Discrete, Circulation-Preserving, and Stable Simplicial Fluids

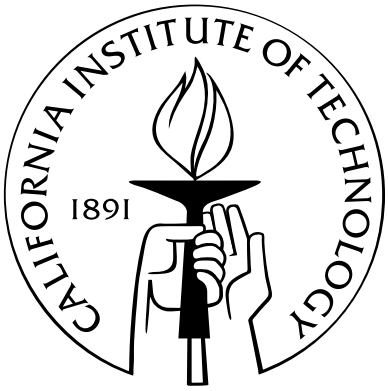
Thesis by

Sharif Elcott

In Partial Fulfillment of the Requirements

for the Degree of

Master of Science



California Institute of Technology

Pasadena, California

2005

(Submitted May 27, 2005)

Abstract

Visual quality, low computational cost, and numerical stability are foremost goals in computer animation. An important ingredient in achieving these goals is the conservation of fundamental motion invariants. For example, rigid and deformable body simulation has benefited greatly from conservation of linear and angular momenta. In the case of fluids, however, none of the current techniques focuses on conserving invariants, and consequently they often introduce a visually disturbing numerical diffusion of *vorticity*. Visually just as important is the resolution of complex simulation domains. Doing so with regular (even if adaptive) grid techniques can be computationally delicate.

In this thesis we describe a novel technique for the simulation of fluid flows. It is designed to respect the defining differential properties, *i.e.*, the *conservation of circulation* along arbitrary loops as they are transported by the flow. Consequently, our method offers several new and desirable properties: (1) arbitrary simplicial meshes (triangles in 2D, tetrahedra in 3D) can be used to define the fluid domain; (2) the computations are efficient due to discrete operators with small support; (3) the method is stable for arbitrarily large time steps; (4) it preserves *discrete circulation* avoiding numerical diffusion of vorticity; and (5) its implementation is straightforward.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Conservation of motion invariants at the discrete computational level is an important ingredient in the construction of numerically stable simulations with a high degree of visual realism [24]. For example, failure to preserve linear and angular momenta in solid mechanics simulations can result in noticeable qualitative inaccuracy. So far, advances of this type have yet to deeply impact fluid flow simulations. Current methods in fluid simulation are rarely designed to conserve *defining physical properties*. Consider, for example, the need in many methods to continually project the numerically updated velocity field onto the set of divergence free velocity fields; or the need to continually reinject vorticity lost due to numerical dissipation as a simulation progresses.

We present a simulation algorithm for incompressible fluids that, by construction, preserves discrete notions of Kelvin's circulation theorem as well as the divergence-free constraint. Instead of simply discretizing the governing differential equations—the Euler equations for inviscid flows and the Navier-Stokes equations for viscous flows—we take a *geometric* approach to the solution of the system. In recent years, there has been a renewed emphasis on the geometric structure of physical systems as a key feature for developing reliable and efficient numerical methods that better respect the underlying physics. Computational Electro-Magnetism (E&M) and Discrete Variational Mechanics, for instance, have independently demonstrated that geometric understanding of the continuous model and proper geometric discretization are crucial for obtaining stable numerical results that conserve charge, momentum, and energy (see, for example, [3, 24, 19, 21, 12]).

The geometric structure of fluid mechanics, specifically Euler's equations for inviscid fluids, has been investigated from a theoretical point of view (see [23] and references therein). In this geometric framework, vorticity plays a central role since Euler's equations can be written directly as a simple vorticity advection (see Section 2.1 for details). Inspired by this geometric viewpoint and the recent

advances in Discrete Exterior Calculus (DEC—see [3, 16]), we propose to mimic these geometric properties on the discrete level through a *discrete differential approach to fluid mechanics*.

A key ingredient in this approach is the location of physical quantities on the appropriate geometric structures (*i.e.*, vertices, edges, faces, or cells). Using the corresponding *discrete calculus on simplicial complexes* we construct a novel integration scheme which employs intrinsically divergence-free variables. This removes the need to enforce the usual divergence-free constraint through a numerically lossy projection step. Our time integration method *by construction* preserves circulation and consequently vorticity. It accomplishes this while being simple, numerically efficient, and unconditionally stable, achieving high visual quality even for very large time steps.

Our approach can be contrasted with Stable Fluids based methods as follows:

- ◇ **our technique is based on a classical *vorticity* formulation of Navier-Stokes and Euler equations**; unlike most vorticity-based methods in CFD and CG, our approach is *Eulerian* as we work only with a fixed mesh and *not* a Lagrangian representation involving vorticity particles (or similar devices);
- ◇ we use an unconditionally-stable, semi-Lagrangian backward advection strategy for vertices just like Stable Fluids; **in contrast to Stable Fluids however we do not point sample velocity, but rather compute *integrals of vorticity***; this simple change removes the need to enforce incompressibility in the updated velocity field through projection;
- ◇ **our strategy exactly preserves circulation along discrete loops in the mesh**; capturing this *geometric property* of fluid dynamics guarantees that vorticity does not get dissipated as is typically the case in Stable Fluids; consequently no vorticity confinement (or other post processes) are required to maintain this important element of visual realism;
- ◇ **our method has multiple advantages from an implementation point of view**: it handles arbitrary meshes (regular grids, hybrid meshes [11], and even cell complexes) with non-trivial topology; the operators involved have very small support leading to very sparse linear systems; all quantities are stored intrinsically (scalars on edges and faces) without reference to global or local coordinate frames; the computational complexity is comparable to previous approaches;

This thesis contains all the details necessary to implement a complete simplicial fluid simulator from scratch. The fluid simulation algorithm and its implementation are discussed in detail in Chapter 2. In Chapter 3 we present our results and analyze the behavior of our algorithm on a few well-studied test cases from the CFD literature. We compare our results to numerical data obtained

from other methods, and investigate convergence properties under h and t refinement. Finally, in order to ease approachability, further implementation details beyond the simulator module are presented in Chapter 4. We discuss our implementation of a tetrahedral mesh data structure suitable for use with any DEC-based algorithm, as well as the implementation and construction of the DEC operators that we use.

Chapter 2 is a modified version of a paper written with coauthors Yiying Tong, Eva Kanso, Mathieu Desbrun and Peter Schröder ([9]). My main contributions are the implementation itself and the numerical experiments and results.

1.1 Previous Work

Fluid Mechanics has been studied extensively in the scientific community both mathematically and computationally. The physical behavior of incompressible fluids is usually modeled by Navier Stokes (NS) equations for viscous fluids and by Euler equations for inviscid (non-viscous) fluids. Numerical approaches in computational fluid dynamics typically discretize the governing equations through Finite Volumes (FV), Finite Elements (FE) or Finite Differences (FD) methods. We will not attempt to review the many methods proposed (an excellent survey can be found in [20]) and instead focus on approaches used for fluids in computer graphics. Some of the first fluid simulation techniques used in the movie industry were based on Vortex Blobs [37] and Finite Differences [14]. To circumvent the ill-conditioning of these iterative approaches for large time steps and achieve unconditional stability, Jos Stam [29, 30] introduced to the graphics community the *method of characteristics* for fluid advection and the Helmholtz-Hodge decomposition to ensure the divergence-free nature of the fluid motion [5]. The resulting algorithm, called *Stable Fluids*, is an extremely successful semi-Lagrangian approach based on a regular grid Eulerian space discretization, that led to many refinements and extensions which have contributed to the enhanced visual impact of fluid animations. Among others, these include the use of staggered grids and monotonic cubic interpolation [10]; improvements in the handling of interfaces [13]; extensions to curved surfaces [31, 33, 27]; inclusion of visco-elastic objects [15]; and goal oriented control of fluid motion [34, 25, 26].

However, the Stable Fluids technique is not without drawbacks. Chief among them is the difficulty of accomodating complex domain boundaries with regular grids. Local adaptivity [22] can greatly help, but the associated octree structures require significant overhead. Additionally, regular

partitions of space (adaptive or not) can suffer from preferred direction sampling, leading to artifacts similar to aliasing in rendering. Finally, due to numerical dissipation, current methods do not enforce important invariants aside from the divergence-free nature of the flow. While exaggerated loss of total energy is often difficult to notice, excessive dissipation of vorticity affects the motion significantly. The presence of vortices in liquids and volutes in smoke is one of the most important visual clues to our perception of fluidity. Vorticity confinement [32, 10] was designed to counteract this dissipation through local reinjection of vorticity. Unfortunately, it is hard to control how much can safely be added back without affecting stability or plausibility of the results.

Chapter 2

A Circulation-Preserving Integration Algorithm

2.1 Geometry of Fluid Motion

Before going into the details of our algorithm we discuss the underlying fluid equations with their relevant properties and how these can be captured over discretized domains.

2.1.1 Euler Fluids

Consider an inviscid, incompressible, and homogeneous fluid on a domain $\mathcal{D}$ in 2 or 3D. The *Euler equations*, governing the motion of this fluid (with no external forces for now), can be written as:

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \;&=\; -\nabla p \,, \\ div(\mathbf{u}) \;=\; 0 \,, \quad \mathbf{u} \;&\parallel\; \partial\mathcal{D} \,. \end{aligned} \tag{2.1}$$

We assume unit density ($\rho = 1$) and use $\mathbf{u}$ to denote the fluid velocity, p the pressure, and $\partial\mathcal{D}$ the boundary of the fluid region $\mathcal{D}$. The pressure term in Eq. (2.1) can be dropped easily by rewriting the Euler equations in terms of *vorticity*. Recall that traditional vector calculus defines vorticity as the curl of the velocity field, $\boldsymbol{\omega} = \nabla \times \mathbf{u}$. Taking the curl ($\nabla\times$) of Eq.(2.1), we obtain

$$\begin{aligned} \frac{\partial \boldsymbol{\omega}}{\partial t} + \mathcal{L}_{\mathbf{u}}\boldsymbol{\omega} \;&=\; \mathbf{0} \,, \\ \boldsymbol{\omega} \;=\; \nabla \times \mathbf{u} \,, \quad div(\mathbf{u}) \;&=\; 0 \,, \quad \mathbf{u} \;\parallel\; \partial\mathcal{D} \,. \end{aligned} \tag{2.2}$$

where $\mathcal{L}_{\mathbf{u}}\boldsymbol{\omega}$ is the Lie derivative, equal in our case to $\mathbf{u} \cdot \nabla\boldsymbol{\omega} - \boldsymbol{\omega} \cdot \nabla\mathbf{u}$. In this form, this vorticity-based equation states that *vorticity is simply advected along the fluid flow*. Note that Equation (2.2)

is equivalent to the more familiar $\frac{D\boldsymbol{\omega}}{Dt} = \boldsymbol{\omega} \cdot \nabla \mathbf{u}$, and therefore already includes the vortex stretching term appearing in *Lagrangian* approaches. Roughly speaking, vorticity measures the local spin of a fluid parcel. Therefore, vorticity advection means that local spin moves with the flow.

Since the integral of vorticity on a given bounded surface equals (by Stokes' theorem) the *circulation* around the bounding loop of the surface, one can explain the geometric nature of an ideal fluid flow in particularly simple terms: **the circulation around any closed loop $\mathcal{C}$ is conserved throughout the motion of this loop in the fluid**. This key result is known as Kelvin's circulation theorem, and is usually written as:

$$\Gamma(t) \;=\; \oint_{\mathcal{C}(t)} \mathbf{u} \cdot \mathbf{dl} \;=\; \text{constant}\,, \tag{2.3}$$

where $\Gamma(t)$ is the circulation of the velocity on the loop $\mathcal{C}$ at time t as it gets advected in the fluid. This will be the key to our time integration algorithm.

2.1.2 Viscous Fluids

In contrast to ideal fluids, incompressible *viscous* fluids generate very different fluid behaviors. They can be modelled by the *Navier-Stokes* equations (compare with Eq. (2.1)):

$$\begin{aligned} \frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \;&=\; -\nabla p + \nu \Delta \mathbf{u}\,, \\ div(\mathbf{u}) \;=\; 0\,, \quad \mathbf{u}|_{\partial \mathcal{D}} &= \mathbf{0}\,. \end{aligned} \tag{2.4}$$

where Δ denotes the Laplace operator, and ν is the *kinematic viscosity*. Note that different types of boundary conditions can be added depending on the chosen model. Despite the apparent similarity between these two models for fluid flows, the added diffusion term dampens the motion, resulting in a slow decay of circulation. This diffusion also implies that the velocity of a viscous fluid at the boundary of a domain must be null, whereas an inviscid fluid could have a non-zero tangential component on the boundary. Here again, one can avoid the pressure term by taking the curl of the equations (compare with Eq. (2.2)):

$$\begin{aligned} \frac{\partial \boldsymbol{\omega}}{\partial t} + \mathcal{L}_{\mathbf{u}} \boldsymbol{\omega} \;&=\; \nu \Delta \boldsymbol{\omega}\,, \\ \boldsymbol{\omega} \;=\; \nabla \times \mathbf{u}\,, \quad div(\mathbf{u}) \;=\; 0\,, \quad \mathbf{u}|_{\partial \mathcal{D}} &= \mathbf{0}\,. \end{aligned} \tag{2.5}$$

2.2 Discrete Setup

For a discrete time and space numerical simulation of Eqs. (2.2) and (2.5) we need a discretized geometry of the domain (given as a simplicial mesh for instance), appropriate discrete analogs of velocity $\mathbf{u}$ and vorticity fields $\boldsymbol{\omega}$, along with the operators which act on them.

2.2.1 Space Discretization

We discretize the spatial domain in which the flow takes place using a locally oriented simplicial complex, *i.e.*, either a tet mesh for 3D domains or a triangle mesh for 2D domains, and refer to this discrete domain as $\mathcal{M}$ (see Figure 2.1). The domain may have non-trivial topology, *e.g.*, it can contain tunnels and voids (3D) or holes (2D), but is assumed to be compact. To ensure good numerical properties in the subsequent simulation we require the simplices of $\mathcal{M}$ to be well shaped (aspect ratios bounded away from zero). This assumption is quite common since many numerical error estimates depend heavily on the element quality. We use meshes generated with the method of [1]. Collectively we refer to the sets of vertices, edges, triangles, and tets as V, E, F, and T.

We will also need the *dual mesh*. It associates with each original simplex (vertex, edge, triangle, tet, respectively) its dual (dual cell, dual edge, dual face, and dual vertex, respectively) (see Fig. 2.2). The geometric realization of the dual mesh uses tet circumcenters as dual vertices and Voronoi cells as dual cells; dual edges are line segments connecting dual vertices across shared tet faces and dual faces are the faces of the Voronoi cells. Notice that storing values on primal simplices or on their associated dual cells is conceptually equivalent, since both sets have the same cardinality. We will see in Section 2.4 that corresponding primal and dual quantities are related through a simple (diagonal) linear operator.

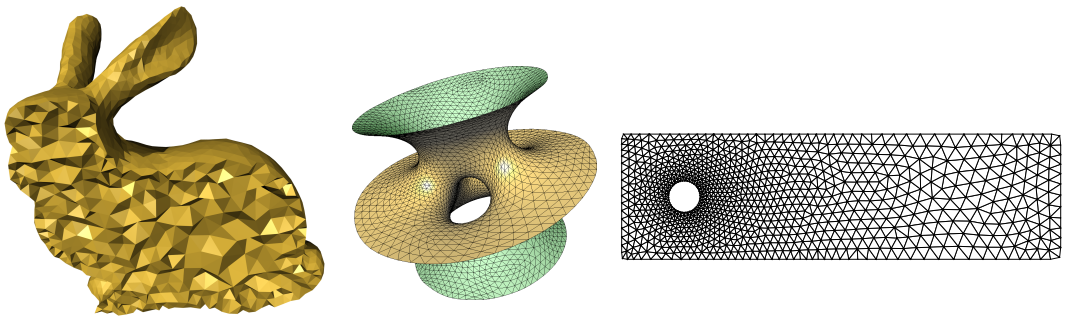


Figure 2.1: *Domain Mesh: our fluid simulator uses a simplicial mesh to discretize the equations of motion; (left) the domain mesh (shown as a cutaway view) used in Fig. 3.1; (middle) the curved triangle mesh used in Fig. 3.3; (right) a coarser version of the flat 2D mesh used in Fig. 3.5.*

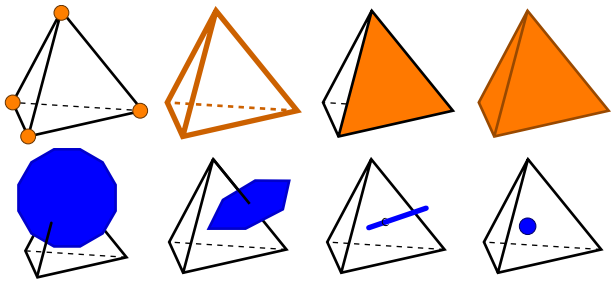


Figure 2.2: *Primal and Dual Cells: the simplices of our mesh are vertices, edges, triangles and tets (up); their circumcentric duals are dual cells, dual faces, dual edges and dual vertices (bottom).*

2.2.2 Discretization of Physical Quantities

In order to faithfully capture the geometric structure of fluid mechanics on the discrete mesh, we define the usual physical quantities, such as velocity and vorticity, through *integral values over the elements of the mesh* $\mathcal{M}$. Depending on whether a given quantity is a point, line, area or volume density, the corresponding discrete representation will "live" at the associated 0, 1, 2, and 3 dimensional mesh elements. These integral values are formally called *discrete differential* k*-forms* for $k = 0, 1, 2, 3$, and are given as integrals of the corresponding differential k-form over the underlying k-cell or k-simplex (we refer the interested readers to a tutorial on this notion [8] for a more comprehensive exposition). In practice we realize k-forms as vectors of `double` of length $|V|$ (for 0-forms), $|E|$ (for 1-forms), $|F|$ (for 2-forms), and $|T|$ (for 3-forms).

Velocity as Discrete Flux To encode a coordinate free (intrinsic) representation of velocity on the mesh we use *flux, i.e.*, the mass of fluid transported across a given surface area per unit time. Note that this makes flux an *integrated*, not pointwise, quantity. On the discrete mesh, fluxes are associated with the triangles of the tet mesh. Thus fluid velocity $\mathbf{u}$ is treated as a 2-form and represented as a vector U of values on faces (size $|F|$). This coordinate-free point of view, also used in [11], is reminiscent of the staggered grid method used in [10] and other non-collocated grid techniques (see [15]). In the staggered grid approach one does not store the x, y, z components of a vector at nodes but rather associates them with the corresponding grid faces. We may therefore think of the idea of storing fluxes on the triangles of our tet mesh as a way of *extending* the idea of staggered grids to the more general simplical mesh setting. This was previously exploited in [4] in the context of E&M computations. It also makes the usual no-transfer boundary conditions easy to encode: boundary faces experience no flux across them. Encoding this boundary condition for velocity vectors stored at vertices is far more cumbersome.

Divergence as Net Flux on Tets Given the incompressibility of the fluid, the velocity field must be divergence-free ($\nabla \cdot \mathbf{u} = 0$). In the discrete setting, the integral of the divergence over a tet becomes particularly simple. According to the generalized Stokes' theorem this integral equals the sum of the fluxes on all four faces, *i.e.*, everything that gets in must get out (see Fig. 2.3). Divergence can therefore be stored as a 3-form, *i.e.*, as a value associated to each tet (a vector of cardinality $|T|$).

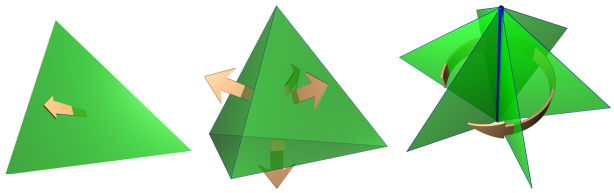


Figure 2.3: *Discrete Physical Quantities: in our geometric discretization, fluid flux lives on faces (left), divergence lives on tets (middle), and vorticity lives on edges (right).*

Vorticity as Flux Spin Finally we need to define vorticity on the mesh. To see the physical intuition behind our definition, consider an edge in the mesh. It has a number of faces incident on it, akin to a paddle wheel (see Figure 2.3). The flux on each face contributes a torque to the edge. The sum of all these, when going around an edge, is the net torque that would "spin" the edge. We can thus give a physical definition of vorticity as a weighted sum of fluxes on all faces incident to a given edge. This quantity is now associated with primal edges—or, equivalently, dual faces—and is thus represented by a vector Ω of size $|E|$.

In Section 2.4, we will see that these physical intuitions can be derived formally from simple algebraic relationships.

2.3 Geometric Integration of Fluid Motion

Since we are using the vorticity formulation of the fluid equations (Eqs. (2.2) or (2.5)) the time integration algorithm must update the discrete vorticity variables which are stored on each primal edge. We have seen that the fluid equations state that vorticity is advected by the velocity field. The fundamental idea of our geometric integration algorithm is thus to ensure that Kelvin's theorem holds in the discrete setting: the circulation around any loop in the fluid remains constant as the loop is advected. This can be achieved by backtracking loops: for any given loop at the current time, determine its backtracked image in the velocity field ("where did it come from?") and compute the

circulation around the backtracked loop. This value is then assigned as the circulation around the original loop at the present time, *i.e.*, circulation is properly advected *by construction* (see Figure 2.4 for a depiction of this loop advection idea).

Since we store vorticity on primal edges, a natural choice for these loops are the bounding loops of the dual faces associated to each primal edge (see Figure 2.2). Notice that these loops are polylines formed by sequences of dual vertices around a given primal edge. Consequently an efficient implementation of this idea requires only that we backtrack *dual vertices* in the velocity field. Once these positions are known *all* backtracked dual loops associated to *all* primal edges are known. These Voronoi loops can indeed generate any discrete, dual loop: the sum of adjacent loops is a larger, outer loop as the interior edges cancel out due to opposite orientation as sketched in Fig. 2.4(right). The evaluation of circulation around these backtracked loops will be quite straightforward . By Stokes' theorem the integral of vorticity over a dual face equals the circulation around its boundary, so we have achieved our goal of updating vorticities and, *by design*, ensured a discrete version of Kelvin's theorem.

The algorithmic details of this geometric approach to time integration of the equations of motion for fluids are given in Section 2.5.

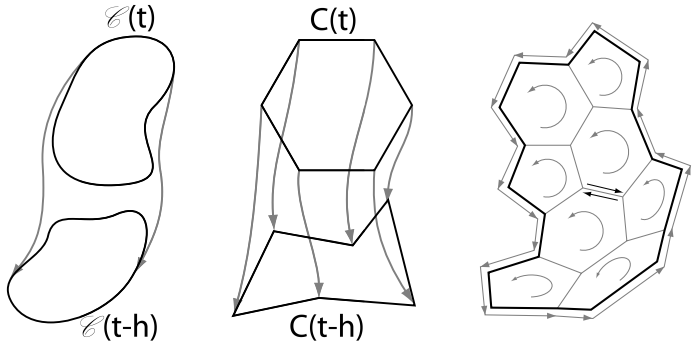


Figure 2.4: *Kelvin's Theorem: (left) in the continuous setting, the circulation on any loop being advected by the flow is constant. (middle) our discrete integration scheme enforces this property on each Voronoi loop, (right) thus on any discrete loop.*

2.4 Computational Machinery

Now that the spatial and physical discretizations are properly defined, we must manipulate the numerics involved in our integration scheme in a principled manner to guarantee proper physical behavior. In this section, we point out that the basic operators to go from fluxes to the divergence,

curl, or Laplacian of the velocity field can be *formally* defined. For a full discussion on the topic of Discrete Exterior Calculus (DEC), which defines precisely a discrete calculus on simplicial complexes, see for instance [8]. We will only present the practical implementation of the few operators we need. *More importantly, we will show that this implementation reduces to simple linear algebra with very sparse matrices.*

2.4.1 Two Basic Operators

The computations involved in our approach only require the definition of two basic operators: one is the exterior derivative d, necessary to compute derivatives, like gradients, divergences, or curls; the other is the Hodge star, to transfer values from primal simplices to dual simplices.

Exterior Derivative d Given an oriented mesh, we implement our first operator by simply assembling the *incidence matrices* of the mesh. These will act on the vectors of our discrete forms and implement the discrete exterior derivative operator d as explained in more details in Appendix A.1. For our 3D implementation, there are three sparse matrices involved, which contain only entries of type 0, $+1$, and -1. Care is required in assembling these incidence matrices, as the orientation must be taken into account in a consistent manner. The first one is d_0, the incidence matrix of vertices and edges ($|E|$ rows and $|V|$ columns). Each row contains a single $+1$ and -1 for the end points of the given edge (and zero otherwise). The sign is determined from the orientation of the edge. The second matrix is, similarly, encoding the incidence relations of edges and faces ($|F|$ rows and $|E|$ columns), with appropriate $+1$ and -1 entries according to the the orientation of edges as one moves around a face. More generally d_k is the incidence matrix of k-cells on $k+1$-cells.

A simple debugging sanity check (necessary but not sufficient) is to compute consecutive products: d_0 followed by d_1 must be a matrix of zeros, and similarly as must be d_1 followed by d_2. This reflects the fact that the boundary of any boundary is the empty set. It also corresponds to the calculus fact that curl of grad is zero as is divergence of curl (see Appendix A.1).

Hodge Star The second operator we need will allow us to transfer quantities back and forth between the primal and dual mesh. We can project a primal k-form to a conceptually-equivalent dual $(3-k)$-form with the *Hodge star*. We will denote $\star_0$ (resp., $\star_1, \star_2, \star_3$) the Hodge star taking a 0-form (resp., 1-form, 2-form, and 3-form) to a dual 3-form (resp., dual 2-form, dual 1-form, dual 0-form). In this work we use what is known as the *diagonal Hodge star* [3]. This operator simply

scales whatever quantity that is stored on mesh cells by the volumes of the corresponding dual and primal cells: let $\mathrm{vol}(.)$ denote the volume of a cell (*i.e.*, 1 for vertices, length for edges, area for triangles, and volume for tets), then

$$(\star_k)_{ii} = \mathrm{vol}(\tilde{\sigma}_i)/\,\mathrm{vol}(\sigma_i)$$

where σ_i is any primal k-simplex, and $\tilde{\sigma}_i$ is its dual. These linear operators, describing the local metric, are diagonal and can be stored as vectors. Conveniently, the inverse matrices going from dual to primal quantities are trivial to compute for this diagonal Hodge star.

Overloading Operators Note that both the d_k and the $\star_k$ operators are *typed*: the subscript k is *implicitly determined* by the dimension of the argument. For example, the velocity field $\mathbf{u}$ is a 2-form stored as a vector U of cardinality $|F|$. Consequently the expression dU implies use of the $|T| \times |F|$-sized matrix d_2. In the implementation this is accomplished with operator overloading (in the sense of `C++`). We will take advantage of this and drop the dimension subscripts from now on.

2.4.2 Offline Matrix Setup

With these overloads of d and $\star$ in place, we can now set up the only two matrices (C and L) that will be used during simulation. They respectively represent the exact discrete analogs of the curl and Laplace operators [8].

Curl Since we store fluxes on faces and gather them in a vector U, the *circulation* of the vector field $\mathbf{u}$ can be derived as values on dual edges through $\star U$. *Vorticity*, typically a 2-form in fluid mechanics [23], is easily computed by then summing this circulation along the dual edges that form the boundary of a dual face. In other words, $\boldsymbol{\omega} = \nabla \times \mathbf{u}$ becomes, in terms of our discrete operators, simply $\Omega = d^T \star U$. We therefore create a matrix C offline as $d^T\star$, *i.e.*, the composition of an incidence matrix with a diagonal matrix.

Laplacian The last matrix we need to define is the discrete Laplacian. The discrete analog of $\Delta\phi = (\nabla\nabla\cdot - \nabla\times\nabla\times)\phi = \boldsymbol{\omega}$ is simply $(\star d \star^{-1} d^T \star + d^T \star d)\ \Phi = \Omega$ as explained in Appendix A.2. This last matrix, a simple composition of incidence and diagonal matrices, is precomputed and stored as L for later use.

2.5 Implementation

To facilitate a direct implementation of our integration scheme, we provide pseudocode (Figure 2.5) along with implementation notes which provide details for specific steps and how these related to the machinery developed in earlier sections.

```
//Load mesh and build incidence matrices
C ← d^T ⋆
L ← ⋆d⋆^{-1} d^T ⋆ + d^T ⋆ d

//Time stepping h
loop
    //Advect Vorticities
    for each dual vertex (tet circumcenter) c_i
        ĉ_i ← PathTraceBackwards(c_i);
        v_i ← InterpolateVelocityField(ĉ_i);
    for each dual face f
        Ω_f ← 0
        for each dual edge (i, j) on the boundary of f
            Ω_f ← Ω_f + (1/2)(v_i + v_j) · (ĉ_i − ĉ_j);

    //Add forces
    Ω ← Ω + h C F

    //Add diffusion for Navier-Stokes
    Ψ ← SolveCG( (⋆ − ν h L)Ψ = Ω );
    Ω ← ⋆ Ψ

    //Convert vorticities back to fluxes
    Φ ← SolveCG( L Φ = Ω );
    U ← dΦ;
```

Figure 2.5: *Pseudocode of our fluid motion integrator.*

2.5.1 External Body Forces

The use of external body forces, like buoyancy, gravity, or stirring, is common practice to create interesting motions. Incorporating external forces into Eq. (2.4) is, fortunately, straightforward, resulting in:

$$\frac{\partial \mathbf{u}}{\partial t} + \mathbf{u} \cdot \nabla \mathbf{u} \;=\; -\nabla p + \nu \Delta \mathbf{u} + \mathbf{f} \; .$$

Again, taking the curl of this equation allows us to recast this equation in terms of vorticity:

$$\frac{\partial \boldsymbol{\omega}}{\partial t} + \mathcal{L}_{\mathbf{u}}\boldsymbol{\omega} \;=\; \nu\Delta\boldsymbol{\omega} + \nabla \times \mathbf{f}\ . \tag{2.6}$$

Thus, we note that an external force influences the vorticity only through the force's curl (the $\nabla \cdot \mathbf{f}$ term is compensated for by the pressure term keeping the fluid divergence-free). Thus, if we express our forces through the vector F of their resulting fluxes in each face, we can directly add the forces to the domain by incrementing Ω by the circulation of F over the time step h, *i.e.*:

$$\Omega \leftarrow \Omega + h\ C\ F.$$

2.5.2 Adding Diffusion

If we desire to simulate a viscous fluid, we must add the diffusion term present in Eq. (2.5). Note that previous methods were sometimes omitting this term because their numerical dissipation was already creating (uncontrolled) diffusion. In our case, however, this diffusion needs to be properly handled if viscosity is desired. This is easily done through an unconditionally-stable implicit integration as done in Stable Fluids (*i.e.*, we also use a fractional step approach). Using the discrete Laplacian in Eq. (A.3) and the current vorticity Ω, we simply solve for the diffused vorticity Ω' using the following linear system:

$$(\star - \nu h L)\ \star^{-1}\Omega' = \Omega.$$

2.5.3 Interpolation of Velocity

In order to perform the backtracking of dual vertices we must first define a velocity field over the entire domain using the data we have on primal faces (fluxes). This is done by computing a unique velocity vector for each dual vertex and then using barycentric interpolation of these vectors over each dual Voronoi cell [36], defining a continuous velocity field over the entire domain. This velocity field can be used to backtrack dual vertices as well as transport particles or dyes (*e.g.*, for visualization purposes) with standard methods.

To see that such a vector, one for each dual vertex, is well defined consider the following argument. The flux on a face corresponds under duality (via the Hodge star) to a circulation along the dual edge of this face. Now, there is a linear relation between fluxes per tet due to the incompress-

ibility condition (fluxes must sum to zero). This translates directly to a linear condition on the four circulations at each tet too. Thus, there is a unique vector (with three components) at the dual vertex whose projection along the dual edges is consistent with the observed circulations.

Relation to k-form Basis Functions The standard method to interpolate k-form data in a piecewise linear fashion over simplicial complexes is based on Whitney forms [3]. In the case of primal 2-forms (fluxes) this results in a piecewise constant (per tet) velocity field. Our argument above, using a Voronoi cell based generalized barycentric interpolation of dual 1-forms (circulation), in fact *extends* the Whitney form machinery to the dual setting. This is a novel contribution which may be useful in other computational applications of discrete forms. We note that the generalized barycentric coordinates have linear accuracy [36], an important requirement in many settings.

2.5.4 Handling Arbitrary Topology

Recall that for a given a velocity field $\mathbf{u}$ there exists a unique vorticity field $\boldsymbol{\omega} = \nabla \times \mathbf{u}$. However the inverse statement is not true in general; a vorticity field does not uniquely specify a velocity field. In particular, adding any vorticity-free field to $\mathbf{u}$ does not affect $\boldsymbol{\omega}$. Because velocity fields that are both vorticity-free *and* divergence-free ("harmonic") cannot exist in closed simple domains, we need only consider this extra degree of freedom when the domain has nontrivial topology (*e.g.*, when the first Betti number is not zero).

By the Helmholtz-Hodge decomposition theorem (Eq. (A.1)), we can represent our velocity field as the sum of a rotational field ($\nabla \times \boldsymbol{\omega}$) and a vorticity-free field ($\mathbf{h}$). Furthermore, in the absence of external forces, $\mathbf{h}$ remains constant. Addressing arbitrary topology domains can therefore be accomplished by augmenting the existing algorithm with the following:

- We have an additional 2-form H that is initialized to 0 and is updated at each timestep with the harmonic component of the forces; assuming F is divergence-free, $H \leftarrow H + h(F - CF)$.
- $U \leftarrow d\Phi$ becomes $U \leftarrow d\Phi + H$ instead.

2.5.5 Handling Boundaries

The algorithm as described above does not constrain the boundaries, thus achieving "open" boundary conditions. No-transfer boundary conditions are easily imposed by setting the fluxes through the boundary triangles to zero. Non-zero flux boundary conditions (*i.e.*, forced fluxes through the

boundary as in the case of Fig. 3.5) are more subtle. First, remark that all these boundary fluxes *must* sum to zero; otherwise, we would have little chance of getting a divergence-free fluid in the domain. Since the total divergence is zero, there exists a harmonic velocity field satisfying exactly these conditions. This is, again, a consequence of the Helmholtz-Hodge decomposition theorem with normal boundary conditions [5]. Thus, this harmonic part H can be computed *once and for all* through a Poisson equation using the same setup as described in Appendix A.2. Then we add H to U at each timestep, as in the previous section.

Vorticity on Boundaries The Voronoi cells at the boundaries are slightly different from the usual, interior ones, since boundary edges do not have a full 1-ring of tets. If the boundaries are unconstrained, this is not a problem; boundary cells may be backtracked and updated just like interior ones. However, if no-transfer boundary conditions are imposed, vorticity must be handled differently on the boundary.

Note that if the boundary fluxes are constrained, we solve the Poisson equation for U using only values of Ω on *interior* dual faces; assigning values on the boundary would over-constrain the linear system. During the diffusion step, however, values of Ω on the boundary are necessary. The first implication of this is that boundary vorticities need not be assigned in inviscid simulations, because there is no diffusion step. For viscous simulations, we must assign vorticity values to boundary cells subject to the constraint that the tangential velocity on the boundary be 0 (see Eq. 2.4).

Recall that $\Omega = d^T \star U$. Said differently, the vorticity on an interior dual face f is equivalent to

$$\Omega_f = \sum_{e \in \partial f} (\star U)_e$$

where $(\star U)_e$ is the circulation on dual edge e. On boundary cells, ∂f contains pieces that are not dual edges, as illustrated in Figure 2.6(b). The vorticity on such a cell can be calculated by simply assigning 0 to the line segments on the boundary of the domain, in accordance with the viscous boundary conditions.

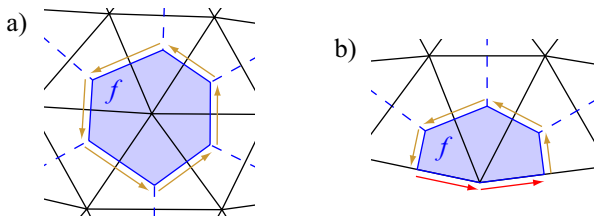


Figure 2.6: *Boundary Cells: Interior cells are bounded only by dual edges (a), while the perimeter of a boundary cell contains line segments that do not lie on the dual mesh (b).*

Chapter 3

Results and Discussion

3.1 Coarse Resolution Simulations

We start with a few simulations that demonstrate the applicability of our method to Computer Graphics applications.

3.1.1 3D Simulations

First we consider a smoke cloud surrounded by air, filling the body of a bunny as an example of flow in a domain with complex boundary. Buoyancy drives the air flow which, in turn, advects the smoke cloud in the three-dimensional domain as shown in Fig. 3.1. The mesh used for this simulation consists of only $7K$ vertices and $32K$ tets, while still effectively resolving the detailed features of the flow in the ears and head.

We also show a snow globe with a bunny inside in Fig. 3.2. We emulate the flow due to an initial

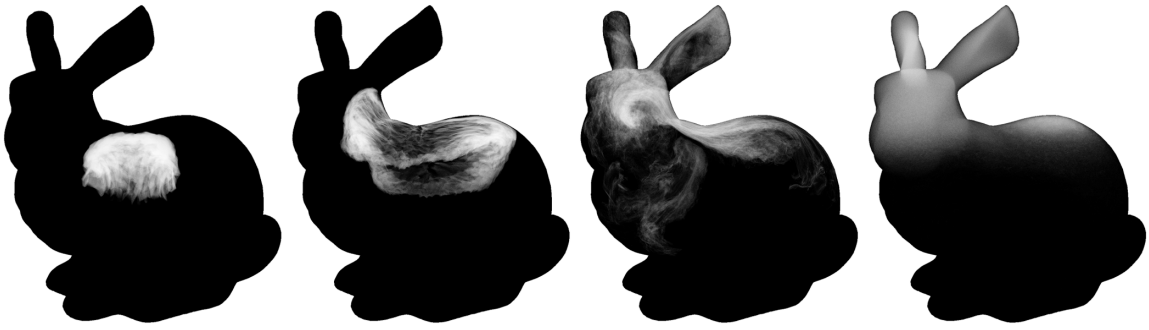


Figure 3.1: *Smoking Bunny: This example demonstrates the power of using tetrahedral meshes for resolving exact boundaries. Here, a hot smoke cloud rises inside a bunny shaped domain of* $7K$ *vertices (*32*K tets), significantly reducing the computational cost of the simulation for such an intricate boundary compared to regular grid-based techniques (*0.47*s/frame on a PIV3GHz).*

spin of the globe using a swirl described as a vorticity field. The snow particles are transported by the flow as they fall down under the effect of gravity. The viscous (no-slip) boundary conditions in this simulation cause the falling particles to stick to the interior boundary as they fall.

Both examples use a timestep of $h = 1/30$, and both took *less than half a second per frame* to compute on a 3GHz Intel Pentium IV, exemplifying the advantage of using tet meshes to resolve fine boundaries.

Rendering In these examples, a large number of passive marker particles are advected through the flow for visualization purposes. The runtimes for this step vary greatly depending on the number of particles used, the time step size, and the order of accuracy of the path tracer. The snow globe example required only a few thousand snow particles, which could be advected and displayed with minimal overhead. The smoke example, on the other hand, uses several million smoke particles for visualization, requiring 20 to 30 seconds on average per frame to advect and render.

Because the particles do not affect the simulation, the user may run interactive preview simulations with less particles in order to adjust parameters as necessary. Furthermore, if an animation is not required, the simulation may be advanced to the point of interest without any visualization overhead, after which the velocity field can be visualized using any flow visualization technique, such as streamlines, implicit stream surfaces, Volume Line Integral Convolution, or others [38, 35, 17, 6].

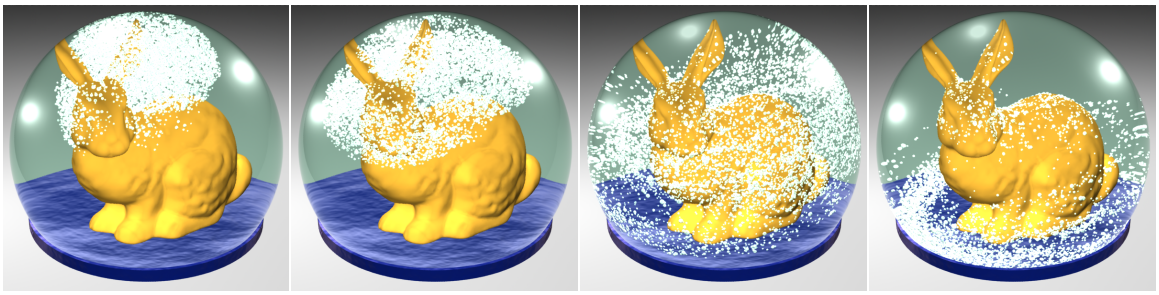


Figure 3.2: *Bunny Snow Globe: the snow in the globe is advected by the inner fluid, initially stirred by a vortex to simulate a spin of the globe.*

3.1.2 Curved Surfaces

We have also considered flow on curved surfaces in 3D with complex topology, as depicted in Fig. 3.3. We were able to easily extend our implementation of two-dimensional flows to this curved case thanks to the intrinsic nature of our approach.

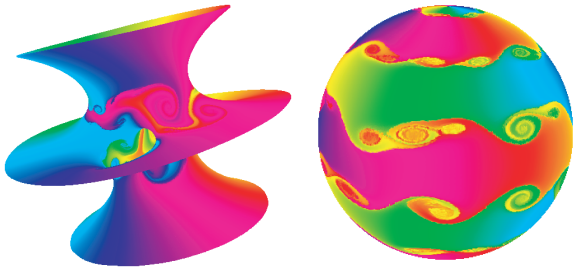


Figure 3.3: *Weather System on Planet Costa: the intrinsic nature of the variables used in our algorithm makes it amenable to the simulation of flows on arbitrary curved surfaces.*

3.1.3 2D Simulations

The behavior of vortex interactions observed in existing experimental results was compared to numerical results based on our novel model and those obtained from the semi-Lagrangian advection method. It is known from theory that two like-signed vortices with a finite vorticity core will merge when their distance of separation is smaller than some critical value. This behavior is captured by the experimental data and shown in the first series of snapshots of Fig. 3.4. As the next row of

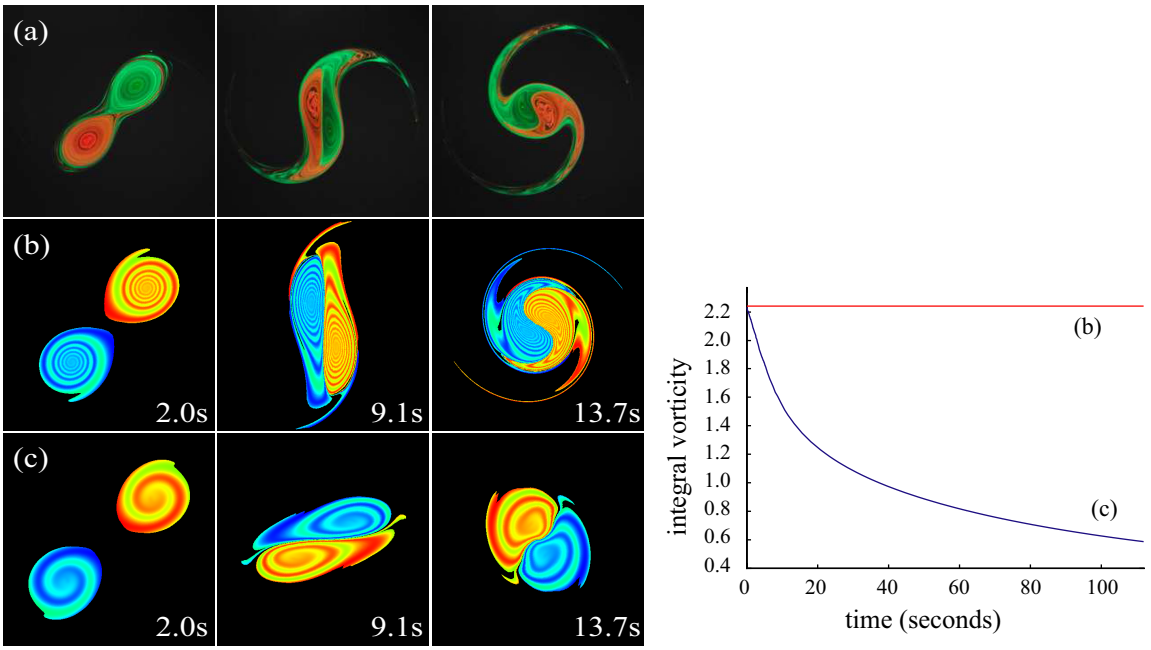


Figure 3.4: *Two Merging Vortices: discrete fluid simulations are compared with a real life experiment (courtesy of Dr. Trieling, Eindhoven University; see* `http://www.fluid.tue.nl/WDY/vort/index.html`*) where two vortices (colored in red and green) merge slowly due to their interaction (a); while our method faithfully captures the merging phenomenon (b), a traditional semi-lagrangian scheme does not capture the correct motion because of vorticity damping (c).*

images indicates, the numerical results that our model generated present striking similarities to the experimental data. In the last row, we see that a traditional semi-Lagrangian advection followed by re-projection misses most of the fine structures of this phenomenon. This can be attributed to the loss of total integral vorticity as evidenced in the graph; in comparison our technique preserves this integral exactly.

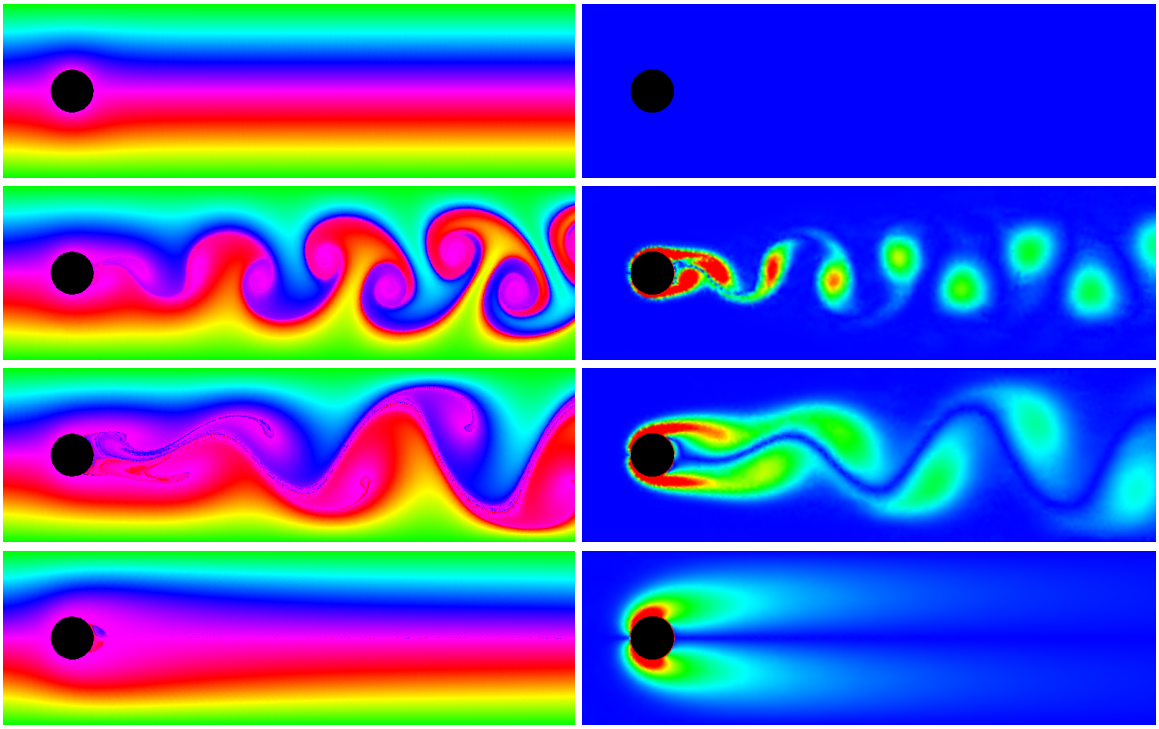


Figure 3.5: *Obstacle Course: in the usual experiment of a flow passing around a disk, the viscosity as well as the velocity can significantly affect the flow appearance; (left) our simulation results for increasing Reynolds number; (right) the vorticity magnitude (shown in false colors) of the same frame. Notice how the usual irrotational flow is obtained (top) for zero viscosity, while the von Karman vortex street appears as viscosity is introduced.*

Finally, we tested our method on the widely studied example of a flow past a cylinder (see Fig. 3.5). Starting with zero vorticity, it is well known that in the case of an inviscid fluid, the flow remains irrotational at all times. By construction, our method does respect this physical behavior since circulation is preserved for Euler equations. We then increase the viscosity of the fluid incrementally, and observe the formation of a vortex wake behind the obstacle, in agreement with physical experiments. As evidenced by the vorticity plots, vortices are shed from the boundary layer as a result of the adherence of the fluid to the obstacle, thanks to our proper treatment of the boundary conditions.

We now use this test case of flow past a cylinder as our basis for more detailed numerical

experiments.

3.2 Flow Past a Cylinder

In this section we investigate the behavior of our method under various parameter settings and discuss convergence properties. The experiment consists of a flow past a stationary cylinder at Reynolds number $Re = 15\,000$. Figure 3.8 illustrates a representative slice of the parameter domain. These images plot the vorticity field near the cylinder at time $t = 5.0s$. For the same set of simulations we graph the total energy and total integral vorticity in the domain over time in Figure 3.9. We expect to see the vorticity graphs remain constant at 0. Our method does not claim to conserve energy, but observing the energy behavior can give us some clues about convergence rates.

3.2.1 Meshes

The domain under consideration is a two-dimensional channel containing a cross-section of a circular cylinder (a disk). The diameter (height) of the channel is 8 and the radius of the cylinder is 1. We ran all of the simulations in this section on the following three meshes representing this domain. We will refer to them as meshes 1, 2, and 3, respectively.

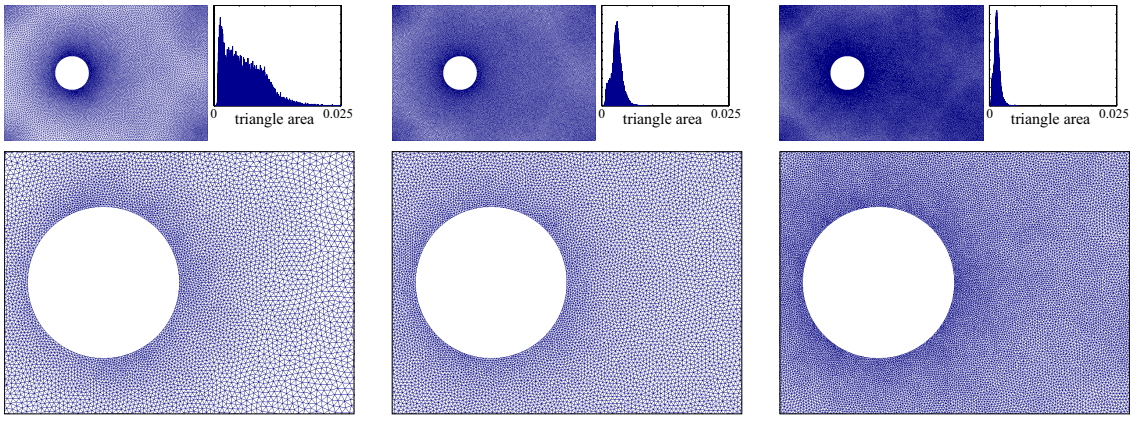


Figure 3.6: *Meshes* 1, 2, *and* 3, *from left to right. The bottom row shows a close up of the region near the cylinder.*

3.2.2 Time Step

We also vary the time step size h. Values of h used were 0.1, 0.033, and 0.01. Here we immediately find the biggest fault of our method. We observe a roughly constant amount of numerical diffusion[1] of vorticity per time step. The simulations therefore do not converge under refinement of dt, because the rate of the loss is inversely proportional to the size of the time step. One of the causes of the diffusion will be addressed in Section 3.2.4, and another in Section 3.4.

3.2.3 Advection

The majority of the algorithm manipulates discrete forms directly; values are stored on simplices, and discrete operators are defined to operate on the discrete forms. Path tracing, however, requires that a velocity field be defined everywhere. In Section 2.5.3 we introduced two prospective interpolants. The first is piecewise-constant per tet, derived using Whitney forms. The second is piecewise-rational (tangentially discontinuous across dual cell boundaries) and linear accurate. We compared the results of using these two interpolants for the backtracking step, using an Euler tracer with the first interpolant and a second order Runge-Kutta tracer with the second interpolant.

We found that RK2 path tracing using the smoother interpolant is no more or less accurate that the piecewise-constant one. The images and graphs are nearly identical; compare Fig. 3.8(a) to Fig. 3.8(c) and Fig. 3.9(a) to Fig. 3.9(c). This can be attributed to the discontinuities in the interpolant, which fails to satisfy the assumptions upon which integrators such as Runge-Kutta are built.

Not surprisingly, using simple forward or backward Euler integration along the piecewise-rational field performs much worse than either of the above, again due to the discontinuities. The lack of coherence between paths of neighboring samples results in highly unstable simulations.

3.2.4 Numerical Quadrature

To assign new vorticities at each step, we integrate the circulation along backtracked loops. In Section 2.5 we did this by sampling the interpolated velocity field at the two endpoints of each edge, averaging them, and performing a dot product with the edge itself. (This corresponds to the trapezoid rule for 1D integrals.) We can perform a more accurate integration by increasing the number of samples of the velocity field along the length of the edge (see Fig. 3.7(b)). Again, because

[1] *Diffusion* refers to "spreading" of vorticity, not to be confused with *dissipation*, or "loss", of vorticity.

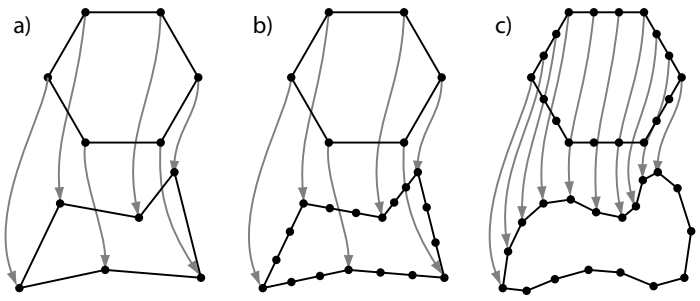


Figure 3.7: *Integrating Circulation With Numerical Quadrature. a) The velocity field is evaluated at the corners of the backtracked loop and the trapezoid rule is used to integrate the circulation on each line segment. b) The velocity field is sampled at more locations on the backtracked loop allowing for a more accurate reconstruction of the velocity field along each edge. c) The loop is split before backtracking and a finer resolution representation of the loop is advected.*

of the discontinuities, we do not use a higher order reconstruction for the line integral but instead perform the integration via a Monte Carlo sampling of the velocity field along the edge.

We find that increasing the accuracy of the numerical quadrature in this manner does indeed mitigate the diffusion problem and exhibits better vorticity preservation properties overall; compare Fig. 3.8(a) to Fig. 3.8(b) and Fig. 3.9(a) to Fig. 3.9(b). Another alternative is to split the edge *before* backtracking, as in Figure 3.7(c), in order to construct a more accurate representation of the backtracked loop. This also works well, but the results are nearly identical to (b). The reason is the following. The diffusion seems to be a result of under-sampling of the vorticity field during the update step. Splitting the edge *before* addresses a different problem: it attempts to correct small deviations in the *positions* of the loops, rather than errors in the integrals themselves. Note also that (c) increases the running time significantly, because path tracing is one of the bottlenecks of the algorithm.

3.2.5 Energy Preservation

Thus far we have been focusing strictly on the behavior of the vorticity field. Perhaps controlling the energy as well may improve the accuracy of our simulations. The total energy of the fluid can be written as:

$$E \;=\; \frac{1}{2}\int_{\mathcal{D}} \parallel \mathbf{u} \parallel^2 , \quad \text{or, equivalently,} \quad \mathrm{E} \;=\; \frac{1}{2}\int_{\mathcal{D}} \boldsymbol{\omega}\cdot\Delta^{-1}\boldsymbol{\omega}\,.$$

Energy preservation can be enforced by ensuring that:

$$(\omega + \delta\omega)\Delta^{-1}(\omega + \delta\omega) - \omega\Delta^{-1}\omega = 0$$

where $\delta\omega$ is the change of vorticity over a time step. Notice that if $\delta\omega\Delta^{-1}(2\omega + \delta\omega) = 0$, the above statement is automatically satisfied. Therefore, for a small time step h for which $\delta\omega$ is small compared to ω, we can use the projection of $\delta\omega$ to the space orthogonal to $\Delta^{-1}(2\omega + \delta\omega)$ as the energy-preserving change of vorticity. Figure 3.8d shows the results of applying this projection.

Unfortunately, this projection is a global operation; that is, far-away vorticities can influence each other, which is not a property of the physical system. In particular, as shown in the figure, non-zero vorticities are introduced in regions that do not lie in the wake of the cylinder. Clearly this is incorrect. Furthermore, the qualitative behavior of the vortex street that forms behind the cylinder is significantly different from that of existing physical and numerical experiments (see Fig. 3.10 and 3.11(d)).

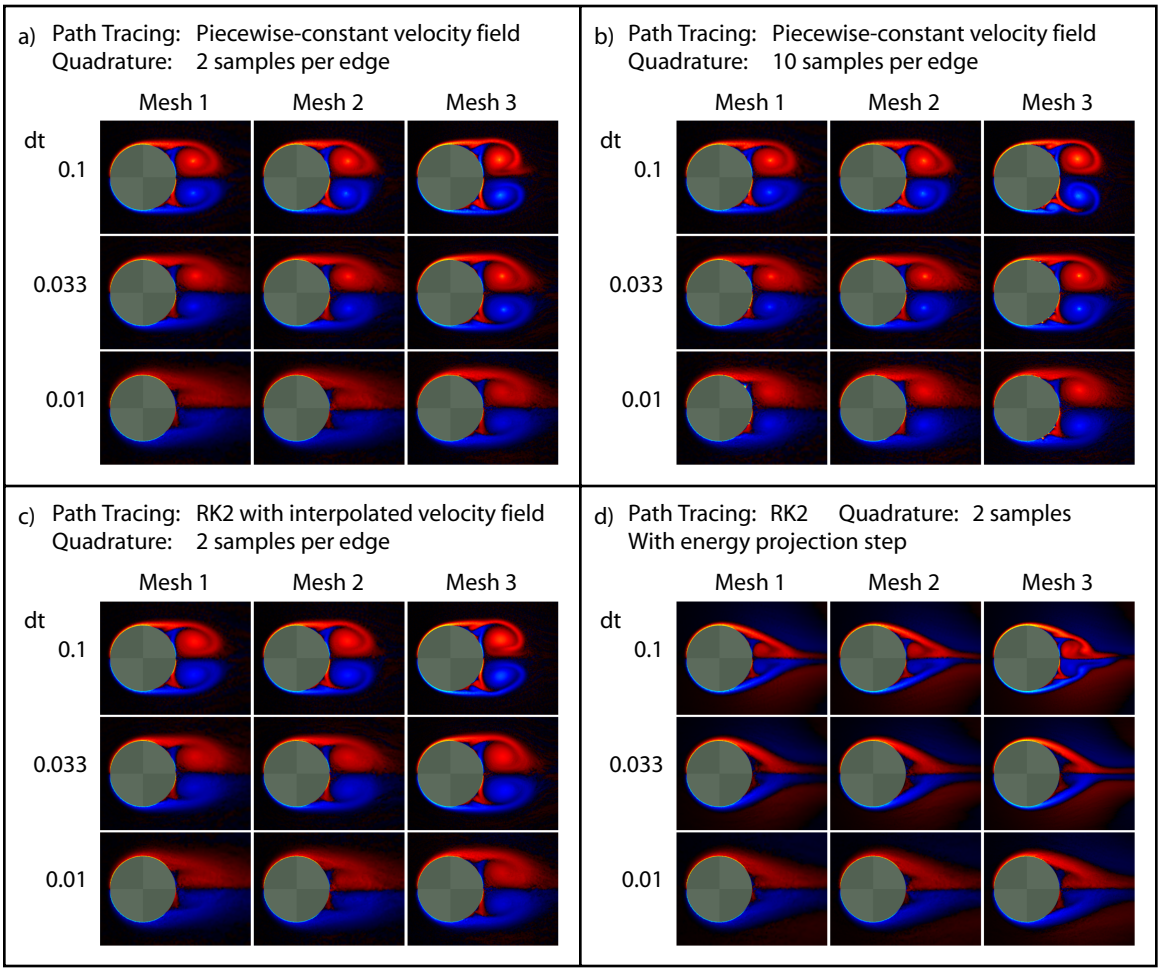


Figure 3.8: *Some representative results from our flow past a cylinder experiment at* $Re = 15\,000$. *Shown are color plots of the vorticity field near the cylinder at* $t = 5$, *where blue represents negative vorticity (counter-clockwise rotation) and red represents positive vorticity (clockwise rotation). We show the effects of using different resolutions of meshes, different time step sizes, different path tracing methods, and different types of numerical quadrature. (d) shows the results of attempting to preserve the total energy with an* L_2 *projection.*

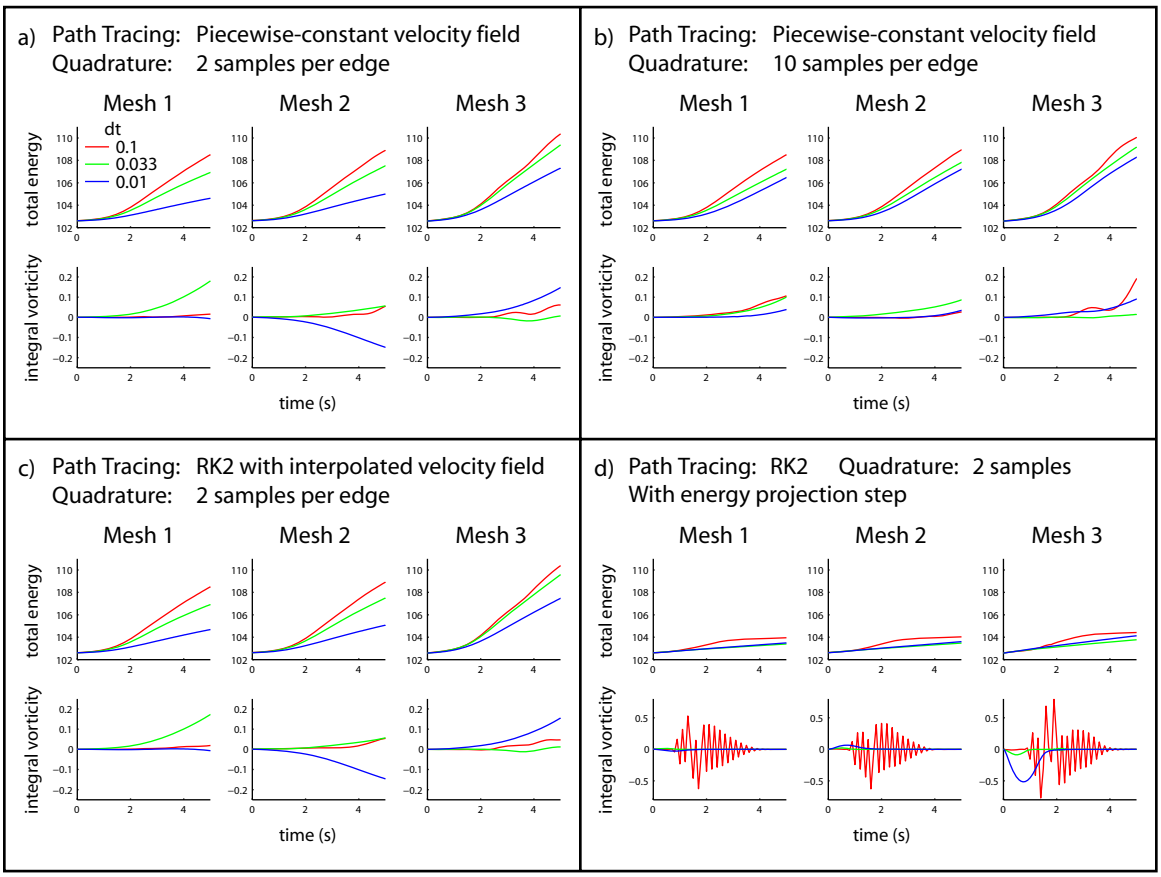


Figure 3.9: *Energy and vorticity behavior of the flow past a cylinder experiment.*

3.3 Flow Past a Cylinder in Rotary Oscillation

This experiment consists of a flow over an oscillating cylinder at Reynolds number $Re = 15\,000$. The cylinder is subjected to a sinusoidal rotation where the tangential velocity on the boundary is given by

$$\frac{\dot{\theta}R}{U_\infty} = 2\sin(\pi t \frac{U_\infty}{R})$$

where R is the radius of the cylinder and U_∞ is the forced horizontal velocity of the fluid on the left boundary of the simulation domain.

We compare our results to those obtained by [28] using a high resolution viscous vortex particle method. This reference simulation uses the particle strength exchange technique ([7]) with 1.7 million computational particles (peak), $\Delta x \approx 0.0015$, and $dt = 0.004$. The running time, taken to $t = 5$, was 40 hours on a 256 processor Cray T3D. We compare this to our simulation using Mesh 3, $dt = 0.033$, the piecewise-constant interpolant for backtracking, and 10 quadrature samples per edge. The running time was 84 minutes on a 3GHz Intel Pentium IV processor.

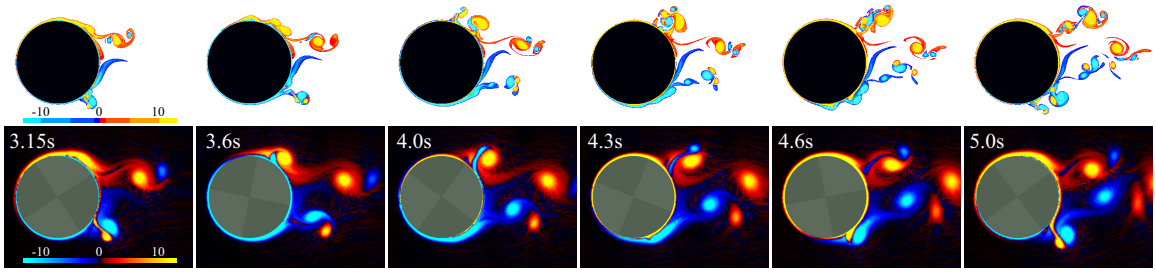


Figure 3.10: *Comparison of our results to those of a high resolution viscous vortex particle method. The top row of images are color vorticity plots from [28] (used with permission), and the bottom row is the result of our method using Mesh* 3, $dt = 0.033$, *the piecewise-constant interpolant for backtracking, and 10 quadrature samples per edge.*

At a coarse scale the simulations exhibit many important similarities. The general pattern of our vortex wake matches that of the reference simulation. The most prominent vortex structures are present, with correct orientations and magnitudes. However, the exact positions of the features deviate slightly; in particular, they are further downstream than they should be. Also, the vorticity field is much smoother because our Eulerian discretization cannot represent discontinuities. But most importantly, our simulation cannot resolve all of the finer structures near the boundary. Although many of the small vortices do indeed form on the boundary, they quickly die out as the flow proceeds and diffuses.

Of all the parameter variations we ran, this one bears the closest resemblance to the reference.

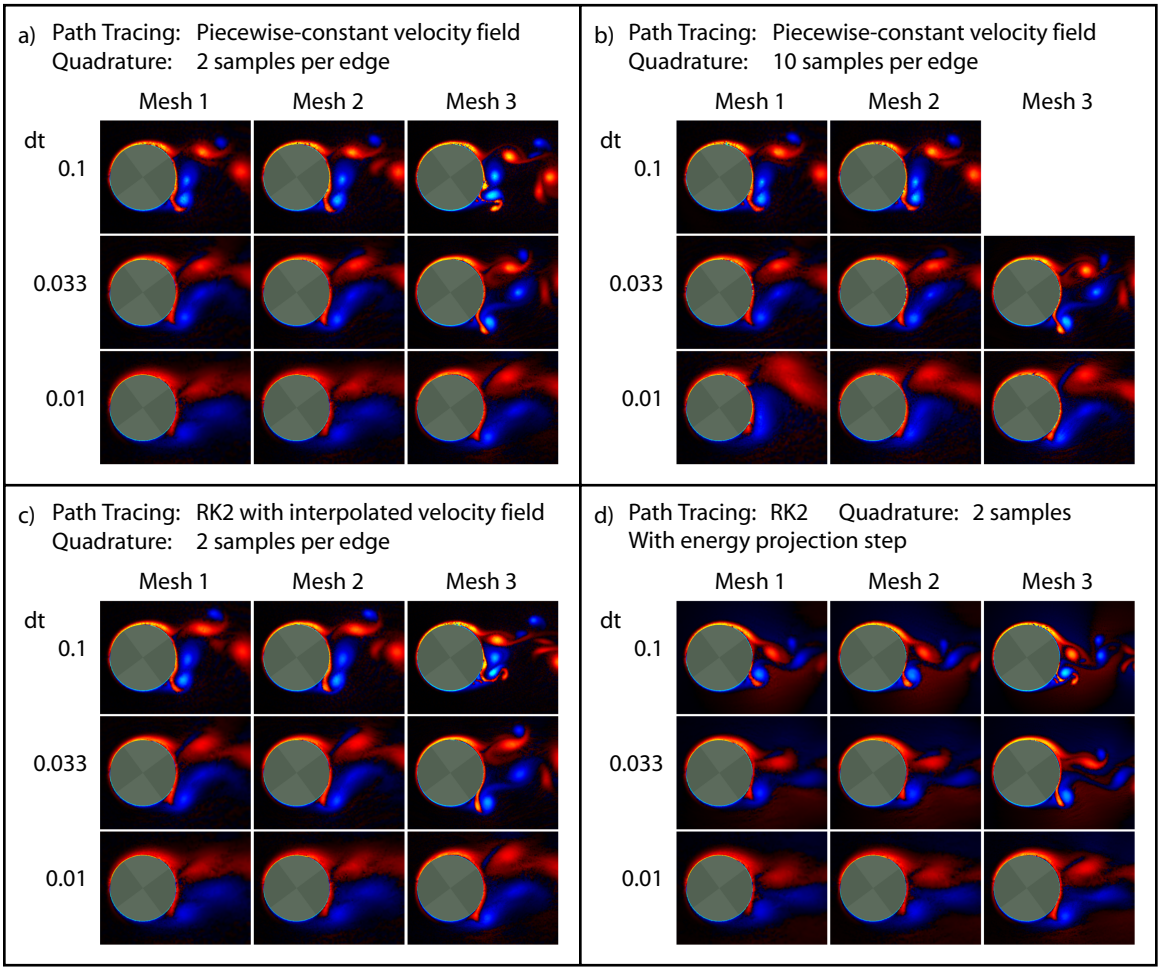


Figure 3.11: *Some representative results from our flow past an oscillating cylinder experiment at* $Re = 15\,000$. *Shown are color plots of the vorticity field near the cylinder at* $t = 5$. *Compare to Fig. 3.8.*

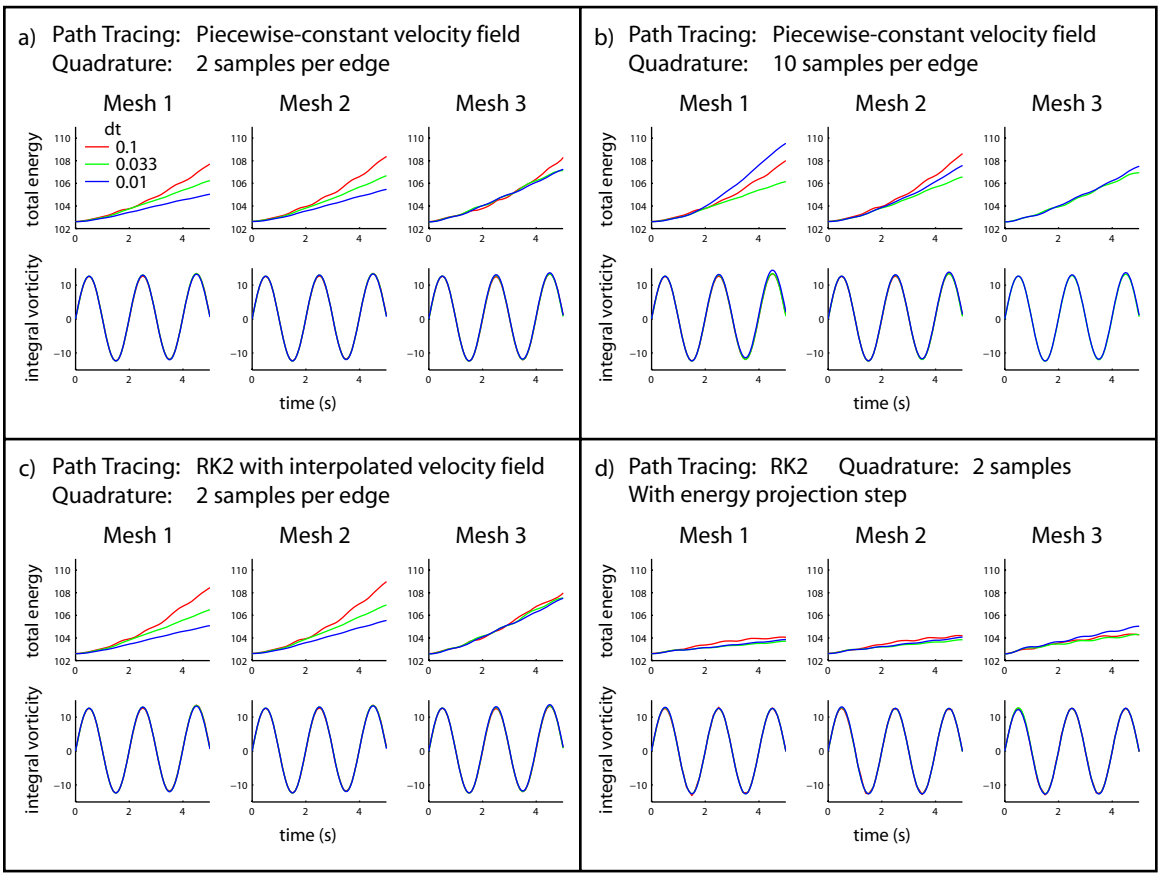


Figure 3.12: *Energy and vorticity behavior of the flow past an oscillating cylinder experiment.*

Images from other simulations are shown in Figure 3.11 for comparison. The overall behavior with respect to variations in mesh size, time step size, etc. echoes the results in Section 3.2.

3.4 Analysis

As our test case we have chosen a very challenging $Re = 15\,000$ flow. The fine scales of relevance at such a high Reynolds number make simulation difficult with any numerical method, and ours is no exception.

We have pointed out that discontinuities in our interpolated velocity field limit the accuracy of the path tracing and the numerical quadrature. And indeed, as Figures 3.8(b) and 3.11(b) demonstrate, the diffusion of the vorticity field is correlated with the accuracy of the numerical quadrature. However, even if the positions of the backtracked loops were exact and the circulation integrals were evaluated exactly, there would still be diffusion due to an intrinsic limitation of Eulerian methods: vorticity can only be represented at discrete locations in space.

Firstly, vortices whose size is below the Nyquist limit of the mesh will be subject to aliasing. Furthermore, all vortices, regardless of size, are affected by re-sampling errors as they move through the mesh. The following example illustrates why. Consider the 1D piecewise-linear curve shown in Figure 3.13(left), represented by values at discrete sample locations along the x-axis. The curve undergoes some translation and then is re-sampled at the original sample locations, yielding a new piecewise-linear curve representing the state of the field at time $t + dt$. The process is then repeated using this new curve as the starting point, and so on. At each step the curve gets shorter and wider, and would eventually, at $t = \infty$, become a straight line. (Importantly, even though the function diffuses, the total integral remains constant.)

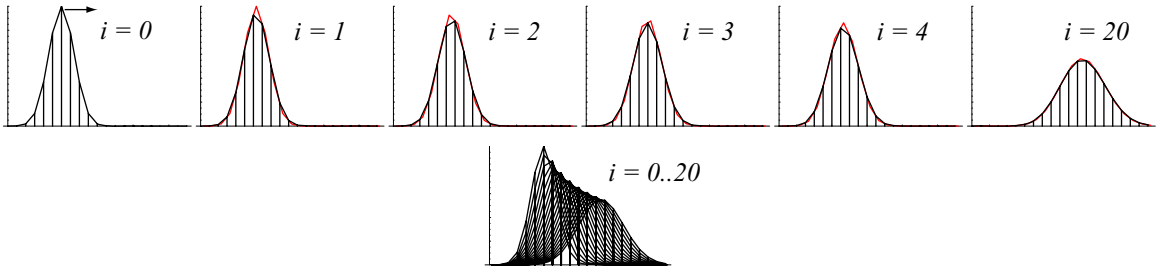


Figure 3.13: *Re-sampling Errors: A curve moving through an Eulerian grid loses its shape over time due to inadequate reconstruction. The red line in each plot is curve* $(i - 1)$ *after translation but before re-sampling.*

This is, in effect, using a triangle reconstruction filter, which is a low-pass filter that aggressively

attenuates even the lower frequencies ([2]). Although this 1D example is not a strictly accurate analogy (because we sample the velocity field, not the vorticity field), it gives a general idea of what is happening. We are currently investigating higher order interpolants to help address this problem.

Another area of future work currently being pursued is the derivation of a variational approach that would preserve energy by construction. As we have seen, a projection or any other such global "post-process" is not the best way to satisfy the conservation laws governing a physical system (see Fig. 3.4(c) and Fig. 3.8(d)). Methods that instead satisfy these properties intrinsically, as ours does with vorticity, exhibit much more accurate and predictable behavior.

3.5 Conclusion

For Computer Graphics applications, the capability of tetrahedral meshes to more effectively resolve domain boundaries is a significant advantage of our method over regular grid based techniques. It allows us to create a very convincing "smoking bunny" animation, for example, at interactive frame rates. Discretizing this domain with hexahedral cells would require significantly more elements and consequently much more time to simulate.

Simulation on curved surfaces has many applications to areas such as special effects and texture generation, for example. Existing methods for addressing this require manipulation of parameterizations and associated Jacobians (see, for example, [31]). Thanks to the intrinsic nature of the variables and operators we use, our algorithm can be applied directly, without modification, to curved domains.

We have also assessed the applicability of this approach to numerical simulation. The remarkable similarity between our results and the results of physical experiments (Fig. 3.4) confirms the validity of our discrete differential approach. Although our method is susceptible to discretization errors, accuracy improves roughly linearly with mesh element size, as expected.

Recommendations Based on the results observed in Sections 3.2 and 3.3 we are able to prioritize certain parameters over others in terms of their effect on accuracy vs. computation time. First, increasing mesh resolution effectively and predictably increases the accuracy of the resulting simulation, as does increasing the number of quadrature samples. The relative cost of each, though, depends on the size of the mesh. Most of the algorithm involves local operations, so increasing the number of cells affects only the Poisson solver. The running time of the Poisson solver (which, in

our case, uses the Conjugate Gradient method) scales as $O(E^{1.5})$ with the number of edges (vertices in 2D) in the mesh. The numerical integration contributes $O(qF)$ to the total running time, where q is the number of quadrature samples per dual edge. Therefore, for small mesh sizes the greatest benefit vs. cost is gained by increasing the mesh size, and for large meshes more samples should be added instead.

For applications with less emphasis on physical accuracy and more concern for producing visually convincing simulations, we recommend first choosing a mesh size based on the size of the features that would be resolved, and then increasing the number of samples based on desired running time. We also recommend the use of the piecewise-constant interpolant for path tracing through the velocity field. If the mesh element size is small relative to the features of the vorticity field, there is no perceptible difference between this and the RK2 path tracer.

Chapter 4

Implementation of a Simplicial Complex Data Structure

The presentation of the fluid simulation algorithm in the previous chapter is intended to be as self-contained as possible. Despite the (possibly intimidating) mathematical theory that went into deriving the algorithms, in the end they lead to a simple, elegant, and straightforward implementation. Although there is sufficient detail about the algorithm itself, readers interested in implementing it should note that the algorithm presumes the existence of a suitable simplicial complex data structure. Such a data structure needs to support local traversal of elements, adjacency information for all dimensions of simplices, a notion of a *dual mesh*, and all simplices must be *oriented*. Unfortunately, most publicly available tetrahedral mesh libraries provide only *unoriented* representations with little more than vertex-tet adjacency information (while we need vertex-edge, edge-triangle, edge-tet, *etc.*). For those eager to implement and build on this and other algorithms based on Discrete Exterior Calculus without having to worry about these details, we discuss here an implementation of a DEC-friendly tetrahedral mesh data structure.

4.1 Motivation

Extending a classic pointer-based mesh data structure to 3D is unwieldy, error-prone, and difficult to debug. We instead take a more abstract set-oriented view in the design of our data structure, by turning to the formal definition of an abstract simplicial complex. This gives our implementation the following desirable properties:

- ◇ We treat the mesh as a graph and perform all of our operations combinatorially.
- ◇ There is no cumbersome pointer-hopping typical of most mesh data structures.

◇ The design easily generalizes to arbitrary dimension.

◇ The final result is very compact and simple to implement.

In effect we are taking advantage of the fact that during assembly of all the necessary structures one can use high level, abstract data structures. That way formal definitions can be turned into code almost verbatim. While these data structures (*e.g.*, sets and maps) may not be the most efficient for *computation*, an approach which uses them during assembly is far less error prone. Once everything has been assembled it can be turned easily into more efficient packed representations (*e.g.*, compressed row storage format sparse matrices) with their more favorable performance during the actual computations which occur, *e.g.*, in physical simulation.

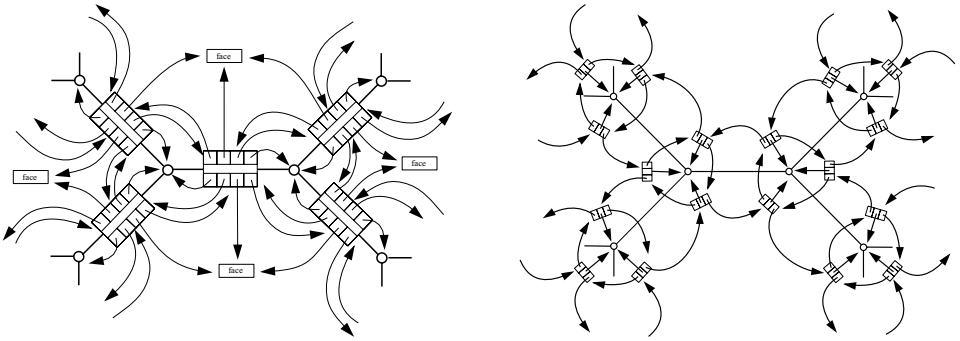


Figure 4.1: *Some typical examples of 2D mesh representations (from [18]; used with permission). Such pointer-based data structures become quite difficult to manage once they are extended to 3D.*

We will begin with a few definitions in Section 4.2, and see how these translate into our tuple-based representation in Section 4.3. The boundary operator, described in Section 4.4, facilitates mesh traversal and implements the discrete exterior derivative. We show how everything is put together in Section 4.5. Finally, we discuss our implementation of the DEC operators in Section 4.6.

4.2 Definitions

We begin by recalling the basic definitions of the objects we are dealing with. The focus here is on the rigorous mathematical definitions in a form which then readily translates into high level algorithms. The underlying concepts are simply what we all know informally as *meshes* in either two (triangle) or three (tet) dimensions.

Simplices A *simplex* is a general term for an element of the mesh, identified by its dimension. 0-simplices are vertices, 1-simplices are edges, 2-simplices are triangles, and 3-simplices are tetra-

hedra.

Abstract Simplicial Complex This structure encodes all the relationships between vertices, edges, triangles, and tets. Since we are only dealing with combinatorics here the atomic element out of which everything is built are the integers $0 \leq i < n$ referencing the underlying vertices. For now they do not yet have point positions in space. Formally, an abstract simplicial complex is a *set of subsets* of the integers $0 \leq i < n$, such that if a subset is contained in the complex then so are all its subsets. For example, a 3D complex is a collection of tetrahedra (4-tuples), triangles (3-tuples), edges (2-tuples), and vertices (singletons), such that if a tetrahedron is present in the complex then so must be its triangles, edges, and vertices. All our simplicial complexes will be proper three or two manifolds, possibly with boundary and may be of arbitrary topology (*e.g.*, containing voids and tunnels).

Manifold The DEC operators that we build on are defined only on meshes which represent manifolds. Practically speaking this means that in a 3D simplicial complex all triangles must have two incident tets only (for a boundary triangle there is only one incident tet). Every edge must have a set of tets incident on it which form a single "ring" which is either open (at the boundary) or closed (in the interior). Finally for vertices it must be true that all incident tets form a topological sphere (or hemisphere at the boundary). These properties should be asserted upon reading the input. For example, for triangles which bound tets one must assert that each such triangle occurs in at most two tets. For an edge the "ring" property of incident tets can be checked as follows. Start with one incident tet and jump across a shared triangle to the next tet incident on the edge. If this walk leads back to the original tet *and* all tets incident on the edge can thusly be visited, the edge passes the test. (For boundary edges such a walk starts at one boundary tet and ends at another.) The test for vertices is more complex. Consider all tets incident on the given vertex. Using the tet/tet adjacency across shared triangles one can build the adjacency graph of all such tets. This graph must be a topological sphere (or hemisphere if the vertex is on the boundary).

Since we need everything to be properly oriented we will only allow *orientable* manifolds (*i.e.*, no Möbius strips or Klein bottles).

Regularity To make life easier on ourselves we also require the simplicial complex to be *strongly regular*. This means that simplices must not have identifications on their boundaries. For example,

edges are not allowed to begin and end in the same vertex. Similarly, the edges bounding a triangle must not be identified nor do we allow edges or triangles bounding a tet to be identified. In practice this is rarely an issue since the underlying geometry would need to be quite contorted for this to occur. Strictly speaking though such identifications are possible in more general, abstract settings without violating the manifold property.

Embedding It is often useful to distinguish between the *topology* (neighbor relationships) and the *geometry* (point positions) of the mesh. A great deal of the operations performed on our mesh can be carried out using only topological information, *i.e.*, without regard to the embedding. The embedding of the complex is given by a map $p : [0, n) \mapsto (x, y, z) \in \mathbb{R}^3$ on the vertices (which is extended piecewise linearly to the interior of all simplices). For example, when we visualize a mesh as being composed of piecewise linear triangles (for 2D meshes) or piecewise linear tets, we are dealing with the geometry. Most of the algorithms we describe below do not need to make reference to this embedding. When implementing these algorithms it is useful to only think in terms of combinatorics. There is only one stage where we care about the geometry: the computation of metric dependent quantities needed in the definition of the Hodge star.

4.3 Simplex Representation

Ignoring orientations for a moment, each k-simplex is represented as a $(k + 1)$-tuple identifying the vertices that bound the simplex. In this view a tet is simply a 4-tuple of integers, a triangle is a 3-tuple of integers, an edge is a 2-tuple, and a vertex is a singleton. Note that all permutations of a given tuple refer to the same simplex. For example, (i, j, k) and (j, i, k) are different *aliases* for the same triangle. In order to remove ambiguities, we must designate one *representative* alias as the representation of the simplex in our data structures. We do this by using the *sorted* permutation of the tuple. Thus each simplex (tuple) is stored in our data structures as its canonical (sorted) representative. Then if we, for example, need to check whether two simplices are in fact the same we only need to compare their representatives element by element.

All this information is stored in lists we designate $\mathbf{V}$, $\mathbf{E}$, $\mathbf{F}$, and $\mathbf{T}$. They contain one representative for every vertex, edge, triangle, and tet, respectively, in the mesh.

4.3.1 Forms

The objects of computation in an algorithm using DEC are forms. Formally, a differential k-form is a quantity that can be integrated over a k dimensional domain. For example, consider the expression $\int f(x)dx$ (x being a scalar). The integrand $f(x)dx$ is called a 1-*form*, because it can be integrated over any 1-dimensional interval. Similarly, the dA in $\int\int dA$ would be a 2-form.

Discrete differential forms are dealt with by storing the results of the integrals themselves, instead of the integrands. That is, discrete k-forms associate one value with each k-simplex, representing the integral of the form over that simplex. With this representation we can recover the integral over any k-dimensional chain (the union of some number of k-simplices) by summing the value on each simplex (using the linearity of the integral).

Since all we have to do is to associate one value with each simplex, for our purposes forms are simply vectors of real numbers where the size of the vector is determined by the number of simplices of the appropriate dimension. 0-forms are vectors of size $|\mathbf{V}|$, 1-forms are vectors of size $|\mathbf{E}|$, 2-forms are vectors of size $|\mathbf{F}|$, and 3-forms are vectors of size $|\mathbf{T}|$. Such a vector representation requires that we assign an index to each simplex. We use the position of a simplex in its respective list ($\mathbf{V}$, $\mathbf{E}$, $\mathbf{F}$, or $\mathbf{T}$) as its index into the form vectors.

4.3.2 Orientation

Because the vectors of values we store represent integrals of the associated k-form over the underlying simplices, we must keep track of orientation. For example, reversing the bounds of integration on $\int_a^b f(x)dx$ flips the sign of the resulting value. To manage this we need an *intrinsic orientation* for each simplex. It is with respect to this orientation that the values stored in the form vectors receive the appropriate sign. For example, suppose we have a 1-form f with value f_{ij} assigned to edge $e = (i, j)$; that is, the real number f_{ij} is the integral of the 1-form f over the line segment (p_i, p_j). If we query the value of this form on the edge (j, i) we should get $-f_{ij}$.

Hence every tuple must be given a sign indicating whether it agrees ($+$) or disagrees ($-$) with the intrinsic orientation of the simplex. Given a set of integers representing a simplex, there are two equivalence classes of orderings of the given tuple: the even and odd permutations of the integers in question. These two equivalence classes correspond to the two possible orientations of the simplex (see Fig. 4.2).

Note that assigning a sign to any one alias (*i.e.*, the representative) implicitly assigns a sign to

all other aliases. Let us assume for a moment that the sign of all representatives is known. Then the sign S of an arbitrary tuple t, with representative r, is

$$\mathbf{S}(t) = \begin{cases} \mathbf{S}(r) & \text{if } t \text{ is in the same equivalence class as } r \\ -\mathbf{S}(r) & \text{if } t \text{ is in the opposite equivalence class.} \end{cases}$$

More formally, let P be the permutation that permutes t into r (*i.e.*, $r = P(t)$). Then

$$\mathbf{S}(t) = \mathbf{S}(P)\mathbf{S}(P(t)).$$

Here $\mathbf{S}(P)$ denotes the sign of the permutation P with $+1$ for even and -1 for odd permutations.

All that remains, then, is to choose an intrinsic orientation for each simplex and set the sign of the representative alias accordingly. In general the assignment of orientations is arbitrary, as long as it is consistent. For all subsimplices we choose the representative to be positively oriented, so that the right-hand-side of the above expression reduces to $\mathbf{S}(P)$. For top-level simplices (tets in 3D, triangles in 2D), we use the convention that a positive volume corresponds to a positively oriented simplex. We therefore require a volume form which, together with an assignment of points to vertices, will allow us to orient all tets. Recall that a volume form accepts three (for 3D; two for 2D) vectors and returns either a positive or negative number (assuming the vectors are linearly independent). So the sign of a 4-tuple is:

$$\mathbf{S}(i_0, i_1, i_2, i_3) = \mathbf{S}(\mathrm{Vol}(p_{i_1} - p_{i_0}, p_{i_2} - p_{i_0}, p_{i_3} - p_{i_0})).$$

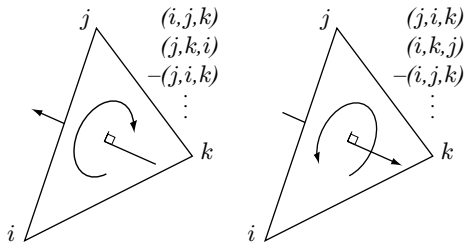


Figure 4.2: *All permutations of a triple (i, j, k) refer to the same triangle, and the sign of the permutation determines the orientation.*

4.4 The Boundary Operator

The *faces* of a k-simplex are the $(k-1)$-simplices that are incident on it, *i.e.*, the subset of one lower dimension. Every k-simplex has $k+1$ faces. Each face corresponds to removing one integer from the tuple, and the relative orientation of the face is $(-1)^i$ where i is the index of the integer that was removed. To clarify:

- ◇ The faces of a tet $+(t_0, t_1, t_2, t_3)$ are $-(t_0, t_1, t_2)$, $+(t_0, t_1, t_3)$, $-(t_0, t_2, t_3)$, and $+(t_1, t_2, t_3)$.
- ◇ The faces of a triangle $+(f_0, f_1, f_2)$ are $+(f_0, f_1)$, $-(f_0, f_2)$, and $+(f_1, f_2)$.
- ◇ The faces of an edge $+(e_0, e_1)$ are $-(e_0)$ and $+(e_1)$.

We can now define the boundary operator ∂ which maps simplices to their their faces. Given the set of tets $\mathbf{T}$ we define $\partial^3 : \mathbf{T} \rightarrow \mathbf{F}^4$ as

$$\begin{aligned} \partial^3(+(i_0, i_1, i_2, i_3)) \quad = \quad & \{-(i_0, i_1, i_2), +(i_0, i_1, i_3), \\ & -(i_0, i_2, i_3), +(i_1, i_2, i_3)\}. \end{aligned}$$

Similarly for $\partial^2 : \mathbf{F} \rightarrow \mathbf{E}^3$ (which maps each triangle to its three edges) and $\partial^1 : \mathbf{E} \rightarrow \mathbf{V}^2$ (which maps each edge to its two vertices).

We represent these operators as sparse adjacency matrices (or, equivalently, signed adjacency lists), containing elements of type $+1$ and -1 only. So ∂^3 is implemented as a matrix of size $|\mathbf{F}| \times |\mathbf{T}|$ with 4 non-zero elements per column, ∂^2 an $|\mathbf{E}| \times |\mathbf{F}|$ matrix with 3 non-zero elements per column, and ∂^1 a $|\mathbf{V}| \times |\mathbf{E}|$ matrix with 2 non-zero elements per column (one $+1$ and one -1). The transposes of these matrices are known as the *coboundary* operators, and they map simplices to their *cofaces*—neighbor simplices of one higher dimension. For example, $(\partial^2)^T$ maps an edge to the "pinwheel" of triangles incident on that edge.

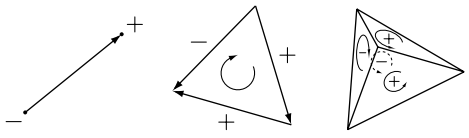


Figure 4.3: *The boundary operator identifies the faces of a simplex as well as their* relative *orientations. In this illustration, arrows indicate intrinsic orientations and signs indicate the relative orientation of a face to a parent.*

These matrices allow us to iterate over the faces or cofaces of any simplex, by walking down the columns or across the rows, respectively. In order to traverse neighbors that are more than one dimension removed (*i.e.*, the tets adjacent to an edge or the faces adjacent to a vertex) we simply

concatenate the appropriate matrices, but without the signs. (If we kept the signs in the matrix multiplication any such consecutive product would simply return the zero matrix reflecting the fact that the boundary of a boundary is always empty.)

4.5 Construction

Although we still need a few auxiliary wrapper and iterator data structures to provide an interface to the mesh elements, the simplex lists and boundary matrices contain the entirety of the topological data of the mesh. All that remains, then, is to fill in this data.

We read in our mesh as a list of (x, y, z) vertex positions and a list of 4-tuples specifying the tets. Reading the mesh in this format eliminates the possibility of many non-manifold scenarios; for example, there cannot be an isolated edge that does not belong to a tet. We assume that all integers in the range $[0, n)$ appear at least once in the tet list (this eliminates isolated vertices), and no integer outside of this range is present.

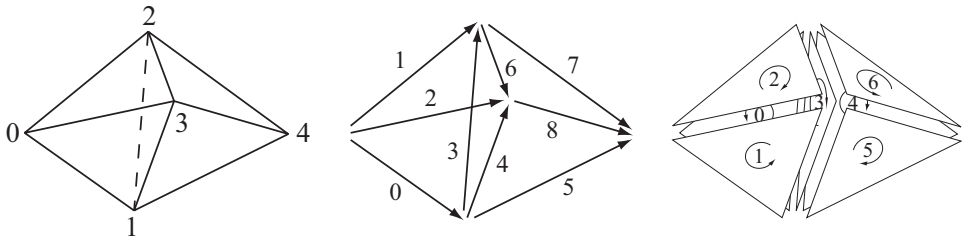
Once $\mathbf{T}$ is read in, building $\mathbf{E}$ and $\mathbf{F}$ is trivial; for each tuple in $\mathbf{T}$, append all subsets of size 2 and 3 to $\mathbf{E}$ and $\mathbf{F}$ respectively. We must be sure to avoid duplicates, either by using a unique associative container, or by sorting the list afterward and removing duplicates. Then the boundary operator matrices are constructed as follows:

for each simplex s
    construct a tuple for each face f of s as described in Section 4.4
    determine the index i of f by locating its representative
    set the entry of the appropriate matrix at row i, column s to $\mathbf{S}(f)$

Figure 4.4 shows a complete example of a mesh and its associated data structure.

4.6 DEC Operators

Now we discuss the implementation of the two most commonly used DEC operators: the exterior derivative and the Hodge star. As we will see, in the end these also amount to nothing more than sparse matrices that can be applied to our form vectors.



$$\partial^3 = \begin{pmatrix} -1 & \\ 1 & \\ -1 & \\ 1 & -1 \\ & 1 \\ & -1 \\ & 1 \end{pmatrix}$$

V	**E**	**F**	**T**
$[0]$	$[0,1]$	$[0,1,2]$	$[0,1,2,3]$
$[1]$	$[0,2]$	$[0,1,3]$	$[1,2,3,4]$
$[2]$	$[0,3]$	$[0,2,3]$	
$[3]$	$[1,2]$	$[1,2,3]$	
$[4]$	$[1,3]$	$[1,2,4]$	
	$[1,4]$	$[1,3,4]$	
	$[2,3]$	$[2,3,4]$	
	$[2,4]$		
	$[3,4]$		

$$\partial^2 = \begin{pmatrix} 1 & 1 & & & & & \\ -1 & & 1 & & & & \\ & -1 & -1 & & & & \\ 1 & & & 1 & 1 & & \\ & 1 & & -1 & & 1 & \\ & & & & -1 & -1 & \\ & & 1 & 1 & & & 1 \\ & & & & 1 & & -1 \\ & & & & & 1 & 1 \end{pmatrix}$$

$$\partial^1 = \begin{pmatrix} -1 & -1 & -1 & & & & & & \\ 1 & & & -1 & -1 & -1 & & & \\ & 1 & & 1 & & & -1 & -1 & \\ & & 1 & & 1 & & 1 & & -1 \\ & & & & & 1 & & 1 & 1 \end{pmatrix}$$

Figure 4.4: *A simple mesh and all associated data structures.*

4.6.1 Exterior Derivative

As we have seen earlier in the course, the discrete exterior derivative is defined using Stokes' theorem such that

$$\int_\sigma d\omega = \int_{\partial\sigma} \omega$$

where ω is a k-form, and σ is a $(k+1)$-simplex. In words, this equation states that the evaluation of $d\omega$ on a simplex is equal to the evaluation of ω on the boundary of that simplex.

Let us try to understand this theorem with a few examples. Consider a 0-form f, *i.e.,* a function giving values at vertices. With that, df is a 1-form which can be integrated along an edge (say with

endpoints denoted a and b) and Stokes' theorem states the well known fact

$$\int_{[a,b]} df = f(b) - f(a).$$

The right hand side is simply the evaluation of the 0-form f on the boundary of the edge (*i.e.*, its endpoints), with appropriate signs indicating the orientation of the edge.

What about triangles? If f is a 1-form (one value per edge), then df is a 2-form that can be evaluated on a triangle abc as

$$\begin{aligned} \int_{\Delta abc} df &= \int_{\partial(\Delta abc)} f \\ &= \int_{[a,b]} f + \int_{[b,c]} f + \int_{[c,a]} f \\ &= f_{ab} + f_{bc} + f_{ca} \end{aligned}$$

using the subscript notation from Section 4.3.2. Again, the right hand side is simply the evaluation of the 1-form f on the boundary of the triangle—its three edges.

We can restate the general form of the theorem for our discrete forms as

$$d\omega_\sigma = \sum_{s \in \partial\sigma} \omega_s$$

Written this way, it is easy to see that this can be implemented as the multiplication of a form vector by the coboundary matrix ∂^T.

4.6.2 The Dual Mesh and the Hodge Star

Every complex has a dual. The dual of a simplicial complex is a *cell complex* where primal k-simplices correspond to dual $(n-k)$-cells. So in our case there are $|\mathbf{V}|$ dual polyhedra, $|\mathbf{E}|$ dual polygons, $|\mathbf{F}|$ dual edges, and $|\mathbf{T}|$ dual vertices, corresponding to primal vertices, edges, triangles, and tetrahedra, respectively (see Fig. 2.2). Note that, since every dual cell is co-located with a primal simplex and the cardinality is the same, in the code there is no explicit representation for the dual mesh. Where appropriate, dual cells are queried through the corresponding primal simplex index.

The operator that transforms a primal k-form into a dual $(n-k)$-form is known as the *Hodge star*. There are many different kinds of Hodge stars, the simplest of which is the *diagonal Hodge star*.

We again attempt to motivate the definition with some intuition. When transferring a quantity from a primal simplex to a dual cell, the quantities must "agree" somehow. Since these are integral values, simply setting the value on the dual to be equal to the value on the primal does not make sense, as the domain of integration is unrelated. Instead, we require that the *integral density* be equal. So, if ω denotes the evaluation of a form on a primal k-simplex σ, then $\star\omega$ is the value on the dual $(n-k)$-cell $\tilde{\sigma}$ such that

$$\frac{\omega}{\mathrm{Vol}(\sigma)} = \frac{\star\omega}{\mathrm{Vol}(\tilde{\sigma})}$$

allowing us to define $\star$ as

$$\star = \frac{\mathrm{Vol}(dual)}{\mathrm{Vol}(primal)}.$$

In effect the diagonal Hodge star requires that the averages of the integrand over the respective domains agree.

This is represented as a diagonal matrix so that, again, application of the operator becomes a simple matrix-vector multiplication. Note that when transforming quantities from the dual to the primal, the inverse of this matrix is used. Since the matrix is diagonal we only store the diagonal entries. There are as many of these as there are simplices of the appropriate dimension. Consequently the diagonal Hodge star can be represented with vectors of length $|\mathbf{V}|$, $|\mathbf{E}|$, $|\mathbf{F}|$, and $|\mathbf{T}|$ respectively.

4.6.2.1 Calculating Dual Volumes

So far the entire implementation has been in terms of the combinatorics of the mesh, but when constructing the Hodge star we must finally introduce the geometry. After all, the purpose of the Hodge star is to capture the metric. The volumes of the primal simplices are straightforward: 1 for vertices, length for edges, area for triangles, and volume for tetrahedra. The dual volumes are similarly defined, but in order to avoid constructing the graph of the dual mesh explicitly, we calculate the dual volumes as follows.

If we use the circumcentric realization of the dual mesh (*i.e.*, dual vertices are at the circumcenters of the associated tets), we can exploit the following facts when calculating the dual volumes.[1]

[1] Circumcentric duals may only be used if the mesh satisfies the Delaunay criterion. If it does not, a barycentric dual mesh may be used. However, care must be taken if a barycentric dual mesh is used, as dual edges are no longer straight lines (they are piecewise linear), dual faces are no longer planar, and dual cells are no longer necessarily convex.

- ◇ A dual edge (dual of a primal triangle t) is linear, is normal to t, and is collinear with the circumcenter of t (though the line segment need not necessarily pass through t).
- ◇ A dual polygon (dual of a primal edge e) is planar, is orthogonal to e, and is coplanar with the center of e (though it need not intersect e).
- ◇ A dual cell (dual of a primal vertex v) is the convex intersection of the half-spaces defined by the perpendicular bisectors of the edges incident on v.

Just as with primal vertices, the volume of a dual vertex is defined to be 1. For the others, we can conceptually decompose each cell into pieces bounded by lower dimensional cells, and sum the volumes of the pieces. For example, a dual polyhedron can be seen as the union of some number of pyramids, where the base of each pyramid is a dual polygon and the apex is the primal vertex. Similarly, a dual polygon can be seen as a union of triangles with dual edges at the bases, and dual edges can be seen as a union of (two) line segments with dual vertices at the bases. The following pseudocode illustrates how the volumes are calculated.

```
vec3 C( Simplex s );  // gives the circumcenter of s

// Initialize all dual volumes to 0.

// Dual edges
for each primal triangle f
    for each primal tet t_f incident on f
        b ← t_f.dualVolume  // 1
        h ← ||C(f) − C(t_f)||
        f.dualVolume ← f.dualVolume + (1/1) bh

// Dual polygons
for each primal edge e
    for each primal triangle f_e incident on e
        b ← f_e.dualVolume
        h ← ||C(e) − C(f_e)||
        e.dualVolume ← e.dualVolume + (1/2) bh

// Dual polyhedra
for each primal vertex v
    for each primal edge e_v incident on v
        b ← e_v.dualVolume
        h ← ||C(v) − C(e_v)||
        v.dualVolume ← v.dualVolume + (1/3) bh
```

Note that, even when dealing with the geometry of the mesh, this part of the implementation still generalizes trivially to arbitrary dimension.

4.7 Summary

All the machinery discussed above can be summarized as follows:

- k-forms as well as the Hodge star are represented as vectors of length $|\mathbf{V}|$, $|\mathbf{E}|$, $|\mathbf{F}|$, and $|\mathbf{T}|$;
- the discrete exterior derivative is represented as (transposes of) sparse adjacency matrices containing only entries of the form $+1$ and -1 (and many zeros); the adjacency matrices are of

dimension $|\mathbf{V}| \times |\mathbf{E}|$ (boundary of edges), $|\mathbf{E}| \times |\mathbf{F}|$ (boundary of triangles), and $|\mathbf{F}| \times |\mathbf{T}|$ (boundary of tets).

In computations these matrices then play the role of operators such as grad, curl, and div and can be composed to construct operators such as the Laplacian (and many others).

While the initial setup of these matrices is best accomplished with associative containers, their final form can be realized with standard sparse matrix representations. Examples include a compressed row storage format, a vector of linked lists (one linked list for each row), or a two dimensional linked list (in effect, storing the matrix and its transpose simultaneously) allowing fast traversal of either rows or columns. The associative containers store integer tuples together with orientation signs. For these we suggest the use of sorted integer tuples (the canonical representatives of each simplex). Appropriate comparison operators needed by the container data structures simply perform lexicographic comparisons.

And that's all there is to it!

Chapter 5

Conclusion

In this thesis we have presented a novel theoretical approach to fluid dynamics, along with its practical implementation and various simulation results. We have carefully discretized the physics of flows to respect the most fundamental geometric structures that characterize their behavior. Amongst the several specific benefits that we demonstrated, the most important is the circulation preservation property of the integration scheme, as evidenced by our numerical examples. The discrete quantities we used are intrinsic, allowing us to go to curved manifolds with no additional complication. Finally, the machinery employed in our approach can be used on any simplicial complex. We wish to emphasize, however, that the same methodology also applies directly to more general spatial partitionings, and in particular, to regular grids and hybrid meshes [11]—rendering our approach widely applicable to existing fluid simulators.

For future work, a rigorous comparison of the current method with standard approaches should be undertaken. Using Bjerknes' circulation theorem for compressible flows may also be an interesting avenue. Finally, we limited ourselves to the investigation of our scheme without focusing on the separate issue of order of accuracy. Coming up with an integration scheme that is higher-order accurate will be the object of further investigation, as it requires a better (denser) Hodge star.

Appendix A

Discrete Operators

A.1 Discrete Exterior Derivative

A key ingredient to defining the discrete version of the exterior derivative d is Stokes' theorem:

$$\int_{\sigma} d\alpha = \int_{\partial\sigma} \alpha,$$

where σ denotes a $(k+1)$-cell and α is a k-form. Stokes' theorem states that the integral of $d\alpha$ (a $(k+1)$-form) over a $(k+1)$-cell equals the integral of the k-form α over the *boundary* of the $(k+1)$-cell (*i.e.*, a k-cell). Stokes' theorem can thus be used as a way to *define* the d operator in terms of the boundary operator ∂. Or, said differently, once we have the boundary operator, the operator d follows immediately if we wish Stokes' theorem to hold on the simplicial complex.

To use a very simple example, consider a 0-form f, *i.e.*, a function giving values at vertices. With that, df is a 1-form which can be integrated along an edge (say with end points denoted a and b) and Stokes' theorem states the well known fact:

$$\int_{[a,b]} df = f(b) - f(a).$$

The right hand side is simply the evaluation of the 0-form f on the boundary of the edge, *i.e.*, its endpoints (with appropriate signs indicating the orientation of the edge). Actually, one can define a hierarchy of these operators that mimic the operators given in the continuous setting by the gradient (∇), curl ($\nabla\times$), and divergence ($\nabla\cdot$), namely,

- d_0: maps 0-forms to 1-forms and corresponds to the **Gradient** modulo a Hodge star;
- d_1: maps 1-forms (values on edges) to 2-forms (values on faces). The value on a given face is

simply the sum (by linearity of the integral) of the 1-form values on the boundary (edges) of the face with the signs chosen according to the local orientation. d_1 corresponds to the **Curl** modulo a Hodge star;

- d_2: maps 2-forms to 3-forms and corresponds to the **Divergence** modulo a Hodge star.

A.2 Discrete Laplacian

We have seen in Section 2.5 how the vorticity can be directly derived from the set of all face fluxes. However, during the simulation, we will also need to recover flux *from* vorticity. For this we employ the Helmholtz-Hodge decomposition theorem, stating that any vector field $\mathbf{u}$ can be decomposed into three components (given appropriate boundary conditions)

$$\mathbf{u} = \nabla \times \phi + \nabla \boldsymbol{\psi} + \mathbf{h}. \tag{A.1}$$

When represented in terms of discrete forms this reads as follows:

$$U = d\Phi + \star d \star \Psi + H \tag{A.2}$$

For the case of incompressible fluids (*i.e.*, with zero divergence), two of the three components are sufficient to describe the velocity field: the curl of a vector potential and a harmonic field. This implies that when decomposing the 2-form U, we may set Ψ to 0. If the topology of the domain is trivial, we can furthermore ignore the harmonic part H (we discuss a full treatment of arbitrary topology in Section 2.5.4), leaving us with $U = d\Phi$.

Thus, we can recover the velocity field solely from the vorticity by solving a Poisson equation to get the vector potential Φ and then applying the curl operator to the potential. The Poisson equation to solve for the 1-form Φ (values on primal edges) is as follows:

$$(\star d \star^{-1} d^T \star + d^T \star d)\Phi = d^T \star U = \Omega \tag{A.3}$$

To arrive at this equation, we applied $d\star$ to both sides of Eq. (A.2), and set the gauge of this Poisson problem as $d^T \star \Phi = 0$. As the Laplacian Δ in *differential calculus* is $d \star d \star + \star\, d \star d$, one can readily verify that the previous equation is, indeed, a discrete version of the Poisson equation. It literally corresponds to $\Delta\phi = (\nabla\nabla\cdot - \nabla\times\nabla\times)\phi = \nabla\times\mathbf{u}$. Notice that the left-side matrix (that we

will denote L) is *symmetric and sparse*, thus ideally suited for fast numerical solvers.

Our linear operators (and, in particular, the discrete Laplacian) differ from another discrete Poisson setup on simplicial complexes proposed in [33]: the ones we use have smaller support, which results in sparser and better conditioned linear systems [3]—an attractive feature in the context of numerical simulation.

Bibliography

[1] ALLIEZ, P., COHEN-STEINER, D., YVINEC, M., AND DESBRUN, M. Variational tetrahedral meshing. *ACM Transactions on Graphics (SIGGRAPH)* (Aug. 2005).

[2] BLINN, J. Return of the jaggy. *IEEE Computer Graphics and Applications 9*, 2 (1989), 82–89.

[3] BOSSAVIT, A. *Computational Electromagnetism*. Academic Press, Boston, 1998.

[4] BOSSAVIT, A., AND KETTUNEN, L. Yee-like schemes on a tetrahedral mesh. *Int. J. Num. Modelling: Electr. Networks, Dev. and Fields 12* (July 1999), 129–142.

[5] CHORIN, A., AND MARSDEN, J. *A Mathematical Introduction to Fluid Mechanics*, 3rd edition ed. Springer-Verlag, 1979.

[6] CRAWFIS, R., SHEN, H.-W., AND MAX, N. Flow visualization techniques for cfd using volume rendering. In *9th International Symposium on Flow Visualization* (2000).

[7] DEGOND, P., AND MAS-GALLIC, S. The weighted particle method for convection-diffusion equations. i - the case of an isotropic viscosity. ii - the anisotropic case. *Mathematics of Computation 53* (Oct. 1989), 485–507.

[8] DESBRUN, M., KANSO, E., AND TONG, Y. Discrete differential forms for computational sciences. *Chapter in ACM SIGGRAPH '05* Discrete Differential Geometry *Courses Notes* (2005).

[9] ELCOTT, S., TONG, Y., KANSO, E., DESBRUN, M., AND SCHRÖDER, P. Discrete, circulation-preserving, and stable simplicial fluids, 2005. Submitted to ACM Symposium on Computer Animation.

[10] FEDKIW, R., STAM, J., AND JENSEN, H. W. Visual simulation of smoke. In *Proceedings of ACM SIGGRAPH* (Aug. 2001), Computer Graphics Proceedings, Annual Conference Series, pp. 15–22.

[11] FELDMAN, B. E., O'BRIEN, J. F., AND KLINGNER, B. M. A method for animating viscoelastic fluids. *ACM Transactions on Graphics (SIGGRAPH)* (Aug. 2005).

[12] FETECAU, R. C., MARSDEN, J. E., ORTIZ, M., AND WEST, M. Nonsmooth lagrangian mechanics and variational collision integrators. *SIAM J. Applied Dynamical Systems 2* (2003), 381–416.

[13] FOSTER, N., AND FEDKIW, R. Practical animation of liquids. In *Proceedings of ACM SIGGRAPH* (Aug. 2001), Computer Graphics Proceedings, Annual Conference Series, pp. 23–30.

[14] FOSTER, N., AND METAXAS, D. Modeling the motion of a hot, turbulent gas. In *Proceedings of SIGGRAPH 97* (Aug. 1997), Computer Graphics Proceedings, Annual Conference Series, pp. 181–188.

[15] GOKTEKIN, T. G., BARGTEIL, A. W., AND O'BRIEN, J. F. A method for animating viscoelastic fluids. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 463–468.

[16] HIRANI, A. *Discrete Exterior Calculus*. PhD thesis, California Institute of Technology, 2003.

[17] INTERRANTE, V., AND GROSCH, C. Strategies for effectively visualizing 3d flow with volume lic. In *VIS '97: Proceedings of the 8th conference on Visualization '97* (Los Alamitos, CA, USA, 1997), IEEE Computer Society Press, pp. 421–ff.

[18] JOY, K. I., LEGAKIS, J., AND MACCRACKEN, R. *Data Structures for Multiresolution Representation of Unstructured Meshes*. Springer-Verlag, Heidelberg, Germany, 2002.

[19] KANE, C., MARSDEN, J. E., ORTIZ, M., AND WEST, M. Variational integrators and the newmark algorithm for conservative and dissipative mechanical systems. *Internat. J. Numer. Methods Engrg. 49* (2000), 1295–1325.

[20] LANGTANGEN, H.-P., MARDAL, K.-A., AND WINTER, R. Numerical methods for incompressible viscous flow. *Advances in Water Resources 25*, 8-12 (Aug-Dec 2002), 1125–1146.

[21] LEW, A., MARSDEN, J. E., ORTIZ, M., AND WEST, M. Asynchronous variational integrators. *Arch. Rational Mech. Anal. 167* (2003), 85–146.

[22] LOSASSO, F., GIBOU, F., AND FEDKIW, R. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 457–462.

[23] MARSDEN, J. E., AND WENSTEIN, A. Coadjoint orbits, vortices and Clebsch variables for incompressible fluids. *Physica D 7* (1983), 305–323.

[24] MARSDEN, J. E., AND WEST, M. Discrete mechanics and variational integrators. *Acta Numerica* (2001), 357–515.

[25] MCNAMARA, A., TREUILLE, A., POPOVIC, Z., AND STAM, J. Fluid control using the adjoint method. *ACM Transactions on Graphics 23*, 3 (Aug. 2004), 449–456.

[26] PIGHIN, F., COHEN, J. M., AND SHAH, M. Modeling and Editing Flows using Advected Radial Basis Functions. In *ACM SIGGRAPH/Eurographics Symposium on Computer Animation* (Aug. 2004), pp. 223–232.

[27] SHI, L., AND YU, Y. Inviscid and Incompressible Fluid Simulation on Triangle Meshes. *Journal of Computer Animation and Virtual Worlds 15*, 3-4 (June 2004), 173–181.

[28] SHIELS, D., AND LEONARD, A. Investigation of a drag reduction on a circular cylinder in rotary oscillation. *Journal of Fluid Mechanics 431* (Mar. 2001), 297–322.

[29] STAM, J. Stable fluids. In *Proceedings of ACM SIGGRAPH* (Aug. 1999), Computer Graphics Proceedings, Annual Conference Series, pp. 121–128.

[30] STAM, J. A simple fluid solver based on the fft. *Journal of Graphics Tools 6*, 2 (2001), 43–52.

[31] STAM, J. Flows on surfaces of arbitrary topology. *ACM Transactions on Graphics 22*, 3 (July 2003), 724–731.

[32] STEINHOFF, J., AND UNDERHILL, D. Modification of the euler equations for *Vorticity Confinement*: Applications to the computation of interacting vortex rings. *Physics of Fluids 6*, 8 (Aug. 1994), 2738–2744.

[33] TONG, Y., LOMBEYDA, S., HIRANI, A. N., AND DESBRUN, M. Discrete multiscale vector field decomposition. *ACM Trans. Graph. 22*, 3 (2003), 445–452.

[34] TREUILLE, A., MCNAMARA, A., POPOVIĆ, Z., AND STAM, J. Keyframe control of smoke simulations. *ACM Transactions on Graphics 22*, 3 (July 2003), 716–723.

[35] VAN WIJK, J. J. Implicit stream surfaces. In *VIS '93: Proceedings of the 4th conference on Visualization '93* (1993), pp. 245–252.

[36] WARREN, J., SCHAEFER, S., HIRANI, A., AND DESBRUN, M. Barycentric coordinates for convex sets, 2004. Preprint.

[37] YAEGER, L., UPSON, C., AND MYERS, R. Combining physical and visual simulation - creation of the planet jupiter for the film 2010. *Computer Graphics (Proceedings of SIGGRAPH 86) 20*, 4 (1986), 85–93.

[38] ZÖCKLER, M., STALLING, D., AND HEGE, H.-C. Interactive visualization of 3d-vector fields using illuminated stream lines. In *VIS '96: Proceedings of the 7th conference on Visualization '96* (Los Alamitos, CA, USA, 1996), IEEE Computer Society Press, pp. 107–ff.