A 3D Implementation of Fluid Simulation Using Laplacian Eigenfunctions in JAX

OGHENEVWOGAGA EBRESAFE, University of Toronto, Canada

1 Introduction

I implemented the paper Fluid Simulation using Laplacian Eigenfunctions [De Witt et al. 2012], which I refer to as *EigenFluids*. Although the method presented in the paper is agnostic to the simulation dimension (2D or 3D), the accompanying implementation is limited to the 2D case, which is considerably simpler than a full 3D implementation. I derived t he necessary details for extending the method to 3D from the accompanying thesis [De Witt 2010a]. My implementation is written in JAX [Bradbury et al. 2018], which greatly simplifies the required linear algebra compared to the original 2D Java implementation provided with the paper [De Witt 2010b]. Also, I use Polyscope for visualization.

2 Underlying Mathematical Details

Here, I split the mathematical details into two subsections. The first subsection presents my intuition for Laplacian eigenfunctions, while the second explains how the fluid dynamics arise from the Navier–Stokes equations.

2.1 Linear Combinations of Laplacian Eigenfunctions

This paper presents an alternative approach to fluid simulation. The key idea is to represent the fluid's velocity field as a linear combination of Laplacian eigenfunctions:

$$u = \sum_{i=1}^{N} \omega_i \Phi_i \tag{1}$$

This is just a linear combination of the eigenfunctions:

$$u = \omega_1 \Phi_1 + \cdots + \omega_N \Phi_N \tag{2}$$

Intuitively, the Laplacian operator Δ measures how much a quantity at a given position differs from its surrounding values on a grid. For a vector field, it can be expressed mathematically as the difference between the gradient of the divergence and the curl of the curl of the field:

$$\Delta = \text{grad}(\text{div } u) - \text{curl}^2 u \tag{3}$$

A Laplacian eigenfunction is then an eigenfunction of this operator:

$$\Delta \Phi_k = \lambda_k \Phi_k \tag{4}$$

Author's Contact Information: Oghenevwogaga Ebresafe, gaga@cs.toronto.edu, University of Toronto, Toronto, Canada.

ACM XXXX-XXXX/2025/12-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Intuitively, each eigenfunction corresponds to a particular mode of "spinning." The weight associated with an eigenfunction indicates how much that mode contributes to the overall motion. When multiple such spinning modes are combined, the result resembles the complex rotational behavior observed in real fluids. This representation also exhibits strikingly artistic characteristics.

The velocity field is expressed as a linear combination of these basis fields, parameterized by a set of weights. Each weight corresponds to a specific Laplacian eigenfunction, and the basis fields themselves are derived from the Laplacian operator.

The divergence equal to zero (the fluid is incomperessible) and the boundary condition at the normal should also be zero:

The velocity field is required to be divergence-free, corresponding to an incompressible fluid, and to satisfy a boundary condition, meaning that its normal component vanishes at the domain boundary. This means that by construction, the eigenfunction statiesfies the desired properties for fluid simulation:

$$\begin{aligned} \text{div}(\Phi_k) &= 0 \\ \Phi_k \cdot \mathbf{n} &= 0 \quad \text{at } \partial D \end{aligned} \tag{5}$$

This implies that the basis eigenfunctions are domain-dependent, and that for certain domains, closed-form expressions for these eigenfunctions exist.

In particular, for a three-dimensional box domain of size $\pi \times \pi \times \pi$, the eigenfunctions can be characterized by the following equations [De Witt 2010a]:

$$\begin{aligned} \Phi_{k,1} = \frac{1}{\lambda_k}\Big(& (k_2^2 + k_3^2)\cos(k_1 x)\sin(k_2 y)\sin(k_3 z)\,\mathbf{a}_x \\ & - k_1 k_2 \sin(k_1 x)\cos(k_2 y)\sin(k_3 z)\,\mathbf{a}_y \\ & - k_1 k_3 \sin(k_1 x)\sin(k_2 y)\cos(k_3 z)\,\mathbf{a}_z\Big) \\ \Phi_{k,2} = \frac{1}{\lambda_k}\Big(& k_1 k_2 \cos(k_1 x)\sin(k_2 y)\sin(k_3 z)\,\mathbf{a}_x \\ & - (k_1^2 + k_3^2)\sin(k_1 x)\cos(k_2 y)\sin(k_3 z)\,\mathbf{a}_y \\ & + k_2 k_3 \sin(k_1 x)\sin(k_2 y)\cos(k_3 z)\,\mathbf{a}_z\Big) \\ \Phi_{k,3} = \frac{1}{\lambda_k}\Big(& - k_3 k_1 \cos(k_1 x)\sin(k_2 y)\sin(k_3 z)\,\mathbf{a}_x \\ & - k_3 k_2 \sin(k_1 x)\cos(k_2 y)\sin(k_3 z)\,\mathbf{a}_y \\ & + (k_1^2 + k_2^2)\sin(k_1 x)\sin(k_2 y)\cos(k_3 z)\,\mathbf{a}_z\Big) \end{aligned} \tag{6}$$

This representation of the vorticity field as a linear combination of eigenfunctions has several beneficial properties.

For instance, the corresponding velocity field can be recovered efficiently from the vorticity field by applying the inverse curl operator:

$$\Phi_k = \text{curl}^{-1}\,\phi_k \tag{7}$$

This contrasts with other methods, in which recovering the velocity field from the vorticity is computationally expensive.

Moreover, the total kinetic energy of the fluid reduces to a simple sum of squares over the basis weights.

2.2 Dynamics via Navier-Stokes

Since the basis functions are expressed in terms of vorticity (with the velocity field recovered afterward), the paper adopts a vorticity-based formulation of the Navier–Stokes equations:

$$\dot{\omega} = \mathrm{Adv}(u, \omega) + \nu \Delta \omega + \mathrm{curl}(f) \tag{8}$$

The equation is projected to the eigenfunction basis to yield:

$$\sum_{k=1}^{N} \dot{\omega}_k \, \phi_k = \sum_{i=1}^{N} \sum_{j=1}^{N} \omega_i \omega_j \, \mathrm{Adv}(\Phi_i, \phi_j) + \nu \sum_{i=1}^{N} \Delta \omega_i \, \phi_i + \mathrm{curl}(f) \tag{9}$$

I will briefly comment on my intuition for each term in the equation. Further details can be found in the paper.

Advection. The advection term is defined as the curl of the product between the weight vector and the velocity field, which is then projected onto the eigenfunction basis. This projection yields a nested summation involving the advection operator applied to pairs of basis eigenfunctions for the velocity and vorticity fields. The interactions between these basis fields are precomputed offline and stored in the so-called *structure coefficient* matrices, denoted C_k.

The resulting expression can therefore be written in closed form, and, in terms of matrix operations, takes the following form:

$$\dot{\omega}_k = \mathbf{w}^\top C_k \mathbf{w} \tag{10}$$

Viscosity. The viscosity term includes the constant ν, which depends on the fluid's physical properties. Importantly, this term involves the Laplacian operator, allowing it to be simplified using the corresponding eigenvalue λ_k, yielding the following contribution:

$$\dot{\omega}_k = \nu \lambda_k \omega_k \tag{11}$$

External Forces. External forces are likewise projected onto the eigenfunction basis. Their combined effect can therefore be summarized as follows:

$$\dot{\omega}_k = f_k \tag{12}$$

Final Time Evolution Equation. The final time evolution equation is then given by the following equation:

$$\dot{\omega}_k = \mathbf{w}^\top C_k \mathbf{w} + \nu \lambda_k \omega_k + f_k \tag{13}$$

Note that the equation is expressed entirely in terms of w, the coefficients associated with each eigenfunction. As a result, at each time step it suffices to solve for these weights, after which the velocity field can be reconstructed as a linear combination of the basis eigenfunctions.

3 Implementation Details

I implemented both the solver and the required precomputation in JAX. I begin by introducing the data structures (and associated types) used to represent the various components of the simulation. I then describe my implementation of the precomputation stage, and finally the details of the fluid solver.

My implementation is informed by the PyTorch-based implementation from the *Scalable EigenFluids* paper [Cui et al. 2018a,b], as well as by the original Java implementation accompanying this paper. To the best of my knowledge (I couldn't find anyone online), no prior 3D implementation of this method exists, which made the implementation particularly challenging.

3.1 Data structure representations

This section describes the data structures used in my implementation. In existing implementations, including the original Java version, I found that the required data components were not sufficiently explicit about their types or shapes. To address this, I use `jax-typing` to precisely annotate the shapes of all components involved in the computation. This explicitness significantly improved my understanding of the system.

We begin with the basis coefficients ω. These are represented as a vector of length N, where N denotes the size of the eigenfunction basis.

```
basis_coef: Float64[Array, "N"]
```

The force is also represented in a simlar way:

```
force_coef: Float64[Array, "N"]
```

The velocity field is represented as a tensor indexed by the spatial coordinates x, y, and z, with each entry yielding a three-dimensional column vector, as expected for a 3D velocity field:

```
velocity_field: Float64[Array, "mx my mz 3"]
```

So far, we have described the *dynamic* data structures—that is, those whose values may change at each time step. We now turn to the precomputed components.

The eigenvalues, λ_k, are presented as follows:

```
eigenvalues: Float64[Array, "N"]
```

Finally the velocity basis fields fields and the strucure coefficient matrix are represented as follows:

```
velocity_basis_fields: Float64[Array, "N mx my mz 3"]
Ck: Float64[Array, "N N N"]
```

3.2 Precomputation

The first components we precompute is the vector wave numbers and the eigenvalues. I define the following functions:

```
def precompute_wave_numbers(self) -> None:

def precompute_eigenvalues(self) -> None
```

For the vector wave numbers, I select integer values up to the cube root of the domain size. The corresponding eigenvalue is given by the sum of the squares of these components, and the eigenvalues are sorted in increasing order.

I also precompute the velocity basis fields. This function implements the closed-form solution for the velocity eigenfunctions on a three-dimensional box domain, as described in [De Witt 2010a].

```
def closed_form_velocity_basis(wave_ns: Int32[Array, "4"
    ],
  x: Float64,
  y: Float64,
  z: Float64,
) -> Float64[Array, "3"]
```

Another function tabulates the data stored into the `velocity_basis_fields` field introduced earlier:

```
def precompute_velocity_basis_fields(self) -> None
```

Since the vorticity can be obtained by applying the curl operator, I compute the required vorticity components on-the-fly as the curl of the velocity basis fields. This computation is performed during the precomputation of the structure coefficient matrices. While the entries of these matrices admit closed-form expressions in some cases, I did not find a closed-form solution for the 3D domain. Thus, I instead use a numerical computation, as described in Algorithm 2 of the paper.

```
def precompute_Ck(self) -> None
```

3.3 Fluid Solver

The solver follows Algorithm 1 in the paper and uses explicit Euler time integration. Since the implementation is written in JAX, which provides convenient linear algebra primitives, the solver can be implemented almost verbatim from the paper. For example, the following snippet illustrates part of the implementation:

```
# Matrix vector product
w_dot = jnp.einsum('i, kij, j -> k', w, self.Ck, w)

# Explicit Euler integration
w_adv = w + w_dot * self.dt

# Calculate energy after time step
e2 = jnp.sum(w_adv**2)
```

Note that the comments are taken verbatim from the paper, and the code closely mirrors Algorithm 1.

4 Running the Code

The code artifacts consist of four files: `eigenfluids.py`, `eigenfluids_cuboid.py` `cube.py`, and `cuboid.py`. The file `eigenfluids.py` contains the numerical computations required for the fluid simulation and is implemented in JAX for the cube. The files `cube.py` and `cuboid.py` implement the first and second scenes, respectively, and make use of the Python bindings for Polyscope.

4.1 Scenes

Gaseous fluid in a cube. The first scene is a gaseous fluid in a cube (i.e square box). This is shown in Figure 1.

Gaseous fluid in a cuboid. The second scene simulates the same gaseous fluid within a cuboid (i.e., a rectangular box). I was unable to find closed-form solutions for eigenfunctions on other types of domains.

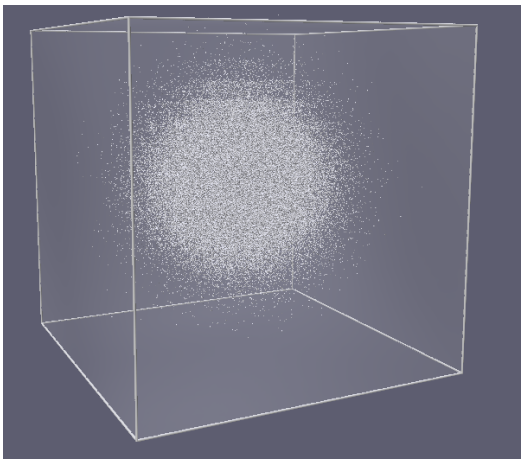

Fig. 1. Gaseous fluid simulation in a cube.

4.2 What to expect

When the code is executed, it opens a Polyscope window that displays the simulation. A checkbox is provided to run the simulation, and an checkbox applies random external forces.

References

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs.* http://github.com/jax-ml/jax

Qiaodong Cui, Pradeep Sen, and Theodore Kim. 2018a. *PyTorch Implementation of Scalable Laplacian Eigenfluids.* https://github.com/bobarna/eigenfluid-control

Qiaodong Cui, Pradeep Sen, and Theodore Kim. 2018b. Scalable laplacian eigenfluids. *ACM Trans. Graph.* 37, 4, Article 87 (July 2018), 12 pages. doi:10.1145/3197517.3201352

Tyler De Witt. 2010a. *Fluid Simulation in Bases of Laplacian Eigenfunctions.* Master's thesis. University of Toronto, Toronto, ON, Canada.

Tyler De Witt. 2010b. Java Implementation of Fluid simulation using Laplacian eigenfunctions. https://www.dgp.toronto.edu/~tyler/fluids/. Accessed: 2025-12-20.

Tyler De Witt, Christian Lessig, and Eugene Fiume. 2012. Fluid simulation using Laplacian eigenfunctions. *ACM Trans. Graph.* 31, 1, Article 10 (Feb. 2012), 11 pages. doi:10.1145/2077341.2077351