace the particles

one time step back in time. This is shown in figure 5c. Then we will look at where the particles end up and calculate the weighted average of the densities of the four closest cells from a particle and set this as the new density of the current cell, also shown in figure 5c. To do this we need to use two different grids: one that contains all the density values from the previous time step and on that contains the new density values. (Stam, 2003)

## 6.3 Velocity Solver

Now we will look at the velocity solver. Just like with the density solver if we look at the equations in section 6.1 we see that the change in velocity in a single time step is influenced by three factors. These are: the addition of forces, viscous diffusion and self-advection. Because the velocity equation looks so much like the density equation we can use the same steps that were shown in section 6.2 to solve it. The only difference is that the velocity solver has an extra step called projection.

### 6.3.1 Projection

This step ensures that mass is conserved. The law of conservation of mass is an important property of real-life fluids. The projection step will be the last step that is executed because the previous steps might not always conserve mass. To make the fluid mass conserving we will use a mathematical concept called Hodge decomposition. Hodge composition states that every vector field is the sum of a mass conserving field and a gradient field. So all we have to do is compute a gradient field and subtract it from the velocity field. The gradient field...TODO: finish this. Computing the gradient field involves the solution of a linear system called a Poisson equation. To solve this system we will again use the Gauss-Seidel method that was also mentioned in section 6.2 in the diffusion step (Stam, 2003).

## 6.4 The Implementation

### 6.4.1 Introduction to the code

Our project is structured as follows:

```

paper
└── neural ..... Contains ML engine
└── simulation/ ..... Contains the physics simulation code
    ├── include/ ..... Our own engine headers
    ├── resources/ ..... Resources like images, fonts, shaders
    ├── src/ ..... Actual engine source
    ├── config.toml ..... Configuration file
    └── Makefile

```

This codebase is written in C++ and uses `raylib` again because it's a really simple and powerful library that allows us to focus on the simulation itself, as it provides a bunch of tools to easily render 3D graphics. The reason this codebase is in C++ was initially because we never got the Eulerian model to work in Go, likely due to a mistake of our own. Nevertheless we think C++ was the right choice, as it's more appropriate for the vector physics and low overhead we need for the simulation.

The main fluid simulation object we use for the simulation is the `Fluid` class. The implementation is available at `/simulation/src/Fluid.cpp`. The interface is as follows:

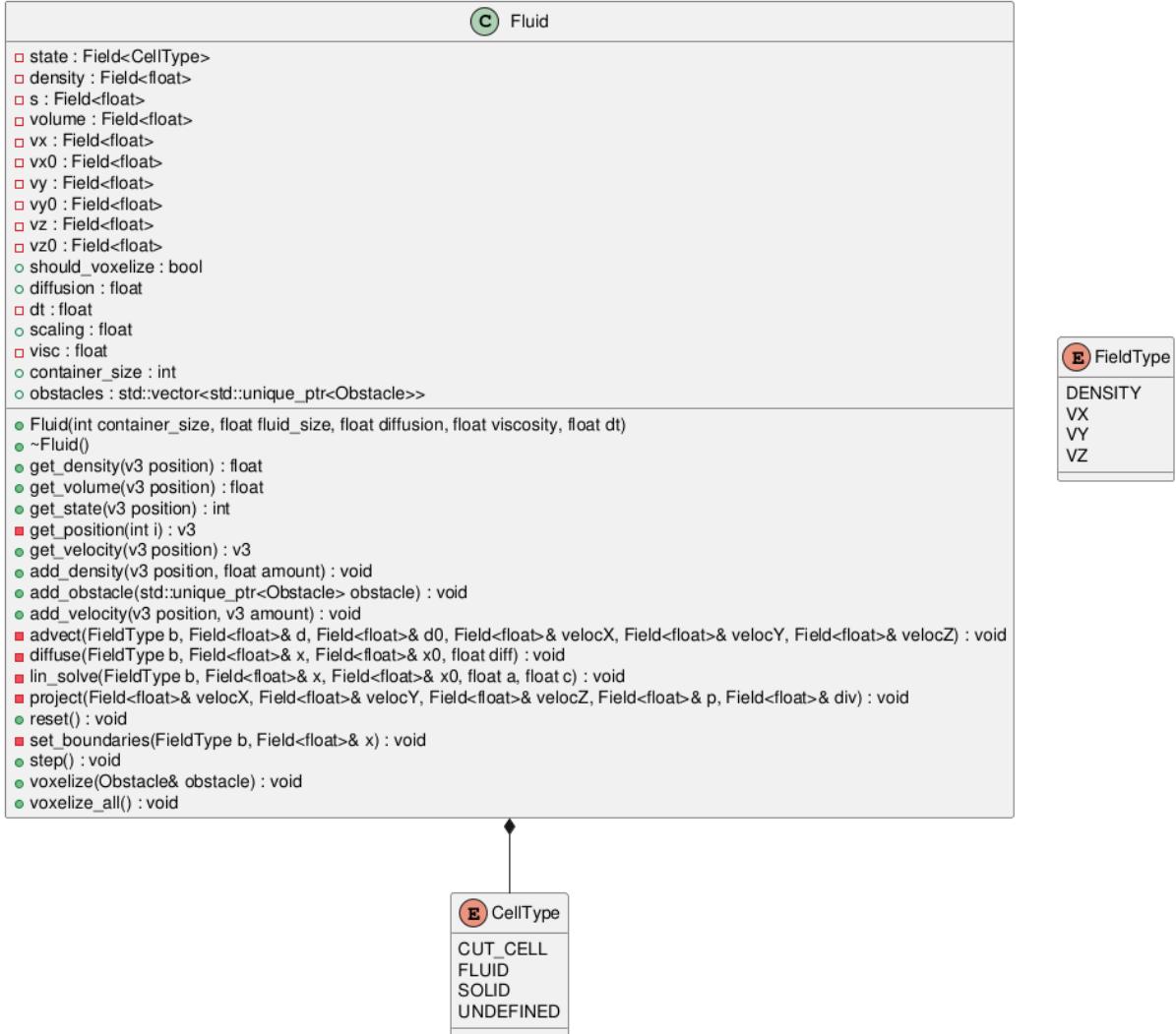


Figure 6: Fluid class diagram

This class contains the vector fields for the container as well as all the functions related to the physics simulation, including the advection, diffusion and projection procedures. These functions are the core of the CFD and this is the part that we borrowed from Mike Ash. After implementing the interface of figure 6 along with some graphical rendering code, we were able to get a basic fluid simulation working. The result is shown in figure 7.



Figure 7: Basic fluid simulation

#### 6.4.2 Voxelizing Geometry

The next step is to implement geometry collision. The graphics library we're using, Raylib, has a function to load geometry from object files. Loading in an arbitrary shape like a torus is easy enough, but the real challenge lies in communicating to the CFD that there's a torus in a given position and it behaving correspondingly.

First, the 3D object that Raylib loads has to be converted to a BVH (Bounding Volume Hierarchy) object. This is a tree structure that contains the object's geometry in a way that makes it easy to check if a point is inside the object. This format is what FCL ([Flexible Collision Library](#)) uses to check for collisions. We are using FCL to easily process flexible collisions between 3D objects.

The next step is to write a voxelization function. Voxelization takes a 3D object's mesh data and iterates over the cells of the 3D vector fields to see if the object occupies a cell or a part of a cell. FCL performs collision detection between two 3D objects. In this case those two objects are our obstacle (like a torus or a plane), and whichever cell is being iterated over. If the cell that's being iterated over contains part of an object, we can pass a corresponding condition to the CFD. This is done for every cell in the grid, and the result is a 3D boolean array that represents the object. The source code is at `Fluid::voxelize`. Each obstacle is passed to this function, and as a result we get a 3D boolean field saved at `Fluid::state`. [8b](#) is a figure of the voxel map, with cells classified as solid highlighted in blue. Note that the edges of the torus are also marked as solid despite their cells obviously only being partial. This is because our implementation of the voxel mapping using FCL isn't working as intended.



Figure 8: Voxel map

This is called the [Cut-Cell method](#) (Tucker, 2000). The reason we even need this is

because the more obvious option is to set up the grid to be high resolution so each cell is small enough to accurately approximate an arbitrarily complex object. The reason this isn't viable is purely due to performance. This might be reasonable with a supercomputer, but for our purposes we'd just like to optimize.

Another method would be to implement an AMR (Adaptive Mesh Refinement) system, which would adapt the mesh's resolution based on what's in a given region (Wikipedia, 2024a). This would generate a high resolution grid around an obstacle's bounds, and a low resolution grid in any empty space. This is an interesting concept, but we would have to rewrite some of the fluid math code to make the irregular grid work, and the Cut-Cell method is more straightforward, so we stuck with that.

#### 6.4.3 Implementing obstacle collision

According to the Cut-Cell method, each cell of the grid now has one of three states: fluid, solid, or cut-cell.

##### 1. Fluid cells

- Cells that are fully inside a fluid region
- Normal Navier-Stokes equations apply

##### 2. Solid cells

- Cells that are fully inside an obstacle
- Applies no-slip condition: velocity is zero

##### 3. Cut-cells

- Cells partially inside an object
- Modify fluid calculations

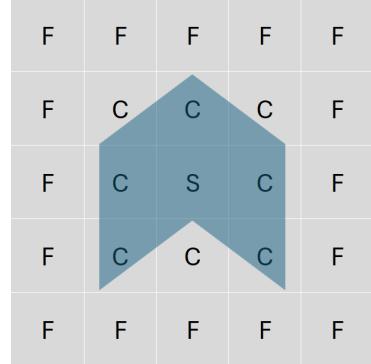


Figure 9: Diagram of cell states

We'd like to preface this by saying that our implementation of Cut-Cell obstacle resolution is not completed or working, so some of the following is theoretical. The idea is that when a cell is classified as a Cut-Cell, we create a new field to track the amount of fluid that's inside the cell. For every cell, we calculate the fractional fluid volume:

$$\text{Fractional fluid volume } \alpha = \frac{\text{volume of fluid in cell}}{\text{total cell volume}}$$

$$\text{Fractional face area } \beta_d = \frac{\text{open d-face area}}{\text{total d-face area}} \text{ where d is one of } x, y, \text{ or } z$$

The fractional fluid volume is the ratio of the volume of fluid in the cell to the total volume of the cell, where a solid cell has a value of 0 and an empty cell has a value of 1. The same goes for the fractional face area, which is the ratio of the open face area to the total area of a given cell.

With this information we can modify the advection step of the CFD to account for a cut-cell. We do this by scaling the advection by the fractional face area. Next we also adjust the [projection step](#). TODO POISSON EQUATION

The last step is to make sure to apply the correct boundaries to the fluid, where any solid cells have a constant velocity of 0.

The reason we failed to implement this is twofold:

1. Incomplete voxelization process. Figure 8b is captured with a modified version of the program that only voxelizes to two states: solid and fluid, with no cut-cells. This is because we weren't capable of implementing the FCL collision detection correctly. We don't know what caused the issue, but unfortunately we didn't have time to debug it thoroughly.
2. We'd forgotten to modify the projection step to account for cut-cells, which explains a bug we experienced, where the fluid would leak through the cut-cells.

Improving on these two points might have yielded better results.

## 7 Machine Learning Theory

The second part of our project was to use machine learning to optimize the shape of the airplane's body. The idea was to use a genetic algorithm to evolve the shape of the airplane's body to maximize performance. Once again since we didn't have a working fluid simulation we never got around to actually writing this part.

That being said here's the outline of how ML (Machine Learning) works: we grab a basic airplane model, and we generate a population of airplane models. We simulate the airflow around each airplane model, and we use the results to calculate a fitness score for each airplane model. We then use a genetic algorithm to evolve the airplane models to maximize the fitness score. We repeat this process until we have a model that approaches a maximum fitness score.

The plan was to write a Python script that utilizes the NEAT algorithm to generate the populations. NEAT is a genetic algorithm, which has a Python library that the user can implement in their scripts.

There's two big parts to this: the first is the fitness function, which is the function that takes the results of the fluid simulation and calculates a score for the airplane model. This is discussed in section 7.1. section 7.2 discusses the genetic algorithm, which is the portion that tweaks the vertex data of each airplane model to try to improve the fitness score.

### 7.1 Fitness Calculation

The fitness function is the function on the CFD-side that measures how good a given polygon is at flying. The metric for this is the lift-to-drag ratio, which is a measure of how much lift the airplane generates compared to how much drag it generates. The lift-to-drag ratio is a good measure of how efficient an airplane is at flying and it's a good way to quantify aerodynamic performance. We want the lift-to-drag ratio to be as high as possible. To achieve that we need to try to maximize lift and minimize drag.

The lift force  $L$  primarily exists because of the pressure difference between the top and the bottom of a wing. The integrals for the lift and drag forces are listed below

(Darak, 2021).

where:

shear stress  $\tau$

pressure  $P$

surface of the wing  $A$

$$F_L = - \int_A (P \sin \theta + \tau_w \cos \theta) dA$$

$$F_D = \int_A (-P \cos \theta + \tau_w \sin \theta) dA$$

We can discreetly approximate these integrals by summing the pressure difference over the surface of the wing:

where  $i$  is a triangle of the wing:

$$F_D = \sum_i (P_i \cos \theta_i + \tau_{w_i} \sin \theta_i) \cdot A_i$$

$$F_L = \sum_i (-P_i \sin \theta_i + \tau_{w_i} \cos \theta_i) \cdot A_i$$

Now that we have a formula for the lift and drag, we can calculate the lift-to-drag ratio by dividing the lift by the drag. This is the fitness score that we will use to evaluate the airplane models.

## 7.2 Genetic Algorithm

This is the final part of the project, where the working fluid simulation and a correct fitness function both converge. Since we never got around to implementing the whole Machine Learning part of the project, we can't really give any specifics on how our genetic algorithm should work. Knowing how to calculate the fitness score is a good start, but the actual population generation and mutation requires some trial and error with the different parameters of the genetic algorithm. What we can talk about is how we would mutate the airplane models. This is the part where the genetic algorithm tweaks the airplane models to try to improve the fitness score.

The idea is to take the vertex data of the airplane model and apply some random mutations to it. This could be anything from changing the position of a vertex to adding a new vertex. The idea is to make small changes to the airplane model and see if the fitness score improves. If it does, we keep the change, if it doesn't we discard it. This is the basic idea behind the genetic algorithm.

## 8 Conclusion

In our paper we tried to answer the following questions:

- How do the different aspects of fluid dynamics work and how do we implement it in a computer program?
- How do we dynamically generate 3D models?
- How does an airplane's wing shape influence its performance?

To answer the first question we looked the different methods that can be used to simulate fluid dynamics and the math behind them. Most CFD models are based on the Navier-Stokes equations. These equations describe the flow of fluid. The two main methods for implementing these equations are: the Lagrangian method and the Eulerian method. We first tried implementing a Lagrangian fluid simulation, in this model fluid is represented by particles. While trying to create this model we ran into some issues and decided to switch to a Eulerian fluid simulation. In a Eulerian fluid simulation the fluid is represented by a grid of cells, where each cell has a velocity and density value. The density solver in our simulation used three steps: adding forces, applying diffusion and moving the densities. The velocity solver used the same steps as the density solver but with an extra step called projection. After having created a basic fluid simulation, we looked at how we could implement fluid behavior when dealing with obstacles. For this, we used the Cut-Cell method.

We weren't able to answer the second question.

To find the answer to the third question we wanted to use a genetic algorithm to evolve the shape of an airplane's wing and analyze the performance of the different shapes in the fluid simulation.

## 9 Discussion: Challenges, Failures & Future Work

Due to the issues with the voxelization process and the object collisions, we weren't actually able to complete the model we wanted. We have a working fluid simulation, but most of the work we did on the obstacles was more theoretical than any tangible working prototype. Here's an outline of what went wrong:

### 9.1 Grid Collision

- **Issue:** air bleeds through obstacles
- **Causes:**
  - Cut-cell method not fully implemented
  - Failure in voxelization process
- **Solution:** it took a lot of trial and error, but no solution was found. We have some hypotheses, but we don't have any time to test, so it's possible that some of the solutions defined in this paper don't work in practice.
- **Result:** a dead end. Most of our project relied on this section working.

### 9.2 Performance

Despite being written in 2006, Mike Ash's isn't the fastest, even with today's hardware. The fluid simulation runs quite reasonably at a resolution, of say  $32 \times 32 \times 32$ , but bumping up the grid size yields worse and worse results. At that grid resolution of  $32^3$ , we see about 20 FPS on my laptop. This isn't abysmal by any means, but  $32^3$  really isn't very high either. So we profiled the program to see what was the slowest.

Samples: 28K of event 'cycles:Pu', Event count (approx.): 31311232985	Overhead	Command	Shared Object	Symbol
54.57%	paper	paper		[.] rlVertex3f
6.87%	paper	paper		[.] rlNormal3f
6.74%	paper	paper		[.] Fluid::lin_solve(FieldType,
4.06%	paper	paper		[.] DrawCube
2.63%	paper	paper		[.] Fluid::advect(FieldType,
2.18%	paper	paper		[.] void fcl::eigen_old<float
1.75%	paper	paper		[.] void std::__introsort_loc
1.36%	paper	paper		[.] ColorFromHSV
1.27%	paper	paper		[.] tryParseDouble
1.24%	paper	paper		[.] is_line_ending
1.10%	paper	paper		[.] main
1.04%	paper	paper		[.] Fluid::get_density(v3)
0.90%	paper	paper		[.] rlMatrixMultiply
0.73%	paper	libc.so.6		[.] 0x000000000016c7a9
0.68%	paper	paper		[.] rlPopMatrix
0.61%	paper	paper		[.] void fcl::detail::getExt
0.61%	paper	paper		[.] void std::__final_inserti
0.61%	paper	paper		[.] Fluid::project(std::vecto
0.61%	paper	libm.so.6		[.] __sqrtf_finite
0.54%	paper	libc.so.6		[.] 0x000000000016c7a4

Figure 10: Profile of the fluid simulation program

Some of the steps involved in the Eulerian fluid simulation require iterating over every cell in the grid, and performing some arithmetic operations on that cell. Both the advection and projection steps require this, and the advection step alone runs four times every step. That's about  $32^3 * 4 = 131\,072$  iterations every step, just for the advection. This is understating the amount of arithmetic operations that are executed, but it already gives an idea of the scale of the problem.

We haven't optimized the code that much, but the most important thing we did was parallelize the these large iterations. Anytime we have any meaningful iteration over the grid, we use [OpenMP](#) to parallelize it. This was a very quick and simple way to get a performance boost, and it works quite well. Still, there's some room for improvement.

We didn't actually go any further but we have some ideas for speed boosts. These are speed boosts and not real algorithmic optimizations because for that we'd have to really look into the Navier-Stokes equations and speed up Jos Stam's algorithms, but that's beyond the scope of what we'd want to do. Rather we're thinking that while we parallelized the long for loops, the code still isn't really asynchronous. OpenMP is a great tool for parallelization, which is what we're using it for, but we could go one step further and refactor our code to be truly asynchronous, and assign a thread to each task, for example.

Another factor is that while the client-side rendering is hardware accelerated, the actual fluid simulation all runs on the CPU. This is quite the bottleneck, as the CPU architecture isn't really designed for this kind of large scale parallel work for high resolution grids. Brute-forcing this would require a lot of compute cores, which CPUs don't have (modern CPUs have between 4 and 16 cores in the consumer market). The most obvious solution is to transfer the load to a dedicated GPU, which we can utilize to perform all these arithmetic operations for us and would yield a massive improvement. Migrating the CFD code to a compute shader or an [OpenCL kernel](#) would be a great way to speed up the simulation. This is something that would be very fun to explore in the future, but once again it's beyond the scope (and time limit) of our project.

### 9.3 Future Work

Considering the amount of ideas that have failed, there's a lot of room for improvement. In the future, there's a very specific order of operations that we'd like to follow to get the project to a working state.

1. Fix the voxelization process
2. Implement a working Cut-Cell (or AMR) method
3. Possible further optimize the fluid simulation
4. Finish with a working genetic algorithm

Once that's all completed, we'd truly have answered all the questions we set out to answer in this paper.

## 10 Bibliography

### References

- Ash, M. (2006). Fluid simulation for dummies. <https://mikeash.com/pyblog/fluid-simulation-for-dummies.html>
- Darak, L. (2021). Flow over a cylinder. <https://skill-lync.com/student-projects/week-2-flow-over-a-cylinder-86>
- d'Lélis, B. (2022). *Semi lagrangian scheme - a visual explanation*. <https://youtu.be/kvBRFxRIJuY>
- Gonkee. (2021). *Coding a fluid simulation with my last 2 brain cells*. <https://youtu.be/uG2mPez44eY>
- Gradience. (2024). *Teaching myself c so i can build a particle simulation*. <https://youtu.be/NorXFOobehY>
- Hakenberg. (2005). Rigid body collision resolution. [https://hakenberg.de/diffgeo/collision/rigid\\_body\\_collision\\_resolution.pdf](https://hakenberg.de/diffgeo/collision/rigid_body_collision_resolution.pdf)
- Ingram, D. M., & Causon, D. (2003). A cartesian cut cell method for incompressible viscous flow. <https://researchgate.net/publication/222518000>
- Lague, S. (2023). *Coding adventure: Simulating fluids*. <https://youtu.be/rSKMYc1CQHE>
- Lift-to-drag ratio. (2024). [https://en.wikipedia.org/wiki/Lift-to-drag\\_ratio](https://en.wikipedia.org/wiki/Lift-to-drag_ratio)
- LinkedIn. (2024). How do you evaluate the performance of different airfoil shapes and configurations. <https://linkedin.com/advice/1/how-do-you-evaluate-performance-different-airfoil>
- Müller, M. (2022). *17 - how to write an eulerian fluid simulator with 200 lines of code*. <https://youtu.be/iKAVRgIrUOU>
- ResolvedAnalytics. (n.d.). Navigating fluid motion: Lagrangian vs. eulerian frames of fluid motion. <https://resolvedanalytics.com/cfd-physics-models/lagrangian-vs-eulerian-phase-models>
- Shiffman, D. (2019). *Coding challenge #132 fluid simulation*. <https://youtu.be/alhpH6ECFvQ>
- Stam, J. (2003). Real-time fluid dynamics for games. [https://dgp.toronto.edu/public\\_user/stam/reality/Research/pdf/GDC03.pdf](https://dgp.toronto.edu/public_user/stam/reality/Research/pdf/GDC03.pdf)
- Tucker, P. (2000). A cartesian cut cell method for incompressible viscous flow. <https://sciencedirect.com/science/article/pii/S0307904X00000056>
- University, E.-R. A. (2024). <https://eaglepubs.erau.edu/introductiontoaerospaceflightvehicles/chapter/airfoil-characteristics/>
- Unknown. (2012). Lattice-botlzmann fluid dynamics. <https://physics.weber.edu/schroeder/javacourse/LatticeBoltzmann.pdf>
- Wikipedia. (2024a). Adaptive mesh refinement. [https://en.wikipedia.org/wiki/Adaptive\\_mesh\\_refinement](https://en.wikipedia.org/wiki/Adaptive_mesh_refinement)
- Wikipedia. (2024b). Computational fluid dynamics. [https://en.wikipedia.org/wiki/Computational\\_fluid\\_dynamics](https://en.wikipedia.org/wiki/Computational_fluid_dynamics)
- Wikipedia. (2024c). Gauss-seidel method. [https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel\\_method](https://en.wikipedia.org/wiki/Gauss%E2%80%93Seidel_method)
- Wikipedia. (2024d). Genetic algorithm. [https://en.wikipedia.org/wiki/Genetic\\_algorithm](https://en.wikipedia.org/wiki/Genetic_algorithm)

Wikipedia. (2024e). Navier-stokes equations. [https://en.wikipedia.org/wiki/Navier%20Stokes\\_equations](https://en.wikipedia.org/wiki/Navier%20Stokes_equations)

Work, P. (2023). *Writing a physics engine from scratch - collision detection optimization.* <https://youtu.be/9IULfQH7E90>