# An Introduction to Classical & Quantum CFD for Navier Stokes Equations

Faisal Shaik

Principal Investigator: Prof. Wagih Ghobriel
Supervising Professor: Prof. Dennis Fernandes

Dept. of Chemical & Physical Sciences
University of Toronto

December 5 , 2025

---

**Abstract**

The Navier-Stokes equations are the holy grail of computational fluid dynamics , governing everything from aerospace flight vehicle design , weather forecasting , plasma magneto-hydrodynamics , and astrophysics. The problem is that **these nonlinear partial differential equations are computationally intractable for classical computers** when dealing with cases such as turbulent flows at high Reynolds numbers , multi-scale phenomena from molecular to macroscopic scales , real-time applications like weather prediction or flight control. A quantum approach to computing flows of a Navier–Stokes fluid may prove to be less computationally intensive , providing the possibility for potential speedups when used in conjunction with classical methods. In this report , we investigate the Navier-Stokes equations , present classical computational algorithms implemented in CUDA , establish the fundamentals of quantum computing , and explore quantum algorithms with improved computational complexity.

---

**Acknowledgements**

Hi.

# Contents

# 1 Introduction to Fluid Mechanics

To begin solving the Navier-Stokes equations , we must first understand what they are. It would be no understatement to call them the heart of fluid mechanics. They apply Newton's second law ($F = ma$) to fluids by describing how the velocity field evolves over time. Given the velocity and pressure distribution throughout a fluid at one moment , plus boundary conditions , these equations predict how the flow will evolve.

The Navier-Stokes equations generally come in a pair of two. The *momentum equation* , which essentially reads "$a = F/m$" , and the *continuity equation* , which enforces that mass must be conserved. For the sake of simplicity , we first introduce these two equations for *incompressible fluids*—fluids that cannot be shrunk down into a smaller volume under pressure.

We will first just state the equations and take a look at them before explaining them. The continuity equation (for incompressible fluids) reads as follows ,

$$\nabla \cdot \mathbf{u} = 0 \tag{1}$$

Where $\mathbf{u}$ is the velocity field of the flow at some point in space and time. Hence , $\nabla \cdot \mathbf{u}$ gives the divergence of $\mathbf{u}$ in space. Likewise , the momentum equation (for incompressible fluids) is ,

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \frac{1}{\rho}\nabla \cdot \boldsymbol{\tau} + \mathbf{f} \tag{2}$$

Where $t$ is time , $\rho$ is the constant fluid density , $p$ is the pressure field , $\boldsymbol{\tau}$ is the viscous stress tensor , and $\mathbf{f}$ represents body forces per unit mass (e.g. , gravity).

## 1.1 Deriving Navier-Stokes Equations

Before any further discussion , we first clarify notation. Let $\mathbf{x}$ denote the spatial coordinates of the system. Typically the fluid lives in 3-dimensions , in which case $\mathbf{x}$ would be a free variable in $\mathbb{R}^3$. Then , whenever we use $\nabla$ with some function $F$ of space (and possibly more) , $\nabla F$ just denotes the Jacobian matrix $\frac{\partial F}{\partial \mathbf{x}}$. For example , $\nabla \mathbf{u}$ is nothing but the Jacobian matrix $\frac{\partial \mathbf{u}}{\partial \mathbf{x}}$. Moreover , every vector quantity as written in equations can be assumed to be a column vector. We denote the $i^{\text{th}}$ coordinate of some vector field or point $p$ as $p_i$. For example , in $\mathbb{R}^3$ we have

$$\mathbf{x} = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \quad \text{and} \quad \mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix}$$

and this gives us the Jacobian matrix ,

$$\nabla \mathbf{u} = \begin{pmatrix} \frac{\partial u_1}{\partial x_1} & \frac{\partial u_1}{\partial x_2} & \frac{\partial u_1}{\partial x_3} \\ \frac{\partial u_2}{\partial x_1} & \frac{\partial u_2}{\partial x_2} & \frac{\partial u_2}{\partial x_3} \\ \frac{\partial u_3}{\partial x_1} & \frac{\partial u_3}{\partial x_2} & \frac{\partial u_3}{\partial x_3} \end{pmatrix}$$

### 1.1.1 Continuity Equation

The divergence $\nabla \cdot \mathbf{u}$ measures how much the fluid "escapes" at a point in space. For example , in 3-dimensions this would be $\nabla \cdot \mathbf{u} = \frac{\partial u_1}{\partial x_1} + \frac{\partial u_2}{\partial x_2} + \frac{\partial u_3}{\partial x_3}$. If we have $\nabla \cdot \mathbf{u} > 0$ at some point, this would mean that more fluid is leaving that point then entering.

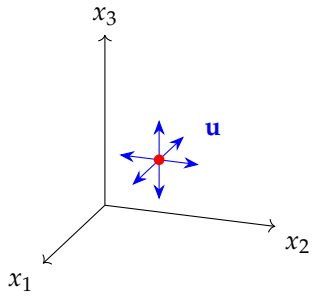


Figure 1: Divergence at a point where $\nabla \cdot \mathbf{u} > 0$: fluid is leaving the point.

Then either the fluid must be "expanding" in the volume it takes , or the point is a "source" where new fluid is being created. Incompressible fluid cannot shrink or expand , which means that $\nabla \cdot \mathbf{u} > 0$ must imply a source. But our system is a regular confined space , so we cannot have a source—that would mean we are making matter out of nothing.

Conversely , if $\nabla \cdot \mathbf{u} < 0$ , more fluid is entering than leaving , creating a "sink". Hence , for incompressible fluids , we require $\nabla \cdot \mathbf{u} = 0$ everywhere , meaning there are no sources or sinks.

### 1.1.2 Momentum Equation

We can show that the momentum equation (2) is just $a = F/m$ for fluids by first: mathematically deriving the left hand side (LHS) from acceleration $\frac{D\mathbf{u}}{Dt}$ , and second: making an empircal argument for the right hand side (RHS) being $F/m$.

Before proceeding , we must first introduce a new perspective of studying fluids. Up until now , we have been using the *Eulerian perspective* , where $\mathbf{x}$ represents a fixed point in space and we observe how fluid properties (like velocity and pressure) change at that location over time. But what if we instead track an individual fluid element as it moves through space ?

How does the position of a specific point in our fluid evolve over time ?

This alternative viewpoint is known as the *Lagrangian perspective*. Here , we follow a single "particle" as it moves around in our fluid—though strictly speaking , we really mean an infinitesimal fluid element. The position of this fluid element becomes a function of time , $\mathbf{x}(t)$. In 3-dimensions , the position is a differentiable function $\mathbf{x} : \mathbb{R} \to \mathbb{R}^3$. This perspective allows us to track the velocity of an individual particle as it evolves through time via the function composition $\mathbf{u}(\mathbf{x}(t), t)$.

The key difference: in the Eulerian view , we ask "what is the velocity at this fixed location ?" , while in the Lagrangian view , we ask "what is the velocity of this specific fluid element ?"

Since velocity $\mathbf{u} : \mathbb{R}^3 \times \mathbb{R} \to \mathbb{R}^3$ is a function of position $\mathbf{x}$ and time $t$ , and $\mathbf{x}$ itself is also a position of time , we get acceleration as

$$\begin{aligned}\frac{D\mathbf{u}}{Dt} &= \frac{\partial \mathbf{u}}{\partial t} + \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \frac{d\mathbf{x}}{dt} \\ &= \frac{\partial \mathbf{u}}{\partial t} + \left( \frac{\partial \mathbf{u}}{\partial \mathbf{x}} \right) \mathbf{u}\end{aligned}$$

The above is derived from an application of chain rule then rewriting $\frac{d\mathbf{x}}{dt}$ as velocity $\mathbf{u}$. By *Lemma 1* (see Appendix A) , we can write

$$\frac{D\mathbf{u}}{Dt} = \frac{\partial \mathbf{u}}{\partial t} + \left( \mathbf{u} \cdot \frac{\partial}{\partial \mathbf{x}} \right) \mathbf{u}$$

Here , $\left( \mathbf{u} \cdot \frac{\partial}{\partial \mathbf{x}} \right) = \left( u_1 \frac{\partial}{\partial x_1} + u_2 \frac{\partial}{\partial x_2} + u_3 \frac{\partial}{\partial x_3} \right)$ is a function operator. Using $\nabla$ notation , we finally yield

$$\frac{D\mathbf{u}}{Dt} = \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla) \mathbf{u} \tag{3}$$

This is exactly the LHS of equation (2). We now argue that the RHS of equation (2) is equal to $F/m$ , where $F$ is every force acting on the fluid at same point. There is no mathematical argument for showing this—it is purely empirical , based on experimentation and other empirically backed theories of physics.

We can split our argument to account for two kinds of accelerations the fluid experiences: acceleration from internal forces the fluid imparts on itself $a_{int}$ , and any acceleration imparted by external forces $a_{ext}$. Then it follows ,

$$a = F/m = a_{ext} + a_{int}$$

Note that the notion of external acceleration is entirely vague ! It depends on the context of the situation. There may be a gravitational field , centripetal acceleration , or something else that imparts acceleration on our fluid—we cannot characterize it further for the general case. For this reason , we just define $\mathbf{f} := a_{ext}$ as all the external force applied per unit mass of the fluid. For example , in many contexts , a system only concerns itself with gravitational acceleration , in which case we would just have $\mathbf{f} = g$.

A more interesting analysis arises when considering internal forces the fluid imparts on itself. Since the acceleration is only local to each point , we have $a_{int} = F_{int}/\rho$ , where $F_{int}$ measures the internal forces the fluid imparts on itself at a point in space and time. The internal force can be decomposed into two distinct mechanisms by which fluid elements interact with one another:

$$F_{int} = F_p + F_\tau$$

where $F_p$ represents pressure forces and $F_\tau$ represents viscous shear stress forces. These two forces arise from fundamentally different physical phenomena.

Pressure is a thermodynamic property that emerges from the statistical mechanics of molecular motion. At the microscopic level , a fluid consists of countless molecules in constant , random motion. These molecules continuously collide with one another and with any surfaces they encounter. Each individual collision imparts a tiny impulse , and when we average over the enormous number of collisions occurring per unit time , we observe a macroscopic force distributed over area—this is pressure. Crucially , pressure acts isotropically , meaning it pushes equally in all directions. If we imagine a small cubic fluid element , the pressure exerts forces perpendicular to each of its faces. When pressure varies spatially , a net force emerges. Consider a fluid element where pressure is higher on one side than the other—the element experiences a greater force from the high-pressure side , resulting in acceleration toward the low-pressure region. Mathematically , this net pressure force per unit volume is given by $-\nabla p$ (see *Derivation 1* in Appendix B). The negative sign indicates that fluid accelerates from high pressure to low pressure , opposing the pressure gradient. Dividing by density $\rho$ to obtain force per unit mass , we have

$$a_p = \frac{F_p}{\rho} = -\frac{1}{\rho} \nabla p$$

This term appears in the momentum equation and captures how spatial variations in pressure drive fluid motion.

While pressure acts normal to surfaces , viscous stresses act tangentially , resisting the relative motion

between adjacent fluid layers. Viscosity is an internal friction that arises from momentum transfer between molecules moving at different velocities. When neighboring fluid layers slide past each other , faster-moving molecules diffuse into slower-moving regions and vice versa , exchanging momentum and creating a drag force that opposes the velocity difference. The viscous stress is characterized by the stress tensor $\boldsymbol{\tau}$ , which encodes how forces are distributed across different orientations within the fluid. For a Newtonian fluid—one where stress is proportional to the rate of strain—the stress tensor depends on the velocity gradient $\nabla \mathbf{u}$. Specifically ,

$$\boldsymbol{\tau} = \mu \left( \nabla \mathbf{u} + (\nabla \mathbf{u})^T \right) + \lambda (\nabla \cdot \mathbf{u}) \mathbf{I}$$

where $\mu$ is the dynamic viscosity , $\lambda$ is the second viscosity coefficient , and $\mathbf{I}$ is the identity matrix. The first term represents shear stresses , while the second accounts for bulk viscosity effects during compression or expansion. For incompressible fluids , $\nabla \cdot \mathbf{u} = 0$ by the continuity equation , so the bulk viscosity term vanishes. The net viscous force per unit volume on a fluid element is given by the divergence of the stress tensor , $\nabla \cdot \boldsymbol{\tau}$. This divergence measures how stress varies spatially , yielding a net force. Dividing by density , we obtain the viscous acceleration:

$$a_\tau = \frac{F_\tau}{\rho} = \frac{1}{\rho} \nabla \cdot \boldsymbol{\tau}$$

Combining both pressure and viscous contributions , the total internal acceleration is

$$a_{int} = -\frac{1}{\rho} \nabla p + \frac{1}{\rho} \nabla \cdot \boldsymbol{\tau}$$

Together with the external forces $\mathbf{f}$ and the material derivative we derived earlier , we arrive at the complete momentum equation (2).

### 1.2 Why Do We Care ?

The Navier-Stokes equations are not merely an academic curiosity—they are the mathematical foundation governing nearly every fluid flow phenomenon we encounter in nature and engineering. Understanding and solving these equations is essential for predicting , controlling , and optimizing fluid behavior across countless applications.

From an engineering perspective , the ability to accurately solve the Navier-Stokes equations translates directly into technological advancement. In aerospace engineering , computational fluid dynamics (CFD) enables the design of more efficient aircraft with reduced drag and improved fuel economy. Numerical solutions allow engineers to explore thousands of design variations virtually before building a single prototype. Similarly , the design of gas turbines , rocket engines , wind turbines , and propulsion systems all rely fundamentally on our ability to solve these equations under extreme conditions.

The equations also play a central role in environmental science and climate modeling. Weather forecasting relies on solving the atmospheric Navier-Stokes equations coupled with thermodynamics. Every time we check tomorrow's forecast , we are relying on computational solutions to these equations running on supercomputers. Climate models , which predict long-term changes in Earth's climate system , similarly depend on accurate representations of atmospheric and oceanic fluid flows.

Perhaps most fundamentally , the Navier-Stokes equations present one of the deepest unsolved problems in mathematics. The Clay Mathematics Institute has designated the question of whether smooth solutions always exist as one of the seven Millennium Prize Problems. Turbulence , which emerges naturally from the Navier-Stokes equations , remains poorly understood theoretically despite being ubiquitous in nature. The computational cost of direct numerical simulation scales roughly as $Re^{9/4}$ in three dimensions , where $Re$ is the Reynolds number. For many practical flows with $Re \sim 10^6$ or higher , direct simulation is completely infeasible even on the most powerful supercomputers—motivating the search for alternative methods , including quantum algorithms that could offer exponential speedups.

### 1.3 Solving Navier-Stokes Equations

Before discussing how to solve the Navier-Stokes equations , we must first clarify what it means to "solve" a partial differential equation (PDE). Unlike ordinary differential equations , where the unknown is a function of a single variable , PDEs involve functions of multiple variables—in our case , velocity $\mathbf{u}(\mathbf{x}, t)$ and pressure $p(\mathbf{x}, t)$ depend on both spatial coordinates $\mathbf{x}$ and time $t$. To solve a PDE means to find an explicit function (or family of functions) that satisfies the equation everywhere in the domain and for all time , subject to given initial and boundary conditions.

A *general solution* to a PDE would be a formula expressing the solution in terms of the problem parameters—initial conditions , boundary conditions , and physical constants like $\rho$ and $\mu$. For simple PDEs like the heat equation or wave equation , such general solutions exist and can be written using techniques like separa-

tion of variables , Fourier series , or Green's functions. However , for the Navier-Stokes equations , **no general solution exists**. The fundamental obstacle is the nonlinear convective term $(\mathbf{u} \cdot \nabla)\mathbf{u}$—this term couples the velocity field to itself in a way that prevents the use of superposition principles or other linear techniques.

To illustrate the nature of solving the Navier-Stokes equations , consider a simplified 2D example: flow between two infinite parallel plates separated by distance $h$ , where the top plate moves with constant velocity $U$ and the bottom plate is stationary. We assume the flow is steady (time-independent) , fully developed (no variation in the flow direction) , and has no pressure gradient. This is known as *plane Couette flow*.

In 2D , with coordinates $(x_1, x_2)$ , we assume the velocity field has the form $\mathbf{u} = (u_1(x_2), 0)$—flow only in the $x_1$ direction , varying only with height $x_2$. The continuity equation becomes

$$\frac{\partial u_1}{\partial x_1} = 0$$

which is automatically satisfied since $u_1$ depends only on $x_2$. The momentum equation in the $x_1$ direction simplifies dramatically. Since $\frac{\partial \mathbf{u}}{\partial t} = 0$ (steady flow) and $(\mathbf{u} \cdot \nabla)\mathbf{u} = 0$ (the nonlinear term vanishes for this particular geometry) , we are left with

$$0 = \frac{\mu}{\rho} \frac{\partial^2 u_1}{\partial x_2^2}$$

assuming $\nabla p = 0$ and no body forces. This reduces to $\frac{d^2 u_1}{dx_2^2} = 0$ , which integrates to give $u_1(x_2) = Ax_2 + B$ for constants $A$ and $B$. Applying boundary conditions $u_1(0) = 0$ and $u_1(h) = U$ yields $B = 0$ and $A = U/h$ , giving the linear velocity profile

$$u_1(x_2) = \frac{U}{h} x_2$$

This is a rare case where we obtain an exact , closed-form solution. But notice how special the setup was: steady flow , simple geometry , no pressure gradients , and crucially , the nonlinear term vanished. For almost any other flow configuration—flow past a cylinder , turbulent pipe flow , atmospheric circulation—the nonlinear term dominates and analytical solutions become impossible.

For more complicated flows , such as turbulent flow over an airfoil or mixing in a combustion chamber , the situation becomes intractable. Turbulent flows exhibit chaotic motion across a vast range of spatial scales , from large energy-containing eddies down to tiny dissipative structures where viscosity converts kinetic energy to heat. To resolve all these scales in a direct numerical simulation (DNS) would require a computational grid with spacing proportional to the *Kolmogorov length scale* $\eta \sim Re^{-3/4}$ , where $Re$ is the Reynolds number. Since we need to resolve a 3D domain , the total number of grid points scales as $Re^{9/4}$. For a typical aerodynamic flow with $Re \sim 10^6$ , this would require roughly $10^{13}$ grid points—far beyond current computational capabilities.

Because exact solutions do not exist and direct simulation is computationally prohibitive , we instead rely on *approximate numerical methods*. These methods discretize the continuous equations in space and time , converting the PDE into a system of algebraic equations that can be solved on a computer. Different discretization strategies give rise to different numerical methods , each with their own advantages and trade-offs. In this paper , we will explore three classical approaches: the **Finite Volume Method** , which discretizes the integral form of the conservation laws and is widely used in engineering CFD , the **Spectral Methods** , which represent the solution as a sum of global basis functions and achieve high accuracy for smooth flows , and the **Lattice Boltzmann Method** , which simulates fluid flow by tracking particle distributions on a lattice. Each method transforms the intractable continuous problem into a finite-dimensional approximation that can be solved numerically—trading mathematical exactness for computational feasibility.

### 1.4 Navier-Stokes for Compressible Fluids

Up until now , we have restricted our attention to incompressible fluids—fluids whose density $\rho$ remains constant throughout space and time. This assumption is valid for liquids like water and for gas flows at low speeds (typically when flow velocities are much less than the speed of sound). However , many important applications involve *compressible flows* where density variations cannot be neglected: high-speed aerodynamics , gas dynamics in engines , shock waves , and atmospheric flows at large scales.

For compressible fluids , the density $\rho = \rho(\mathbf{x}, t)$ becomes a field variable that varies in space and time. This fundamentally changes the continuity equation. Mass conservation for a compressible fluid is expressed by

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{4}$$

This equation states that the rate of change of density at a point plus the divergence of mass flux $\rho\mathbf{u}$ must equal zero—mass cannot be created or destroyed. Expanding the divergence term using the product rule

gives

$$\frac{\partial \rho}{\partial t} + \mathbf{u} \cdot \nabla \rho + \rho \nabla \cdot \mathbf{u} = 0$$

Notice that for incompressible flow where $\rho$ is constant , $\frac{\partial \rho}{\partial t} = 0$ and $\nabla \rho = 0$ , which immediately recovers $\nabla \cdot \mathbf{u} = 0$ as we had before. However , for compressible flow , $\nabla \cdot \mathbf{u} \neq 0$ in general—the velocity field can have sources and sinks corresponding to local compression or expansion of the fluid.

The momentum equation for compressible flow takes a similar form , but we must account for the fact that density is no longer constant. The most general form is

$$\rho \left( \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} \right) = -\nabla p + \nabla \cdot \boldsymbol{\tau} + \rho \mathbf{f} \qquad (5)$$

The key difference from equation (2) is that density $\rho$ now appears as a coefficient that varies in space and time , rather than being factored out as a constant. The viscous stress tensor $\boldsymbol{\tau}$ for a compressible fluid also changes slightly:

$$\boldsymbol{\tau} = \mu \left( \nabla \mathbf{u} + (\nabla \mathbf{u})^T \right) + \left( \lambda - \frac{2}{3}\mu \right) (\nabla \cdot \mathbf{u}) \mathbf{I}$$

where the bulk viscosity term no longer vanishes since $\nabla \cdot \mathbf{u} \neq 0$. This term accounts for viscous stresses arising from volumetric expansion or compression.

The compressible Navier-Stokes equations form a coupled system: equations (4) and (5) contain four unknowns ($\rho$, $\mathbf{u}$, and $p$) but only provide four equations. To close the system , we need an additional equation relating pressure , density , and temperature—the *equation of state*. For an ideal gas , this is $p = \rho R T$ where $R$ is the specific gas constant and $T$ is temperature. In practice , we also need an energy equation to track temperature evolution , making compressible flow simulation significantly more complex than the incompressible case.

### 1.5 Navier-Stokes for Newtonian & Non-Newtonian Fluids

Thus far , we have implicitly assumed a specific relationship between the viscous stress tensor $\boldsymbol{\tau}$ and the velocity gradient $\nabla \mathbf{u}$. This relationship defines what is known as a *Newtonian fluid*—a fluid where the stress is linearly proportional to the rate of strain. However , not all fluids obey this simple linear relationship , and understanding the distinction between Newtonian and non-Newtonian behavior is crucial for modeling many real-world flows.

A **Newtonian fluid** is one in which the viscous stress tensor depends linearly on the rate of deformation. The defining characteristic is that the viscosity $\mu$ remains constant regardless of the applied shear rate. Common examples include water , air , most gases , simple oils , and alcohols. For these fluids , doubling the shear rate doubles the shear stress. Mathematically , the stress tensor for a Newtonian fluid is given by

$$\boldsymbol{\tau} = \mu \left( \nabla \mathbf{u} + (\nabla \mathbf{u})^T \right) + \lambda (\nabla \cdot \mathbf{u}) \mathbf{I}$$

For incompressible Newtonian fluids , where $\nabla \cdot \mathbf{u} = 0$ , this simplifies to

$$\boldsymbol{\tau} = \mu \left( \nabla \mathbf{u} + (\nabla \mathbf{u})^T \right)$$

Substituting this into the momentum equation and using the identity $\nabla \cdot \left( \nabla \mathbf{u} + (\nabla \mathbf{u})^T \right) = \nabla^2 \mathbf{u} + \nabla (\nabla \cdot \mathbf{u})$ , along with $\nabla \cdot \mathbf{u} = 0$ , we obtain the familiar incompressible Navier-Stokes momentum equation:

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho} \nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f} \qquad (6)$$

where $\nu = \mu / \rho$ is the kinematic viscosity and $\nabla^2 \mathbf{u}$ is the Laplacian of the velocity field. This is the classical form most commonly seen in fluid mechanics textbooks. It is the most simplest general form of the momentum equation.

For compressible Newtonian fluids , the stress tensor retains the bulk viscosity term , giving

$$\boldsymbol{\tau} = \mu \left( \nabla \mathbf{u} + (\nabla \mathbf{u})^T \right) + \left( \lambda - \frac{2}{3}\mu \right) (\nabla \cdot \mathbf{u}) \mathbf{I}$$

and the momentum equation becomes equation (5) from the previous section.

In contrast , **non-Newtonian fluids** exhibit a nonlinear relationship between stress and strain rate. The effective viscosity can depend on the shear rate itself , the history of deformation , or even the duration of applied stress. These fluids are ubiquitous in everyday life and industrial processes. Examples include:

- *Shear-thinning fluids* (pseudoplastic): Viscosity decreases with increasing shear rate. Examples include ketchup , paint , blood , and polymer solutions. These fluids flow more easily when stirred or shaken.
- *Shear-thickening fluids* (dilatant): Viscosity increases with increasing shear rate. The classic

example is oobleck—a cornstarch and water mixture that behaves like a liquid under gentle stirring but solidifies under rapid impact. Other examples include some clay slurries and quicksand.

- *Bingham plastics*: These fluids behave as solids below a critical yield stress but flow like viscous fluids above it. Examples include toothpaste , mayonnaise , and drilling mud.
- *Viscoelastic fluids*: These exhibit both viscous and elastic properties , showing memory effects where the stress depends on the deformation history. Examples include silly putty , molten polymers , and biological fluids like saliva.

For non-Newtonian fluids , the stress tensor $\boldsymbol{\tau}$ cannot be expressed as a simple linear function of $\nabla \mathbf{u}$. Instead , we write the momentum equation in its most general form:

$$\rho \left( \frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} \right) = -\nabla p + \nabla \cdot \boldsymbol{\tau}(\nabla \mathbf{u}, \dot{\gamma}, \ldots) + \rho \mathbf{f} \tag{7}$$

where $\boldsymbol{\tau}$ is now a more complex functional of the velocity gradient , the shear rate $\dot{\gamma}$ , and potentially other variables like deformation history. Specific constitutive models—such as the power-law model , Carreau model , or Maxwell model—provide particular forms for $\boldsymbol{\tau}$ depending on the fluid's rheological behavior.

It is important to recognize the hierarchical relationship between these formulations. The general non-Newtonian momentum equation (7) applies to *all* fluids , with Newtonian fluids being a special case where $\boldsymbol{\tau}$ happens to be linear in $\nabla \mathbf{u}$. Similarly , the compressible Navier-Stokes equations apply to all fluids , with incompressible flow being a special case where $\rho$ is constant. The incompressible Newtonian Navier-Stokes equation (6) thus represents a doubly-specialized case: both incompressible *and* Newtonian. While this special case covers many important flows , the full generality of equation (7) is necessary for modeling the vast array of complex fluids encountered in nature and industry.

## 2 Classical Methods

In this section , we explore classical computational approaches for solving the Navier-Stokes equations numerically. For simplicity and clarity , we will focus primarily on **incompressible Newtonian fluids**—the case described by equations (1) and (6). This restriction covers a wide range of practical applications including water flow , air flow at low speeds , and many industrial processes , while avoiding the additional complexity of variable density and non-Newtonian rheology. The methods we discuss can be extended to compressible and non-Newtonian cases , but the fundamental algorithmic principles are most clearly illustrated in the incompressible Newtonian regime. With this foundation , we now examine three major classes of numerical methods: the Finite Volume Method , Spectral Methods , and the Lattice Boltzmann Method.

### 2.1 Introduction to Parallel Programming

The numerical methods we are about to explore—Finite Volume , Spectral , and Lattice Boltzmann—all share a common computational structure: they discretize the continuous fluid domain into a finite set of degrees of freedom (grid points , spectral coefficients , or lattice nodes) and then perform repetitive calculations at each location. For a 3D simulation with $N$ points per dimension , we might have $N^3 \sim 10^6$ to $10^9$ computational nodes , and at each time step , we must update velocities , pressures , and other quantities at every single node. Moreover , turbulent flows or high Reynolds number simulations may require thousands or millions of time steps to reach steady state or capture transient dynamics. This creates a massive computational burden that sequential processing—executing one operation at a time—cannot handle in reasonable time.

The key observation is that many of these computations are *independent*: updating the velocity at grid point $(i, j, k)$ often requires only information from nearby neighbors , not from the entire domain. This independence suggests a natural parallelization strategy: why not compute updates for multiple grid points simultaneously ? This is the essence of **parallel computing**—performing many operations at once rather than one after another.

The most powerful paradigm for this type of computation is **SIMD** (Single Instruction , Multiple Data). In SIMD , the same operation is applied to many different pieces of data in parallel. For example , if we need to add viscous diffusion terms to velocity values at a million grid points , SIMD allows us to execute the same "compute diffusion" instruction on thousands of points simultaneously. This is in stark contrast to traditional CPU architecture , where a few cores execute different instructions on different data.

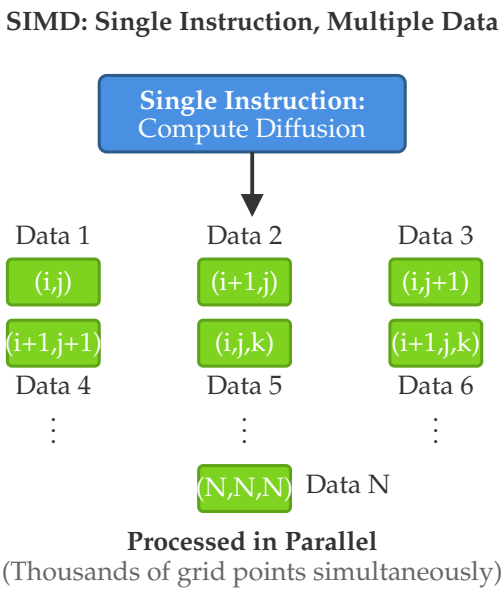


Figure 2: SIMD architecture: a single instruction operates on multiple data elements in parallel.

Modern **Graphics Processing Units (GPUs)** are specifically designed for SIMD computation. Originally built to render millions of pixels in parallel for computer graphics , GPUs have evolved into general-purpose parallel processors capable of accelerating scientific computing. A typical GPU contains thousands of small processing cores , each capable of executing the same instruction on different data. While an individual GPU core is much slower than a CPU core , the sheer number of cores operating in parallel provides enormous computational throughput for data-parallel problems like CFD simulations.

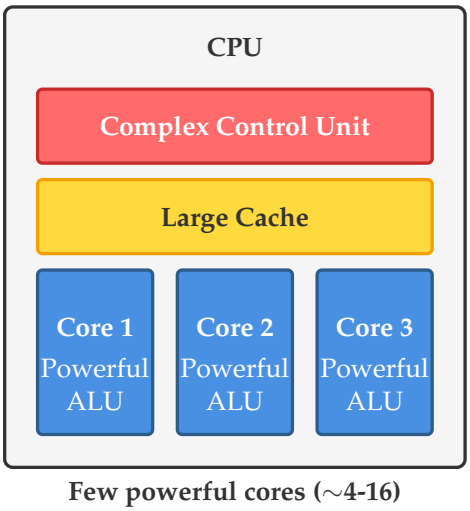


Figure 3: High level view of CPU architecture: CPUs have few powerful cores optimized for sequential processing.

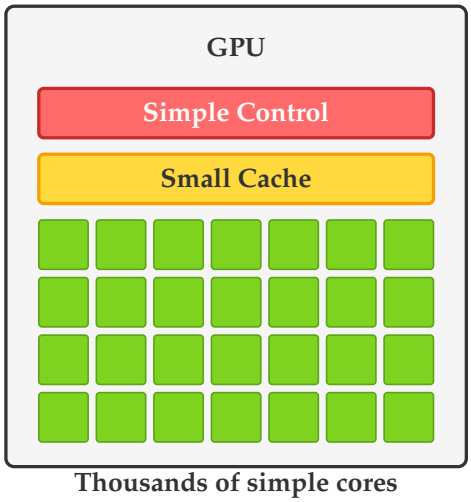


Figure 4: High level view of GPU architecture: GPUs have thousands of simpler cores for parallel computation.

To understand how GPUs achieve this parallelism , we need to examine their architecture more closely. A GPU organizes computation into a hierarchy of **threads**. A thread is the smallest unit of execution—a single instance of a program running on one core. Thousands of threads execute simultaneously on a GPU , each processing a different piece of data. Threads are grouped into **blocks** , and blocks are organized into a **grid**. This hierarchy allows programmers to map computational problems naturally: for a 3D fluid simulation , we might create a 3D grid of thread blocks , where each thread computes the update for one grid point. See Figure 11 in Appendix C for visualization.

GPU programming typically involves two distinct code regions: **host code** and **device code**. The host is the CPU , which manages the overall program flow , allocates memory , and launches GPU computations. The device is the GPU , which executes the parallel computations. Data must be explicitly transferred between host memory (RAM) and device memory (GPU memory). The host code launches **kernels**—functions that run on the GPU—and specifies how many threads should execute the kernel and how they should be organized into blocks and grids.

Let us illustrate this with a simple example: computing the dot product of two vectors $\mathbf{a}, \mathbf{b} \in \mathbb{R}^N$. The dot product $\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{N} a_i b_i$ involves $N$ independent multiplications followed by a sum. In CUDA (NVIDIA's GPU programming framework) , we write:

```
__global__ void dotProductKernel(float* a, float*
 ↪  b,
float* partial, int N) {
    int idx = blockIdx.x * blockDim.x +
     ↪  threadIdx.x;
    if (idx < N) {
        partial[idx] = a[idx] * b[idx];
    }
}
```

Figure 5: CUDA device code for parallel vector dot product computation.

In this example , each thread computes one element of the element-wise product $a_i \cdot b_i$ in parallel. The variable `idx` uniquely identifies each thread's position in the global thread array. The kernel function marked with `__global__` runs on the GPU device , while the host code (see Figure 10 in Appendix C for the complete implementation) allocates GPU memory , copies data to the device , launches the kernel with the appropriate number of blocks and threads , and retrieves results. While this example shows the basic structure , real GPU programming involves many subtleties: memory coalescing (accessing memory in patterns that maximize bandwidth) , shared memory usage (fast on-chip cache shared by threads in a block) , thread synchronization , occupancy optimization (keeping all GPU cores busy) , and register pressure management (avoiding register spills to slower memory).

These optimizations can dramatically affect performance—a naive GPU implementation might be only marginally faster than CPU code , while a well-optimized kernel can achieve over 10,000x speedups for highly parallelizable problems. The details of these optimizations are highly technical and device-specific , involving intimate knowledge of GPU memory hierarchies , warp scheduling , and hardware constraints. We will not delve deeply into these implementation details , as they are beyond the scope of this report. Instead , we focus on the algorithmic principles of the numerical methods themselves , understanding that efficient GPU implementations require careful attention to hardware characteristics.

With this foundation in parallel computing , we now turn to the specific numerical methods for solving the Navier-Stokes equations , keeping in mind that all of them are naturally amenable to GPU acceleration due to their data-parallel structure.

## 2.2 Finite Volume Method

The Finite Volume Method (FVM) is one of the most widely used techniques in computational fluid dynamics , particularly in industrial applications. Unlike methods that discretize the differential form of the Navier-Stokes equations directly , the FVM discretizes the *integral form* of the conservation laws. This fundamental difference gives the FVM several advantageous properties: it inherently conserves mass , momentum , and energy at the discrete level , it naturally handles complex geometries through unstructured meshes , and it provides a clear physical interpretation where each computational cell represents a control volume with fluxes entering and leaving through its boundaries.

The core philosophy of the Finite Volume Method is deceptively simple: divide the computational domain into small , non-overlapping control volumes (cells) , and enforce conservation laws on each cell by tracking fluxes across cell boundaries. The velocity field is stored at cell centers , while fluxes are computed at cell faces. This staggered arrangement , combined with careful numerical schemes for flux computation , yields a robust and physically consistent discretization.

For incompressible Newtonian fluids , recall that we must solve the coupled system of equations (1) and (6):

$$\nabla \cdot \mathbf{u} = 0$$
$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = -\frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{f}$$

The FVM discretizes these equations by integrating over each control volume and applying the divergence theorem to convert volume integrals of divergences into surface integrals of fluxes. This transformation is the mathematical heart of the method.

### 2.2.1 Spatial Discretization

We begin by discretizing the spatial domain into a uniform Cartesian grid. Let the computational domain be a rectangular box $\Omega = [0, L_x] \times [0, L_y] \times [0, L_z]$. We divide this domain into $N_x \times N_y \times N_z$ cubic cells , where each cell has side length $\Delta x = L_x / N_x$ (assuming uniform spacing in all directions for simplicity). The cell centered at grid point $(i, j, k)$ occupies the region

$$\Omega_{i,j,k} = [x_i - \Delta x/2, x_i + \Delta x/2] \times [y_j - \Delta x/2, y_j + \Delta x/2] \times [z_k - \Delta x/2, z_k + \Delta x/2]$$

where $(x_i, y_j, z_k) = (i\Delta x, j\Delta x, k\Delta x)$ is the center of the cell. We use the index notation $(i, j, k)$ to uniquely identify each cell , with $i \in \{0, 1, \ldots, N_x - 1\}$ , $j \in \{0, 1, \ldots, N_y - 1\}$ , and $k \in \{0, 1, \ldots, N_z - 1\}$.

[INSERT DIAGRAM]

At each cell center , we store the three components of the velocity vector: $\mathbf{u}_{i,j,k} = (u_{i,j,k}, v_{i,j,k}, w_{i,j,k})$ , representing the velocity in the $x$, $y$, and $z$ directions respectively. We also store the pressure $p_{i,j,k}$ at cell centers. This data structure forms the foundation of our discretization—each cell contains all the fluid state variables needed to describe the local flow.

The key to the FVM is computing fluxes across cell faces. Each cubic cell has six faces: left/right (perpendicular to $x$-axis) , front/back (perpendicular to $y$-axis) , and bottom/top (perpendicular to $z$-axis). For a cell $(i, j, k)$ , we denote the six neighboring cells as $(i \pm 1, j, k)$ , $(i, j \pm 1, k)$ , and $(i, j, k \pm 1)$. The flux computation at a face requires interpolating or extrapolating values from neighboring cell centers to the face location.

[INSERT DIAGRAM]

Consider the momentum equation. Integrating over cell $\Omega_{i,j,k}$ gives

$$\iiint_{\Omega_{i,j,k}} \frac{\partial \mathbf{u}}{\partial t}\, dV + \iiint_{\Omega_{i,j,k}} \nabla \cdot (\mathbf{u}\mathbf{u})\, dV = -\frac{1}{\rho} \iiint_{\Omega_{i,j,k}} \nabla p\, dV + \nu \iiint_{\Omega_{i,j,k}} \nabla^2 \mathbf{u}\, dV + \iiint_{\Omega_{i,j,k}} \mathbf{f}\, dV$$

Assuming the cell is small enough that quantities are approximately constant within it , the time derivative term simplifies to

$$\iiint_{\Omega_{i,j,k}} \frac{\partial \mathbf{u}}{\partial t}\, dV \approx \frac{\partial \mathbf{u}_{i,j,k}}{\partial t} \cdot \Delta V$$

where $\Delta V = (\Delta x)^3$ is the cell volume. For the convective term $\nabla \cdot (\mathbf{u}\mathbf{u})$ , we apply the divergence theorem to convert the volume integral into a surface integral over the cell faces:

$$\iiint_{\Omega_{i,j,k}} \nabla \cdot (\mathbf{u}\mathbf{u})\, dV = \oiint_{\partial\Omega_{i,j,k}} (\mathbf{u}\mathbf{u}) \cdot \mathbf{n}\, dS$$

where $\partial\Omega_{i,j,k}$ is the boundary (six faces) of the cell and $\mathbf{n}$ is the outward unit normal vector. This surface integral represents the net convective flux leaving the cell through all its faces. Similarly , the pressure gradient and viscous terms can be written as surface integrals using the divergence theorem.

The challenge now is to approximate these surface integrals numerically. For a uniform Cartesian grid , each face has area $(\Delta x)^2$ , and the outward normals are simply $\pm\mathbf{e}_x$ , $\pm\mathbf{e}_y$ , $\pm\mathbf{e}_z$ (unit vectors along coordinate axes). The surface integral decomposes into a sum over the six faces. For example , the convective flux through the right face (at $x = x_i + \Delta x/2$) in the $x$-direction is

$$\text{Flux}_{\text{right}} = u \cdot u \cdot (\Delta x)^2$$

evaluated at the face location. But we only have velocity values at cell centers , not at faces ! This necessitates interpolation. The simplest approach is linear interpolation between adjacent cell centers:

$$u_{\text{face}} = \frac{u_{i,j,k} + u_{i+1,j,k}}{2}$$

However , naive interpolation can lead to numerical instability , particularly for convection-dominated flows. The FVM therefore employs various *upwind schemes* and *flux limiters* to ensure stability and accuracy. We will discuss this in detail when examining the advection step.

### 2.2.2 Temporal Discretization and Operator Splitting

The momentum equation contains three distinct physical processes: convection (advection by the flow) , diffusion (viscous spreading) , and pressure forces. A powerful technique for solving this coupled system is *operator splitting* , also known as fractional step methods. The idea is to decompose the full problem into a sequence of simpler sub-problems , each handling one physical process at a time. While mathematically this introduces splitting error , the error can be made arbitrarily small by choosing sufficiently small time steps , and the computational benefits are enormous.

The most common operator splitting for incompressible flow is the *projection method* , introduced by Chorin in 1967. The algorithm proceeds in several stages at each time step:

1. **Source Term**: Add any external body forces $\mathbf{f}$ to the velocity field
2. **Advection**: Update velocity due to convection $(\mathbf{u} \cdot \nabla)\mathbf{u}$
3. **Diffusion**: Update velocity due to viscous diffusion $\nu\nabla^2\mathbf{u}$

4. **Projection**: Enforce incompressibility $\nabla \cdot \mathbf{u} = 0$ by solving for pressure and correcting velocity

5. **Boundary Conditions**: Enforce no-slip or other boundary conditions on walls

This splitting allows us to tackle each physical mechanism with specialized numerical techniques optimized for that particular type of equation. Let us examine each stage in detail.

#### 2.2.3 Source Term Addition

The source term $\mathbf{f}$ represents external body forces per unit mass acting on the fluid. The most common example is gravity , $\mathbf{f} = g\mathbf{e}_z$ where $g$ is gravitational acceleration and $\mathbf{e}_z$ is the upward unit vector. In our implementation , we also use the source term to inject momentum into the fluid to drive the flow—this is purely for simulation purposes and represents a continuous forcing that keeps the flow active.

Adding the source term is computationally trivial. For each cell $(i, j, k)$ , we simply perform a forward Euler step:

$$\mathbf{u}^*_{i,j,k} = \mathbf{u}^n_{i,j,k} + \Delta t\, \mathbf{f}_{i,j,k}$$

where the superscript $n$ denotes the time level and $*$ denotes the intermediate velocity after adding sources. In our code , the `addSourceKernel` implements this by adding an upward velocity in a localized region at the bottom of the domain:

```
if (k < nz / 4) {  // Bottom quarter of domain
    float strength = 0.15f * exp(-r*r / 0.1f) *
    dt;
    w[idx] += strength;  // Upward velocity
    u[idx] += -dy_pos * strength * 0.5f;  //
    Rotation
    v[idx] += dx_pos * strength * 0.5f;
}
```

This creates a localized "jet" of fluid rising from the bottom , with a Gaussian spatial profile that decays with distance $r$ from the center. The rotation components add swirl to make the flow more visually interesting. From a computational perspective , this is an embarrassingly parallel operation: each thread updates one grid cell independently , with no communication between threads. The computational complexity is $O(N_x N_y N_z)$ with perfect parallelization—if we have enough GPU cores , this step can be completed in constant time regardless of grid size.

#### 2.2.4 Advection (Convection)

The advection step solves the pure transport equation

$$\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = 0$$

which describes how the velocity field is transported by the flow itself. This is the *nonlinear convection term* that makes the Navier-Stokes equations so difficult. The term $(\mathbf{u} \cdot \nabla)\mathbf{u}$ can be understood as follows: at each point , the velocity $\mathbf{u}$ acts as a "transport vector" that advects the velocity field itself. If we imagine sitting on a fluid particle moving with velocity $\mathbf{u}$ , this term describes how the velocity we observe changes as we move through the spatially-varying velocity field.

Numerically solving advection equations is notoriously challenging. The fundamental issue is that simple finite difference schemes tend to be either *numerically diffusive* (smoothing out sharp features unrealistically) or *dispersive* (creating spurious oscillations). For high Reynolds number flows where sharp gradients and turbulent structures are important , these numerical artifacts can completely corrupt the solution.

Our implementation uses the **Semi-Lagrangian method** , an elegant approach that sidesteps many of these difficulties. The key insight is to exploit the method of characteristics. For the pure advection equation $\frac{\partial \mathbf{u}}{\partial t} + (\mathbf{u} \cdot \nabla)\mathbf{u} = 0$ , the solution along characteristic curves (particle trajectories) is constant. That is , if we follow a fluid particle moving with the flow , the velocity we observe does not change due to advection alone.

Mathematically , consider a fluid particle at position $\mathbf{x}$ at time $t^{n+1}$. Where was this particle at the previous time $t^n$ ? If the particle moved with velocity $\mathbf{u}$ , it came from position

$$\mathbf{x}_{\text{prev}} = \mathbf{x} - \Delta t\, \mathbf{u}(\mathbf{x}, t^{n+1})$$

Since velocity is constant along characteristics , we have

$$\mathbf{u}(\mathbf{x}, t^{n+1}) = \mathbf{u}(\mathbf{x}_{\text{prev}}, t^n)$$

This is the *semi-Lagrangian advection scheme*: to find the new velocity at grid point $\mathbf{x}$ , we trace backwards in time to find where a particle at $\mathbf{x}$ came from , then look up the velocity at that previous location. The scheme is called "semi-Lagrangian" because we trace Lagrangian particle paths but maintain an Eulerian grid.

[INSERT DIAGRAM]

The algorithm proceeds as follows for each grid cell $(i, j, k)$:

1. Compute the departure point: $\mathbf{x}_{\text{prev}} = (i, j, k) - \Delta t\, \mathbf{u}^*_{i,j,k}/\Delta x$
2. The departure point typically does not lie on a grid point—it falls somewhere between grid points
3. Use trilinear interpolation to evaluate $\mathbf{u}(\mathbf{x}_{\text{prev}}, t^n)$ from the surrounding 8 grid points
4. Set $\mathbf{u}^{**}_{i,j,k} = \mathbf{u}(\mathbf{x}_{\text{prev}}, t^n)$

The trilinear interpolation is the computational bottleneck of this method. Given the departure point $(x, y, z)$ , we first find the 8 surrounding grid vertices:

$$\begin{aligned} i_0 &= \lfloor x \rfloor, \quad i_1 = i_0 + 1 \\ j_0 &= \lfloor y \rfloor, \quad j_1 = j_0 + 1 \\ k_0 &= \lfloor z \rfloor, \quad k_1 = k_0 + 1 \end{aligned}$$

The interpolation weights are determined by the fractional position within the cell:

$$\begin{aligned} s_x &= x - i_0, \quad t_x = 1 - s_x \\ s_y &= y - j_0, \quad t_y = 1 - s_y \\ s_z &= z - k_0, \quad t_z = 1 - s_z \end{aligned}$$

The interpolated value is then

$$\begin{aligned} \mathbf{u}(x, y, z) = {} & t_x t_y t_z\, \mathbf{u}_{i_0,j_0,k_0} + s_x t_y t_z\, \mathbf{u}_{i_1,j_0,k_0} \\ & + t_x s_y t_z\, \mathbf{u}_{i_0,j_1,k_0} + s_x s_y t_z\, \mathbf{u}_{i_1,j_1,k_0} \\ & + t_x t_y s_z\, \mathbf{u}_{i_0,j_0,k_1} + s_x t_y s_z\, \mathbf{u}_{i_1,j_0,k_1} \\ & + t_x s_y s_z\, \mathbf{u}_{i_0,j_1,k_1} + s_x s_y s_z\, \mathbf{u}_{i_1,j_1,k_1} \end{aligned}$$

This requires 8 memory reads (to fetch the 8 surrounding velocity values) and 24 multiply-add operations (8 trilinear weight computations times 3 velocity components). In our CUDA implementation , each thread handles one grid cell:

```
// Trace backwards
float x = i - dt * u[idx] / dx;
float y = j - dt * v[idx] / dx;
float z = k - dt * w[idx] / dx;

// Clamp to grid boundaries
x = fmaxf(0.5f, fminf(nx - 1.5f, x));
y = fmaxf(0.5f, fminf(ny - 1.5f, y));
z = fmaxf(0.5f, fminf(nz - 1.5f, z));

// Find surrounding cells
int i0 = (int)x, i1 = i0 + 1;
int j0 = (int)y, j1 = j0 + 1;
int k0 = (int)z, k1 = k0 + 1;

// Interpolation weights
float sx = x - i0, sy = y - j0, sz = z - k0;
float tx = 1.0f - sx, ty = 1.0f - sy, tz = 1.0f -
     sz;

// Trilinear interpolation (shown for u-component
    )
u_new[idx] = tx*ty*tz * u[i0 + nx*j0 + nx*ny*k0]
           + sx*ty*tz * u[i1 + nx*j0 + nx*ny*k0]
           + tx*sy*tz * u[i0 + nx*j1 + nx*ny*k0]
           + ... // (8 terms total)
```

The computational complexity of advection is $O(N_x N_y N_z)$ per time step. However , the memory access pattern is problematic for GPU performance. The 8 surrounding grid points are typically not contiguous in memory (due to the 3D array layout) , leading to *non-coalesced memory accesses*. Modern GPUs achieve peak bandwidth when threads in a warp access consecutive memory addresses , but here each thread reads from scattered locations. This can reduce memory bandwidth by 4-8x compared to optimal access patterns.

Despite this inefficiency , the semi-Lagrangian method has major advantages: it is unconditionally stable (no CFL restriction) , handles large time steps well , and is relatively simple to implement. The stability comes from the fact that we are essentially solving the advection exactly along characteristics—there is no numerical diffusion or dispersion from discretizing derivatives. The price we pay is some numerical dissipation (velocities get slightly smoothed during interpolation) , but this is generally acceptable for visual simulations.

#### 2.2.5 Diffusion (Viscous Term)

The diffusion step solves the heat equation for each velocity component:

$$\frac{\partial \mathbf{u}}{\partial t} = \nu \nabla^2 \mathbf{u}$$

where $\nu$ is the kinematic viscosity and $\nabla^2$ is the Laplacian operator. In 3D , the Laplacian of a scalar field $\phi$ is

$$\nabla^2 \phi = \frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} + \frac{\partial^2 \phi}{\partial z^2}$$

Physically , the diffusion term represents momentum spreading due to viscosity. Regions of high velocity transfer momentum to neighboring low-velocity regions , smoothing out velocity gradients over time.

The diffusion equation is parabolic , fundamentally different from the hyperbolic advection equation. While advection transports information along characteristics at finite speed , diffusion spreads information instantaneously (in the mathematical idealization) to all points in space.

The standard centered finite difference approximation for the Laplacian at grid point $(i, j, k)$ is

$$(\nabla^2 \mathbf{u})_{i,j,k} \approx \Big(\mathbf{u}_{i+1,j,k} + \mathbf{u}_{i-1,j,k} + \mathbf{u}_{i,j+1,k} + \mathbf{u}_{i,j-1,k} + \mathbf{u}_{i,j,k+1} + \mathbf{u}_{i,j,k-1} - 6\mathbf{u}_{i,j,k}\Big) \frac{1}{(\Delta x)^2}$$

This is a 7-point stencil: the value at $(i, j, k)$ and its 6 nearest neighbors. The discretization is second-order accurate , meaning the truncation error decreases as $(\Delta x)^2$ as the grid is refined.

[INSERT DIAGRAM]

For time integration , we face a critical choice: explicit or implicit ? An **explicit scheme** like forward Euler would give

$$\mathbf{u}^{n+1}_{i,j,k} = \mathbf{u}^{n}_{i,j,k} + \Delta t\, \nu\, (\nabla^2 \mathbf{u}^n)_{i,j,k}$$

This is trivial to implement but suffers from a severe stability restriction. Stability analysis (von Neumann stability) shows that explicit diffusion requires

$$\Delta t \leq \frac{(\Delta x)^2}{6\nu}$$

For fine grids , this CFL-like condition forces extremely small time steps. For example , with $\Delta x = 0.02$ and $\nu = 0.0001$ , we would need $\Delta t \lesssim 0.0007$ , far smaller than the $\Delta t = 0.016$ we use for visual smoothness at 60 FPS. This makes explicit diffusion impractical for most applications.

The solution is an **implicit scheme** , where we evaluate the Laplacian at the new time level:

$$\mathbf{u}^{n+1}_{i,j,k} = \mathbf{u}^{**}_{i,j,k} + \Delta t\, \nu\, (\nabla^2 \mathbf{u}^{n+1})_{i,j,k}$$

Note that the unknown $\mathbf{u}^{n+1}$ appears on both sides of the equation. Expanding the Laplacian gives

$$\mathbf{u}^{n+1}_{i,j,k} = \mathbf{u}^{**}_{i,j,k} + \Big(\mathbf{u}^{n+1}_{i+1,j,k} + \mathbf{u}^{n+1}_{i-1,j,k} + \mathbf{u}^{n+1}_{i,j+1,k} + \mathbf{u}^{n+1}_{i,j-1,k} + \mathbf{u}^{n+1}_{i,j,k+1} + \mathbf{u}^{n+1}_{i,j,k-1} - 6\mathbf{u}^{n+1}_{i,j,k}\Big) \frac{\Delta t\, \nu}{(\Delta x)^2}$$

Rearranging , we obtain

$$(1 + 6\alpha)\, \mathbf{u}^{n+1}_{i,j,k} - \alpha\Big(\mathbf{u}^{n+1}_{i+1,j,k} + \mathbf{u}^{n+1}_{i-1,j,k} + \mathbf{u}^{n+1}_{i,j+1,k} + \mathbf{u}^{n+1}_{i,j-1,k} + \mathbf{u}^{n+1}_{i,j,k+1} + \mathbf{u}^{n+1}_{i,j,k-1}\Big) = \mathbf{u}^{**}_{i,j,k}$$

where $\alpha = \Delta t\, \nu / (\Delta x)^2$ is the dimensionless diffusion coefficient. This is a massive linear system: we have one equation like this for every grid point , and all equations are coupled together. For a $64 \times 64 \times 64$ grid , this is a system of 262,144 equations with 262,144 unknowns. The coefficient matrix is sparse (each equation involves only 7 unknowns) but solving it exactly would still require prohibitive memory and computation.

The standard approach is to use an **iterative solver**. Our implementation uses the *Jacobi method* , one of the simplest iterative schemes. The idea is to split the coefficient matrix as $A = D - L - U$ where $D$ is diagonal , $L$ is strictly lower triangular , and $U$ is strictly upper triangular. The Jacobi iteration is

$$\mathbf{u}^{(m+1)} = D^{-1}(\mathbf{b} + (L + U)\mathbf{u}^{(m)})$$

where $\mathbf{b}$ is the right-hand side vector and $m$ is the iteration index. For our diffusion equation , this simplifies beautifully. At each iteration , we update:

$$\mathbf{u}^{(m+1)}_{i,j,k} = \Big[\mathbf{u}^{**}_{i,j,k} + \alpha\Big(\mathbf{u}^{(m)}_{i+1,j,k} + \mathbf{u}^{(m)}_{i-1,j,k} + \mathbf{u}^{(m)}_{i,j+1,k} + \mathbf{u}^{(m)}_{i,j-1,k} + \mathbf{u}^{(m)}_{i,j,k+1} + \mathbf{u}^{(m)}_{i,j,k-1}\Big)\Big] \frac{1}{1 + 6\alpha}$$

This is a beautifully local update rule: the new value at each point depends only on its 6 neighbors from the previous iteration. It is perfectly suited for GPU parallelization—each thread computes the update for one grid cell. The implementation is straightforward:

```
float alpha = dt * nu / (dx * dx);
float rbeta = 1.0f / (1.0f + 6.0f * alpha);

for (int iter = 0; iter < 20; iter++) {
    // Each thread updates one grid point
    u_new[idx] = (u[idx] + alpha * (
        u_new[i-1,j,k] + u_new[i+1,j,k] +
        u_new[i,j-1,k] + u_new[i,j+1,k] +
        u_new[i,j,k-1] + u_new[i,j,k+1]
    )) * rbeta;

    cudaDeviceSynchronize();  // Wait for all
    threads
    swap(u, u_new);           // Swap pointers
}
```

We perform 20 Jacobi iterations per time step. Why 20 ? This is an empirical choice balancing accuracy and performance. Each iteration reduces the error by a factor roughly equal to the spectral radius of the iteration matrix. For the Laplacian on a uniform grid , the Jacobi method converges but relatively slowly—it is not the most efficient iterative solver. More sophisticated methods like Gauss-Seidel , Successive Over-Relaxation (SOR) , or Conjugate Gradient would converge faster , but they are harder to parallelize effectively on GPUs because they introduce dependencies between grid points within a single iteration.

The computational cost of diffusion is $O(K \cdot N_x N_y N_z)$ where $K$ is the number of iterations. With $K = 20$ and parallelization , this is still manageable. Each iteration requires reading 6 neighbor values and writing 1 new value per grid point. The memory access pattern is better than advection—neighbors in the $x$-direction are often in the same cache line—but still not perfectly coalesced.

An important subtlety: implicit diffusion is unconditionally stable but not unconditionally accurate. Large time steps lead to excessive numerical diffusion , over-smoothing the velocity field. The choice of $\Delta t$ should still be guided by accuracy requirements , not just stability. In practice , we choose $\Delta t$ based on the desired frame rate and accept some over-diffusion as the price for large time steps.

#### 2.2.6 Projection (Enforcing Incompressibility)

After advection and diffusion , the velocity field $\mathbf{u}^{**}$ is not necessarily divergence-free. The projection step enforces the incompressibility constraint $\nabla \cdot \mathbf{u} = 0$ by decomposing the velocity into divergence-free and gradient components. This is the mathematical heart of the projection method and relies on the *Helmholtz decomposition theorem*.

The Helmholtz decomposition states that any sufficiently smooth vector field $\mathbf{v}$ can be uniquely decomposed as

$$\mathbf{v} = \mathbf{w} + \nabla\phi$$

where $\nabla \cdot \mathbf{w} = 0$ (divergence-free part) and $\phi$ is a scalar potential. Applying this to our intermediate velocity $\mathbf{u}^{**}$ , we can write

$$\mathbf{u}^{**} = \mathbf{u}^{n+1} + \nabla\phi$$

where $\mathbf{u}^{n+1}$ is the desired divergence-free velocity and $\nabla\phi$ is the gradient part we need to subtract. Taking the divergence of both sides:

$$\nabla \cdot \mathbf{u}^{**} = \nabla \cdot \mathbf{u}^{n+1} + \nabla^2\phi = 0 + \nabla^2\phi$$

This gives us a *Poisson equation* for $\phi$:

$$\nabla^2\phi = \nabla \cdot \mathbf{u}^{**}$$

Once we solve for $\phi$ , we can project out the gradient component:

$$\mathbf{u}^{n+1} = \mathbf{u}^{**} - \nabla\phi$$

In the context of incompressible flow , the potential $\phi$ is related to pressure. Comparing with the momentum equation , we identify $\phi = p/\rho$ (up to a constant). The projection step is therefore equivalent to solving for the pressure field and using it to correct the velocity.

The algorithm proceeds in three sub-steps:

**Sub-step 4a: Compute Divergence**

First , we compute the divergence of the intermediate velocity field at each grid point. Using centered finite differences:

$$(\nabla \cdot \mathbf{u})_{i,j,k} = \frac{u_{i+1,j,k} - u_{i-1,j,k}}{2\Delta x} + \frac{v_{i,j+1,k} - v_{i,j-1,k}}{2\Delta x} + \frac{w_{i,j,k+1} - w_{i,j,k-1}}{2\Delta x}$$

[INSERT DIAGRAM]

This is a simple stencil operation requiring 6 memory reads per grid point. The implementation is:

```
div[idx] = -0.5f * dx * (
    u[i+1,j,k] - u[i-1,j,k] +
    v[i,j+1,k] - v[i,j-1,k] +
    w[i,j,k+1] - w[i,j,k-1]
);
```

Note the factor of $-0.5\Delta x$ instead of division by $2\Delta x$—this is an optimization that folds the $\Delta x$ scaling into the pressure solve. The divergence computation is $O(N_x N_y N_z)$ and perfectly parallel.

**Sub-step 4b: Solve Poisson Equation for Pressure**

We must now solve $\nabla^2 p = \nabla \cdot \mathbf{u}^{**}$ for the pressure $p$. This is structurally identical to the diffusion equation—a Poisson equation on the grid. We use the same Jacobi iteration approach. Discretizing the Laplacian:

$$\Big(p_{i+1,j,k} + p_{i-1,j,k} + p_{i,j+1,k} + p_{i,j-1,k} + p_{i,j,k+1} + p_{i,j,k-1} - 6p_{i,j,k}\Big)\frac{1}{(\Delta x)^2} = (\nabla \cdot \mathbf{u})_{i,j,k}$$

Solving for $p_{i,j,k}$:

$$p_{i,j,k} = \frac{1}{6}\Big[p_{i+1,j,k} + p_{i-1,j,k} + p_{i,j+1,k} + p_{i,j-1,k} + p_{i,j,k+1} + p_{i,j,k-1} - (\Delta x)^2(\nabla \cdot \mathbf{u})_{i,j,k}\Big]$$

The Jacobi iteration becomes:

$$p_{i,j,k}^{(m+1)} = \frac{1}{6}\Big[p_{i+1,j,k}^{(m)} + p_{i-1,j,k}^{(m)} + p_{i,j+1,k}^{(m)} + p_{i,j-1,k}^{(m)} + p_{i,j,k+1}^{(m)} + p_{i,j,k-1}^{(m)} - (\Delta x)^2(\nabla \cdot \mathbf{u})_{i,j,k}\Big]$$

We perform 30 iterations (more than diffusion because pressure projection accuracy directly affects divergence):

```
for (int iter = 0; iter < 30; iter++) {
    p_new[idx] = (div[idx] +
        p[i-1,j,k] + p[i+1,j,k] +
        p[i,j-1,k] + p[i,j+1,k] +
        p[i,j,k-1] + p[i,j,k+1]
    ) / 6.0f;

    cudaDeviceSynchronize();
    swap(p, p_new);
}
```

The computational cost is $O(K_p \cdot N_x N_y N_z)$ where $K_p = 30$ is the iteration count. This is often the most expensive part of the entire time step because we need high accuracy to properly enforce incompressibility. Insufficient pressure iterations lead to visible artifacts: the flow "leaks" through boundaries or exhibits unphysical expansion/compression.

**Sub-step 4c: Pressure Correction**

Finally , we subtract the pressure gradient from the intermediate velocity:

$$\mathbf{u}_{i,j,k}^{n+1} = \mathbf{u}_{i,j,k}^{**} - \frac{\nabla p_{i,j,k}}{\rho}$$

For incompressible flow with constant $\rho$ , we typically work in units where $\rho = 1$. The pressure gradient is computed using centered differences:

$$u_{i,j,k}^{n+1} = u_{i,j,k}^{**} - \frac{p_{i+1,j,k} - p_{i-1,j,k}}{2\Delta x}$$
$$v_{i,j,k}^{n+1} = v_{i,j,k}^{**} - \frac{p_{i,j+1,k} - p_{i,j-1,k}}{2\Delta x}$$
$$w_{i,j,k}^{n+1} = w_{i,j,k}^{**} - \frac{p_{i,j,k+1} - p_{i,j,k-1}}{2\Delta x}$$

[INSERT DIAGRAM]

Implementation:

```
u[idx] -= 0.5f * (p[i+1,j,k] - p[i-1,j,k]) / dx;
v[idx] -= 0.5f * (p[i,j+1,k] - p[i,j-1,k]) / dx;
w[idx] -= 0.5f * (p[i,j,k+1] - p[i,j,k-1]) / dx;
```

This is again $O(N_x N_y N_z)$ and perfectly parallel. After this correction , the velocity field satisfies $\nabla \cdot \mathbf{u}^{n+1} = 0$ to within the tolerance of our iterative pressure solve.

The projection method is mathematically elegant but computationally expensive. The Poisson solve dominates the total cost—even with 30 iterations of Jacobi , convergence is slow for fine grids. State-of-the-art CFD codes use multigrid methods or preconditioned Krylov solvers (like GMRES or BiCGSTAB) to solve the pressure Poisson equation much faster , often achieving optimal $O(N)$ complexity. However , these advanced solvers are complex to implement and their GPU parallelization is non-trivial.

### 2.2.7 Boundary Conditions

The final stage enforces boundary conditions at the walls of the computational domain. For our simulation , we use *no-slip boundary conditions*: the velocity is zero at all wall surfaces. Physically , this represents the fact that fluid molecules stick to solid surfaces due to intermolecular forces.

Implementing no-slip conditions is trivial—we simply set all velocity components to zero at boundary grid points:

```
if (i == 0 || i == nx-1 || j == 0 || j == ny-1 ||
    k == 0 || k == nz-1) {
    u[idx] = 0.0f;
    v[idx] = 0.0f;
    w[idx] = 0.0f;
}
```

This is executed by every thread , but only boundary threads actually modify data. The computational cost is negligible—$O(N_x N_y + N_y N_z + N_z N_x) \ll O(N_x N_y N_z)$ since only surface grid points are affected.

More sophisticated boundary conditions exist: free-slip (tangential velocity allowed , normal velocity zero) , inflow/outflow (prescribed velocity or pressure) , periodic (opposite boundaries identified) , etc. Each requires specialized handling. For complex geometries with curved boundaries , immersed boundary methods or cut-cell approaches are needed to accurately represent the interface.

#### 2.2.8 Overall Algorithm Summary and Parallelization

Combining all stages , the complete FVM algorithm for one time step is:

1. **Add sources**: $\mathbf{u}^* = \mathbf{u}^n + \Delta t\, \mathbf{f}$ [Complexity: $O(N)$ , perfect parallel]
2. **Advect**: $\mathbf{u}^{**}$ = semi-Lagrangian backtrack from $\mathbf{u}^*$ [Complexity: $O(N)$ , scattered memory access]
3. **Diffuse**: Jacobi iteration for $\mathbf{u}^{***}$ from $\mathbf{u}^{**}$ [Complexity: $O(K_d \cdot N)$ , $K_d = 20$]
4. **Project**:
   (a) Compute divergence [Complexity: $O(N)$]
   (b) Solve Poisson for pressure [Complexity: $O(K_p \cdot N)$ , $K_p = 30$]
   (c) Correct velocity [Complexity: $O(N)$]
5. **Boundaries**: Set wall velocities to zero [Complexity: $O(N^{2/3})$]

where $N = N_x N_y N_z$ is the total number of grid points. The total complexity per time step is $O((1 + K_d + K_p) \cdot N) \approx O(50N)$ for our parameter choices. This is excellent—linear scaling with problem size.

The parallelization strategy is embarrassingly simple: each CUDA thread handles one grid cell. For a $64^3$ grid , we launch $64^3 = 262{,}144$ threads organized as:

```
dim3 threads(8, 8, 8);  // 512 threads per block
dim3 blocks(
    (nx + 7) / 8,  // 8 blocks in x
    (ny + 7) / 8,  // 8 blocks in y
    (nz + 7) / 8   // 8 blocks in z
);
// Total: 8*8*8 = 512 blocks * 512 threads =
    262,144 threads
```

Each kernel launch processes the entire grid in parallel. The only synchronization is between kernel launches (via `cudaDeviceSynchronize()`) to ensure all threads complete before the next stage begins. Within a kernel , threads operate independently with no communication—this is the key to GPU efficiency.

The main performance bottleneck is memory bandwidth. Each grid point stores 4 values (u , v , w , p) as 32-bit floats , requiring 16 bytes. For a $64^3$ grid , this is 4 MB of data. Modern GPUs have $\sim$500 GB/s memory bandwidth , so a single pass through the grid (reading and writing all values) takes $\sim$0.02 ms. Our algorithm makes roughly 100 passes per time step (20 diffusion iterations + 30 pressure iterations + advection reads) , suggesting a theoretical lower bound of $\sim$2 ms per time step. In practice , our implementation achieves $\sim$5-10 ms per step , indicating reasonable efficiency given non-optimal memory access patterns.

For comparison , a CPU implementation running on a single core would take $\sim$100-500 ms per time step—a speedup of 10-50x from GPU parallelization. The speedup is less than the theoretical maximum ($\sim$1000x based on core count) because we are memory-bound , not compute-bound. Further optimization would focus on memory access patterns: using shared memory for frequent neighbor accesses , coalescing memory transactions , and employing texture memory for interpolation.

#### 2.2.9 Stability , Accuracy , and Convergence

The FVM as presented has several important numerical properties that determine its effectiveness for different types of flows.

**Stability:** The semi-Lagrangian advection is unconditionally stable—there is no CFL restriction on time step size from the convective term. The implicit diffusion is also unconditionally stable. The pressure projection involves solving an elliptic equation and does not introduce instability. Therefore , the overall method is stable for arbitrarily large $\Delta t$ from a numerical perspective. However , large time steps lead to temporal discretization errors and loss of physical accuracy.

**Accuracy:** The spatial discretization is second-order accurate in $\Delta x$ for smooth flows—the truncation error scales as $O((\Delta x)^2)$. The temporal accuracy is first-order due to operator splitting: the splitting error is $O(\Delta t)$. For smooth steady flows , refining the grid by a factor of 2 reduces spatial error by a factor of 4 , while halving the time step only reduces temporal error by a factor of 2. For time-dependent or turbulent flows , higher-order time stepping (such as Runge-Kutta methods) can improve temporal accuracy , but this significantly increases computational cost.

**Convergence:** The Jacobi iteration for pressure and diffusion converges geometrically—the error decreases by a constant factor each iteration. However , the convergence rate deteriorates as the grid is refined. For a grid with spacing $\Delta x$ , the number of Jacobi iterations required for a fixed error tolerance scales as $O(1/(\Delta x)^2)$. This makes Jacobi impractical for very fine grids. Multigrid methods overcome this by solv-

ing the problem on a hierarchy of grids , achieving $O(N)$ complexity independent of resolution.

**Conservation:** The FVM exactly conserves mass at the discrete level—after projection , the divergence is numerically zero (within iteration tolerance). This is a major advantage over some other discretizations that only conserve mass approximately. Momentum and energy are not exactly conserved due to numerical diffusion from interpolation and iterative solvers , but the violation is small and decreases with grid refinement.

For practical CFD simulations , these properties mean the FVM is most effective for:

- Moderate Reynolds numbers ($Re \sim 10^2$ to $10^4$) where viscosity is significant
- Geometrically simple domains (rectangles , cylinders) where Cartesian grids are natural
- Steady or slowly-varying flows where first-order temporal accuracy is acceptable
- Applications where exact mass conservation is critical (e.g. , incompressible flow with moving boundaries)

For very high Reynolds numbers (turbulent flows) or complex geometries , more advanced techniques are needed: higher-order schemes , adaptive mesh refinement , unstructured grids , or alternative methods like Lattice Boltzmann or spectral methods (which we will explore next).

### 2.3 Spectral Methods

While the Finite Volume Method discretizes the Navier-Stokes equations on a spatial grid using local stencils , **Spectral Methods** take a fundamentally different approach: they represent the velocity and pressure fields as expansions in global basis functions. Instead of storing discrete values at grid points , spectral methods store the coefficients of basis functions—typically sines and cosines (Fourier series) or polynomials (Chebyshev , Legendre). This global representation leads to dramatically different numerical properties compared to FVM.

The core idea is to write the velocity field as

$$\mathbf{u}(\mathbf{x}, t) = \sum_{k_x, k_y, k_z} \hat{\mathbf{u}}_{k_x,k_y,k_z}(t)\, e^{i(k_x x + k_y y + k_z z)}$$

where $\hat{\mathbf{u}}_{k_x,k_y,k_z}(t)$ are the Fourier coefficients (complex amplitudes) and the sum runs over a discrete set of wavenumbers. Similarly , pressure can be expanded as $p(\mathbf{x}, t) = \sum_k \hat{p}_k(t) e^{i\mathbf{k}\cdot\mathbf{x}}$. The Navier-Stokes equations are then transformed into equations for these coefficients.

The key advantage of spectral methods is **spectral accuracy**: for smooth solutions , the error decreases exponentially with the number of basis functions , rather than algebraically as in FVM. While FVM achieves second-order accuracy $O((\Delta x)^2)$ , spectral methods can achieve errors like $O(e^{-cN})$ where $N$ is the number of modes and $c$ is a constant. This means spectral methods can capture smooth flows with far fewer degrees of freedom than grid-based methods.

However , this exponential accuracy comes with severe restrictions. Spectral methods work best for:

- *Smooth flows*: Turbulence with sharp gradients or shocks destroys spectral accuracy (Gibbs phenomenon)
- *Periodic domains*: Fourier spectral methods naturally assume periodicity , limiting geometry
- *Simple boundaries*: Non-periodic boundaries require polynomial bases and complex boundary treatments

**Computational Structure:** Computing derivatives in spectral methods is remarkably simple in Fourier space. The spatial derivative $\frac{\partial}{\partial x}$ becomes multiplication by $ik_x$ in Fourier space:

$$\widehat{\frac{\partial \mathbf{u}}{\partial x}} = ik_x \hat{\mathbf{u}}$$

This means the viscous term $\nu\nabla^2\mathbf{u}$ becomes $-\nu(k_x^2 + k_y^2 + k_z^2)\hat{\mathbf{u}}$ in Fourier space—no finite difference stencils needed ! However , the nonlinear convection term $(\mathbf{u}\cdot\nabla)\mathbf{u}$ is problematic. Nonlinearity in physical space becomes convolution in Fourier space , which is expensive to compute. The standard approach is to use the *pseudospectral method*: transform to physical space , compute the nonlinear term there , then transform back to Fourier space. This requires Fast Fourier Transforms (FFTs) at each time step.

The FFT is the computational workhorse of spectral methods. For an $N^3$ grid , a naive Fourier transform would cost $O(N^6)$ , but the FFT reduces this to $O(N^3 \log N)$. Modern GPU FFT libraries (like cuFFT) are highly optimized , making spectral methods competitive with FVM for moderate problem sizes.

**Comparison with FVM:** The fundamental trade-off between spectral methods and FVM is *accuracy vs. generality*. Spectral methods achieve superior accuracy for

smooth , periodic flows in simple geometries. FVM is more robust for complex geometries , discontinuities , and local refinement. For Direct Numerical Simulation (DNS) of homogeneous turbulence in a periodic box , spectral methods are the gold standard—they can resolve the same physical phenomena with 2-4x fewer grid points per dimension than FVM , translating to 8-64x reduction in total degrees of freedom. However , for engineering applications with complex geometries (airfoils , engines , buildings) , FVM's flexibility makes it far more practical.

Memory access patterns also differ fundamentally. FVM operates with local stencils , accessing only neighboring grid points—this is cache-friendly and parallelizes naturally. Spectral methods perform global FFTs , which require all-to-all communication patterns. On GPUs , FFTs are bandwidth-limited and suffer from the same scattered memory access issues as FVM advection. The FFT effectively shuffles data across the entire grid multiple times per time step , making it challenging to optimize for modern hardware.

For the projection step , spectral methods have a beautiful advantage: the pressure Poisson equation $\nabla^2 p = \nabla \cdot \mathbf{u}$ becomes trivial in Fourier space. We simply have $\hat{p} = \frac{\widehat{\nabla \cdot \mathbf{u}}}{-k^2}$ where $k^2 = k_x^2 + k_y^2 + k_z^2$. No iterative Jacobi solver needed ! This is one of the major computational advantages of spectral methods over FVM.

### 2.4 Lattice Boltzmann Method

The **Lattice Boltzmann Method (LBM)** represents a radical departure from both FVM and spectral methods. Rather than discretizing the Navier-Stokes equations directly , LBM simulates fluid flow at the mesoscopic level by tracking the evolution of particle distribution functions on a discrete lattice. The approach is rooted in kinetic theory: instead of solving for macroscopic quantities (velocity , pressure) , we solve for the probability distribution of particles moving with different velocities.

The fundamental variable in LBM is $f_i(\mathbf{x}, t)$—the distribution function representing the density of particles at position $\mathbf{x}$ and time $t$ moving with discrete velocity $\mathbf{c}_i$. For the popular D3Q19 lattice (3D with 19 discrete velocity directions) , we track 19 distribution functions at each grid point. The macroscopic velocity and density are recovered as moments:

$$\rho(\mathbf{x}, t) = \sum_{i=0}^{18} f_i(\mathbf{x}, t)$$

$$\rho \mathbf{u}(\mathbf{x}, t) = \sum_{i=0}^{18} f_i(\mathbf{x}, t) \mathbf{c}_i$$

The LBM algorithm consists of two steps repeated at each time step:

1. **Collision**: Particles at each lattice site redistribute according to $f_i^*(\mathbf{x}, t) = f_i(\mathbf{x}, t) - \frac{1}{\tau}(f_i(\mathbf{x}, t) - f_i^{\text{eq}}(\mathbf{x}, t))$ where $f_i^{\text{eq}}$ is the equilibrium distribution and $\tau$ is the relaxation time

2. **Streaming**: Particles propagate to neighboring sites: $f_i(\mathbf{x} + \mathbf{c}_i \Delta t, t + \Delta t) = f_i^*(\mathbf{x}, t)$

Through Chapman-Enskog expansion , it can be shown that this simple collision-streaming process recovers the incompressible Navier-Stokes equations in the macroscopic limit , with viscosity $\nu = c_s^2(\tau - 0.5)\Delta t$ where $c_s$ is the lattice sound speed.

**Comparison with FVM and Spectral Methods:** LBM occupies a unique position in the landscape of CFD methods. Compared to FVM , LBM has several distinctive features:

- *No pressure solver*: The pressure Poisson equation that dominates FVM computational cost simply does not exist in LBM. Pressure emerges naturally from the equation of state $p = c_s^2 \rho$. This is a massive simplification.

- *Explicit time stepping*: LBM is fully explicit—no iterative solvers , no linear systems. Each grid point updates independently based on local information. This makes LBM embarrassingly parallel.

- *Complex boundaries*: LBM handles complex geometries remarkably well through simple "bounce-back" boundary conditions. A solid boundary is implemented by reversing particle directions—particles that would stream into a wall bounce back. This is far simpler than FVM cut-cell methods.

- *Memory overhead*: Storing 19 distribution functions per grid point (for D3Q19) requires significantly more memory than FVM's 3 velocity components. For a $64^3$ grid , LBM uses $\sim$20 MB compared to FVM's $\sim$4 MB.

Compared to spectral methods , LBM is even more different in philosophy. Spectral methods work in Fourier space with global basis functions , while LBM works on a lattice with purely local interactions. LBM has no spectral accuracy—it is typically second-order in space and time , similar to FVM. However , LBM's strength lies not in accuracy but in simplicity and parallel efficiency.

**Computational Efficiency:** On modern GPUs , LBM can be extraordinarily efficient. The collision step is entirely local—each thread computes the collision for one lattice site using only local data. The streaming step involves simple memory copies to neighboring sites. Both operations map perfectly to SIMD architectures. Memory bandwidth is the bottleneck: each time step requires reading and writing 19 floats per grid point. With careful optimization (two-lattice schemes , memory coalescing) , GPU implementations can achieve near-peak memory bandwidth.

LBM's Achilles heel is stability. The explicit time stepping requires $\Delta t \leq \Delta x / c_{\max}$ where $c_{\max}$ is the maximum particle velocity. For high Reynolds number flows , this CFL-like restriction forces very small time steps. Additionally , numerical stability deteriorates for $\tau$ close to 0.5 , limiting the range of accessible viscosities. Various stabilization techniques (entropic LBM , multiple relaxation times) address these issues at the cost of increased complexity.

The three methods represent different points in the design space. FVM offers versatility and robustness , spectral methods offer accuracy for smooth flows , and LBM offers simplicity and parallel efficiency. For turbulent flow in complex geometries , FVM remains dominant in industry. For DNS of homogeneous turbulence , spectral methods are preferred in research. For applications requiring ease of implementation and excellent parallel scaling (especially with complex moving boundaries) , LBM shines. The choice depends critically on the specific problem , available computational resources , and required accuracy.

# 3 Quantum Methods

## 3.1 Introduction to Quantum Computing

Quantum computing represents a fundamentally different computational paradigm from classical computing. While classical bits exist in definite states 0 or 1 , quantum bits (qubits) exist in superpositions of basis states , enabling exponentially large state spaces. For $n$ qubits , a classical computer requires $2^n$ bits to represent all possible states simultaneously , but a quantum computer represents this naturally in $n$ qubits. This exponential advantage in state representation is the source of quantum computing's potential power for certain problems.

A single qubit is mathematically described by a unit vector in a two-dimensional complex Hilbert space $\mathbb{C}^2$. The computational basis states are denoted $|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$ and $|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ , and a general qubit state is a superposition

$$|\psi\rangle = \alpha |0\rangle + \beta |1\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix}$$

where $\alpha, \beta \in \mathbb{C}$ satisfy the normalization condition $|\alpha|^2 + |\beta|^2 = 1$. Upon measurement in the computational basis , the qubit collapses to $|0\rangle$ with probability $|\alpha|^2$ and to $|1\rangle$ with probability $|\beta|^2$.

For $n$ qubits , the state space is the tensor product $(\mathbb{C}^2)^{\otimes n} \cong \mathbb{C}^{2^n}$. A general $n$-qubit state is

$$|\psi\rangle = \sum_{x=0}^{2^n-1} \alpha_x |x\rangle$$

where $|x\rangle$ denotes the computational basis state corresponding to the binary representation of $x$ , and $\sum_{x=0}^{2^n-1} |\alpha_x|^2 = 1$.

### 3.1.1 Quantum Gates and Circuits

Quantum computation proceeds through *quantum gates*—unitary operators that evolve the quantum state. A unitary operator $U$ satisfies $U^\dagger U = U U^\dagger = I$ where $U^\dagger$ is the conjugate transpose (see Definition A in Appendix A). Unitarity ensures that evolution preserves normalization.

The most fundamental single-qubit gates include the **Pauli gates**:

$$X = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, \quad Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$

where $X$ is the quantum NOT gate , $Z$ applies a phase flip , and $Y = iXZ$. The **Hadamard gate** creates superpositions:

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}$$

It maps $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$. Applying $H$ to all qubits in state $|0\rangle^{\otimes n}$ produces an equal superposition over all $2^n$ basis states.

The fundamental two-qubit gate is the **CNOT (Controlled-NOT)**:

$$\text{CNOT} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

CNOT flips the second (target) qubit if and only if the first (control) qubit is $|1\rangle$. Starting from $|+\rangle|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)|0\rangle$ , applying CNOT yields the maximally entangled Bell state:

$$\text{CNOT}(|+\rangle|0\rangle) = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$

**Example: Bell State Preparation Circuit.** Consider the circuit shown in Figure 6 that prepares the Bell state $|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$. Starting from $|11\rangle$ , we apply Hadamard to the first qubit: $H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ , giving state $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)|1\rangle$. The CNOT then creates entanglement , and a final Hadamard on the second qubit completes the transformation to $|\Psi^+\rangle$. This illustrates how simple gates combine to create complex quantum correlations.

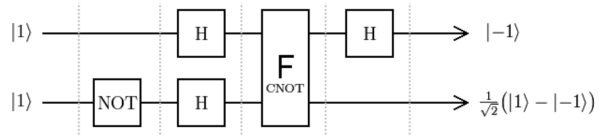


Figure 6: Bell state preparation circuit using Hadamard and CNOT gates , transforming $|11\rangle$ into the entangled state $\frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$.

### 3.1.2 Quantum Parallelism and Interference

The power of quantum computing emerges from *quantum parallelism* and *quantum interference*. Consider a function $f : \{0,1\}^n \to \{0,1\}$ implemented as unitary $U_f$ with $U_f|x\rangle|y\rangle = |x\rangle|y \oplus f(x)\rangle$. Starting from the superposition $|\psi_0\rangle = \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}|x\rangle|-\rangle$ where $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ , applying $U_f$ gives:

$$U_f|\psi_0\rangle = \frac{1}{\sqrt{2^n}}\sum_{x=0}^{2^n-1}(-1)^{f(x)}|x\rangle|-\rangle$$

This evaluates $f(x)$ for all $2^n$ inputs simultaneously , with function values encoded in phases. By applying gates that constructively interfere desired amplitudes while destructively interfering others , we extract global properties of $f$ efficiently.

The Deutsch-Jozsa algorithm exploits this to determine whether $f$ is constant or balanced with a single query versus $2^{n-1}+1$ queries classically—an exponential speedup. After applying $U_f$ and Hadamard gates $H^{\otimes n}$ , measurement yields $|0\rangle^{\otimes n}$ if and only if $f$ is constant.

### 3.1.3 Quantum Amplitude Encoding

For quantum CFD algorithms , classical data must be encoded into quantum states. **Amplitude encoding** represents an $N$-dimensional vector $\mathbf{v} = (v_0, \ldots, v_{N-1})^T$ as:

$$|\mathbf{v}\rangle = \frac{1}{\|\mathbf{v}\|}\sum_{j=0}^{N-1} v_j|j\rangle$$

where $N = 2^n$. A velocity field on a $64^3$ grid has $N = 262{,}144$ points , requiring only $n = 18$ qubits—exponential compression. However , preparing $|\mathbf{v}\rangle$ generally requires $O(N)$ gates , negating the advantage unless downstream quantum processing provides sufficient speedup. State preparation remains a major bottleneck.

### 3.1.4 Quantum Complexity and Speedups

Quantum complexity distinguishes *query complexity* (oracle calls) , *gate complexity* (total gates) , and *circuit depth* (sequential layers). Key quantum speedups include:

- **Grover's algorithm**: Searches $N$ items in $O(\sqrt{N})$ queries versus $O(N)$ classically (quadratic speedup)
- **Quantum Linear Solver (HHL)**: Solves $A\mathbf{x} = \mathbf{b}$ in time $O(\log N \cdot \kappa^2 \cdot \text{poly}(\log(1/\epsilon)))$ versus classical $O(N\kappa\log(1/\epsilon))$ for sparse $A$ (exponential in $\log N$)
- **Quantum Amplitude Estimation**: Estimates expectation values in $O(1/\epsilon)$ versus $O(1/\epsilon^2)$ classically (quadratic speedup)

Critical caveat: HHL outputs quantum state $|\mathbf{x}\rangle$ , not classical vector $\mathbf{x}$. Extracting all $N$ components requires $O(N)$ measurements , eliminating the speedup. The algorithm is useful when only properties like $\|\mathbf{x}\|$ or $\mathbf{u}^T\mathbf{x}$ are needed , or when $|\mathbf{x}\rangle$ feeds into further quantum processing.

For CFD , this is problematic: we typically need full velocity fields for visualization and analysis. However , for specific tasks—computing drag coefficients ,

estimating turbulent statistics , solving pressure Poisson equations within classical workflows—quantum speedups may prove valuable.

#### 3.1.5 Quantum Error and Decoherence

Real quantum computers suffer from *decoherence*—loss of quantum coherence from environmental interactions. Current superconducting qubits have coherence times $T_2 \sim$ 10-100 $\mu s$ , limiting circuit depth to thousands of gates. Gate error probability $p \sim 10^{-3}$ means a circuit with $d$ gates has success probability $(1-p)^d \approx e^{-pd}$. For $d \sim 10^6$ gates , success is negligible without error correction.

**Quantum Error Correction (QEC)** encodes logical qubits in many physical qubits. The surface code requires $\sim$ 1000 physical qubits per logical qubit for error rates below $10^{-15}$. The threshold theorem states that if physical error rates are below $p_{\text{th}} \sim 10^{-4}$ to $10^{-2}$ , arbitrarily long computations are possible with polynomial overhead. Current hardware approaches but has not reliably crossed this threshold. *Fault-Tolerant Quantum Computing (FTQC)* likely requires millions of physical qubits.

Near-term *Noisy Intermediate-Scale Quantum (NISQ)* devices with $\sim$ 100-1000 qubits use shallow circuits (depth $\sim$ 10-100) without error correction. NISQ algorithms like VQE and QAOA rely on error mitigation and hybrid classical-quantum approaches. For CFD , NISQ might address coarse-grained sub-problems while deferring full solutions to future fault-tolerant devices.

#### 3.1.6 Relevance to Computational Fluid Dynamics

Connecting quantum computing to CFD is non-obvious. Classical CFD excels at simulating continuous PDEs on grids. Quantum algorithms offer speedups for linear algebra , but Navier-Stokes equations are fundamentally nonlinear. Several approaches bridge this gap:

- **Linearization and operator splitting**: FVM already splits Navier-Stokes into linear subproblems (diffusion , pressure Poisson). Quantum linear solvers could accelerate the pressure solve.
- **Monte Carlo methods**: Turbulence statistics via Monte Carlo sampling benefit from quantum amplitude estimation's quadratic speedup.
- **Carleman linearization**: Nonlinear PDEs rewritten as infinite-dimensional linear systems via Carleman embedding , truncated and solved with quantum linear solvers.
- **Lattice-based encodings**: LBM's discrete velocity structure aligns with quantum states. Quantum walks and cellular automata could simulate kinetic models.

In the following sections , we examine quantum amplitude estimation for turbulent flow statistics and quantum linear solvers for pressure Poisson equations—illustrating both promise and practical challenges of quantum-accelerated CFD.

### 3.2 Quantum Hardware

### 3.3 Turbulent Flows

Turbulent flows represent one of the most computationally challenging regimes in fluid dynamics. The nonlinear nature of the Navier-Stokes equations gives rise to chaotic motion across a vast hierarchy of spatial and temporal scales , from large energy-containing eddies down to tiny dissipative structures. Direct numerical simulation (DNS) of turbulence requires resolving all these scales simultaneously , leading to computational costs that scale as $Re^{9/4}$ in three dimensions , where $Re$ is the Reynolds number. For realistic high-Reynolds-number flows—such as atmospheric turbulence with $Re \sim 10^6$ or boundary layer flows over aircraft with $Re \sim 10^7$—classical computational resources become prohibitively expensive.

It is precisely in this regime of rough , non-smooth flows that quantum algorithms may offer substantial computational advantages. Gaitan[1] demonstrated that quantum approaches to solving the Navier-Stokes equations can provide speedups when the driver function $f(U)$ exhibits low smoothness—a characteristic feature of turbulent flows. We examine two complementary approaches: using quantum amplitude estimation to accelerate Monte Carlo sampling of turbulent statistics , and applying quantum linear solvers to the implicit time-stepping schemes required for stable turbulent flow simulation.

#### 3.3.1 Monte Carlo Approximation

Many practical applications of turbulence do not require the full spatiotemporal resolution of every eddy in the flow. Instead , engineers and scientists often seek statistical properties: mean velocity profiles , Reynolds stresses , turbulent kinetic energy dissipation rates , drag coefficients , or probability distributions of velocity fluctuations. These quantities can be

estimated using Monte Carlo methods , which sample the flow field at random points and times to build up statistical averages.

Classical Monte Carlo methods converge slowly. To estimate an expectation value $\mathbb{E}[f]$ to within error $\epsilon$ with high probability requires $O(1/\epsilon^2)$ samples. For turbulent flows where each sample requires solving the Navier-Stokes equations at a point (or evaluating a surrogate model) , this quadratic dependence on accuracy creates a significant computational burden. For example , achieving 0.1% accuracy ($\epsilon = 10^{-3}$) requires approximately one million samples.

Quantum amplitude estimation[1] provides a quadratic speedup for this task. By encoding the Monte Carlo sampling procedure into quantum amplitudes and using quantum interference , the algorithm achieves $O(1/\epsilon)$ complexity—requiring only one thousand samples to reach the same 0.1% accuracy. This improvement is particularly valuable for turbulent flow statistics where expensive simulations limit the number of samples classical methods can afford.

The quantum advantage emerges from the wave-like nature of quantum computation. Classical Monte Carlo relies on the law of large numbers: averaging many independent random samples to reduce statistical fluctuations. The error decreases as $1/\sqrt{N}$ where $N$ is the number of samples , necessitating four times as many samples to halve the error. Quantum amplitude estimation , in contrast , uses constructive and destructive interference of quantum probability amplitudes to extract information more efficiently. The algorithm repeatedly applies controlled rotations and measurements , with the number of required repetitions scaling linearly with $1/\epsilon$ rather than quadratically.

However , several practical considerations temper this theoretical advantage. First , the quantum algorithm requires encoding the turbulent flow sampling procedure into a quantum oracle—a unitary operator that maps computational basis states to flow statistics. Creating this oracle may itself require classical preprocessing or expensive state preparation. Second , the quadratic speedup applies to the *query complexity* (number of oracle calls) , but each quantum oracle call may be more expensive than a classical sample if the quantum circuit is deep or requires many gates. Third , current NISQ (Noisy Intermediate-Scale Quantum) devices have limited qubit counts and gate fidelities , restricting the size and fidelity of implementable quantum circuits.

For feasibility , we must consider the smoothness parameter $q$ of the underlying flow. Gaitan[1] showed that the quantum speedup is most pronounced when $q \ll 1$ , corresponding to rough , highly non-smooth functions. Turbulent flows , with their sharp gradients and intermittent structures , naturally fall into this regime. In the rough limit with $q, \gamma \ll 1$ , the quantum algorithm's complexity becomes $O(1/\epsilon)$ while classical randomized algorithms require $O(1/\epsilon^2)$—a quadratic speedup. For smooth flows with $q \gg 1$ , both quantum and classical algorithms achieve similar complexity $\Theta((1/\epsilon)^{1/q})$ , and the quantum advantage vanishes.

Therefore , quantum-accelerated Monte Carlo is most promising for statistical analysis of fully-developed turbulence at high Reynolds numbers , where the flow exhibits significant roughness. Applications include estimating turbulent drag on aerospace vehicles , computing probability distributions of velocity increments for turbulence modeling , or sampling extreme events in atmospheric flows. The quadratic speedup could enable statistical convergence studies that are currently infeasible—for instance , accurately quantifying uncertainty in large-eddy simulation models or validating Reynolds-averaged Navier-Stokes closures against high-fidelity DNS data.

#### 3.3.2 Quantum Amplitude Estimation

To understand how quantum amplitude estimation accelerates turbulent flow statistics , we briefly outline the algorithm's structure following Gaitan[1]. Suppose we wish to estimate the expectation value $\mu = \mathbb{E}[g(x)]$ where $x$ is sampled from some distribution relevant to the turbulent flow (for example , spatial locations in the domain or time instances along a trajectory) and $g(x)$ is a quantity of interest (such as local velocity magnitude or vorticity).

Classically , we would sample $N$ independent values $\{x_1, \ldots, x_N\}$ , compute $\{g(x_1), \ldots, g(x_N)\}$ , and estimate $\mu \approx \frac{1}{N}\sum_{i=1}^{N} g(x_i)$. The standard error in this estimate scales as $\sigma/\sqrt{N}$ where $\sigma$ is the standard deviation of $g$ , so achieving error $\epsilon$ requires $N = O(\sigma^2/\epsilon^2)$ samples.

The quantum algorithm proceeds differently. First , we prepare a quantum state that encodes the probability distribution:

$$|\psi\rangle = \sum_x \sqrt{P(x)}\,|x\rangle\,|0\rangle$$

where $P(x)$ is the sampling distribution. Second , we apply a quantum oracle $U_g$ that maps $|x\rangle\,|0\rangle$ to $|x\rangle\,(\sqrt{1 - g(x)/g_{\max}}\,|0\rangle + \sqrt{g(x)/g_{\max}}\,|1\rangle)$ , effectively encoding $g(x)$ as an amplitude in an ancilla

qubit. This creates the state:

$$|\phi\rangle = \sum_x \sqrt{P(x)}\,|x\rangle \left( \sqrt{1 - g(x)/g_{\max}}\,|0\rangle + \sqrt{g(x)/g_{\max}}\,|1\rangle \right)$$

The key observation is that the probability of measuring the ancilla qubit in state $|1\rangle$ is exactly $\sum_x P(x)g(x)/g_{\max} = \mu/g_{\max}$. Quantum amplitude estimation exploits quantum phase estimation to determine this probability—and hence $\mu$—with $O(1/\epsilon)$ applications of controlled versions of the state preparation and oracle operations.

The algorithm constructs the *Grover operator* $Q = (2|\phi\rangle\langle\phi| - I)(2|\psi\rangle\langle\psi| - I)$ which rotates in the two-dimensional subspace spanned by the "good" states (ancilla = 1) and "bad" states (ancilla = 0). By applying quantum phase estimation to $Q$ , we extract the rotation angle $\theta$ where $\sin^2(\theta) = \mu/g_{\max}$ , determining $\mu$ to precision $\epsilon$ with $O(g_{\max}/\epsilon)$ applications of $Q$.

For turbulent flow applications , the oracle $U_g$ must encode the relationship between spatial/temporal location $x$ and the flow quantity $g(x)$. If we have a surrogate model (such as a trained neural network or reduced-order model) that predicts $g(x)$ , this can be compiled into a quantum circuit. Alternatively , if we have precomputed DNS data on a grid , $g(x)$ can be stored in quantum memory and accessed via amplitude encoding. The feasibility of the quantum algorithm thus depends critically on the cost of implementing $U_g$.

For rough turbulent flows , Gaitan[1] showed that the overall complexity—including the cost of constructing and applying the quantum oracle—remains $O(1/\epsilon)$ for Hölder class functions with low smoothness parameter $q \ll 1$. In contrast , classical deterministic algorithms have complexity $\Theta((1/\epsilon)^{1/q})$ , which becomes exponential in $1/q$ for rough functions. This suggests that quantum amplitude estimation could provide exponential speedup over deterministic classical methods for turbulent statistics.

However , we must address practical implementation challenges. Current quantum hardware has limited coherence times and gate fidelities. A quantum circuit for amplitude estimation with error $\epsilon = 0.001$ might require $O(1000)$ applications of the Grover operator , each involving the state preparation and oracle. If the oracle depth is hundreds of gates , the total circuit depth could exceed coherence limits of NISQ devices. Error mitigation techniques or future fault-tolerant quantum computers would be necessary to realize the theoretical speedup.

Additionally , the quadratic speedup is most significant when the number of required samples is large. For coarse-grained turbulent statistics that only need $10^4$ to $10^5$ samples , the difference between classical $O(1/\epsilon^2)$ and quantum $O(1/\epsilon)$ may not justify the overhead of quantum hardware. The quantum advantage becomes compelling for high-precision statistics requiring $10^8$ to $10^{12}$ samples—regimes where classical Monte Carlo becomes prohibitively expensive but quantum amplitude estimation remains tractable.

In summary , quantum amplitude estimation offers a promising path to accelerate statistical analysis of turbulent flows , particularly for high-Reynolds-number regimes where flow roughness is pronounced and high-precision statistics are needed. The quadratic speedup over classical Monte Carlo and potentially exponential speedup over deterministic methods could enable new classes of turbulence studies. Realizing this potential requires advances in quantum hardware (coherence , gate fidelity) , efficient compilation of flow oracles into quantum circuits , and careful assessment of when the quantum advantage outweighs implementation overhead.

### 3.4 Linear Systems from Implicit Time Discretization

Implicit time-stepping schemes are essential for practical CFD simulations , offering unconditional numerical stability and permitting larger time steps than explicit methods. However , this stability comes at a computational cost: implicit discretization transforms the Navier-Stokes equations into large sparse linear systems that must be solved at each time step. For the Finite Volume Method described in Section 2.1 , the implicit Euler scheme discretizes the temporal term as[2]

$$\frac{V_j}{\Delta t}\Delta\mathbf{U}^n + \mathbf{R}(\mathbf{U}^{n+1}) = 0$$

where $V_j$ is the cell volume , $\Delta t$ is the time step , $\Delta\mathbf{U}^n = \mathbf{U}^{n+1} - \mathbf{U}^n$ is the change in conserved variables , and $\mathbf{R}(\mathbf{U})$ is the residual vector containing flux terms. Linearizing the residual via Taylor expansion $\mathbf{R}(\mathbf{U}^{n+1}) \approx \mathbf{R}(\mathbf{U}^n) + \mathbf{J}(\mathbf{U}^n)\Delta\mathbf{U}^n$ where $\mathbf{J} = \frac{\partial\mathbf{R}}{\partial\mathbf{U}}$ is the Jacobian matrix , we obtain the linear system

$$\left[\frac{V_j}{\Delta t}\mathbf{I} + \mathbf{J}(\mathbf{U}^n)\right]\Delta\mathbf{U}^n = -\mathbf{R}(\mathbf{U}^n)$$

This must be solved for $\Delta\mathbf{U}^n$ at every cell $j$ and every time step $n$. For a 3D grid with $N_x \times N_y \times N_z$ cells and $m$ conserved variables per cell , this is a linear system of dimension $N = m \cdot N_x N_y N_z$. The coefficient matrix

is sparse—each row has at most $7m$ nonzero entries (the center cell plus six neighbors)—but for realistic simulations with $N \sim 10^6$ to $10^9$ , solving this system dominates computational cost.

Classical iterative solvers like GMRES or BiCGSTAB are the workhorses of industrial CFD codes , requiring $O(N)$ to $O(N \log N)$ operations per iteration depending on sparsity and conditioning. Quantum linear solvers offer the tantalizing prospect of exponential speedup: the HHL algorithm[1] achieves complexity $O(\log N \cdot s^2\kappa^2/\epsilon)$ where $s$ is the sparsity , $\kappa$ is the condition number , and $\epsilon$ is the desired accuracy. However , this theoretical advantage faces severe practical obstacles that we must address for quantum CFD to be viable.

### 3.4.1 Quantum Linear Solvers

Quantum linear solvers fall into two categories: direct methods like HHL that encode the solution in quantum amplitudes via phase estimation , and variational methods like VQLS that iteratively minimize a cost function in a hybrid quantum-classical loop[2]. Each approach has distinct advantages and limitations for CFD applications.

The **HHL algorithm** (Harrow-Hassidim-Lloyd[1]) is the canonical quantum linear solver. For a linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ where $\mathbf{A}$ is an $N \times N$ Hermitian matrix , HHL constructs a quantum state $|\mathbf{x}\rangle$ proportional to the solution via three main steps[2]: (1) encode $\mathbf{b}$ into a quantum state $|\mathbf{b}\rangle$ using amplitude encoding , (2) apply quantum phase estimation to extract eigenvalues of $\mathbf{A}$ , and (3) perform controlled rotations to invert eigenvalues and construct $|\mathbf{x}\rangle \propto \sum_j \beta_j \lambda_j^{-1} |u_j\rangle$ where $\lambda_j$ and $|u_j\rangle$ are eigenvalues and eigenvectors of $\mathbf{A}$ , and $\mathbf{b} = \sum_j \beta_j |u_j\rangle$.

The algorithmic complexity is $O(\log N \cdot s^2\kappa^2/\epsilon)$ where $s$ is the maximum number of nonzero entries per row (sparsity) and $\kappa = \lambda_{\max}/\lambda_{\min}$ is the condition number. This represents exponential speedup in $\log N$ over classical methods. However , the algorithm requires several formidable resources. First , it needs $n = \lceil \log_2 N \rceil + \lceil \log_2 \kappa \rceil + 1$ qubits—for a modest $64^3$ grid with 4 conserved variables per cell , this is $N = 4 \times 262{,}144 \approx 2^{20}$ requiring at least 20 qubits just for the system , plus additional qubits for phase estimation depending on $\kappa$. Second , the circuit depth scales with the complexity , requiring coherent evolution through thousands of gates. Third , the output is the quantum state $|\mathbf{x}\rangle$ , not the classical vector $\mathbf{x}$—extracting all $N$ components via measurement negates the quantum advantage , so HHL is only useful when downstream processing can work directly with $|\mathbf{x}\rangle$ or when only aggregate quantities like $\|\mathbf{x}\|$ are needed.

For CFD , the sparsity $s \sim 7$ is favorable (six neighbors plus center in 3D) , but condition numbers are problematic. Without preconditioning , $\kappa \sim 10^3$ to $10^6$ for typical implicit CFD systems , dramatically inflating both qubit count and circuit depth. Classical preconditioners like incomplete LU factorization can reduce $\kappa$ to $\sim 10$ to $100$[2] , but encoding preconditioned systems into quantum circuits adds complexity. Moreover , the fundamental limitation remains: we typically need the full velocity field $\mathbf{u}(\mathbf{x}, t)$ for visualization , boundary condition updates , and nonlinear term computation—not just its quantum encoding.

The **Variational Quantum Linear Solver (VQLS)** offers an alternative approach better suited to NISQ hardware[2]. VQLS constructs a Hamiltonian $H = \mathbf{A}^\dagger(\mathbf{I} - |\mathbf{b}\rangle\langle\mathbf{b}|)\mathbf{A}$ whose ground state is the normalized solution $|\mathbf{x}\rangle$. A parameterized quantum circuit (ansatz) $|\psi(\boldsymbol{\theta})\rangle$ approximates this ground state , and a classical optimizer adjusts parameters $\boldsymbol{\theta}$ to minimize the cost function

$$C(\boldsymbol{\theta}) = \langle\psi(\boldsymbol{\theta})| H |\psi(\boldsymbol{\theta})\rangle = \|\mathbf{A}|\psi(\boldsymbol{\theta})\rangle\|^2 - |\langle\mathbf{b}|\mathbf{A}|\psi(\boldsymbol{\theta})\rangle|^2$$

The gradient $\nabla C(\boldsymbol{\theta})$ is computed via quantum measurements using parameter-shift rules or finite differences , and classical gradient descent updates $\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \beta\nabla C(\boldsymbol{\theta}_t)$ until convergence.

VQLS has several advantages for near-term quantum hardware. First , it uses shallow , variable-depth circuits that can be tailored to available qubit connectivity and coherence times. Typical ansätze like hardware-efficient ansatz require only $n = \lceil \log_2 N \rceil$ qubits and depth $O(d)$ where $d$ is the number of ansatz layers—far less demanding than HHL's deep phase estimation circuits. Second , VQLS is more robust to noise since the variational approach naturally averages over many measurements , and imperfect gates simply slow convergence rather than causing catastrophic failure. Third , Hamiltonian morphing techniques[2] can improve convergence for ill-conditioned systems by gradually transitioning from an easy problem (identity matrix) to the target system.

However , VQLS also faces challenges. Convergence is not guaranteed and depends critically on ansatz design—poor ansatze may get trapped in local minima or require exponentially many parameters to represent the solution. The number of iterations to reach accuracy $\epsilon$ is not rigorously bounded , though empirically it often requires $O(\text{poly}(\kappa, 1/\epsilon))$ iterations. Each iteration involves multiple quantum circuit executions (to evaluate cost and gradient) and classical optimization steps , creating overhead. For CFD where we

solve similar linear systems at consecutive time steps , warm-starting with parameters from the previous time step can significantly reduce iterations , but this has not been extensively studied.

Most critically , both HHL and VQLS face the same output problem: they produce quantum states , not classical vectors. Quantum state tomography—the process of reconstructing the classical vector from measurement statistics—requires $O(N)$ measurements and post-processing , destroying the quantum advantage. Sparse tomography methods[2] can reduce this to $O(s \log N)$ for $s$-sparse vectors , but CFD velocity fields are not generally sparse in the computational basis. Recent work has explored reading out only specific quantities (drag coefficients , energy norms) without full tomography , but this limits applicability.

#### 3.4.2 Sub-QLS with Krylov Subspace Methods

The mismatch between quantum linear solvers (which produce quantum states) and CFD requirements (which need classical vectors) can be partially resolved through **hybrid quantum-classical Krylov subspace methods**[2]. The key insight is to use quantum solvers not on the original $N$-dimensional system , but on a much smaller $k$-dimensional system where $k \ll N$ , making full state tomography tractable.

Krylov subspace methods like GMRES construct an approximate solution in the Krylov subspace $\mathcal{K}_k = \text{span}\{\mathbf{r}_0, \mathbf{A}\mathbf{r}_0, \mathbf{A}^2\mathbf{r}_0, \ldots, \mathbf{A}^{k-1}\mathbf{r}_0\}$ where $\mathbf{r}_0 = \mathbf{b} - \mathbf{A}\mathbf{x}_0$ is the initial residual. Using the Arnoldi process , we construct an orthonormal basis $\mathbf{V}_k = [\mathbf{v}_1, \ldots, \mathbf{v}_k]$ for $\mathcal{K}_k$ such that $\mathbf{A}\mathbf{V}_k = \mathbf{V}_{k+1}\mathbf{H}_k$ where $\mathbf{H}_k$ is a $(k+1) \times k$ upper Hessenberg matrix. The original linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ is projected onto this subspace , yielding the small system

$$\mathbf{H}_k \mathbf{y} = \|\mathbf{r}_0\| \mathbf{e}_1$$

where $\mathbf{e}_1 = (1, 0, \ldots, 0)^T$. Solving this $k \times k$ system for $\mathbf{y}$ gives the approximate solution $\mathbf{x} \approx \mathbf{x}_0 + \mathbf{V}_k \mathbf{y}$.

The **SUB-QLS (Subspace Quantum Linear Solver)** framework[2] uses a quantum solver (HHL or VQLS) to solve the small projected system $\mathbf{H}_k \mathbf{y} = \|\mathbf{r}_0\| \mathbf{e}_1$. Since $k$ is typically 5 to 20 for CFD applications (chosen to balance convergence and quantum resource requirements) , the quantum solver operates on only $\lceil \log_2 k \rceil \approx 3$ to 5 qubits. This dramatically reduces circuit depth and makes full quantum state tomography feasible—reading out a 5-dimensional vector requires only $2^5 = 32$ measurement bases. The algorithm proceeds iteratively: construct Krylov subspace on classical computer $\rightarrow$ encode $\mathbf{H}_k$ into quantum circuit $\rightarrow$ solve with quantum linear solver $\rightarrow$ read out $\mathbf{y}$ via tomography $\rightarrow$ update solution $\mathbf{x} = \mathbf{x}_0 + \mathbf{V}_k \mathbf{y} \rightarrow$ restart if not converged.

The computational complexity becomes hybrid. Classical Arnoldi iteration costs $O(ksN)$ operations to construct $\mathbf{V}_k$ and $\mathbf{H}_k$ where $s$ is sparsity. The quantum solver costs $O(\log k \cdot \kappa_k^2)$ for HHL or $O(\text{poly}(\kappa_k))$ iterations for VQLS , where $\kappa_k$ is the condition number of $\mathbf{H}_k$ (often better than $\kappa$ for $\mathbf{A}$). Tomography costs $O(k \cdot M)$ where $M$ is the number of measurement shots per basis. If we require $m$ outer Krylov restarts to converge , the total classical cost is $O(mksN)$ and quantum cost is $O(m \log k \cdot \kappa_k^2)$ per restart.

Crucially , the quantum speedup is now over the $k$-dimensional solve , not the full $N$-dimensional system. Classical methods solve the $k \times k$ system in $O(k^3)$ or $O(k^2)$ operations , while quantum methods achieve $O(\log k)$ or $O(\text{poly}(\log k))$. For small $k$ , this speedup is modest in absolute terms (solving a $10 \times 10$ system classically is trivial) , but the quantum advantage compounds across thousands of time steps and nonlinear iterations in a CFD simulation. Moreover , the framework is compatible with classical preconditioning: apply preconditioner $\mathbf{M}^{-1}$ classically to get $\mathbf{M}^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}^{-1}\mathbf{b}$ , then use SUB-QLS on the preconditioned system with improved $\kappa$.

The feasibility of SUB-QLS for CFD has been demonstrated on NISQ hardware[2]. Recent work solved linear systems of dimension $N \sim 5000$ arising from 2D Navier-Stokes simulations using subspace dimension $k = 8$ on a superconducting quantum computer , achieving agreement with classical GMRES to within a few percent. For 3D simulations with $N \sim 10^6$ , the algorithm remains tractable on near-term devices since the quantum solver only handles the $k$-dimensional projected system.

However , several practical considerations temper enthusiasm. First , the overhead of quantum state preparation , tomography , and classical-quantum communication must be accounted for. If preparing the quantum state for $\mathbf{H}_k$ takes longer than solving the $k \times k$ system classically , the quantum approach offers no advantage. Second , quantum solver accuracy directly impacts Krylov convergence. If the quantum solution to $\mathbf{H}_k \mathbf{y} = \|\mathbf{r}_0\| \mathbf{e}_1$ has error $\epsilon_q$ , this propagates through $\mathbf{x} = \mathbf{x}_0 + \mathbf{V}_k \mathbf{y}$ , potentially requiring more Krylov restarts or reducing the achievable CFD solution accuracy. VQLS convergence tolerance and HHL approximation error must be carefully chosen relative to CFD residual targets. Third , for unsteady simulations , the time spent on quantum linear solves must not exceed the physical time step—otherwise the simulation cannot keep pace with real-time applications like flow control or weather forecasting.

Despite these challenges , SUB-QLS represents a pragmatic path toward quantum-accelerated CFD in the NISQ era[2]. By confining quantum computation to small subspace solves where near-term hardware can excel , while leveraging classical computers for spatial discretization , Arnoldi iteration , and post-processing , the framework achieves a practical division of labor. As quantum hardware improves (increasing qubit count , coherence , and gate fidelity) , the subspace dimension $k$ can be increased and eventually the full system solved directly with HHL or VQLS , realizing the full theoretical quantum advantage. For now , the hybrid approach offers a feasible stepping stone , demonstrating quantum utility for CFD while working within current hardware constraints.

# 4 Applications

## 4.1 Aerospace Flight Vehicle Design

## 4.2 Weather Forecasting

## 4.3 Astrophysics

## 4.4 Blood Flows

## A Mathematical Definitions & Proofs

**Lemma 1.**
Let $f : \mathbb{R}^m \times \mathbb{R}^k \to \mathbb{R}^m$. We can characterize the input of $f$ as $(x_1, \ldots, x_m, x_{m+1}, \ldots, x_{m+k})$. Let $x = (x_1, \ldots, x_m) \in \mathbb{R}^m$ denote the first $m$ components of the input. Then ,

$$\left[\left\langle f, \frac{\partial}{\partial x} \right\rangle\right] f = \left(\frac{\partial f}{\partial x}\right) f$$

where $\langle \cdot, \cdot \rangle$ denotes the dot product , and $\frac{\partial f}{\partial x}$ is the Jacobian matrix with entries $\left(\frac{\partial f}{\partial x}\right)_{ij} = \frac{\partial f_i}{\partial x_j}$ for $i, j \in \{1, \ldots, m\}$.

♦

*Proof.*

$$\begin{aligned}
\left[\left\langle f, \frac{\partial}{\partial x} \right\rangle\right] f &= \left[f_1 \frac{\partial}{\partial x_1} + \ldots + f_m \frac{\partial}{\partial x_m}\right] \begin{pmatrix} f_1 \\ \vdots \\ f_m \end{pmatrix} \\
&= \begin{pmatrix} f_1 \frac{\partial f_1}{\partial x_1} + \ldots + f_m \frac{\partial f_1}{\partial x_m} \\ \vdots \\ f_1 \frac{\partial f_m}{\partial x_1} + \ldots + f_m \frac{\partial f_m}{\partial x_m} \end{pmatrix} \\
&= \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_m} \end{pmatrix} \begin{pmatrix} f_1 \\ \vdots \\ f_m \end{pmatrix} \\
&= \left(\frac{\partial f}{\partial x}\right) f
\end{aligned}$$

□

**Definition 1.** Conjugate Transpose (Hermitian Adjoint)
Let $A$ be an $m \times n$ complex matrix. The *conjugate transpose* (or *Hermitian adjoint*) of $A$ , denoted $A^\dagger$ , is the $n \times m$ matrix defined by

$$(A^\dagger)_{ij} = \overline{A_{ji}}$$

where $\overline{z}$ denotes the complex conjugate of $z$. Equivalently , $A^\dagger = (\bar{A})^T$ where $\bar{A}$ is the element-wise complex conjugate and $(\cdot)^T$ is the transpose.

For a vector $|\psi\rangle \in \mathbb{C}^n$ written in column form , the conjugate transpose $\langle\psi| = |\psi\rangle^\dagger$ is a row vector. The inner product of $|\phi\rangle$ and $|\psi\rangle$ is $\langle\phi|\psi|\phi|\psi\rangle = \langle\phi|\psi\rangle = \sum_{i=1}^n \overline{\phi_i}\psi_i$.

**Properties:**

- $(A^\dagger)^\dagger = A$
- $(AB)^\dagger = B^\dagger A^\dagger$
- $(A + B)^\dagger = A^\dagger + B^\dagger$
- $(cA)^\dagger = \bar{c}A^\dagger$ for scalar $c \in \mathbb{C}$

⧫

**Definition 2.** Unitary Matrix
A square complex matrix $U \in \mathbb{C}^{n\times n}$ is *unitary* if

$$U^\dagger U = UU^\dagger = I$$

where $I$ is the identity matrix. Equivalently , $U^{-1} = U^\dagger$.

**Properties:**

- Unitary matrices preserve inner products: $\langle U\phi|U\psi|U\phi|U\psi\rangle = \langle\phi|\psi|\phi|\psi\rangle$
- Unitary matrices preserve norms: $\|U\,|\psi\rangle\,\| = \|\,|\psi\rangle\,\|$
- The columns (and rows) of a unitary matrix form an orthonormal basis
- The determinant satisfies $|\det(U)| = 1$
- Eigenvalues of unitary matrices have modulus 1: $|\lambda| = 1$

In quantum mechanics , all reversible evolutions are represented by unitary operators.

⧫

**Definition 3.** Tensor Product
Let $V$ and $W$ be vector spaces. The *tensor product* $V \otimes W$ is a vector space whose elements are formal linear combinations of pairs $(v, w)$ with $v \in V$ and $w \in W$ , denoted $v \otimes w$.

For finite-dimensional spaces with bases $\{v_i\}$ for $V$ and $\{w_j\}$ for $W$ , the tensor product $V \otimes W$ has basis $\{v_i \otimes w_j\}$. If $\dim(V) = m$ and $\dim(W) = n$ , then $\dim(V \otimes W) = mn$.

For vectors $|\psi\rangle = \sum_i \alpha_i\,|v_i\rangle \in V$ and $|\phi\rangle = \sum_j \beta_j\,|w_j\rangle \in W$ , the tensor product is

$$|\psi\rangle \otimes |\phi\rangle = \sum_{i,j} \alpha_i\beta_j\,|v_i\rangle \otimes |w_j\rangle$$

In component form for $|\psi\rangle \in \mathbb{C}^m$ and $|\phi\rangle \in \mathbb{C}^n$ , the tensor product is the $mn$-dimensional vector

$$|\psi\rangle \otimes |\phi\rangle = \begin{pmatrix} \psi_1\phi_1 \\ \psi_1\phi_2 \\ \vdots \\ \psi_1\phi_n \\ \psi_2\phi_1 \\ \vdots \\ \psi_m\phi_n \end{pmatrix}$$

For matrices $A \in \mathbb{C}^{m\times m}$ and $B \in \mathbb{C}^{n\times n}$ , the tensor product $A \otimes B \in \mathbb{C}^{mn\times mn}$ is

$$A \otimes B = \begin{pmatrix} A_{11}B & A_{12}B & \cdots & A_{1m}B \\ A_{21}B & A_{22}B & \cdots & A_{2m}B \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1}B & A_{m2}B & \cdots & A_{mm}B \end{pmatrix}$$

**Properties:**

- $(A \otimes B)(C \otimes D) = (AC) \otimes (BD)$
- $(A \otimes B)^\dagger = A^\dagger \otimes B^\dagger$
- $\mathrm{Tr}(A \otimes B) = \mathrm{Tr}(A) \cdot \mathrm{Tr}(B)$

In quantum computing , an $n$-qubit system is the tensor product of $n$ single-qubit spaces: $(\mathbb{C}^2)^{\otimes n}$.

♦

**Definition 4.** Dirac Notation (Bra-Ket)
*Dirac notation* is a standard notation in quantum mechanics for vectors and linear functionals in Hilbert spaces.

- A *ket* $|\psi\rangle$ represents a column vector (state) in $\mathbb{C}^n$
- A *bra* $\langle\phi|$ represents a row vector (dual state) , defined as $\langle\phi| = |\phi\rangle^\dagger$
- The *inner product* is $\langle\phi|\psi|\phi|\psi\rangle = \sum_i \overline{\phi_i}\psi_i \in \mathbb{C}$
- The *outer product* is $|\psi\rangle\langle\phi|$ , an $n \times n$ matrix with entries $(|\psi\rangle\langle\phi|)_{ij} = \psi_i\overline{\phi_j}$

For computational basis states , we write $|0\rangle = \begin{pmatrix}1\\0\end{pmatrix}$ and $|1\rangle = \begin{pmatrix}0\\1\end{pmatrix}$. Multi-qubit basis states are tensor products: $|01\rangle = |0\rangle \otimes |1\rangle$. In general , $|x\rangle$ denotes the basis state corresponding to integer $x$ in binary.

**Useful identities:**

- $\langle\psi|\psi|\psi|\psi\rangle = \| |\psi\rangle \|^2 = \sum_i |\psi_i|^2$
- $|\psi\rangle\langle\psi|$ is the projector onto $|\psi\rangle$
- Completeness: $\sum_i |i\rangle\langle i| = I$ for orthonormal basis $\{|i\rangle\}$

♦

**Definition 5.** Bloch Sphere
Any single-qubit pure state can be represented as a point on the unit sphere in $\mathbb{R}^3$ , called the *Bloch sphere*. A general qubit state is

$$|\psi\rangle = \cos(\theta/2)\,|0\rangle + e^{i\phi}\sin(\theta/2)\,|1\rangle$$

where $\theta \in [0, \pi]$ and $\phi \in [0, 2\pi)$ are the polar and azimuthal angles on the Bloch sphere. The state $|\psi\rangle$ corresponds to the point $(\sin\theta\cos\phi, \sin\theta\sin\phi, \cos\theta)$ in Cartesian coordinates.

Special points:

- North pole: $|0\rangle$ (eigenstate of $Z$ with eigenvalue $+1$)
- South pole: $|1\rangle$ (eigenstate of $Z$ with eigenvalue $-1$)
- Equator: superpositions like $|+\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$ and $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$

Rotations on the Bloch sphere correspond to unitary operations. For example , $R_z(\phi)$ rotates around the $z$-axis by angle $\phi$ , and $R_x(\theta)$ rotates around the $x$-axis by angle $\theta$.

**Note:** The Bloch sphere representation only applies to single qubits. Multi-qubit states , especially entangled states , cannot be visualized on a single Bloch sphere.

♦

**Definition 6.** Quantum Entanglement

A multi-qubit state $|\psi\rangle \in (\mathbb{C}^2)^{\otimes n}$ is *entangled* if it cannot be written as a tensor product of single-qubit states. Equivalently , $|\psi\rangle$ is entangled if it cannot be decomposed as

$$|\psi\rangle \neq |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots \otimes |\psi_n\rangle$$

for any choice of single-qubit states $|\psi_i\rangle \in \mathbb{C}^2$.

**Bell states** are maximally entangled two-qubit states:

$$|\Phi^+\rangle = \frac{1}{\sqrt{2}}(|00\rangle + |11\rangle)$$
$$|\Phi^-\rangle = \frac{1}{\sqrt{2}}(|00\rangle - |11\rangle)$$
$$|\Psi^+\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle)$$
$$|\Psi^-\rangle = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle)$$

Entanglement is a uniquely quantum resource with no classical analog. Measuring one qubit of an entangled state instantaneously affects the measurement statistics of the other qubit , regardless of spatial separation. This is not due to classical correlation but genuine quantum correlation.

For pure bipartite states $|\psi\rangle_{AB}$ , entanglement can be quantified by the *von Neumann entropy* of the reduced density matrix. A state is separable (not entangled) if and only if its reduced density matrices are pure.

♦

**Definition 7.** Quantum Oracle

A *quantum oracle* (or black-box) is a unitary operator $U_f$ that encodes a classical function $f : \{0,1\}^n \to \{0,1\}^m$ into quantum gates. The oracle is typically defined by its action on computational basis states:

$$U_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle$$

where $|x\rangle \in (\mathbb{C}^2)^{\otimes n}$ , $|y\rangle \in (\mathbb{C}^2)^{\otimes m}$ , and $\oplus$ denotes bitwise XOR (addition modulo 2).

The oracle leaves the input register $|x\rangle$ unchanged and XORs the function value $f(x)$ into the output register $|y\rangle$. By linearity , if the input is in superposition , the oracle acts on all basis states simultaneously:

$$U_f \left( \sum_x \alpha_x |x\rangle \right) |0\rangle = \sum_x \alpha_x |x\rangle |f(x)\rangle$$

Oracles are used in quantum algorithms to model access to input data or problem structure. The *query complexity* of an algorithm is the number of oracle calls required. Many quantum speedups (Grover , Deutsch-Jozsa , Simon) are formulated in the oracle model.

**Phase oracle:** An alternative formulation uses a phase kickback. If the output register is initialized to $|-\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ , then

$$U_f |x\rangle |-\rangle = (-1)^{f(x)} |x\rangle |-\rangle$$

The function value is encoded as a phase rather than in a separate register.

♦

## B Physical Derivations

**Derivation 1.** Pressure Force in a Fluid

We first begin by briefly stating where pressure arises from. At the microscopic level , pressure exists because of molecular collisions with surfaces. When molecules strike a surface , they transfer momentum , creating a force distributed over the area. This momentum transfer per unit area defines the pressure $p$ at that location.

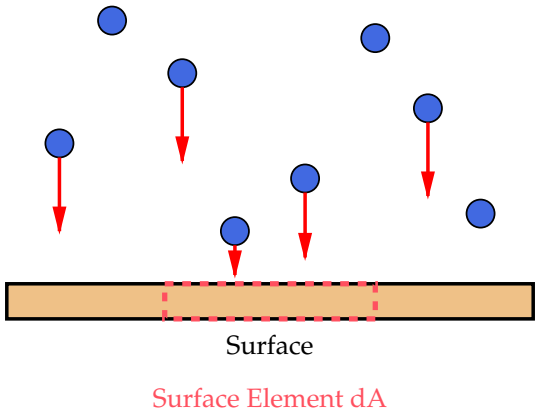


Figure 7: Molecular collisions with a surface element creating pressure.

Having established the microscopic origin , we now seek to determine how pressure creates forces within a fluid. To do this , we consider an infinitesimal rectangular element with one corner at the point $\mathbf{x}' = (x_1', x_2', x_3') \in \mathbb{R}^3$ and dimensions $\Delta x_1 \times \Delta x_2 \times \Delta x_3$. We define this rectangular prism as:

$$\mathcal{P} = \{(a, b, c) \in \mathbb{R}^3 : x_1' \leq a \leq x_1' + \Delta x_1,\ x_2' \leq b \leq x_2' + \Delta x_2,\ x_3' \leq c \leq x_3' + \Delta x_3\}$$

with volume $\Delta V = \Delta x_1 \Delta x_2 \Delta x_3$. Below is a diagram ,

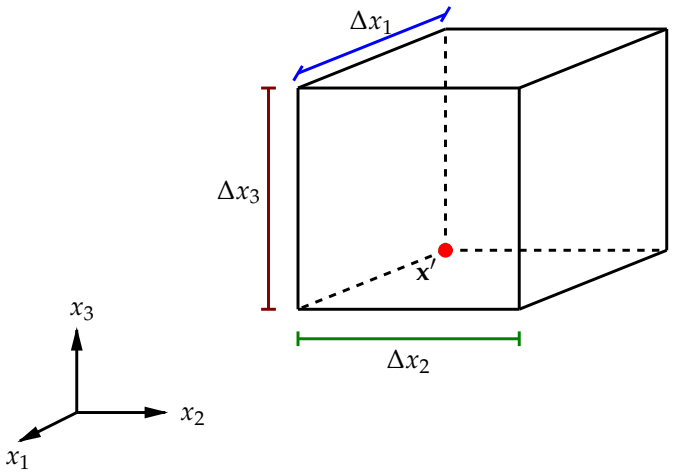


Figure 8: Infinitesimal fluid prism $\mathcal{P}$ used to derive the pressure force.

By analyzing the forces on this element in each direction and taking the limit as $\Delta x_i \to 0$ , we obtain the pressure force per unit volume at the point $\mathbf{x}'$.

It is crucial to note that we imagine the rectangular element itself contains no fluid—it is merely a mathematical control volume. All pressure forces acting on this element come from the external fluid surrounding it , pushing inward on each face from the outside.

Let us now analyze the forces in the $x_1$ direction. Consider the two faces of $\mathcal{P}$ perpendicular to the $x_1$ axis. We define these faces as:

$$\mathcal{R}_1 = \{(x_1', b, c) \in \mathbb{R}^3 : x_2' \le b \le x_2' + \Delta x_2,\ x_3' \le c \le x_3' + \Delta x_3\}$$
$$\mathcal{R}_2 = \{(x_1' + \Delta x_1, b, c) \in \mathbb{R}^3 : x_2' \le b \le x_2' + \Delta x_2,\ x_3' \le c \le x_3' + \Delta x_3\}$$

The face $\mathcal{R}_1$ (the "left wall" at position $x_1'$) experiences pressure from the fluid to its left , pushing the element rightward (in the $+x_1$ direction). The face $\mathcal{R}_2$ (the "right wall" at position $x_1' + \Delta x_1$) experiences pressure from the fluid to its right , pushing the element leftward (in the $-x_1$ direction). Below is a projection along the $x_1x_3$ plane showing this.

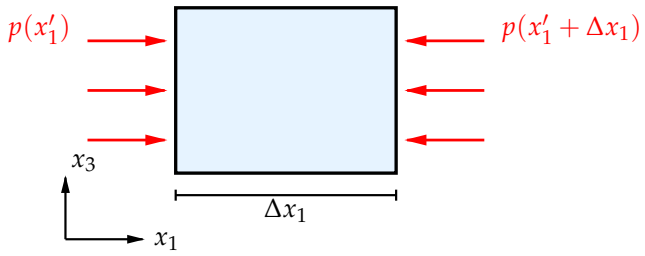


Figure 9: Projection of Fluid Prism on the $x_1x_3$ plane.

The force on each face is given by integrating the pressure over that face:

$$F_1^{(\text{left})} = \iint_{\mathcal{R}_1} p(x_1', x_2, x_3)\, dx_2\, dx_3$$
$$F_1^{(\text{right})} = \iint_{\mathcal{R}_2} p(x_1' + \Delta x_1, x_2, x_3)\, dx_2\, dx_3$$

By the *Mean Value Theorem* for integrals , there exist points $\bar{x}_2, \bar{x}_3 \in [x_2', x_2' + \Delta x_2] \times [x_3', x_3' + \Delta x_3]$ on $\mathcal{R}_1$ and $\bar{x}_2', \bar{x}_3' \in [x_2', x_2' + \Delta x_2] \times [x_3', x_3' + \Delta x_3]$ on $\mathcal{R}_2$ such that:

$$F_1^{(\text{left})} = p(x_1', \bar{x}_2, \bar{x}_3) \cdot \Delta x_2 \Delta x_3$$
$$F_1^{(\text{right})} = p(x_1' + \Delta x_1, \bar{x}_2', \bar{x}_3') \cdot \Delta x_2 \Delta x_3$$

The net force in the $x_1$ direction is the force pushing right minus the force pushing left:

$$F_1 = F_1^{(\text{left})} - F_1^{(\text{right})} = [p(x_1', \bar{x}_2, \bar{x}_3) - p(x_1' + \Delta x_1, \bar{x}_2', \bar{x}_3')]\Delta x_2 \Delta x_3$$

This makes physical sense—if pressure is higher on the left than on the right , the element experiences a net rightward force toward lower pressure.

We now compute the force per unit volume before taking any limits. Factoring out the negative sign:

$$\frac{F_1}{\Delta V} = -\frac{p(x_1' + \Delta x_1, \bar{x}_2', \bar{x}_3') - p(x_1', \bar{x}_2, \bar{x}_3)}{\Delta x_1 \Delta x_2 \Delta x_3} \cdot \Delta x_2 \Delta x_3$$

The $\Delta x_2$ and $\Delta x_3$ terms cancel:

$$\frac{F_1}{\Delta V} = -\frac{p(x_1' + \Delta x_1, \bar{x}_2', \bar{x}_3') - p(x_1', \bar{x}_2, \bar{x}_3)}{\Delta x_1}$$

We now take the continuum limit carefully. First , take the limit as $\Delta x_2 \to 0$. Since $\bar{x}_2, \bar{x}_2' \in [x_2', x_2' + \Delta x_2]$ , as $\Delta x_2 \to 0$ both $\bar{x}_2 \to x_2'$ and $\bar{x}_2' \to x_2'$. Thus:

$$\lim_{\Delta x_2 \to 0} \frac{F_1}{\Delta V} = -\frac{p(x_1' + \Delta x_1, x_2', \bar{x}_3') - p(x_1', x_2', \bar{x}_3)}{\Delta x_1}$$

Next , take the limit as $\Delta x_3 \to 0$. Since $\bar{x}_3, \bar{x}_3' \in [x_3', x_3' + \Delta x_3]$ , as $\Delta x_3 \to 0$ both $\bar{x}_3 \to x_3'$ and $\bar{x}_3' \to x_3'$. Thus:

$$\lim_{\Delta x_3 \to 0} \lim_{\Delta x_2 \to 0} \frac{F_1}{\Delta V} = -\frac{p(x_1' + \Delta x_1, x_2', x_3') - p(x_1', x_2', x_3')}{\Delta x_1}$$

Finally , take the limit as $\Delta x_1 \to 0$. By the definition of the partial derivative:

$$\begin{aligned}
\lim_{\Delta x_1 \to 0} \lim_{\Delta x_3 \to 0} \lim_{\Delta x_2 \to 0} \frac{F_1}{\Delta V} &= \frac{F_1}{dV} \\
&= -\lim_{\Delta x_1 \to 0} \frac{p(x_1' + \Delta x_1, x_2', x_3') - p(x_1', x_2', x_3')}{\Delta x_1} \\
&= -\frac{\partial p}{\partial x_1}\bigg|_{\mathbf{x}'}
\end{aligned}$$

Repeating this analysis for the $x_2$ and $x_3$ directions yields:

$$\frac{F_2}{dV} = -\frac{\partial p}{\partial x_2}\bigg|_{\mathbf{x}'} \quad \text{and} \quad \frac{F_3}{dV} = -\frac{\partial p}{\partial x_3}\bigg|_{\mathbf{x}'}$$

Combining all three components into a vector , the total pressure force at an infinitesmal volume element at point $\mathbf{x}'$ is thus

$$\frac{\mathbf{F}_p}{dV} = -\begin{pmatrix} \frac{\partial p}{\partial x_1} \\ \frac{\partial p}{\partial x_2} \\ \frac{\partial p}{\partial x_3} \end{pmatrix}\bigg|_{\mathbf{x}'} = -\nabla p(\mathbf{x}')$$

More compactly we write ,

$$\frac{\mathbf{F}_p}{dV} = -\nabla p$$

This result shows that fluid elements are pushed from regions of high pressure toward regions of low pressure , with the force to the pressure gradient.

♦

## C Additional Figures

```
// Device code: runs on GPU
__global__ void dotProductKernel(float* a, float* b,
float* partial, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N) {
        partial[idx] = a[idx] * b[idx];
    }
}

// Host code: runs on CPU
void computeDotProduct(float* a, float* b, int N) {
    float *d_a, *d_b, *d_partial;

    // Allocate device memory
    cudaMalloc(&d_a, N * sizeof(float));
    cudaMalloc(&d_b, N * sizeof(float));
    cudaMalloc(&d_partial, N * sizeof(float));

    // Copy data from host to device
    cudaMemcpy(d_a, a, N * sizeof(float),
    cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, b, N * sizeof(float),
    cudaMemcpyHostToDevice);

    // Launch kernel with N threads
    int threads = 256;
    int blocks = (N + threads - 1) / threads;
    dotProductKernel<<<blocks, threads>>>(d_a, d_b,
    d_partial, N);

    // Sum partial results on CPU (simplified)
    float result = 0;
    float* h_partial = new float[N];
    cudaMemcpy(h_partial, d_partial, N * sizeof(float),
    cudaMemcpyDeviceToHost);
    for (int i = 0; i < N; i++) result += h_partial[i];

    // Free memory
    cudaFree(d_a); cudaFree(d_b); cudaFree(d_partial);
}
```

Figure 10: CUDA code for parallel vector dot product computation.

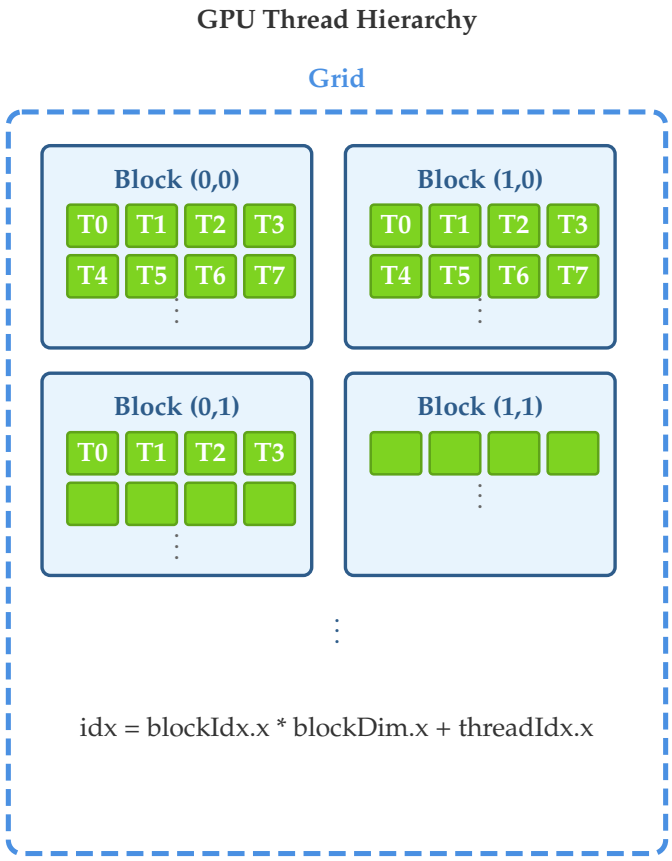


Figure 11: GPU thread hierarchy: threads are organized into blocks , and blocks form a grid. Each thread has a unique global index.

## References


[1] Frank Gaitan. Finding flows of a navier–stokes fluid through quantum computing. *npj Quantum Information*, 6(1):61, 2020.

[2] Chuang-Chao Ye, Ning-Bo An, Teng-Yang Ma, Meng-Han Dou, Wen Bai, De-Jun Sun, Zhao-Yun Chen, and Guo-Ping Guo. A hybrid quantum-classical framework for computational fluid dynamics. *Physics of Fluids*, 36(12), December 2024.