FLUID SIMULATION
SIGGRAPH 2007 Course Notes

Robert Bridson[1]
University of British Columbia

Computer Science Department
201-2366 Main Mall
Vancouver, V6T 1Z4, Canada

Matthias Müller-Fischer[2]
AGEIA Inc.

AGEIA PhysX
Technoparkstrasse 1
8005 Zurich, Switzerland

August 10, 2007

[1]rbridson@cs.ubc.ca
[2]mmueller@ageia.com

Course Description

Animating fluids like water, smoke, and fire by physics-based simulation is increasingly important in visual effects and is starting to make an impact in real-time games. This course goes from the basics of 3D fluid flow to the state of the art in graphics. We will begin with an intuitive explanation of the important concepts in fluid simulation, and as we progress demonstrate how to implement an effective smoke and water simulation system, complete with irregular curved boundaries and surface tension. The last half of the course will cover advanced topics such as fire and explosions, adaptive grid methods, real-time-capable algorithms together with the latest technology in hardware acceleration, and non-Newtonian fluids like sand. Intuition and implementation details will be underscored throughout.

Level of Difficulty: Advanced.

Intended Audience

Developers in industry who want to implement or at least understand the state of the art in graphics fluid simulations, and researchers new to the field.

Prerequisites

Familiarity with differential equations at the graphics level, Newtonian physics, basic numerical methods. You should know what a finite difference is and what $F = ma$, work, and energy mean.

For your reference, in appendix A there is a quick review of some of this prequisite material.

On the Web

For these notes and other updates or resources, see the companion web-site,
`http://www.cs.ubc.ca/~rbridson/fluidsimulation`

Approximate Timeline

Wednesday 1:45pm–5:30pm.

1:45 Bridson: The basics of fluid flow (105 minutes)

- 1:45 Welcome and overview (5 minutes)
- 1:50 The equations of fluids (20 minutes)
- 2:10 Advection algorithms (20 minutes)
- 2:30 Making fluids incompressible (15 minutes)
- 2:45 Advanced boundary conditions (10 minutes)
- 2:55 Turning it into smoke (10 minutes)
- 3:05 Turning it into liquid (25 minutes)

3:30 (15 minute break)

3:45 Müller-Fischer: Real-time fluids (60 minutes)

- 3:45 Reduced dimension models (30 minutes)
- 4:15 3D Smoothed Particle Hydrodynamics (30 minutes)

4:45 Bridson: Advanced topics (45 minutes)

- 4:45 Coupling fluids and solids (25 minutes)
- 5:10 Advances in surface tracking (20 minutes)

Presenters

Robert Bridson

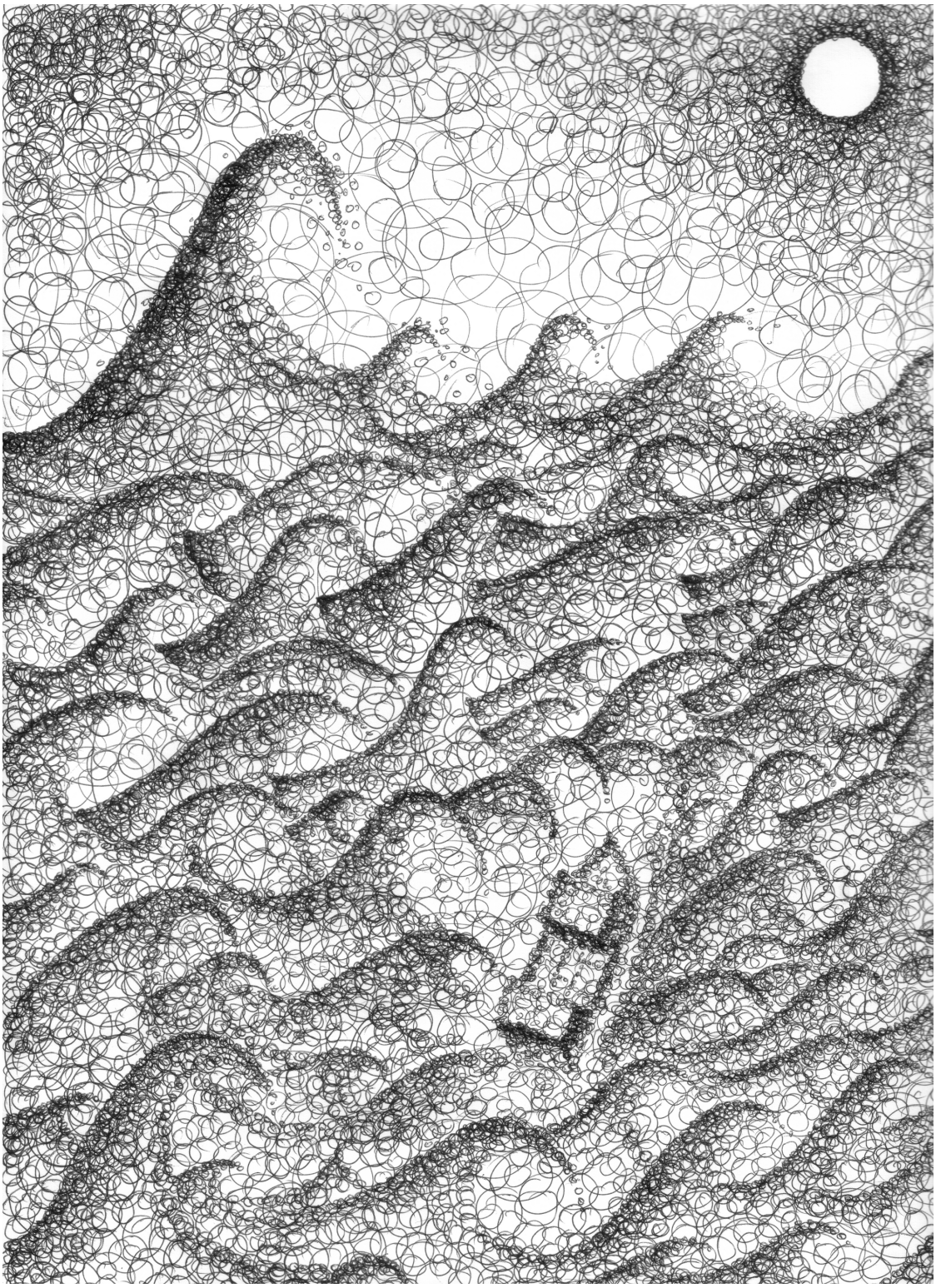
Professor of Computer Science, University of British Columbia.

Bridson received his Ph.D. in Scientific Computing and Computational Mathematics in 2003 at Stanford University before arriving at UBC. He has published several computer graphics papers on physics-based animation, from fluids to cloth, as well as a number of scientific computing papers. He codeveloped physics-based animation software for feature film visual effects in use at Industrial Light + Magic and elsewhere, consults at a variety of visual effects studios, and serves on the software technical advisory board for AGEIA Inc.

Matthias Müller-Fischer

Senior Software Engineer and Researcher, AGEIA Inc.

Müller-Fischer received his Ph.D. on atomistic simulation of dense polymer systems in 1999 from ETH Zurich. During his post-doc with the MIT Computer Graphics Group 1999-2001 he changed fields to macroscopic physically-based simulations. He has published papers on particle-based water simulation, Finite Element-based soft bodies, deformable object collision detection and fracture simulation. In 2002 he co-founded NovodeX, which develops a physics simulation library for gaming. In 2004, when NovodeX became a subsidiary of AGEIA, he began porting the library to the world's first Physics Processing Unit developed by AGEIA. He currently works on real-time cloth simulation and fluid rendering.



The Foamy Brine, Robert Bridson. An artistic rendition of particle-based fluid simulation.

Contents

Preface

These course notes are designed to give you a practical introduction to fluid simulation for graphics. The field of fluid dynamics, even just in animation, is vast and so not every topic will be covered. The focus of these notes is animating fully three-dimensional incompressible flow, from understanding the math and the algorithms to actual implementation. However, we will include a small amount of material on heightfield simplifications which are important for real-time animation.

In general we follow Einstein's dictum, "everything should be made as simple as possible, but not simpler." Constructing a fluid solver for computer animation is not the easiest thing in the world—there end up being a lot of little details that need attention—but is perhaps easier than it may appear from surveying the literature. We will also provide pointers to some more advanced topics here and there.

Part I

The Basics

Chapter 1

The Equations of Fluids

The fluid flow animators are interested in is governed by the famous incompressible Navier-Stokes equations, a set of partial different equations that are supposed to hold throughout the fluid. They are usually written as:

$$\frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} + \frac{1}{\rho} \nabla p = \vec{g} + \nu \nabla \cdot \nabla \vec{u} \tag{1.1}$$

$$\nabla \cdot \vec{u} = 0 \tag{1.2}$$

Pretty complicated at first glance! We'll soon break them down into easy-to-understand parts (and in appendix B provide a more rigorous explanation), but first let's define what each of the symbols is.

1.1 Symbols

The letter $\vec{u}$ is traditionally used in fluid mechanics for the velocity of the fluid. Why not $\vec{v}$? It's hard to say, but it fits another useful convention, to call the three components of 3D velocity (u, v, w).

The Greek letter ρ stands for the density of the fluid. For water, this is roughly $1000 kg/m^3$ and for air this is roughly $1.3 kg/m^3$, a ratio of about 700:1.

The letter p stands for **"pressure"**, the force per unit area that the fluid exerts on anything.

The letter $\vec{g}$ is the familiar acceleration due to gravity, usually $(0, -9.81, 0) m/s^2$. Now is a good time to mention that in these notes we'll take as a convention that the y-axis is pointing vertically upwards, and the x- and z-axes are horizontal. We should add that in animation, additional control accelerations (to make the fluid behave in some desired way) might be added on top of gravity—we'll lump all of these into the one symbol $\vec{g}$. More generally people call these **"body forces"**, because they are applied throughout the whole body of fluid, not just on the surfaces.

The Greek letter ν is technically called the **"kinematic viscosity"**. It measures how viscous the fluid is. Fluids like molasses have high viscosity, and fluids like alcohol have low viscosity: it measures how much the fluid resists deforming while it flows (or more intuitively, how difficult it is to stir).

1.2 The Momentum Equation

The first differential equation (1.1), which is actually three in one wrapped up as a vector equation, is called the **"momentum equation"**. This really is good old Newton's equation $\vec{F} = m\vec{a}$ in disguise. It tells us how the fluid accelerates due to the forces acting on it. We'll try to break this down before moving onto the second differential equation (1.2), which is called the **"incompressibility condition"**.

Let's first imagine we were simulating a fluid using a particle system (later in the course we will actually use this as a practical method, but for now let's just use it as a thought experiment). Each particle would represent a little blob of fluid. It would have a mass m, a volume V, and a velocity $\vec{u}$. To integrate the system forward in time all we would need is to figure out what the forces acting on each particle are: $\vec{F} = m\vec{a}$ then tells us how the particle accelerates, from which we get its motion. We'll write the acceleration of the particle in slightly odd notation (which we'll later relate to the momentum equation above):

$$\vec{a} \equiv \frac{D\vec{u}}{Dt} \tag{1.3}$$

The big D derivative notation is called the material derivative. Newton's law is now:

$$m\frac{D\vec{u}}{Dt} = \vec{F} \tag{1.4}$$

So what are the forces acting on the particle? The simplest is of course gravity: $m\vec{g}$. The rest of the fluid (the other particles) also exerts forces though.

The first of the fluid forces is pressure. High pressure regions push on lower pressure regions. Note that what we really care about is the net force on the particle, though. For example, if the pressure is equal in every direction there's going to be net force of zero. What really matters is the imbalance of higher pressure on one side of the particle than the other, resulting in a force pointing away from the high pressure and towards the low pressure. In the appendices we show how to rigorously derive this, but for now let's just point out that the simplest way to measure the imbalance in pressure at the position of the particle is simply to take the negative gradient of pressure: $-\nabla p$. We'll need to integrate this over the volume of our blob of fluid to get the pressure force. As a simple approximation, we'll just multiply by the volume V. You might be asking yourself, but what is the pressure? We'll skip over this until later, when we talk about incompressibility, but for now you can think of it being whatever it takes to keep the fluid at constant volume.

The other fluid force is due to viscosity. A viscous fluid tries to resist deforming. In the appendices again we will rigorously derive this, but for now let's intuitively develop this as a force that tries to make our particle move at the average velocity of the nearby particles, that tries to minimize differences in velocity between nearby bits of fluid. You may remember from image processing, digital geometry processing, the physics of diffusion or heat dissipation, or many other domains, that the differential operator which measures how far a quantity is from the average around it is the Laplacian $\nabla \cdot \nabla$. (Now is a good time to mention that there is a quick review of vector calculus in the appendices, including differential operators like the Laplacian.) This will provide our viscous force then, once we've integrated it over the volume of the blob. We'll use the **"dynamic viscosity coefficient"** (dynamic means we're getting a **force** out of it; the kinematic viscosity from before is used to get an **acceleration** instead), which is denoted with the Greek letter μ. I'll note here that for fluids with variable viscosity, this term ends up being a little more complicated, but won't be handled in these notes.

Putting it all together, here's how a blob of fluid moves:

$$m\frac{D\vec{u}}{Dt} = m\vec{g} - V\nabla p + V\mu\nabla\cdot\nabla\vec{u} \tag{1.5}$$

Obviously we're making errors when we approximate a fluid with a small finite number of particles. We will take the limit, then, as our number of particles goes to infinity and the size of each blob goes to zero. This poses a problem in our particle equation, because the mass m and volume V of the particle is then going to zero. We can fix this by dividing the equation by the volume, and then take the limit. Remembering m/V is just the fluid density ρ, we get:

$$\rho\frac{D\vec{u}}{Dt} = \rho\vec{g} - \nabla p + \mu\nabla\cdot\nabla\vec{u} \tag{1.6}$$

Looking familiar? We'll divide by the density and rearrange the terms a bit to get

$$\frac{D\vec{u}}{Dt} + \frac{1}{\rho}\nabla p = \vec{g} + \frac{\mu}{\rho}\nabla\cdot\nabla\vec{u} \tag{1.7}$$

To simplify things even a little more we'll define the kinematic viscosity as $\nu = \mu/\rho$ to get

$$\frac{D\vec{u}}{Dt} + \frac{1}{\rho}\nabla p = \vec{g} + \nu\nabla\cdot\nabla\vec{u} \tag{1.8}$$

We've almost made it to the momentum equation! In fact this form, using the material derivative D/Dt, is actually more important to us in computer graphics and will guide us how to numerically solve the equation. But we still will want to understand what the material derivative is, and how it relates back to the traditional form of the momentum equation. For that, we'll need to understand the difference between the **"Lagrangian"** and **"Eulerian"** viewpoints.

1.3 Lagrangian and Eulerian Viewpoints

When we think about a continuum (like a fluid or a deformable solid) moving, there are two approaches to tracking this motion, the Lagrangian viewpoint and the Eulerian viewpoint.

The Lagrangian approach (named after the French mathematician Lagrange) is what you're probably most familiar with. It treats the continuum just like a particle system. Each point in the fluid or solid is labeled as a separate particle, with a position $\vec{x}$ and a velocity $\vec{u}$. You could even think of each particle as being one molecule of the fluid. Nothing too special here! Solids are almost always simulated in a Lagrangian way, with a discrete set of particles usually connected up in a mesh.

The Eulerian approach (named after the Swiss mathematician Euler) takes a different tactic, that's usually used for fluids. Instead of tracking each particle, we instead look at fixed points in space and see how the fluid quantities (such as density, velocity, temperature, etc.) measured at those points change in time. The fluid is probably flowing past those points, contributing one sort of change (e.g. as a warm fluid moves past followed by a cold fluid, the temperature at the fixed point in space will go down—even though the temperature of any individual particle in the fluid is not changing!). In addition the fluid variables can be changing in the fluid, contributing the other sort of change that might be measured at a fixed point (e.g. the temperature at a fixed point in space might decrease as the fluid everywhere cools off).

One way to think of the two viewpoints is in doing a weather report. In the Lagrangian viewpoint you're in a balloon floating along with the wind, measuring the pressure and temperature and humidity etc. of the air that's flowing alongside you. In the Eulerian viewpoint you're stuck on the ground, measuring the pressure and temperature and humidity etc. of the air that's flowing past.

Numerically, the Lagrangian viewpoint corresponds to a particle system (with or without a mesh connecting up the particles) and the Eulerian viewpoints corresponds to using a fixed grid that doesn't change in space even as the fluid flows through it.

It might seem the Eulerian approach is unnecessarily complicated: why not just stick with Lagrangian particle systems? Indeed, later on in chapter 9 we will show how to simulate fluids exactly as a particle system. But the rest of these notes will stick to the Eulerian approach for a number of reasons:

- It's easier analytically to work with the spatial derivatives like the pressure gradient and viscosity in the Eulerian viewpoint
- It's much easier numerically to approximate those spatial derivatives on a fixed Eulerian mesh than on a cloud of arbitrarily moving particles

The key to connecting the two viewpoints is the material derivative. We'll start with a Lagrangian description: there are particles with positions $\vec{x}$ and velocities $\vec{u}$. Let's look at a generic quantity we'll call q: each particle has a value for q. (q might be density, or velocity, or temperature, or many other things.) In particular, the function $q(t, \vec{x})$ tells us the value of q at time t for the particle that happens to be at position $\vec{x}$ then: this is an Eulerian variable. So how fast is q changing for that particle (the Lagrangian question)? Take the total derivative (a.k.a. the Chain Rule):

$$\frac{d}{dt}q(t,\vec{x}) = \frac{\partial q}{\partial t} + \nabla q \cdot \frac{d\vec{x}}{dt} \tag{1.9}$$

$$= \frac{\partial q}{\partial t} + \nabla q \cdot \vec{u} \tag{1.10}$$

$$\equiv \frac{Dq}{Dt} \tag{1.11}$$

This is the material derivative!

Let's review the two terms that go into the material derivative. The first is $\partial q/\partial t$, which is just how fast q is changing at that fixed point in space (an Eulerian measurement). The second term, $\nabla q \cdot \vec{u}$, is correcting for how much of that change is due just to differences in the fluid flowing past (e.g. the temperature changing because hot air is being replaced by cold air, not because the temperature of any molecule is changing).

Just for completeness, let's write out the material derivative in full, with all the partial derivatives:

$$\frac{Dq}{Dt} = \frac{\partial q}{\partial t} + u\frac{\partial q}{\partial x} + v\frac{\partial q}{\partial y} + w\frac{\partial q}{\partial z} \tag{1.12}$$

Obviously in 2D, we can just get rid of the w and z term.

Note that I keep talking about how the quantity, or molecules, or particles, move with the velocity field $\vec{u}$. This is called **"advection"** (or sometimes **"convection"** or **"transport"**; they all mean the same thing). An **"advection**

equation" is just one that uses the material derivative, at its simplest setting it to zero:

$$\frac{Dq}{Dt} = 0 \tag{1.13}$$

$$\text{i.e.} \quad \frac{\partial q}{\partial t} + \vec{u} \cdot \nabla q = 0 \tag{1.14}$$

This just means the quantity is moving around, but isn't changing in the Lagrangian viewpoint.

1.3.1 An Example

Hopefully to lay the issue to rest, let's work through an example in one dimension. Instead of q we'll use T for temperature. We'll say that at one instant in time, the temperature profile is

$$T(x) = 10x \tag{1.15}$$

That is, it's freezing at the origin (if we're using Celsius!) and gets warmer as we look further to the right, to a temperature of 100 at $x = 10$. Now let's say there's a steady wind of speed c blowing, i.e. the fluid velocity is c everywhere:

$$\vec{u} = c \tag{1.16}$$

We'll assume that the temperature of each particle of air isn't changing, they're just moving. So the material derivative, measuring things in the Lagrangian viewpoint, says the change is zero:

$$\frac{DT}{Dt} = 0 \tag{1.17}$$

If we expand this out we have:

$$\frac{\partial T}{\partial t} + \nabla T \cdot \vec{u} = 0 \tag{1.18}$$

$$\frac{\partial T}{\partial t} + 10 \cdot c = 0 \tag{1.19}$$

$$\Rightarrow \qquad \frac{\partial T}{\partial t} = -10c \tag{1.20}$$

$$\tag{1.21}$$

That is, at a fixed point in space, the temperature is changing at a rate $-10c$. If the wind has stopped, $c = 0$, nothing changes. If the wind is blowing to the right at speed $c = 1$, the temperature at a fixed point will drop at rate -10. If the wind is blowing faster to the left at speed $c = -2$, the temperature at a fixed point will increase at rate 20. So even though the Lagrangian derivative is zero, in this case the Eulerian derivative can be anything depending on how fast and in what direction the flow is moving.

1.3.2 Advecting Vector Quantities

One point of common confusion is what the material derivative means when applied to vector quantities, like RGB colors, or most confusing of all, the velocity field $\vec{u}$ itself. The simple answer is: treat each component separately.

Let's write out for the color vector $\vec{C} = (R, G, B)$ what the material derivative is:

$$\frac{D\vec{C}}{Dt} = \begin{bmatrix} DR/Dt \\ DG/Dt \\ DB/Dt \end{bmatrix} = \begin{bmatrix} \partial R/\partial t + \vec{u} \cdot \nabla R \\ \partial G/\partial t + \vec{u} \cdot \nabla G \\ \partial B/\partial t + \vec{u} \cdot \nabla B \end{bmatrix} = \frac{\partial \vec{C}}{\partial t} + \vec{u} \cdot \nabla \vec{C} \tag{1.22}$$

So although the notation $\vec{u} \cdot \nabla \vec{C}$ might not strictly make sense (is the gradient of a vector a matrix? what is the dot-product of a vector with a matrix?[1]) it's not hard to figure out if we split up the vector into scalar components.

Let's do the same thing for velocity itself, which really is no different except $\vec{u}$ appears in two places, as the velocity field that the fluid is moving in and as the fluid quantity that is getting advected. People sometimes call this **"self-advection"** as if this was some terribly complicated thing: it isn't, anymore than particles possessing a velocity variable and moving at that same velocity is complicated. So just by copying and pasting, here is the advection of velocity $\vec{u} = (u, v, w)$ spelled out:

$$\frac{D\vec{u}}{Dt} = \begin{bmatrix} Du/Dt \\ Dv/Dt \\ Dw/Dt \end{bmatrix} = \begin{bmatrix} \partial u/\partial t + \vec{u} \cdot \nabla u \\ \partial v/\partial t + \vec{u} \cdot \nabla v \\ \partial w/\partial t + \vec{u} \cdot \nabla w \end{bmatrix} = \frac{\partial \vec{u}}{\partial t} + \vec{u} \cdot \nabla \vec{u} \tag{1.23}$$

or if you want to get right down to the nuts and bolts of partial derivatives:

$$\frac{D\vec{u}}{Dt} = \begin{bmatrix} \frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} \\ \frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} \\ \frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} \end{bmatrix} \tag{1.24}$$

If you want to go even further, advecting matrix quantities around, it's no different: just treat each component separately.

1.4 Incompressibility

Real fluids, even liquids like water, change their volume. In fact, that's just what sound waves are: perturbations in the volume (and thus density and pressure) of the fluid. You may once have been taught that the difference between liquids and gases was that gases change their volume but liquids don't, but that's not really true: otherwise you wouldn't be able to hear underwater!

However, the crucial thing is that usually fluids don't change their volume very much. It's next to impossible, even with an incredibly powerful pump, to change the volume of water much at all. Even air won't change its volume much unless you stick it in a pump, or are dealing with really extreme situations like sonic booms and blast waves. The study of how fluids behave in these situations is generally called **"compressible flow"**. It's complicated and expensive to simulate, and apart from acoustics doesn't enter much into everday life. And even sound waves are such tiny perturbations in the volume, and have so small of an effect on how fluids move at a macroscopic level (water sloshing, smoke billowing, etc.), that they're practically irrelevant for animation.

[1] With slightly more sophisticated tensor notation, this can be put on a firm footing, but traditionally people stick with the dot-product.

What this means is that in animation we can treat all fluids as **"incompressible"**, which means their volume doesn't change.[2] What does this mean mathematically? There's a more rigorous explanation in appendix B again, but we can sketch out a quick argument now.

Pick an arbitrary chunk of fluid to look at at some instant in time. We'll call this volume Ω, and its boundary surface $\partial\Omega$ (these are the traditional symbols). We can measure how fast the volume of this chunk of fluid is changing by integrating the normal component of its velocity around the boundary:

$$\frac{d}{dt}\text{Volume}(\Omega) = \iint_{\partial\Omega} \vec{u}\cdot\hat{n} \tag{1.25}$$

For an incompressible fluid, the volume had better say constant, i.e. this rate of change is zero:

$$\iint_{\partial\Omega} \vec{u}\cdot\hat{n} = 0 \tag{1.26}$$

Now we can use the Divergence Theorem to change this to a volume integral. Basically, this is a multi-dimensional version of the Fundamental Theorem of Calculus: if you integrate the derivative of a function, you get the original evaluated at the bounds of your integration (see appendix A for a review if you need to brush up on your vector calculus). In this case, we get:

$$\iiint_{\Omega} \nabla\cdot\vec{u} = 0 \tag{1.27}$$

Now, here's the magical part: this equation should be true for any choice of Ω (any region of fluid). The only function that integrates to zero independent of the volume of integration is zero itself. Thus the integrand has to be zero everywhere:

$$\nabla\cdot\vec{u} = 0 \tag{1.28}$$

This is the **incompressibility condition**, the other part of the incompressible Navier-Stokes equations.

A vector-field that satisfies the incompressiblity condition is called **"divergence-free"** for obvious reasons. One of the tricky parts of simulating incompressible fluids is making sure that the velocity field stays divergence-free. This is where the pressure comes in.

The way to think about pressure is that it's whatever it takes to keep the velocity divergence-free. If you're familiar with constrained dynamics, you can think of the incompressibility condition as a constraint, and the pressure field as the Lagrange multiplier needed to satisfy that constraint subject to the principle of zero virtual work. If you're not, don't worry. Let's derive exactly what the pressure has to be.

The pressure only shows up in the momentum equation, and we want to somehow relate it to the divergence of the velocity. Therefore, let's take the divergence of both sides of the momentum equation:

$$\nabla\cdot\frac{\partial\vec{u}}{\partial t} + \nabla\cdot(\vec{u}\cdot\nabla\vec{u}) + \nabla\cdot\frac{1}{\rho}\nabla p = \nabla\cdot(\vec{g} + \nu\nabla\cdot\nabla\vec{u}) \tag{1.29}$$

We can change the order of differentation in the first term, to being the time derivative of divergence:

$$\frac{\partial}{\partial t}\nabla\cdot\vec{u} \tag{1.30}$$

[2] Even if we have sonic booms and blast waves, they're basically invisible and most audiences have no idea really how they behave, so it's generally a much better idea to hack together something that looks cool than try to simulate them accurately.

If the incompressibility condition always holds, this had better be zero. Subsequently rearranging equation (1.29) gives us an equation for pressure:

$$\nabla \cdot \frac{1}{\rho} \nabla p = \nabla \cdot (-\vec{u} \cdot \nabla \vec{u} + \vec{g} + \nu \nabla \cdot \nabla \vec{u}) \tag{1.31}$$

This isn't exactly relevant for our numerical simulation, but it's worth seeing because we'll go through almost exactly the same steps, from looking at how fast a volume is changing to an equation for pressure, when we discretize.

1.5 Dropping Viscosity

In some situations, viscosity forces are extremely important: e.g. simulating honey or tiny water droplets. But in most other cases that we wish to animate, viscosity plays a minor role, and thus we often drop it: the simpler the equations are, the better. In fact, most numerical methods for simulating fluids unavoidably introduce errors which can be physically reinterpreted as viscosity—so even if we drop viscosity in the equations, we will still get something that looks like it (and in fact, one of the big challenges in computational fluid dynamics is avoiding this viscous error as much as possible). Thus for the rest of these notes we will assume viscosity has been dropped.

The Navier-Stokes equations without viscosity are called the **"Euler equations"**, and such an ideal fluid with no viscosity is called **"inviscid"**. Just to make it clear what has been dropped, here are the incompressible Euler equations, using the material derivative to emphasize the simplicity:

$$\frac{D\vec{u}}{Dt} + \frac{1}{\rho} \nabla p = \vec{g} \tag{1.32}$$

$$\nabla \cdot \vec{u} = 0 \tag{1.33}$$

It is these equations that we'll mostly be using.

1.6 Boundary Conditions

Most of the, ahem, "fun" in numerically simulating fluids is in getting the boundary conditions correct. So far we've only talked about what's happening in the interior of the fluid: so what goes on at the boundary?

In these notes we will only focus on two boundary conditions, **"solid walls"** and **"free surfaces"**. One important case we won't cover is the boundary between two different fluids: most often this isn't needed in animation, but if you are interested, see papers such as [HK05].

A solid wall boundary is where the fluid is in contact with, you guessed it, a solid. It's simplest to phrase this in terms of velocity: the fluid better not be flowing into the solid or out of it, thus the normal component of velocity has to be zero:

$$\vec{u} \cdot \hat{n} = 0 \tag{1.34}$$

That is, if the solid isn't moving itself. In general, we need the normal component of the fluid velocity to match the normal component of the solid's velocity:

$$\vec{u} \cdot \hat{n} = \vec{u}_{\text{solid}} \cdot \hat{n} \tag{1.35}$$

In both these equations, $\hat{n}$ is of course the normal to the solid boundary. This is sometimes called the "no-stick" condition, since we're only restricting the normal component of velocity, allowing the fluid to freely slip past in the tangential direction.

So that's what the velocity does; how about the pressure at a solid wall? We again go back to the idea that pressure is "whatever it takes to make the fluid incompressible." We'll add to that, "and enforce the solid wall boundary conditions." The $\nabla p/\rho$ term in the momentum equation applies even on the boundary, so for the pressure to control $\vec{u} \cdot \hat{n}$ at a solid wall, obviously that's saying something about $\nabla p \cdot \hat{n}$, otherwise known as the normal derivative of pressure: $\partial p/\partial \hat{n}$. We'll wait until we get into numerically handling the boundary conditions before we get more precise.

That's all there is to a solid wall boundary for an **inviscid** fluid. If we do have viscosity, life gets a little more complicated. In that case, the stickiness of the solid might have an influence on the tangential component of the fluid's velocity. The simplest case is the **"no slip"** boundary condition, where we simply say

$$\vec{u} = 0 \tag{1.36}$$

or if the solid is moving,

$$\vec{u} = \vec{u}_{\text{solid}} \tag{1.37}$$

Again, we'll avoid a discussion of exact details until we get into numerical implementation.

As a side note, sometimes the solid wall actually is a vent or a drain that fluid **can** move through: in that case, we obviously want $\vec{u} \cdot \hat{n}$ to be different from the wall velocity, but instead be the velocity at which fluid is being pumped in or out of the simulation at that point.

The other boundary condition that we're interested in is the free surface. This is where we stop modeling the fluid. For example, if we simulate water splashing around, then the water surfaces that are **not** in contact with a solid wall are free surfaces. In this case there really is another fluid, air, but we may not want the hassle of simulating the air as well: and since air is 700 times lighter than water, it's not able to have that big of an effect on the water anyhow. So instead we make the modeling simplification that the air can simply be represented as a region with atmospheric pressure. In actual fact, since only **differences** in pressure matter (in incompressible flow), we can set the air pressure to be any arbitrary constant: zero is the most convenient. Thus a free surface is one where $p = 0$, and we don't control the velocity in any particular way.

The other case in which free surfaces arise is where we are trying to simulate a bit of fluid that is part of a much larger domain: for example, simulating smoke in the open air. We obviously can't afford to simulate the entire atmosphere of the Earth, so we will just make a grid that covers the region we expect to be "interesting." At the boundaries of that region the fluid continues, but we're not simulating it; we allow fluid to enter and exit the region as it wants, so it's natural to consider this a free surface, $p = 0$, even though there's no visible surface.

One final note on free surfaces: for smaller scale liquids, surface tension can be very important. At the underlying molecular level, surface tension exists because of varying strengths of attraction between molecules of different types. For example, water molecules are more strongly attracted to other water molecules than they are to air molecules: therefore the water molecules at the surface separating water and air try to move to be as surrounded by water as much as possible. From a geometric perspective, physical chemists have modeled this as a force that tries to minimize the surface area, or equivalently tries to reduce the mean curvature of the surface. You can interpret the first idea (minimizing surface area) as a tension which constantly tries to shrink the surface, hence the name surface tension. We will instead use the second idea, which is based on the mean curvature of the surface. (Later, in chapter

6 we'll talk about how to actually measure mean curvature and exactly what it means.) In short, the model is that there is actually a jump in pressure between the two fluids, proportional to the mean curvature:

$$[p] = \gamma\kappa \tag{1.38}$$

The $[p]$ notation means the jump in pressure, i.e. the difference in pressure measured on the water side and measured on the air side, γ is the surface tension coefficient which you can look up (for water and air at room temperature it is approximately $\gamma \approx 0.073 N/m$), and κ is the mean curvature, measured in $1/m$. What this means for a free surface with surface tension is that the pressure at the surface of the water is the air pressure (which we assume to be zero) plus the pressure jump:

$$p = \gamma\kappa \tag{1.39}$$

Free surfaces do have one major problem: air bubbles immediately collapse. While air is much lighter than water, and so might not be able to transfer much momentum to water usually, it is still incompressible. An air bubble inside water keeps its volume. Modeling the air bubble with a free surface will let the bubble collapse and vanish. To handle this kind of thing, you need either hacks based on adding bubble particles to a free surface flow, or more generally a simulation of both air and water (called **"two phase"** flow, because there are two phases or types of fluid involved). Again, we won't get into that in these notes.

Chapter 2

Overview of Numerical Simulation

Now that we know and understand the basic equations, how do we discretize them to numerically simulate fluids on the computer? There are an awful lot of choices for how to do this, and people are continuing to invent new ways. We will mainly focus on one approach that works very well, at least for graphics, but at the end of the notes in chapter 9 we'll cover a particle-based alternative.

2.1 Splitting

The approach works on the basis of something called **"splitting"**: we split up a complicated equation into its component parts and solve each one separately in turn. If we say that the rate of change of one quantity is the sum of several terms, we can numerically update it by computing each term and adding it in one by one.

Let's make that clearer with an incredibly simple "toy" example, a single ordinary differential equation:

$$\frac{dq}{dt} = 1 + 2 \tag{2.1}$$

Well, gee, you already know the answer is $q(t) = 3t + q(0)$, but let's work out a numerical method based on splitting. We'll split it into two steps, each one of which looks like a simple Forward Euler update (if you want to remind yourself what Forward Euler is, refer to appendix A):

$$\tilde{q} = q^n + 1\Delta t \tag{2.2}$$

$$q^{n+1} = \tilde{q} + 2\Delta t \tag{2.3}$$

The notation used here is that q^n is the value of q computed at time step n, and Δt is the amount of time between consecutive time steps.[1] What we have done is split the equation up into two steps: after the first step (2.2) we get an intermediate quantity $\tilde{q}$ which includes the contribution of the first term ($= 1$) but not the second ($= 2$), and then the second step (2.3) goes from the intermediate value to the end by adding in the missing term's contribution. In this example, obviously, we get exactly the right answer, and splitting didn't buy us anything.

[1] In particular, do not get confused with raising q to the power of n or $n+1$: this is an abuse of notation, but is so convenient it's consistently used in fluid simulation. On the rare occasion that we do raise a quantity to some exponent, we'll very clearly state that: otherwise assume the superscript indicates at what time step the quantity is.

Let's upgrade our example to something more interesting:

$$\frac{dq}{dt} = f(q) + g(q) \tag{2.4}$$

Here $f()$ and $g()$ are some blackbox functions representing separate software modules. We could do splitting with Forward Euler again:

$$\tilde{q} = q^n + \Delta t f(q^n) \tag{2.5}$$
$$q^{n+1} = \tilde{q} + \Delta t g(\tilde{q}) \tag{2.6}$$

A simple Taylor series analysis shows that this is still a first order accurate algorithm if you're worried (if you're not, ignore this):

$$q^{n+1} = (q^n + \Delta t f(q^n)) + \Delta t g(q^n + \Delta t f(q^n)) \tag{2.7}$$
$$= q^n + \Delta t f(q^n) + \Delta t \left(g(q^n) + O(\Delta t)\right) \tag{2.8}$$
$$= q^n + \Delta t (f(q^n) + g(q^n)) + O(\Delta t^2) \tag{2.9}$$
$$= q^n + \frac{dq}{dt}\Delta t + O(\Delta t^2) \tag{2.10}$$
$$\tag{2.11}$$

Wait, you say, that hasn't bought you anything beyond what simple old Forward Euler without splitting gives you. Aha! Here's where we get a little more sophisticated. Let's assume that the reason we've split $f()$ and $g()$ into separate software modules is that we have special numerical methods which are really good at solving the simpler equations

$$\frac{dr}{dt} = f(r) \tag{2.12}$$
$$\frac{ds}{dt} = g(s) \tag{2.13}$$

This is precisely the motivation for splitting: we may not be able to easily deal with the complexity of the whole equation, but it's built out of separate terms that we **do** have good methods for. I'll call the special integration algorithms $\mathtt{F}(\Delta t, r)$ and $\mathtt{G}(\Delta t, s)$. Our splitting method is then:

$$\tilde{q} = \mathtt{F}(\Delta t, q^n) \tag{2.14}$$
$$q^{n+1} = \mathtt{G}(\Delta t, \tilde{q}) \tag{2.15}$$

If $\mathtt{F}()$ and $\mathtt{G}()$ were just Forward Euler, then this is exactly the same as equations (2.5, 2.6), but the idea is again that they're something better. If you do the Taylor series analysis, you can show we still have a first order accurate method[2] but I'll leave that as an exercise.

Splitting really is just the principle of divide-and-conquer applied to differential equations: solving the whole problem may be too hard, but you can split it into pieces that are easier to solve and then combine the solutions.

If you're on the ball, you might have thought of a different way of combining the separate parts: instead of sequentially taking the solution from $\mathtt{F}()$ and then plugging it into $\mathtt{G}()$, you could run $\mathtt{F}()$ and $\mathtt{G}()$ in parallel and add their

[2]There are more complicated ways of doing splitting in fluid dynamics which can get higher order accuracy, but for animation we won't need to bother with them.

contributions together. The reason we're **not** going to do this, but will stick to sequentially working through the steps, is that our special algorithms (the integrators `F()` and `G()` in this example) will guarantee special things about their output that are needed as preconditions for the input of other algorithms. Doing it in the **right** sequence will make everything work, but doing it in parallel will mess up those guarantees. We'll talk more about what those guarantees and preconditions are in the next section.

2.2 Splitting the Fluid Equations

We're going to use splitting on the incompressible fluid equations. In particular, we'll separate out the advection part, the body forces (gravity) part, and the pressure/incompressibility part.

That is, we'll work out methods for solving these simpler equations:

$$\frac{Dq}{Dt} = 0 \quad \textit{(advection)} \tag{2.16}$$

$$\frac{\partial \vec{u}}{\partial t} = \vec{g} \quad \textit{(body forces)} \tag{2.17}$$

$$\frac{\partial \vec{u}}{\partial t} + \frac{1}{\rho}\nabla p = 0 \quad \text{s.t.} \quad \nabla \cdot \vec{u} = 0 \quad \textit{(pressure/incompressibility)} \tag{2.18}$$

I used the generic quantity q in the advection equation because we may be interested in advecting other things, not just velocity $\vec{v}$.

Let's call our algorithm for solving the advection equation (2.16) `advect`$(\vec{u}, \Delta t, q)$: it advects quantity q through the velocity field $\vec{u}$ for a time interval Δt. Chapter 3 will cover how to do this.

For the body force equation (2.17) we'll just use Forward Euler.

For the pressure/incompressibility part (2.18) we'll develop an algorithm called `project`$(\Delta t, \vec{u})$ that calculates and applies just the right pressure to make $\vec{u}$ divergence-free, and also enforces the solid wall boundary conditions. Chapter 4 deals with this part (and explains the odd choice of word, "project").

The important precondition/guarantee issue we mentioned in the previous section is that advection should only be done in a divergence-free velocity field. When we move fluid around, if it's going to conserve volume, then the velocity field that we are moving it in must be divergence-free: we covered that already in chapter 1. So we want to make sure we only run `advect()` with the output of `project()`: the sequence of our splitting matters a lot!

Putting it together, here is our basic fluid algorithm:

- Start with an initial divergence-free velocity field $\vec{u}^{(0)}$.
- For time step $n = 0, 1, 2, \ldots$
  - Determine a good time step Δt to go from time t_n to time t_{n+1}
  - Set $\vec{u}^A =$ `advect`$(\vec{u}^n, \Delta t, \vec{u}^n)$
  - Add $\vec{u}^B = \vec{u}^A + \Delta t \vec{g}$
  - Set $\vec{u}^{n+1} =$ `project`$(\Delta t, \vec{u}^B)$

2.3 Time Steps

Determining a good time step size is the first step of the algorithm. Our first concern is that we don't want to go past the duration of the current animation frame: if we pick a Δt and $t_n + \Delta t > t_{frame}$, then we should clamp it to $\Delta t = t_{frame} - t_n$ and set a flag that alerts us to the fact we've hit the end of the frame. (Note that checking if $t_{n+1} = t_{frame}$ is a bad idea, since inexact floating-point arithmatic may mean t_{n+1} isn't exactly equal to t_{frame} even when calculated this way!) At the end of each frame we'll presumably do something special like save the state of the fluid animation to disk, or render it on the screen.

Subject to that clamping, we want to then select a Δt that satisfies any requirements made by the separate steps of the simulation: advection, body forces, etc. We'll discuss these in the chapters that follow. Selecting the minimum of these suggested time steps is safe.

However, in some situations we may have a performance requirement which won't let us take lots of small time steps every frame. If we only have time for, say, three time steps per frame, we had better make sure Δt is at least a third of the frame time. This might be larger than the suggested time step sizes from each step, so we will make sure that all the methods we use can tolerate the use of larger-than-desired time steps—they should generate plausible results in this case, even if they're quantitatively inaccurate.

2.4 Grids

In this numerical section, so far we have only talked about discretizing in time, not in space. While we will go into more detail about this in the subsequent chapters, we'll introduce the basic grid structure here.

In the early days of computational fluid dynamics Harlow and Welch introduced the Marker-and-Cell (MAC) method [HW65] method for solving incompressible flow problems. One of the fundamental innovations of this paper was a new grid structure that (as we will see later) makes for a very effective algorithm.

The so-called **"MAC grid"** is a **"staggered"** grid, i.e. where the different variables are stored at different locations. Let's look at it in two dimensions first, illustrated in figure 2.1. The pressure in grid cell (i, j) is sampled at the centre of the cell, indicated by $p_{i,j}$. The velocity is split into its two Cartesian components. The horizontal u component is sampled at the centers of the vertical cell faces, for example indicated by $u_{i+1/2,j}$ for the horizontal velocity between cells (i, j) and $(i+1, j)$. The vertical v component is sampled at the centers of the horizontal cell faces, for example indicated by $v_{i,j+1/2}$ for the vertical velocity between cells (i, j) and $(i, j+1)$. Note that for grid cell (i, j) we have sampled the **normal** component of the velocity at the center of each of its faces: this will very naturally allow us to estimate the amount of fluid flowing into and out of the cell.

In three dimensions, the MAC grid is set up the same way, with pressure at the grid cell centers and the three different components of velocity split up so that we have the normal component of velocity sampled at the center of each cell face: see figure 2.2.

We'll go into more detail about why we use this staggered arrangement in chapter 4, but briefly put it's so that we can use accurate **"central differences"** for the pressure gradient and for the divergence of the velocity field without the usual disadvantages of the method. Consider just a one-dimension example: estimating the derivative of a quantity q sampled at grid locations $\ldots, q_{i-1}, q_i, q_{i+1}, \ldots$. To estimate $\partial q/\partial x$ at grid point i without any bias, the natural

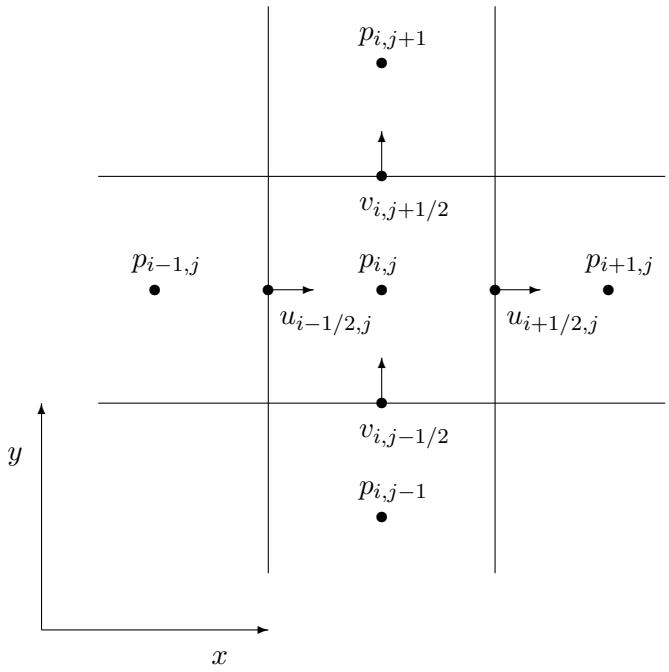


Figure 2.1: The two-dimensional MAC grid.

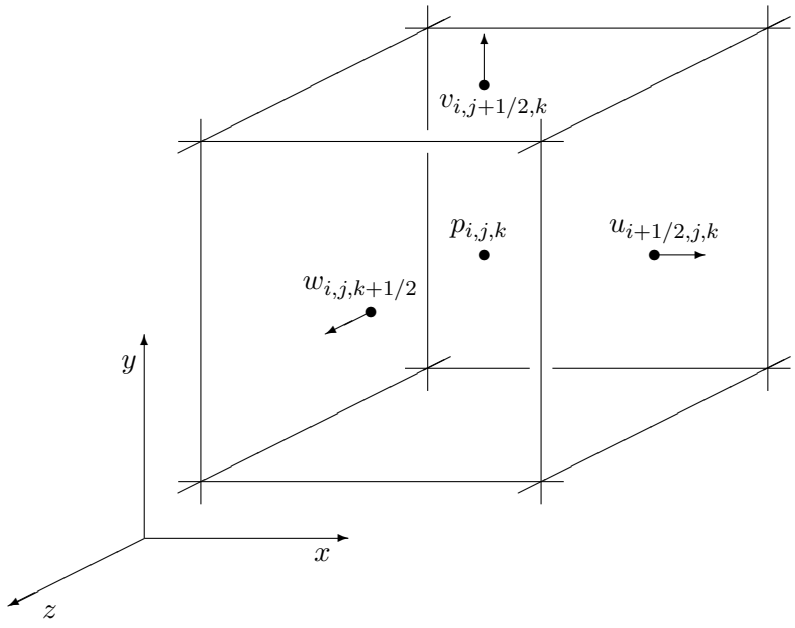


Figure 2.2: One cell from the three-dimensional MAC grid.

formula is the first central difference:

$$\left(\frac{\partial q}{\partial x}\right)_i \approx \frac{q_{i+1} - q_{i-1}}{2\Delta x} \tag{2.19}$$

This is unbiased and accurate to $O(\Delta x^2)$, as opposed to a forward or backward difference, such as:

$$\left(\frac{\partial q}{\partial x}\right)_i \approx \frac{q_{i+1} - q_i}{\Delta x} \tag{2.20}$$

which is biased to the right and only accurate to $O(\Delta x)$. However, formula (2.19) has a major problem in that the derivative estimate at grid point i completely ignores the value q_i sampled there! To see why this is so terrible, recall that a constant function can be defined as one whose first derivative is zero. If we require that the finite difference (2.19) is zero, we are allowing q's that aren't necessarily constant—q_i could be quite different from q_{i-1} and q_{i+1} and still the central difference will report that the derivative is zero as long as $q_{i-1} = q_{i+1}$. In fact, a very jagged function like $q_i = (-1)^i$ which is far from constant will register as having zero derivative according to formula (2.19). On the other hand, only truly constant functions satisfy the forward difference (2.20) equal to zero. The problem with formula (2.19) is technically known as having a nontrivial **"null-space"**: the set of functions where the formula evaluates to zero contains more than just the constant functions it should be restricted to.

How can we get the unbiased second-order accuracy of a central difference without this null-space problem? The answer is using a staggered grid: sample the q's at the half-way points, $q_{i+1/2}$ instead. Then we naturally can estimate the derivative at grid point i as:

$$\left(\frac{\partial q}{\partial x}\right)_i \approx \frac{q_{i+1/2} - q_{i-1/2}}{\Delta x} \tag{2.21}$$

This is unbiased and accurate to $O(\Delta x^2)$, but it doesn't skip over any values of q like formula (2.19), and so if we set this equal to zero we can only have q constant: the null-space is correct. The MAC grid is set up so that we use this staggered form of the central difference wherever we need to estimate a derivative in the pressure solve (i.e. the incompressibility condition).

The staggered MAC grid is perfectly suited for handling pressure and incompressibility, but it's a little awkward for other uses. For example, if we actually want to evaluate the full velocity vector somewhere we will always need to use some kind of interpolation even if we're looking at a grid point! At an arbitrary location in space, we'll do separate bilinear or trilinear interpolation for each component of velocity, but since those components are offset from each other we will need to compute a different set of interpolation weights for each component. At the grid locations themselves, this boils down to some simple averaging. In two dimensions these averages are:

$$\vec{u}_{i,j} = \left(\frac{u_{i-1/2,j} + u_{i+1/2,j}}{2}, \quad \frac{v_{i,j-1/2} + v_{i,j+1/2}}{2}\right) \tag{2.22}$$

$$\vec{u}_{i+1/2,j} = \left(u_{i+1/2,j}, \quad \frac{v_{i,j-1/2} + v_{i,j+1/2} + v_{i+1,j-1/2} + v_{i+1,j+1/2}}{4}\right) \tag{2.23}$$

$$\vec{u}_{i,j+1/2} = \left(\frac{u_{i-1/2,j} + u_{i+1/2,j} + u_{i-1/2,j+1} + u_{i+1/2,j+1}}{4}, \quad v_{i,j+1/2}\right) \tag{2.24}$$

$$\tag{2.25}$$

In three dimensions the formulas are similar:

$$\vec{u}_{i,j,k} = \left(\frac{u_{i-1/2,j,k} + u_{i+1/2,j,k}}{2}, \quad \frac{v_{i,j-1/2,k} + v_{i,j+1/2,k}}{2}, \quad \frac{w_{i,j,k-1/2} + w_{i,j,k+1/2}}{2} \right) \tag{2.26}$$

$$\vec{u}_{i+1/2,j,k} = \left(u_{i+1/2,j,k}, \quad \frac{\begin{array}{c} v_{i,j-1/2,k} + v_{i,j+1/2,k} \\ +v_{i+1,j-1/2,k} + v_{i+1,j+1/2,k} \end{array}}{4}, \quad \frac{\begin{array}{c} w_{i,j,k-1/2} + w_{i,j,k+1/2} \\ +w_{i+1,j,k-1/2} + w_{i+1,j,k+1/2} \end{array}}{4} \right) \tag{2.27}$$

$$\vec{u}_{i,j+1/2,k} = \left(\frac{\begin{array}{c} u_{i-1/2,j,k} + u_{i+1/2,j,k} \\ +u_{i-1/2,j+1,k} + u_{i+1/2,j+1,k} \end{array}}{4}, \quad v_{i,j+1/2,k}, \quad \frac{\begin{array}{c} w_{i,j,k-1/2} + w_{i,j,k+1/2} \\ +w_{i,j+1,k-1/2} + w_{i,j+1,k+1/2} \end{array}}{4} \right) \tag{2.28}$$

$$\vec{u}_{i,j,k+1/2} = \left(\frac{\begin{array}{c} u_{i-1/2,j,k} + u_{i+1/2,j,k} \\ +u_{i-1/2,j,k+1} + u_{i+1/2,j,k+1} \end{array}}{4}, \quad \frac{\begin{array}{c} v_{i,j-1/2,k} + v_{i,j+1/2,k} \\ +v_{i,j-1/2,k+1} + v_{i,j+1/2,k+1} \end{array}}{4}, \quad w_{i,j,k+1/2} \right) \tag{2.29}$$

(2.30)

Chapter 3

Advection Algorithms

In the previous chapter we saw that a crucial step of the fluid simulation is solving the advection equation:

$$Dq/Dt = 0. \tag{3.1}$$

We will encapsulate this in a numerical routine

$$q^{n+1} = \texttt{advect}(\vec{u}, \Delta t, q^n) \tag{3.2}$$

which given a velocity field $\vec{u}$ (discretized on a MAC grid), a time step size Δt and the current field quantity q^n returns an approximation to the result of advecting q through the velocity field over that duration of time.

The brute force approach to solving Dq/Dt for a time step is to simply write out the PDE, e.g. in one dimension:

$$\frac{\partial q}{\partial t} + u\frac{\partial q}{\partial x} = 0 \tag{3.3}$$

and then replace the derivatives with finite differences. For example, if we use Forward Euler for the time derivative and an accurate central difference for the spatial derivative we get:

$$\frac{q_i^{n+1} - q_i^n}{\Delta t} + u_i^n \frac{q_{i+1}^n - q_{i-1}^n}{2\Delta x} = 0 \tag{3.4}$$

which can be arranged into an explicit formula for the new values of q:

$$q_i^{n+1} = q_i^n - \Delta t u_i^n \frac{q_{i+1}^n - q_{i-1}^n}{2\Delta x} \tag{3.5}$$

At first glance this seems just fine. But there are disastrous problems lurking here!

First off, it turns out that Forward Euler is unconditionally **unstable** for this discretization of the spatial derivative: no matter how small we make Δt, it will always eventually blow up! (If you know about the stability region of Forward Euler, what's happening is that the eigenvalues of the Jacobian generated by the central difference are pure imaginary, thus always outside the region of stability. If you don't, don't worry: we'll get to a method that works soon enough!)

Even if we replace Forward Euler with a more stable time integration technique, in fact even if were to somehow *exactly* solve the time part of the PDE, the spatial discretization will give us major troubles. Remember from last chapter that discussion of the naughty null-space of standard central differences? Well, it raises its ugly head here too: **"high frequency"**[1] jagged components of the solution, like $(-1)^i$, erroneously register as having zero or near-zero spatial derivative, and so don't get evolved forward in time, or move much more slowly than they should. Meanwhile the low frequency components are handled accurately and move at the right speed. This boils down to the low frequency components separating out from the high frequency components, and you are left with all sorts of strange high frequency wiggles and oscillations appearing and sticking around that **shouldn't be there**!

We won't go into a more rigorous analysis of the problems of simple central differences, but rest assured there is plenty of high powered numerical analysis which not only carefully identifies the disease but also supplies a cure with more sophisticated finite difference formulas for the spatial derivative.

We will instead take a different, simpler and physically-motivated approach called the **"semi-Lagrangian"** method. The word Lagrangian should remind you that the advection equation $Dq/Dt = 0$ is utterly trivial in the Lagrangian framework, and if we were using particle system methods it's solved automatically when we move our particles through the velocity field. That is, the new value of q at some point $\vec{x}$ in space is just what the old value of q was for the particle that ends up at $\vec{x}$.

We can apply that reasoning on our grid to get the semi-Lagrangian method introduced to graphics by Stam [Sta99]. We want to figure out the new value of q at a grid point, and to do that in a Lagrangian way we need to figure out the old value of q that the particle that ends up at the grid point possesses. The particle is moving through the velocity field $\vec{u}$, and we know where it ends up. To find where it started we simply run backwards through the velocity field from the grid point. We can grab the old value of q from this start-point, and we've got the new value of q at the grid point! But wait, you say, what if that start point wasn't on the grid? In that case we simply interpolate the old value of q from the old values on the grid, and we're done.

Let's go through that again, slowly and with formulas. We'll say that the location in space of the grid point we're looking at is $\vec{x}_G$. We want to find the new value of q at that point, which we'll call q_G^{n+1}. We know from our understanding of advection that if a hypothetical particle with old value q_P^n ends up at $\vec{x}_G$, when it moves through the velocity field for the time step Δt, then $q_G^{n+1} = q_P^n$. So the question is, how do we figure out q_P^n?

The first step is figuring out where this imaginary particle would have started from, a position we'll call $\vec{x}_P$. The particle moves according to the simple ordinary differential equation:

$$\frac{d\vec{x}}{dt} = \vec{u} \tag{3.6}$$

and ends up at $\vec{x}_G$ after time Δt. If we now run time backwards, we can go in reverse from $\vec{x}_G$ to the start point of the particle. That is, where a particle would end up under the reverse velocity field $-\vec{u}$ "starting" from $\vec{x}_G$. The simplest possible way to estimate this is to use one step of Forward Euler:

$$\vec{x}_P = \vec{x}_G - \Delta t\, \vec{u}_G \tag{3.7}$$

where we use the velocity evaluated at the grid point to take a Δt step backwards through the flow field. It turns out Forward Euler is often adequate, but somewhat better results can be obtained using a slightly more sophisti-

[1] By frequency in this context we mean frequency in space, as if you performed a Fourier transform of the function, expressing it as a sum of sine or cosine waves of different spatial frequencies. The high frequency components correspond to sharp features that vary over a small distance, and the low frequency components correspond to the smooth large-scale features.

cated technique like Modified Euler, a second order Runge-Kutta (RK2) method. See appendix A to review time integration methods.

We now know where the imaginary particle started from; next we have to figure out what old value of q it had. Most likely $\vec{x}_P$ is not on the grid, so we don't have the exact value, but we can get a good approximation by interpolating from q^n at nearby grid points. Bilinear (trilinear in three dimensions) interpolation is usually used, though more accurate schemes have been proposed (see chapter 5 for one scheme documented in [FSJ01]).

Putting this together into a formula, our simplest semi-Lagrangian formula is:

$$q_G^{n+1} = \texttt{interpolate}(q^n, \vec{x}_G - \Delta t\, \vec{u}_G) \tag{3.8}$$

Note that the particle I've described is purely hypothetical. No particle is actually created in the computer: we simply use Lagrangian particles to conceptually figure out the update formula for the Eulerian advection step. Because we are almost using a Lagrangian approach to do an Eulerian calculation, this is called the semi-Lagrangian method.

Just for completeness, let's illustrate this in one dimension again, using Forward Euler and linear interpolation for the semi-Lagrangian operations. For grid point x_i, the particle is traced back to $x_P = x_i - \Delta t u_i$. Assuming this lies in the interval $[x_j, x_{j+1}]$, and letting $\alpha = (x_P - x_j)/\Delta x$ be the fraction of the interval the point lands in, the linear interpolation is $q_P^n = (1-\alpha)q_j^n + \alpha q_{j+1}^n$. So our update is:

$$q_i^{n+1} = (1-\alpha)q_j^n + \alpha q_{j+1}^n \tag{3.9}$$

In practice we will need to advect the velocity field, and perhaps additional variables such as smoke density or temperature. Usually the additional variables are stored at the grid cell centers, but the velocity components are stored at the staggered grid locations discussed in the previous chapter. In each case, we will need to use the approriate averaged velocity, given at the end of the previous chapter, to estimate the particle trajectory.

3.1 Boundary Conditions

If the starting point of the imaginary particle is in the interior of the fluid, then doing the interpolation is no problem. What happens if the estimated starting point happens to end up outside of the fluid boundaries though? This could happen because fluid is flowing in from outside the domain (and the particle is "new" fluid), or it could happen due to numerical error (the true trajectory of the particle actually stayed inside the fluid, but our Forward Euler or RK2 step introduced error that put us outside).

This is really the question of boundary conditions. In the first case, where we have fluid flowing in from the outside, we should know what the quantity is that's flowing in: that's part of stating the problem correctly. For example, if we say that fluid is flowing in through a grating on one side of the domain at a particular velocity $\vec{U}$, then any particle whose starting point ends up past that side of the domain should get velocity $\vec{U}$.

In the second case, where we simply have a particle trajectory which strayed outside the fluid boundaries due to numerical error, the appropriate strategy is to extrapolate the quantity from the nearest point on the boundary—this is our best bet as to the quantity that the true trajectory (which should have stayed inside the fluid) would pick up. Sometimes that extrapolation can be easy: if the boundary we're closest to has a specified velocity we simply use

that. For example, for simulating smoke in the open air we could assume a constant wind velocity $\vec{U}$ (perhaps zero) outside of the simulation domain.

The trickier case is when the quantity isn't known *a priori* but has to be numerically extrapolated from the fluid region where it is known. We will go into more detail on this extrapolation in chapter 6 on water. For now, let's just stick with finding the closest point that is on the boundary of the fluid region, and interpolating the quantity from the fluid values stored on the grid near there. In particular, this is what we will need to do for finding velocity values when our starting point ends up inside a solid object, or for free-surface flows (water) if we end up in the free space.

Note that taking the fluid velocity at a solid boundary is *not* the same as the solid's velocity in general. As we discussed earlier, the normal component of the fluid velocity had better be equal to the normal component of the solid's velocity, but apart from in viscous flows, the tangential component can be completely different. Thus we usually interpolate the fluid velocity at the boundary, and don't simply take the solid velocity. However, for the particular case of viscous flows (or at least, a viscous fluid-solid interaction), we can indeed take the shortcut of just using the solid's velocity instead.

3.2 Time Step Size

A primary concern for any numerical method is whether it is stable: will it blow up? Happily the semi-Lagrangian approach is **"unconditionally stable"**: no matter how big Δt is, we can never blow up. It's easy to see why: wherever the particle starting point ends up, we interpolate from old values of q to get the new values for q. Linear/bilinear/trilinear interpolation always produces values that lie between the values we're interpolating from: we can't create larger or smaller values of q than were already present in the previous time step. So q stays bounded. This is really very attractive: we can select the time step based purely on the accuracy vs. speed trade-off curve. If we want to run at real-time rates regardless of the accuracy of the simulation, we can pick Δt equal to the frame time for example.

In practice, the method can produce some strange results if we are too aggressive with the time step size. It has been suggested[FF01] that an appropriate strategy is to limit Δt so that the furthest a particle trajectory is traced is five grid cell widths:

$$\Delta t \leq \frac{5\Delta x}{u_{max}} \tag{3.10}$$

where u_{max} is an estimate of the maximum velocity in the fluid. This could be as simple as the maximum velocity currently stored on the grid. A more robust estimate takes into account velocities that might be induced due to acceleration g from gravity (or other body forces like buoyancy) over the time step. In that case

$$u_{max} = \max(|u^n|) + \Delta t\,|g| \tag{3.11}$$

and after manipulating inequality (3.10) we can approximate this as:

$$u_{max} = \max(|u^n|) + \sqrt{5\Delta x\, g} \tag{3.12}$$

This has the advantage of always being positive, even when the initial velocities are zero, so we avoid a divide-by-zero in inequality (3.10).

3.2.1 The CFL Condition

Before leaving the subject of time step sizes for advection, I can't resist setting the record straight on a pet peeve of mine. The **"CFL condition"** is one of most abused terms in physics-based animation, and to try to make things clearer, I'll explain it fully here. This section can be safely skipped if you're not interested in some more technical aspects of numerical analysis.

The CFL condition, named for applied mathematicians R. Courant, K. Friedrichs and H. Lewy, is a simple and very intuitive necessary condition for convergence.[2] Convergence means that if you repeat a simulation with smaller and smaller Δt and Δx, in the limit going to zero, then your numerical solutions approch the exact solution.[3]

The solution of a time dependent partial differential equation, such as the advection equation, at a particular point in space $\vec{x}^\star$ and time $t^\star$ depends on some or all of the initial conditions. That is, it may be possible that the value of the intial conditions at one point can be changed without changing the value of the solution at $\vec{x}^\star$ and $t^\star$, whereas if you modify the initial conditions at other points the value of the solution there will change. In the case of the constant-velocity advection equation, the value $q(\vec{x}^\star, t^\star)$ is exactly $q(\vec{x}^\star - t^\star\vec{u}, 0)$, so it only depends on a single point in the initial conditions. For other PDE's, such as the heat equation $\partial q/\partial t = \nabla \cdot \nabla q$, each point of the solution depends on *all* points in the initial conditions. The **"domain of dependence"** for a point is the set of points that have an effect on the value of the solution at that point.

Each point of a numerical solution also has a domain of dependence: again, the set of locations in the initial conditions that have an effect on the value of the solution at that point. It should be intuitively obvious that the numerical domain of dependence, in the limit, must contain the true domain of dependence if we want to get the correct answer. This is, in fact, the CFL condition: convergence is only possible in general if, in the limit as $\Delta x \to 0$ and $\Delta t \to 0$, the numerical domain of dependence for each point contains the true domain of dependence.

For semi-Lagrangian methods, the CFL condition is automatically satisfied: in the limit, the particle trajectories we trace converge to the true ones, so we interpolate from the correct grid cells and get the correct dependence. So talking about the CFL condition in the context of semi-Lagrangian methods is pretty much vacuous: there isn't one!

That said, for standard explicit finite difference methods for the advection equation, where the new value of a grid point q_i^{n+1} is calculated from a few of the old values at neighbouring grid points, i.e. from points only $C\Delta x$ away for a small integer constant C, there is a meaningful CFL condition. In particular, the true solution is going at speed $|\vec{u}|$, so the speed at which numerical information is transmitted, i.e. $C\Delta x/\Delta t$, must be at least as fast. That is:

$$\frac{C\Delta x}{\Delta t} \geq |\vec{u}| \tag{3.13}$$

which turns into a condition on the time step:

$$\Delta t \leq \frac{C\Delta x}{|\vec{u}|} \tag{3.14}$$

Now, here's where most of the confusion arises. This is often the same, up to a small constant factor, as the maximum stable time step for the method. (There are of course exceptions: the first method in this chapter was unstable no matter how small the time step, and it is possible to devise explicit methods which are stable no matter how large

[2] It is **not** three downs and 110 yards.

[3] As an aside, this is a sticky point for the three dimensional incompressible Navier-Stokes equations, as at the time of this writing nobody has managed to prove that they do in fact have a unique solution for all time. It has already been proven in two dimensions, but a million dollar prize has been offered from the Clay Institute for the first person to finish the proof in three dimensions.

the time step is—though of course they give the wrong answer unless the CFL condition is satisfied!) Because the two are often about the same, people got the notion that somehow the CFL condition is always the same as the requirement for stability, when it is has nothing at all to do with it, or even that it's always the condition (3.14) regardless of the numerical method you're using. Thus you may see written that the time step we talked about earlier, inequality (3.10), is five times the CFL condition—when that doesn't make any sense whatsoever!

So, now you know. Spread the word. Don't abuse the CFL condition when you really mean stability condition or accuracy condition.

3.3 Dissipation

Notice that in the interpolation step of semi-Lagrangian advection we are taking a weighted average of values from the previous time step. That is, with each advection step, we are doing an averaging operation. Averaging tends to smooth out sharp features, a process called **"dissipation"**. A lot of the rest of these notes is in fact about methods that try to avoid this dissipation: it can be fairly catastrophic in some cases!

Let's try to understand this smoothing behaviour more physically. We'll use a clever technique called **"modified PDEs"**. The common way of looking at numerical error in solving equations is that our solution gets perturbed from the true solution by some amount. The different approach that we'll now use, sometimes also called **"backwards error analysis"**, is to say that we *are* solving a problem exactly—it's just the problem isn't quite the same as the one we started out with, and has been perturbed in some way. Often interpreting the error this way, and understanding the perturbation to the problem being solved, is extremely useful.

To make our analysis as simple as possible, we'll solve the advection problem in one dimension with a constant velocity $u > 0$:

$$\frac{\partial q}{\partial t} + u\frac{\partial q}{\partial x} = 0 \tag{3.15}$$

We'll assume $\Delta t < \Delta x/u$, i.e. that the particle trajectories are less than a grid cell—the analysis easily extends to larger time steps too, but nothing significant changes. In that case, the starting point of the trajectory that ends on grid point i is in the interval $[x_{i-1},\, x_i]$. Doing the linear interpolation between q^n_{i-1} and q^n_i at point $x_i - \Delta t u$ gives:

$$q^{n+1}_i = \frac{\Delta t\, u}{\Delta x}\, q^n_{i-1} + \left(1 - \frac{\Delta t\, u}{\Delta x}\right) q^n_i \tag{3.16}$$

We can rearrange this to get:

$$q^{n+1}_i = q^n_i - \Delta t\, u\, \frac{q^n_i - q^n_{i-1}}{\Delta x} \tag{3.17}$$

which is in fact exactly the Eulerian scheme of Forward Euler in time and a one-sided finite difference in space.[4] Now recall the Taylor series for q^n_{i-1}:

$$q^n_{i-1} = q^n_i - \left(\frac{\partial q}{\partial x}\right)^n_i \Delta x + \left(\frac{\partial^2 q}{\partial x^2}\right)^n_i \frac{\Delta x^2}{2} + O(\Delta x^3) \tag{3.18}$$

[4] If you're interested, note that the side the finite difference is biased to is the side from which the fluid is flowing: this is no coincidence, and in general, biasing a finite difference to the direction that flow is coming from is called **"upwinding"**. You get information from upwind, not downwind, in advection. Most advanced Eulerian schemes are upwind schemes that do this with more advanced one-sided finite difference formulas.

Subsituting this into equation (3.17) and doing the cancellation gives:

$$q_i^{n+1} = q_i^n - \Delta t\, u \frac{1}{\Delta x}\left(\left(\frac{\partial q}{\partial x}\right)_i^n \Delta x - \left(\frac{\partial^2 q}{\partial x^2}\right)_i^n \frac{\Delta x^2}{2} + O(\Delta x^3)\right) \tag{3.19}$$

$$= q_i^n - \Delta t\, u \left(\frac{\partial q}{\partial x}\right)_i^n + \Delta t\, u \Delta x \left(\frac{\partial^2 q}{\partial x^2}\right)_i^n + O(\Delta x^2) \tag{3.20}$$

$$\tag{3.21}$$

Up to a second order truncation error, this is Forward Euler in time applied to the **modified PDE**:

$$\frac{\partial q}{\partial t} + u\frac{\partial q}{\partial x} = u\Delta x \frac{\partial^2 q}{\partial x^2} \tag{3.22}$$

This is the advection equation plus a viscosity-like term with coefficient $u\Delta x$! That is, when we use the simple semi-Lagrangian method to try to solve the advection equation *without* viscosity, our results look like we were simulating a fluid *with* viscosity. It's called **"numerical dissipation"** (or numerical viscosity, or numerical diffusion—they all mean the same thing).

Fortunately the coefficient of this numerical dissipation goes to zero as $\Delta x \to 0$, so we get the right answer in the limit. Unfortunately, in computer graphics we don't have the patience to let Δx go to zero: we want to see good-looking results with Δx as large as possible!

So how bad is it? It depends on what we're trying to simulate. If we're trying to simulate a viscous fluid, which has plenty of natural dissipation already, then the extra numerical dissipation will hardly be noticed—and more importantly, looks plausibly like real dissipation. But, most often we're trying to simulate nearly inviscid fluids, and this is a serious annoyance which keeeps smoothing the interesting features like small vortices from our flow. As bad as this is for velocity, in chapter 6 we'll see it can be much much worse for other fluid variables.

Thus we need techniques to fix this. More accurate interpolation is a partial answer, proposed for example in [FSJ01], but can be expensive and is still usually inadequate. Later chapters will propose some more effective solutions.

Chapter 4

Making Fluids Incompressible

In this chapter we'll look at making the fluid incompressible and simultaneously enforcing boundary conditions: the implementation of the routine `project`$(\Delta t, \vec{u})$ we mentioned earlier in chapter 2. We'll first cover the classical approach to this, and then later talk about more accurate discretizations.

The `project` routine will subtract off the pressure gradient from the intermediate velocity field $\vec{u}$:

$$\vec{u}^{n+1} = \vec{u} - \Delta t \frac{1}{\rho} \nabla p \tag{4.1}$$

so that the result satisfies incompressibility inside the fluid:

$$\nabla \cdot \vec{u}^{n+1} = 0 \tag{4.2}$$

and satisfies the solid wall boundary conditions:

$$\vec{u}^{n+1} \cdot \hat{n} = \vec{u}_{\text{solid}} \cdot \hat{n} \tag{4.3}$$

Thus the first thing we need to do is write down the discretization of this pressure update: how do we approximate the pressure gradient on the MAC grid? After that we'll look at defining the discrete divergence on the MAC grid, and putting the two together come up with a system of linear equations to solve to find the pressure; we'll cover both the system and an effective way to solve it.

The classical MAC grid approach we'll outline first really only applies to boundaries that are aligned with the grid: it's not obvious how to handle sloped or curved boundaries. We will finish the chapter with a reinterpretation of the pressure equation that leads to a simple way of handling these irregular boundary conditions accurately and robustly.

4.1 The Discrete Pressure Gradient

The raison d'être of the MAC grid, as we briefly discussed before, is that the staggering makes accurate central differences robust. For example, where we need to subtract the $\partial/\partial x$-component of the pressure gradient from the u component of velocity, there are two pressure values lined up perfectly on either side of the u component just waiting to be differenced. You might want to keep referring back to figure 2.1 to see how this works.

So without further ado, here are the formulas for the pressure update in two dimensions, using the central difference approximations for $\partial p/\partial x$ and $\partial p/\partial y$:

$$u_{i+1/2,j}^{n+1} = u_{i+1/2,j} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x} \tag{4.4}$$

$$v_{i,j+1/2}^{n+1} = v_{i,j+1/2} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1} - p_{i,j}}{\Delta x} \tag{4.5}$$

and in three dimensions, including $\partial p/\partial z$ too:

$$u_{i+1/2,j,k}^{n+1} = u_{i+1/2,j,k} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x} \tag{4.6}$$

$$v_{i,j+1/2,k}^{n+1} = v_{i,j+1/2,k} - \Delta t \frac{1}{\rho} \frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x} \tag{4.7}$$

$$w_{i,j,k+1/2}^{n+1} = w_{i,j,k+1/2} - \Delta t \frac{1}{\rho} \frac{p_{i,j,k+1} - p_{i,j,k}}{\Delta x} \tag{4.8}$$

Note that these pressure updates apply to every velocity component that borders a grid cell that contains fluid.

In later sections we'll work out how to determine the pressure in the grid cells that contain fluid. However, the formulas above, applied on the boundary faces of the fluid region, involve pressures in grid cells that lie **outside** of the fluid region. Thus we need to specify our pressure boundary conditions. We will for now assume that grid cells can be simply classified as fluid cells (ones that contain fluid), solid cells (ones that are fully occupied by a solid), or air cells (ones that have nothing in them—as in free surface water simulations). We'll postpone dealing with cells that are only partially filled with solid and/or fluid until the end of the chapter: for now, our solids must line up with the grid perfectly, and we ignore the exact mix of air and water in grid cells at the free surface and just label them as fluid cells.

The easier pressure boundary condition is at a free surface, as in water simulation. Here we assume that the pressure is simply zero outside the fluid: in our update formulas we replace the $p_{i,j,k}$'s that lie in air cells with zero. This is called a **"Dirichlet"** boundary condition if you're interested in the technical lingo: Dirichlet means we're directly specifying the value of the quantity at the boundary.

The more difficult pressure boundary condition is at solid walls. Since the solid wall boundaries line up with the grid cell faces, the component of velocity that we store on those faces is in fact $\vec{u} \cdot \hat{n}$. In this case, subtracting the pressure gradient there should enforce the boundary condition $\vec{u}^{n+1} \cdot \hat{n} = \vec{u}_{\text{solid}} \cdot \hat{n}$. This works out to be a specification of the normal derivative of pressure (which technically is known as a **"Neumann"** boundary condition). By substituting in the pressure update this turns into a simple linear equation to solve for the pressure inside the solid wall. For example, supposing grid cell (i, j) was fluid and grid cell $(i + 1, j)$ was solid, we would update $u_{i+1/2,j}$ with:

$$u_{i+1/2,j}^{n+1} = u_{i+1/2,j} - \Delta t \frac{1}{\rho} \frac{p_{i+1,j} - p_{i,j}}{\Delta x} \tag{4.9}$$

and we know that $u_{i+1/2,j}^{n+1}$ actually is u_{solid}. Rearranging the update equation gives:

$$p_{i+1,j} = p_{i,j} + \frac{\rho \Delta x}{\Delta t} (u_{i+1/2,j} - u_{\text{solid}}) \tag{4.10}$$

Hmm. You might be scratching your head at this point, thinking wouldn't it be much simpler just to directly set the value of $u_{i+1/2,j}^{n+1}$ to what it's supposed to be, rather than figuring out the value of pressure we need to subtract off to

make it that way? In fact, many fluid simulators do it like that, first setting the velocities on the solid walls, and then worrying about pressure later. But, as we'll come back to at the end of the chapter, it actually will pay off to do it in this fashion: we're staying more true to the continuum equations. We also will still need to know what that pressure is when it comes time to formulate equations to solve for the pressure inside the fluid.

Another odd point that needs to be mentioned here, about the boundary condition, is that we figure out the solid grid cell pressure for each boundary face **separately**. That is, if a solid grid cell has two or more faces adjacent to fluid cells, we are going to calculate two or more completely independent pressure values for the cell! If this seems weird just remember that properly speaking, at the continuum level, pressure isn't defined in the interior of the solid: we just use the pressure gradient at the boundary of the solid. Numerically it's convenient to express that as a finite difference (it fits in with all the other formulas) but the pressure we talk about inside that solid grid cell is really just a convenient figment of our mathematical imagination. All that counts is the pressure difference on the boundary faces. So we don't store pressure explicitly inside the solid grid cell (since there could be multiple values for it) but instead just use formulas like equation (4.10) whenever we need to know the pressure difference across a boundary face.

Phwew! Boundary conditions can be complicated—it's no joke that the folklore of computational fluid dynamics says 90% of the sweat, blood and tears is spent on getting boundary conditions right. It could be worth your time going over this section slowly, with a drawing of the MAC grid (like figure 2.1) in front of you, looking at different configurations of solid, fluid and air cells until you feel confident about all this.

4.2 The Discrete Divergence

Now for the easy part of the chapter! In the continuum case, we want our fluid to be incompressible: $\nabla \cdot \vec{u} = 0$. On the grid, we will approximate this condition with finite differences, and require that the divergence estimated at each fluid grid cell be zero for $\vec{u}^{n+1}$.

Remember the divergence in two dimensions is

$$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \tag{4.11}$$

and in three dimensions is

$$\nabla \cdot \vec{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \tag{4.12}$$

Using the obvious central differences (take a look at the MAC grid again) we approximate the two dimensional divergence in fluid grid cell (i, j) as

$$(\nabla \cdot \vec{u})_{i,j} \approx \frac{u_{i+1/2,j} - u_{i-1/2,j}}{\Delta x} + \frac{v_{i,j+1/2} - v_{i,j-1/2}}{\Delta x} \tag{4.13}$$

and in three dimensions, for fluid grid cell (i, j, k):

$$(\nabla \cdot \vec{u})_{i,j,k} \approx \frac{u_{i+1/2,j,k} - u_{i-1/2,j,k}}{\Delta x} + \frac{v_{i,j+1/2,k} - v_{i,j-1/2,k}}{\Delta x} + \frac{w_{i,j,k+1/2} - w_{i,j,k-1/2}}{\Delta x} \tag{4.14}$$

Note that we are only ever going to evaluate divergence for a grid cell that is marked as fluid.

Another way of interpreting the discrete divergence we have defined here is through a direct estimate of the total rate of fluid entering or exiting the grid cell. Remember that this (in the exact continuum setting) is just the integral of the normal component of velocity around the faces of the grid cell:

$$\iint_{\partial \text{cell}} \vec{u} \cdot \hat{n} \tag{4.15}$$

This is the sum of the integrals over each grid cell face. Since we have the normal component of velocity stored at the center of each face, we can estimate the integral easily by just multiplying the normal component by the area of the face (though be careful with signs here—in the above integral the normal is always outwards-pointing, whereas the velocity components stored on the grid always are for the same directions, as shown in figure 2.1 for example). After rescaling, this leads to exactly the same central difference formulas—I'll let you work this out for yourself if you're interested. This numerical technique, where we directly estimate the integral of a quantity around the faces of a grid cell instead of looking at the differential equation formulation, is called the **"finite volume"** method.

Finally we note why the MAC grid is so useful. If we used a regular **"collocated"** grid, where all components of velocity were stored together at the grid points, we would have a difficult time with the divergence. If we used the central difference formula, for example:

$$(\nabla \cdot \vec{u})_{i,j} \approx \frac{u_{i+1,j} - u_{i-1,j}}{2\Delta x} + \frac{v_{i,j+1} - v_{i,j-1}}{2\Delta x} \tag{4.16}$$

then we have exactly the null-space issues we mentioned back in chapter 2. Some highly divergent velocity fields such as $\vec{u}_{i,j} = ((-1)^i, (-1)^j)$ will evaluate to zero divergence. Therefore the pressure solve won't do anything about correcting them, and so high frequency oscillations in the velocity field will persist and probably even will grow unstably when this is combined with advection. There are two possible fixes to get around this but still use a collocated grid. The first is to use a biased, one-sided difference approximation—and while this works, it does introduce a peculiar bias to the simulation which can be disturbingly obvious. The second is to smooth out the high-frequency divergent modes before doing the pressure solve, to get rid of them—unfortunately, our major goal is to **avoid** numerical smoothing wherever possible, so this isn't a good idea for animation either. Thus we stick to the MAC grid.

4.3 The Pressure Equations

We now have the two ingredients we will need to figure out incompressibility: how to update velocities with the pressure gradient, and how to estimate the divergence.

Recall that we want the final velocity, $\vec{u}^{n+1}$ to be divergence-free inside the fluid. To find the pressure that achieves this, we simply subsitute the pressure update formulas for $\vec{u}^{n+1}$, equations (4.4) in 2D and (4.6) in 3D, into the divergence formula, equation (4.13) in 2D and (4.14) in 3D. This gives us a linear equation for each fluid grid cell (remember we only evaluate divergence for a grid cell containing fluid), with the pressures as unknowns.

Let's write this out explicity in 2D, for fluid grid cell (i, j):

$$\frac{u^{n+1}_{i+1/2,j} - u^{n+1}_{i-1/2,j}}{\Delta x} + \frac{v^{n+1}_{i,j+1/2} - v^{n+1}_{i,j-1/2}}{\Delta x} = 0 \tag{4.17}$$

$$\frac{1}{\Delta x}\left[\left(u_{i+1/2,j}-\Delta t\frac{1}{\rho}\frac{p_{i+1,j}-p_{i,j}}{\Delta x}\right)-\left(u_{i-1/2,j}-\Delta t\frac{1}{\rho}\frac{p_{i,j}-p_{i-1,j}}{\Delta x}\right)\right.$$
$$\left.+\left(v_{i,j+1/2}-\Delta t\frac{1}{\rho}\frac{p_{i,j+1}-p_{i,j}}{\Delta x}\right)-\left(v_{i,j-1/2}-\Delta t\frac{1}{\rho}\frac{p_{i,j}-p_{i,j-1}}{\Delta x}\right)\right]=0 \tag{4.18}$$

$$\frac{\Delta t}{\rho}\left(\frac{4p_{i,j}-p_{i+1,j}-p_{i,j+1}-p_{i-1,j}-p_{i,j-1}}{\Delta x^2}\right)=-\left(\frac{u_{i+1/2,j}-u_{i-1/2,j}}{\Delta x}+\frac{v_{i,j+1/2}-v_{i,j-1/2}}{\Delta x}\right) \tag{4.19}$$

And now in 3D for fluid grid cell (i,j,k):

$$\frac{u^{n+1}_{i+1/2,j,k}-u^{n+1}_{i-1/2,j,k}}{\Delta x}+\frac{v^{n+1}_{i,j+1/2,k}-v^{n+1}_{i,j-1/2,k}}{\Delta x}+\frac{w^{n+1}_{i,j,k+1/2}-w^{n+1}_{i,j,k-1/2}}{\Delta x}=0 \tag{4.20}$$

$$\frac{1}{\Delta x}\left[\left(u_{i+1/2,j,k}-\Delta t\frac{1}{\rho}\frac{p_{i+1,j,k}-p_{i,j,k}}{\Delta x}\right)-\left(u_{i-1/2,j,k}-\Delta t\frac{1}{\rho}\frac{p_{i,j,k}-p_{i-1,j,k}}{\Delta x}\right)\right.$$
$$+\left(v_{i,j+1/2,k}-\Delta t\frac{1}{\rho}\frac{p_{i,j+1,k}-p_{i,j,k}}{\Delta x}\right)-\left(v_{i,j-1/2,k}-\Delta t\frac{1}{\rho}\frac{p_{i,j,k}-p_{i,j-1,k}}{\Delta x}\right) \tag{4.21}$$
$$\left.+\left(w_{i,j,k+1/2}-\Delta t\frac{1}{\rho}\frac{p_{i,j,k+1}-p_{i,j,k}}{\Delta x}\right)-\left(w_{i,j,k-1/2}-\Delta t\frac{1}{\rho}\frac{p_{i,j,k}-p_{i,j,k-1}}{\Delta x}\right)\right]=0$$

$$\frac{\Delta t}{\rho}\left(\frac{6p_{i,j,k}-p_{i+1,j,k}-p_{i,j+1,k}-p_{i,j,k+1}-p_{i-1,j,k}-p_{i,j-1,k}-p_{i,j,k-1}}{\Delta x^2}\right)=-\left(\frac{u_{i+1/2,j,k}-u_{i-1/2,j,k}}{\Delta x}+\frac{v_{i,j+1/2,k}-v_{i,j-1/2,k}}{\Delta x}+\frac{w_{i,j,k+1/2}-w_{i,j,k-1/2}}{\Delta x}\right) \tag{4.22}$$

Observe that equations (4.19) and (4.22) are numerical approximations to the **"Poisson"** problem $-\Delta t/\rho\nabla\cdot\nabla p=-\nabla\cdot\vec{u}$.

If a fluid grid cell is at the boundary, recall that the new velocities on the boundary faces involve pressures outside the fluid that we have to define through boundary conditions: we need to use that here. For example, if grid cell $(i,j+1)$ is an air cell, then we replace $p_{i,j+1}$ in equation (4.19) with zero. If grid cell $(i+1,j)$ is a solid cell, then we replace $p_{i+1,j}$ with the value we compute from the boundary condition there, as in formula (4.10). Assuming $(i-1,j)$ and $(i,j-1)$ are fluid cells, this would reduce the equation to the following:

$$\frac{\Delta t}{\rho}\left(\frac{4p_{i,j}-\left[p_{i,j}+\frac{\rho\Delta x}{\Delta t}\left(u_{i+1/2,j}-u_{\text{solid}}\right)\right]-0-p_{i-1,j}-p_{i,j-1}}{\Delta x^2}\right)$$
$$=-\left(\frac{u_{i+1/2,j}-u_{i-1/2,j}}{\Delta x}+\frac{v_{i,j+1/2}-v_{i,j-1/2}}{\Delta x}\right) \tag{4.23}$$

$$\frac{\Delta t}{\rho}\left(\frac{3p_{i,j}-p_{i-1,j}-p_{i,j-1}}{\Delta x^2}\right)=-\left(\frac{u_{i+1/2,j}-u_{i-1/2,j}}{\Delta x}+\frac{v_{i,j+1/2}-v_{i,j-1/2}}{\Delta x}\right)+\left(\frac{u_{i+1/2,j}-u_{\text{solid}}}{\Delta x}\right)$$
$$=-\left(\frac{u_{\text{solid}}-u_{i-1/2,j}}{\Delta x}+\frac{v_{i,j+1/2}-v_{i,j-1/2}}{\Delta x}\right) \tag{4.24}$$

Observe three things about this example that hold in general, and point out how to implement this in code. First, for the air cell boundary condition, we simply just delete mention of that p from the equation. Second, for the solid cell boundary condition, we delete mention of that p but also reduce the coefficient in front $p_{i,j}$, which is normally four, by one—in other words, the coefficient in front of $p_{i,j}$ is equal to the number of non-solid grid cell neighbours

(this is the same in three dimensions). Third, we changed the divergence measured on the right hand side to use the solid wall velocity u_{solid} instead of the old fluid velocity $u_{i+1/2,j}$ there: we would make similar replacements if other boundaries were against a solid surface.

4.3.1 Putting Them In Matrix-Vector Form

We have now defined a large system of linear equations for the unknown pressure values. We can conceptually think of it as a large coefficient matrix, A, times a vector consisting of all pressure unknowns, p, equal to a vector consisting of the divergences in each fluid grid cell, d (with appropriate modifications to divergence at solid wall boundaries):

$$Ap = d \tag{4.25}$$

In an implementation, of course, p and d are probably stored in a two or three-dimensional grid structure (since each entry corresponds to a grid cell), and if not all of the grid cells are fluid, they will not in fact be stored as contiguous vectors in memory.

We also needn't store A directly as a matrix. Notice that each row of A corresponds to one equation, i.e. one fluid cell. For example, if grid cell (i, j, k) is fluid, then there will be a row of the matrix that we can label with the indices (i, j, k). The entries in that row are the coefficients of all the pressure unknowns in that equation: almost all of these are zero except possibly for the seven entries corresponding to $p_{i,j,k}$ and its six neighbours, $p_{i\pm1,j,k}$, $p_{i,j\pm1,k}$ and $p_{i,j,k\pm1}$. (In two dimensions there are at most four neighbours of course.) We only have nonzeros (i, j, k) and its fluid cell neighbours. It is of course pointless to store all the zeros: this is a **"sparse"** matrix.

Let's take a closer look at A. In the equation for (i, j, k), the coefficients for neighbouring fluid cells are all equal to $-\Delta t/(\rho\Delta x^2)$, and if there are $n_{i,j,k}$ fluid or air cell neighbours the coefficient for $p_{i,j,k}$ is $n_{i,j,k}\Delta t/(\rho\Delta x^2)$. It's a little inconvenient to keep that $\Delta t/(\rho\Delta x^2)$ constant around, so we could optionally divide both sides of the equation by it. Then the modified matrix A would have off-diagonal entries either zero or -1, and diagonal entries $n_{i,j,k}$. This is such a simple matrix, then, that you needn't even store it at all! However, when we get to more advanced discretizations at the end of the chapter, A won't be quite as simple so we'll assume that you will store it.

One of the nice properties of the matrix A is that it is symmetric. For example, $A_{(i,j,k),(i+1,j,k)}$, the coefficient of $p_{i+1,j,k}$ in the equation for grid cell (i, j, k), has to be equal to $A_{(i+1,j,k),(i,j,k)}$. Either it's zero if one of those two cells is not fluid, or it's the same nonzero value. This symmetry property will hold even with the more advanced discretization at the end of the chapter. Thus we only have to store half of the nonzero entries in A, since the other half are just copies!

This leads us to the following structure for storing A. In two dimensions, we will store three numbers at every grid cell: the diagonal entry $A_{(i,j),(i,j)}$, and the entries for the neighbouring cells in the positive directions, $A_{(i,j),(i+1,j)}$ and $A_{(i,j),(i,j+1)}$. We could call these entries `Adiag(i,j)`, `Aplusi(i,j)` and `Aplusj(i,j)` in our code. In three dimensions, we would similarly have `Adiag(i,j,k)`, `Aplusi(i,j,k)`, `Aplusj(i,j,k)` and `Aplusk(i,j,k)`. When we need to refer to an entry like $A_{(i,j),(i-1,j)}$ we use symmetry and instead refer to $A_{(i-1,j),(i,j)}$ =`Aplusi(i-1,j)`.

4.3.2 The Conjugate Gradient Algorithm

The matrix A is a very well known type of matrix, sometimes referred to as the five- or seven-point Laplacian matrix (in two or three dimensions respectively). It has been exhaustively studied, the subject of countless numerical linear algebra papers, the first example of a sparse matrix in just about any setting. More effort has been put into solving linear systems with this type of matrix than probably all other sparse matrices put together! We won't look very far into this vast body of work, beyond one particular algorithm that is both efficient and simple to implement, called **"MICCG(0)"**, or more fully **"Modified Incomplete Cholesky Conjugate Gradient, Level Zero"**. Quite a mouthful! Let's go through it slowly.

One of the many properties that A has is that it is **"symmetric positive definite"** (SPD). Technically this means that $q^T Aq > 0$ for any nonzero vector q.

Actually, before going on I should be a little more careful. A might just be symmetric positive semi-definite, meaning that $q^T Aq \geq 0$. If there is some fluid region entirely surrounded by solid walls, with no empty air cells, then A will not be strictly positive definite. In that case, A is singular in fact—it doesn't have an inverse. That doesn't necessarily mean there isn't a solution however. If the divergences (the right hand side) satisfy a **"compatibility condition"** then life is good and there is a solution. The compatibility condition is simply that the velocities of the solid walls are compatible with the fluid contained within being incompressible—i.e. the fluid-solid boundary faces have wall velocities that add up to zero, so that the flow in is balanced by the flow out if there is any flow. In fact, not only will there be a solution, but there are infinitely many solutions! You can take any solution for pressure and add an arbitrary constant to it and get another one, it turns out: but when we take the pressure gradient for the velocity update, the constants cancel so we don't care which solution we get. They're all good.

One particularly useful algorithm for solving symmetric positive (semi-)definite linear systems is called the Conjugate Gradient algorithm, usually abbreviated as CG. It's an iterative method, meaning that we start with a guess at the solution, and in each iteration improve that, stopping when we think we are accurate enough. CG chooses the iterative updates to the guess to minimize a particular measure of the error, and thus can be guaranteed to converge to the solution eventually. Another very nice aspect of CG, as compared to Gaussian Elimination for example, is that each iteration only involves multiplying A by a vector, adding vectors, multiplying vectors by scalar numbers, and computing a few dot-products—all of which are very easy to code, and easy to code to run fast.

The problem with CG, however, is that the larger the grid, the longer it takes to converge. It can be shown that the number of iterations it takes to converge to some desired accuracy is proportional to the width of the grid: the maximum number of grid cells in any one direction. However, there is a trick up our sleeve that can speed this up, called **"preconditioning"**. Preconditioned Conjugate Gradient (PCG) is what we will be using.

More generally, CG takes more iterations the further A is from being the identity matrix, I. It should be immediately obvious that solving a system with the identity matrix is pretty easy—the solution of $Ip = d$ is just $p = d$! How exactly we measure how far A is from the identity is beyond the scope of these notes—something called the **"condition number"** of A. The idea behind preconditioning is that the solution of $Ap = d$ is the same as the solution of $MAp = Md$ for some matrix M. If M is approximately the inverse of A, so that MA is really close to being the identity matrix, then CG should be able to solve the preconditioned equations $MAp = Md$ really fast. PCG is just a clever way of applying CG to these preconditioned equations without actually having to form them.

Before we actually get to the details of PCG, we need to talk about convergence. When do we know to stop? How do we check to see that our current guess is close enough to the solution? Ideally we would just measure the norm

of the difference between our current guess and the exact solution—but of course that requires knowing the exact solution! So we will instead look at a vector called the **"residual"**:

$$r_i = d - Ap_i \tag{4.26}$$

That is, if p_i is the i'th guess at the true solution, the residual r_i is just how far away it is from satisfying the equation $Ap = d$. When we hit the exact solution, the residual is exactly zero. Therefore, we stop our iteration when the norm of the residual is small enough, below some tolerance.

That brings us to the next question: how small is small enough? In fact, what norm do you use to measure r_i? Think back to what r_i means physically. These equations resulted from deriving that $d - Ap$ is the finite difference estimate of the divergence of $\vec{u}^{n+1}$, which we want to be zero. Thus the residual is exactly how much divergence there will be in the velocity field after we've updated it with our estimate of the pressure.[1] It seems sensible, then, to take the infinity norm of the residual (the maximum absolute value of any entry) and compare that to some small number `tol`, so that we know the worst of how compressible our new velocity field could possibly be. The dimensions of `tol` are one over time: $O(1/\texttt{tol})$ is a lower bound on how long it will take our inexact velocity field to compress the fluid by some fraction. Thus `tol` probably should be inversely proportional to the time length of our simulation. In practice, this either doesn't vary a lot (a few seconds for most shots) or we are doing interactive animation with a potentially infinite time length—so in the end we just pick an arbitrary small fixed number for `tol`, like $10^{-6}s^{-1}$. (The s^{-1} is one over seconds: remember again that this quantity has dimensions one over time.) Smaller tolerances will result in less erroneous divergence, but will take more iterations (and time) to compute, so there's a clear trade-off in adjusting this number up or down.

Typically we will also want to guard against inexact floating-point arithmetic causing the algorithm not to converge properly, so we stop at a certain maximum number of iterations. Or, we may have a real-time constraint that limits the number of iterations we can use. Setting this is another case of trade-offs: starting out your simulator with the value 100 is reasonable.

Another last issue is what the initial guess for pressure should be. One nice thing about PCG is that if we start with a good guess, we can get to an acceptable solution much faster. In some circumstances, say when the fluid has settled down and isn't changing much, the pressure from the last time step is a very good guess for the next pressure: it won't change much either. However, these situations usually can easily be solved from an initial guess of all zeros also. In the more interesting cases (and really, why would you be simulating fluids that are just sitting still?) the pressure can change significantly from one time step to the next, or in fact may be defined on different grid cells anyhow (e.g. as some change from fluid to air or vice versa) and can't be used. Therefore we usually use the vector of all zeros as the initial guess.

The PCG algorithm is shown in figure (4.1). Note it needs storage for an "auxiliary" vector and a "search" vector (the same size as p, d, and r), and calls subroutine `applyA` to multiply the coefficient matrix A times a vector, and subroutine `applyPreconditioner` to multiply M by a vector (which we will talk about next). Also, be aware that this statement of PCG uses different symbols than most text books: I've tried to avoid the use of letters like x and ρ which have other meanings in the context of fluids.

[1] Please note: this is assuming we didn't divide out by $\Delta/(\rho\Delta x^2)$ which obviously rescales the equations.

- Set initial guess $p = 0$ and residual vector $r = d$ (If $r = 0$ then return p)
- Set auxiliary vector $z = \texttt{applyPreconditioner}(r)$, and search vector $s = z$
- $\sigma = \texttt{dotproduct}(z, r)$
- Loop until done (or maximum iterations exceeded):
  - Set auxiliary vector $z = \texttt{applyA}(s)$
  - $\alpha = \rho / \texttt{dotproduct}(z, s)$
  - Update $p+ = \alpha s$ and $r- = \alpha z$
  - If $\max |r| \leq \texttt{tol}$ then return p
  - Set auxiliary vector $z = \texttt{applyPreconditioner}(r)$
  - $\sigma_{\text{new}} = \texttt{dotproduct}(z, r)$
  - $\beta = \sigma_{\text{new}} / \rho$
  - Set search vector $s = z + \beta s$
  - $\sigma = \sigma_{\text{new}}$
- Return p (and report iteration limit exceeded)

Figure 4.1: The Preconditioned Conjugate Gradient (PCG) algorithm for solving $Ap = d$.

4.3.3 Incomplete Cholesky

We still have the question of defining the preconditioner. From the standpoint of convergence the perfect preconditioner would be A^{-1}, except that's far too expensive to compute. The true ideal is something that is both fast to compute and apply, and is effective in speeding up convergence, so as to minimize the total solution time.

There are many, many choices of preconditioner, with more being invented each year. Our recommended choice, though, is quite an old preconditioner from the **"Incomplete Cholesky"** (IC) family. For regular PC's (as opposed to massively parallel supercomputers) and for the size of problem we care about in animation (as opposed to the problems that run on those supercomputers) it's really hard to beat. It's fast, it's robust (even when your fluid has splashed around into very irregular shapes), and it's very easy to implement.

Recall how you might directly solve a system of linear equations, with Gaussian Elimination. That is, you do row reductions on your system until the matrix is upper triangular, and then you use back substitution to get the solution one entry at a time. Mathematically this is equivalent to factoring the matrix A as a product of a lower and an upper triangular matrix, and then solving the triangular systems one after the other. In the case of a symmetric positive definite A, we can actually do it so that the two triangular matrices are transposes of each other:

$$A = LL^T \tag{4.27}$$

This is called the **"Cholesky"** factorization. The original system $Ap = d$ is the same as $L(L^T p) = d$, which we can solve as:

$$\begin{aligned} \text{solve} \quad & Lq = d \quad \text{with forward substitution} \\ \text{solve} \quad & L^T p = q \quad \text{with backward substitution} \end{aligned} \tag{4.28}$$

The main reason we don't use this method is that although A has very few nonzeros, L can have a lot: in three dimensions it's particularly bad. You would run out of memory for any decent sized simulation.

Incomplete Cholesky tackles this problem with a very simple idea: whenever the Cholesky algorithm tries to create a new nonzero, in a location that is zero in A, cancel it—keep it as zero. On the one hand the resulting L is just as sparse as A, and memory is no longer an issue. On the other hand, we deliberately made errors: $A \neq LL^T$ now. *However*, hopefully our incomplete factorization is close enough to A that doing the solves in (4.28) is close enough to applying A^{-1} that we have a useful preconditioner for PCG!

Technically, performing Incomplete Cholesky only allowing nonzeros in L where there are nonzeros in A is called Level Zero: IC(0). There are variations which allow a limited number of nonzeros in other locations, but we will not broach that topic here.

To make this more precise, IC(0) constructs a lower triangle matrix L with the same nonzero pattern as the lower triangle of A, such that $LL^T = A$ wherever A is nonzero. The only error is that LL^T is nonzero in some locations where A is zero.

Assume we order our grid cells (and the corresponding rows and columns of A) lexicographically, say along the i dimension first, then the j dimensions, and then the k dimension (if in three dimensions). Which order we take the dimensions doesn't actually matter. Suppose we split A up into its strict lower triangle F and diagonal D:

$$A = F + D + F^T \tag{4.29}$$

Then it can be shown—though we won't do it here—that the IC(0) factor L is of the form

$$L = FE^{-1} + E \tag{4.30}$$

where E is a diagonal matrix. That is, all we need to compute and store are the diagonal entries of L, and we can infer the others just from A!

Crunching through the algebra gives the following formulas for computing the entries along the diagonal of E. In two dimensions:

$$E_{(i,j)} = \sqrt{A_{(i,j),(i,j)} - (A_{(i-1,j),(i,j)}/E_{(i-1,j)})^2 - (A_{(i,j-1),(i,j)}/E_{(i,j-1)})^2} \tag{4.31}$$

In three dimensions:

$$E_{(i,j,k)} = \sqrt{\begin{aligned} A_{(i,j,k),(i,j,k)} - (A_{(i-1,j,k),(i,j,k)}/E_{(i-1,j,k)})^2 - (A_{(i,j-1,k),(i,j,k)}/E_{(i,j-1,k)})^2 \\ -(A_{(i,j,k-1),(i,j,k)}/E_{(i,j,k-1)})^2 \end{aligned}} \tag{4.32}$$

In these equations, we replace terms referring to a non-fluid cell (or cell that lies off the grid) with zero.

4.3.4 Modified Incomplete Cholesky

Incomplete Cholesky is a great preconditioner which can effectively reduce our iteration count when solving the pressure equations. But, for almost no extra cost, we can do better! **Modified** Incomplete Cholesky (MIC) will square-root our iteration count: if our grid is n grid cells wide, we'll need $O(n^{1/2})$ iterations, which for the values of n we use in computer graphics remains pretty small.[2]

[2]There are significantly more effective methods available, such as multigrid which in theory can reduce the number of iterations required to $O(1)$. However, these can be tricky to make work in fluid situations with complicated geometry, or when additional phenomena are added to the simulation; we'll stick with MIC-PCG for now.

- Set tuning constant $\tau = 0.97$
- For `i=1` to `nx`, `j=1` to `ny`, `k=1` to `nz`:
  - If cell (i, j, k) is fluid:
    - Set

$$\begin{aligned} e = \texttt{Adiag}_{i,j,k} &- (\texttt{Aplusi}_{i-1,j,k} * \texttt{precon}_{i-1,j,k})^2 \\ &- (\texttt{Aplusj}_{i,j-1,k} * \texttt{precon}_{i,j-1,k})^2 \\ &- (\texttt{Aplusk}_{i,j,k-1} * \texttt{precon}_{i,j,k-1})^2 \\ -\tau \big[&\texttt{Aplusi}_{i-1,j,k} * (\texttt{Aplusj}_{i-1,j,k} + \texttt{Aplusk}_{i-1,j,k}) * \texttt{precon}^2_{i-1,j,k} \\ &+ \texttt{Aplusj}_{i,j-1,k} * (\texttt{Aplusi}_{i,j-1,k} + \texttt{Aplusk}_{i,j-1,k}) * \texttt{precon}^2_{i,j-1,k} \\ &+ \texttt{Aplusk}_{i,j,k-1} * (\texttt{Aplusi}_{i,j,k-1} + \texttt{Aplusj}_{i,j,k-1}) * \texttt{precon}^2_{i,j,k-1}\big] \end{aligned}$$

    - $\texttt{precon}_{i,j,k} = 1/\sqrt{e + 10^{-30}}$ (small number to guard against accidental divide-by-zero)

Figure 4.2: The calculation of the MIC(0) preconditioner in three dimensions.

Modified Incomplete Cholesky works like Incomplete Cholesky, except instead of discarding those unwanted nonzeros, we account for them by adding them to the diagonal.

To make this more precise, MIC(0) constructs a lower triangular matrix L with the same nonzero pattern as the lower triangle of A, such that:

1 The offdiagonal nonzero entries of A are equal to the corresponding ones of (LL^T)

2 The sum of each row of A is equal to the sum of each row of (LL^T)

This boils down to a slightly different calculation for the diagonal entries: the modified L is also equal to $FE^{-1} + E$, just for a different E. In two dimensions:

$$E_{(i,j)} = \sqrt{\begin{aligned} &A_{(i,j),(i,j)} - (A_{(i-1,j),(i,j)}/E_{(i-1,j)})^2 - (A_{(i,j-1),(i,j)}/E_{(i,j-1)})^2 \\ &\qquad - A_{(i-1,j),(i,j)} A_{(i-1,j),(i-1,j+1)}/E^2_{(i-1,j)} \\ &\qquad - A_{(i,j-1),(i,j)} A_{(i,j-1),(i+1,j-1)}/E^2_{(i,j-1)} \end{aligned}} \tag{4.33}$$

In three dimensions:

$$E_{(i,j,k)} = \sqrt{\begin{aligned} &A_{(i,j,k),(i,j,k)} - (A_{(i-1,j,k),(i,j,k)}/E_{(i-1,j,k)})^2 - (A_{(i,j-1,k),(i,j,k)}/E_{(i,j-1,k)})^2 \\ &\qquad - (A_{(i,j,k-1),(i,j,k)}/E_{(i,j,k-1)})^2 \\ &\quad - A_{(i-1,j,k),(i,j,k)}(A_{(i-1,j,k),(i-1,j+1,k)} + A_{(i-1,j,k),(i-1,j,k+1)})/E^2_{(i-1,j,k)} \\ &\quad - A_{(i,j-1,k),(i,j,k)}(A_{(i,j-1,k),(i+1,j-1,k)} + A_{(i,j-1,k),(i,j-1,k+1)})/E^2_{(i,j-1,k)} \\ &\quad - A_{(i,j,k-1),(i,j,k)}(A_{(i,j,k-1),(i+1,j,k-1)} + A_{(i,j,k-1),(i,j+1,k-1)})/E^2_{(i,j,k-1)} \end{aligned}} \tag{4.34}$$

In practice, you can squeeze out even better performance by taking a weighted average between the regular Incomplete Cholesky formula and the Modified one, typically something like 0.97. See figure 4.2 for pseudo-code to

- (First solve $Lq = r$)
- For `i=1` to `nx`, `j=1` to `ny`, `k=1` to `nz`:
  - If cell (i, j, k) is fluid:
    - Set
$$t = r_{i,j,k} - \texttt{Aplusi}_{i-1,j,k} * \texttt{precon}_{i-1,j,k} * q_{i-1,j,k} - \texttt{Aplusj}_{i,j-1,k} * \texttt{precon}_{i,j-1,k} * q_{i,j-1,k} - \texttt{Aplusk}_{i,j,k-1} * \texttt{precon}_{i,j,k-1} * q_{i,j,k-1}$$
    - $q_{i,j,k} = t * \texttt{precon}_{i,j,k}$
- (Next solve $L^T z = q$)
- For `i=nx` down to `1`, `j=ny` down to `1`, `k=nz` down to `1`:
  - If cell (i, j, k) is fluid:
    - Set
$$t = q_{i,j,k} - \texttt{Aplusi}_{i,j,k} * \texttt{precon}_{i,j,k} * z_{i+1,j,k} - \texttt{Aplusj}_{i,j,k} * \texttt{precon}_{i,j,k} * z_{i,j+1,k} - \texttt{Aplusk}_{i,j,k} * \texttt{precon}_{i,j,k} * z_{i,j,k+1}$$
    - $z_{i,j,k} = t * \texttt{precon}_{i,j,k}$

Figure 4.3: Applying the MIC(0) preconditioner in three dimensions ($z = Mr$).

implement this in three dimensions. We actually compute and store the *reciprocals* of the diagonal entries of E, in a grid variable called `precon`, to avoid divides when applying the preconditioner.

All that's left is how to apply the preconditioner, that is perform the triangular solves. This is illustrated in figure 4.3 for three dimensions.

Finally, before going on I should note that the nesting of loops is an important issue for performance. Due to cache effects, it's far faster if you can arrange your loops to walk through memory sequentially. For example, if $p_{i,j,k}$ and $p_{i,j,k+1}$ are stored next to each other in memory, then the k loop should be innermost.

4.4 Projection

Putting the following together, the `project`$(\Delta t, \vec{u})$ routine does the following:

- Calculate the divergence d (the right-hand side) with modifications at solid wall boundaries.
- Set the entries of A (stored in `Adiag` etc.).
- Construct the MIC(0) preconditioner.
- Solve $Ap = d$ with MICCG(0), i.e. the PCG algorithm with MIC(0) as preconditioner.
- Compute the new velocities $\vec{u}^{n+1}$ according to the pressure gradient update to $\vec{u}$.

We still haven't explained *why* this routine is called `project`. You can skip over this section if you're not interested.

If you recall from your linear algebra, a projection is a special type of linear operator such that if you apply it twice, you get the same result as applying it once. For example, a matrix P is a projection if $P^2 = P$. It turns out that our transformation from $\vec{u}$ to $\vec{u}^{n+1}$ is indeed a linear projection.[3]

If you want, you can trace through the steps to establish the linearity: the entries of d are linear combinations of the entries of $\vec{u}$, the pressures $p = A^{-1}d$ are linear combinations of the entries of d, and the new velocities $\vec{u}^{n+1}$ are linear combinations of $\vec{u}$ and p.

Physically it's very obvious that this transformation has to be a projection. The resulting velocity field, $\vec{u}^{n+1}$, has discrete divergence equal to zero. So if we repeated the pressure step with this as input, we'd first evaluate $d = 0$, which would give constant pressures and no change to the velocity.

4.5 Accurate Curved Boundaries

Reducing the geometry of the problem to just labeling grid cells as solid, fluid or air can introduce major artifacts when the boundaries are actually at an angle or curved. For example, for a water surface we are eliminating the ability of the fluid simulation to "see" small waves (less than a grid cell high)—when water splashes around and comes to a rest, it probably will continue to have small static bumps on the surface because the simulator just sees a flat plane of fluid cells. Similarly, a solid slope becomes in the eyes of the simulator a sequence of flat stair steps: obviously water flows down stairs very differently than an inclined slope. If you render an inclined slope but your water pools at regular intervals along it instead of flowing down, people will complain.

4.5.1 Free Surface Boundaries

The first problem, treating the fluid-air interface accurately (the $p = 0$ boundary condition), has been solved for example by Gibou et al. [GFCK02]. The trick is to estimate "ghost" pressures in the air cells such that when you linearly interpolate p between the fluid cell and its air cell neighbour, $p = 0$ at the actual fluid-air boundary—not at the center of the air cell. We thus enforce the $p = 0$ condition at the correct place. Let's go through the math on this.

[3] Technically speaking, if we have nonzero solid wall velocities then this is an affine transformation rather than linear, but it still is a projection. For the purposes of simplicity we'll ignore this case here.

Dropping extraneous subscripts, let us just consider a cell i whose center is in the fluid and the neighbouring cell $i+1$ whose center is in the air. We will assume we can determine from the geometry of the free surface that it cuts through the line between grid cell centers i and $i+1$ at a fraction θ. That is, $x_F = (1-\theta)x_i + \theta x_{i+1}$ is the location of the free surface on that line.

We want to implicitly capture the condition $p(x_F) = 0$ without actually adding a pressure value there (which would ruin our regular grid). So we construct a ghost pressure value in the air, p_{i+1}.

Note that the *real* pressure of the air (or vacuum) at x_{i+1} is 0. However, the real pressure isn't smooth across the free surface (it's a constant zero in the air and then suddenly starts increasing when we move into the water, its derivative jumping from zero to a nonzero value discontinuously). So it doesn't make sense to construct a finite difference using the real pressure value—trying to take a derivative of a non-smooth function puts you on shaky ground! Thus instead we think of what the pressure would be if the water continued up into the air, hitting zero at x_F along the way.

Since we know pressure is smooth inside a fluid, the simplest guess that makes sense is to linearly extrapolate it. Fitting a line through $p(x_i) = p_i$ and $p(x_F) = 0$ gives:

$$p_{i+1} = -\frac{1-\theta}{\theta} p_i \tag{4.35}$$

You can check that if you linearly interpolate between p_i and p_{i+1} you do get zero at x_F.

Equation (4.35) might look a little scary if $\theta \approx 0$, i.e. the free surface is really close to the center of grid cell i. To guard against divide-by-zero, we can clamp θ to always be at least some small number, say 10^{-6}. But rest assured: the closer p_i is to the $p = 0$ boundary condition, the closer p_i will be to zero, so dividing by θ won't be a problem.

So, to handle the free surface well in our pressure equations and the pressure gradient velocity update, whenever we need a pressure value in the air we use formula 4.35 instead of just zero. Continuing the example above, when we update $u_{i+1/2}$ we do:

$$\begin{aligned} u_{i+1/2}^{n+1} &= u_{i+1/2} - \frac{\Delta t}{\rho} \frac{p_{i+1} - p_i}{\Delta x} \\ &= u_{i+1/2} - \frac{\Delta t}{\rho} \frac{-\frac{1-\theta}{\theta} p_i - p_i}{\Delta x} \end{aligned} \tag{4.36}$$

In the pressure equation, then, the only modification from before is to add

$$\frac{\Delta t}{\rho \Delta x^2} \frac{1-\theta}{\theta} \tag{4.37}$$

to the diagonal entry. The right-hand side (the divergence of the old velocity) remains the same.

4.5.2 Solid Wall Boundaries

The second problem, treating the solid-fluid interface accurately, is a little more delicate since we are dealing with the derivative of pressure rather than the pressure itself. While some *ad hoc* fixes have been proposed in the animation literature, they generally violate the projection property of the pressure update and can cause strange artifacts as a

result. Very recently, Batty and Bridson [BB06] proposed a simple but consistent solution to this problem which we'll explain here.

Let's do an energy budget of the pressure update. The work done by the pressure is the sum of the work it does on the fluid itself with the work it does on the solid boundaries. We can measure the first term just as the change in kinetic energy:

$$W_{\text{fluid}}[p] = \iiint_{\Omega} \tfrac{1}{2}\rho\|\vec{u}^{n+1}\|^2 - \iiint_{\Omega} \tfrac{1}{2}\rho\|\vec{u}\|^2 \tag{4.38}$$

where Ω is the fluid region and $\vec{u}^{n+1} = \vec{u} - \Delta t/\rho \nabla p$ as we have discussed before. The work done on the solid is simply the integral of the force applied dotted with the displacement. The force pressure exerts on a unit area is just the pressure times the outward normal (that is, the normal pointing out from the fluid into the solid). The displacement we can estimate simply as Δt times the solid velocity, giving:

$$W_{\text{solid}}[p] = \iint_{S} p\hat{n} \cdot \Delta t \vec{u}_{\text{solid}} \tag{4.39}$$

Here we use S to represent the part of the fluid surface in contact with a solid—the rest of the fluid surface if there is any we will call F, for free surface. The total change in mechanical energy of the system due to pressure is then:

$$W[p] = \iiint_{\Omega} \tfrac{1}{2}\rho\|\vec{u}^{n+1}\|^2 - \iiint_{\Omega} \tfrac{1}{2}\rho\|\vec{u}\|^2 + \iint_{\Omega} p\hat{n} \cdot \Delta t \vec{u}_{\text{solid}} \tag{4.40}$$

Now, let's think back to our intuitive idea of a fluid as composed of many particles that interact with various forces. The pressure force between two particles is a perfectly inelastic interaction: since the incompressible model eliminates sound waves, there's no possibility for two particles to bounce elastically. The fluid particles also interact inelastically with the solid boundaries. One way to characterize a purely inelastic force is that it dissipates as much kinetic energy as possible: i.e. minimizes W. Thus our pressure problem in fact can be stated succintly as:

$$\min_{\substack{p \\ p = 0 \text{ on } F}} W[p] \tag{4.41}$$

where we include the constraint that pressure has to be zero on the free surface F. Note that the kinetic energy of the intermediate velocity $\vec{u}$ is a constant in $W[p]$, thus doesn't affect the minimization, so from now on we will simply drop it.

In appendix B we show this is in fact mathematically equivalent to the usual PDE statement of the pressure problem:

$$\begin{aligned} \nabla \cdot \frac{\Delta t}{\rho} \nabla p &= \nabla \cdot \vec{u} && \text{inside } \Omega \\ p &= 0 && \text{on } F \\ \frac{\Delta t}{\rho} \nabla p \cdot \hat{n} &= \vec{u} \cdot \hat{n} - \vec{u}_{\text{solid}} \cdot \hat{n} && \text{on } S \end{aligned} \tag{4.42}$$

However, here we will focus on the minimization statement instead. The main advantage of this is that it provides for a simple discretization that can consistently handle irregular boundaries. We will do this by discretizing the integrals first, which isn't hard to do consistently even on an irregular domain, and then finding pressure values which minimize the resulting discrete expression.

Internal Work

Let's turn our attention first to the kinetic energy term in $W[p]$ (which also covers the important case of unmoving solid boundaries, when the work done on the solid is zero). The first simplication is that we can split the kinetic energy integral up into separate integrals for each component, e.g. in three dimensions:

$$\begin{aligned}\iiint_\Omega \tfrac{1}{2}\rho \left\| \frac{\Delta t}{\rho}\nabla p - \vec{u} \right\|^2 &= \iiint_\Omega \tfrac{1}{2}\rho \left[\left(\frac{\Delta t}{\rho}\frac{\partial p}{\partial x} - u \right)^2 + \left(\frac{\Delta t}{\rho}\frac{\partial p}{\partial y} - v \right)^2 + \left(\frac{\Delta t}{\rho}\frac{\partial p}{\partial z} - w \right)^2 \right] \\ &= \iiint_\Omega \tfrac{1}{2}\rho \left(\frac{\Delta t}{\rho}\frac{\partial p}{\partial x} - u \right)^2 + \iiint_\Omega \tfrac{1}{2}\rho \left(\frac{\Delta t}{\rho}\frac{\partial p}{\partial y} - v \right)^2 + \iiint_\Omega \tfrac{1}{2}\rho \left(\frac{\Delta t}{\rho}\frac{\partial p}{\partial z} - w \right)^2\end{aligned} \tag{4.43}$$

We'll approximate each of these integrals separately. Let's just look at the first integral in detail. We have sampled $u_{i+1/2,j,k}$ on the MAC grid, and at each of those sample points we can also estimate $\partial p/\partial x$ with the central difference $(p_{i+1,j,k} - p_{i,j,k})/\Delta x$. So we just need to know how to weight these samples to approximate the integral with a sum. The simplest consistent weight is just ρ times the volume (or area in 2D) of the fluid contained in grid-cell-sized cell centered on this sample point. Note this cell is offset from the normal grid cells so as to be centered on the u sample point, so we will call it a u-cell. This weight is actually the mass of the fluid contained in the u-cell, so we will call it $m_{i+1/2,j,k}$.

In the interior of the fluid, where the u-cell is completely filled with fluid, the mass is just $\rho\Delta x^3$ (or $\rho\Delta x^2$ in 2D). It only gets interesting at the boundaries. Calculating this if the fluid only partially fills the u-cell depends on how we are representing our geometry. The simplest consistent approximation is just to take the fraction α of the line between $x_{i,j,k}$ and $x_{i+1,j,k}$, and scale the mass of a full fluid cell by that:

$$m_{i+1/2,j,k} = \alpha\rho\Delta x^3 \tag{4.44}$$

This is conveniently in line with our ghost fluid approach for free surfaces (which uses a similar fraction θ), and it can be shown that this indeed simplifies the right-hand side in that case. A more accurate approximation is to take the cross-sectional area $A_{i+1/2,j,k}$ of the fluid (in the face on which $u_{i+1/2,j,k}$ lies) and multiply by $\rho\Delta x$:

$$m_{i+1/2,j,k} = \rho\Delta x A_{i+1/2,j,k} \tag{4.45}$$

It can be shown that this allows us to reinterpret the method as a **"finite volume"** approximation, but there still are advantages to thinking in terms of energy minimization as we shall see.

Computing similar masses for the v-cells and w-cells, we arrive at our discrete approximation to the kinetic energy integral:

$$\begin{aligned}K^{n+1} = &\sum_{i,j,k} \tfrac{1}{2} m_{i+1/2,j,k} \left(\frac{p_{i+1,j,k} - p_{i,j,k}}{\Delta x} - u_{i+1/2,j,k} \right)^2 \\ &+ \sum_{i,j,k} \tfrac{1}{2} m_{i,j+1/2,k} \left(\frac{p_{i,j+1,k} - p_{i,j,k}}{\Delta x} - v_{i,j+1/2,k} \right)^2 \\ &+ \sum_{i,j,k} \tfrac{1}{2} m_{i,j,k+1/2} \left(\frac{p_{i,j,k+1} - p_{i,j,k}}{\Delta x} - w_{i,j,k+1/2} \right)^2\end{aligned} \tag{4.46}$$

Note that this sum may include ghost pressures inside the solid—which are left as unknowns to be solved for—as well as ghost pressures inside the air, which we have already figured out based on the neighbouring fluid pressure. In the case where we have a ghost air pressure minus a ghost air pressure, we simply set them both to zero in the above expression: there's nothing to extrapolate from.

To minimize K^{n+1}, we take the gradient with respect to all the pressures and set that equal to zero. This gives us a set of equations for the pressures. In fact, since K^{n+1} is just a quadratic function of the pressures, these equations (the gradient) are just a big linear system—just like before! Let's differentiate K^{n+1} with respect to $p_{i,j,k}$ to see what the i, j, k equation is. It appears in two terms of each of the sums, giving:

$$\begin{aligned}\frac{\partial K^{n+1}}{\partial p_{i,j,k}} = &- m_{i+1/2,j,k}\frac{\Delta t}{\rho \Delta x}\left(\frac{\Delta t}{\rho}\frac{p_{i+1,j,k}-p_{i,j,k}}{\Delta x} - u_{i+1/2,j,k}\right)\\ &+ m_{i-1/2,j,k}\frac{\Delta t}{\rho \Delta x}\left(\frac{\Delta t}{\rho}\frac{p_{i,j,k}-p_{i-1,j,k}}{\Delta x} - u_{i-1/2,j,k}\right)\\ &- m_{i,j+1/2,k}\frac{\Delta t}{\rho \Delta x}\left(\frac{\Delta t}{\rho}\frac{p_{i,j+1,k}-p_{i,j,k}}{\Delta x} - v_{i,j+1/2,k}\right)\\ &+ m_{i,j-1/2,k}\frac{\Delta t}{\rho \Delta x}\left(\frac{\Delta t}{\rho}\frac{p_{i,j,k}-p_{i,j-1,k}}{\Delta x} - v_{i,j-1/2,k}\right)\\ &- m_{i,j,k+1/2}\frac{\Delta t}{\rho \Delta x}\left(\frac{\Delta t}{\rho}\frac{p_{i,j,k+1}-p_{i,j,k}}{\Delta x} - w_{i,j,k+1/2}\right)\\ &+ m_{i,j,k-1/2}\frac{\Delta t}{\rho \Delta x}\left(\frac{\Delta t}{\rho}\frac{p_{i,j,k}-p_{i,j,k-1}}{\Delta x} - w_{i,j,k-1/2}\right)\end{aligned} \tag{4.47}$$

Setting this equal to zero gives:

$$\begin{aligned}\frac{\Delta t^2}{\rho^2 \Delta x^2}\begin{pmatrix} M_{i,j,k}p_{i,j,k} & - m_{i+1/2,j,k}p_{i+1/2,j,k} - m_{i-1/2,j,k}p_{i-1/2,j,k}\\ & - m_{i,j+1/2,k}p_{i,j+1/2,k} - m_{i,j-1/2,k}p_{i,j-1/2,k}\\ & - m_{i,j,k+1/2}p_{i,j,k+1/2} - m_{i,j,k-1/2}p_{i,j,k-1/2}\end{pmatrix} =\\ - \frac{\Delta t}{\rho \Delta x}\begin{pmatrix}(m_{i+1/2,j,k}u_{i+1/2,j,k} - m_{i-1/2,j,k}u_{i-1/2,j,k})\\ +(m_{i,j+1/2,k}v_{i,j+1/2,k} - m_{i,j-1/2,k}v_{i,j-1/2,k})\\ +(m_{i,j,k+1/2}w_{i,j,k+1/2} - m_{i,j,k-1/2}w_{i,j,k-1/2})\end{pmatrix}\end{aligned} \tag{4.48}$$

Here $M_{i,j,k}$ is the sum of the masses of all the overlapping cells, $m_{i-1/2,j,k} + m_{i+1/2,j,k} + m_{i,j-1/2,k} + \ldots$. Note that we of course drop the terms and equations for which $M_{i,j,k} = 0$, where there is no fluid involved at all. This means, for example, that the pseudo-code presented before for constructing and applying the preconditioner should be altered—instead of checking if a grid cell is marked as fluid, we check if $M_{i,j,k} = 0$ or not.

In the interior of the fluid, where all the m's are the same, this is exactly the same as our previous pressure equation up to a rescaling! (In fact, to make physical sense of the residual when checking for convergence in PCG, we may want to divide by that factor, $\Delta t \Delta x^3$ in 3D or $\Delta t \Delta x^2$ in 2D.)

It's also easy to prove that this approach is a projection. We are finding the update that minimizes K^{n+1}, so if we do the update and then try again there's no way we can minimize it any further: the subsequent update will just be zero.

One worry about more complicated schemes for solving the pressure equation is whether the matrix A is still symmetric positive (semi-)definite as required for PCG to work; this is trivially true in this case since A is the Hessian

of K^{n+1} which is always nonnegative (being a postively-weighted sum of squares). A further issue is if, in the semi-definite case (and A is singular), the system of linear equations has a solution—but this again is trivially true since K^{n+1} *must* have a minimum (being a quadratic ≥ 0) thus we don't ever have to worry about compatibility conditions.

Another of the properties guaranteed by this approach is that the simplest of all cases, fluid at rest in a gravity field, is handled exactly. If the top of the fluid (where $p = 0$) is flat, and $\vec{u}$ is a constant $\Delta t g$ pointing down, then it's easy to see the exact hydrostatic pressure $p = -\rho g y$ exactly zeroes out $\vec{u}^{n+1}$, giving $K^{n+1} = 0$ which is the absolute minimum possible.

Solid Boundary Work

In the final version of these notes, we will discuss the discretization of the boundary integral of work done by pressure on the solid, and how that ends up modifying the right hand side of the pressure equations.

Part II

Different Types of Fluids

Chapter 5

Smoke

The first visual fluid phenomena we will consider is smoke, following the paper by Fedkiw et al. [FSJ01]. Our fluid in this case is the air in which the smoke particles are suspended. To model the most important effects of smoke, we need two extra fluid variables: the temperature T of the air and the concentration s of smoke particles (the thing we actually can see).

In reality, temperature changes cause the air to slightly expand or contract, causing a change in density, and the mass of the smoke particles also effectively increases the density. However, these variations are very small, so we will use what is called the **"Boussinesq"** approximation and assume constant density as before but replace the gravitational force with a buoyancy force f_{buoy} that is derived from these variations. Simplifying our life further, we assume buoyancy just depends linearly on the temperature and smoke concentration. Letting the ambient background temperature be T_{amb} (say $300K$, or $270K$ in Canada) we take the buoyancy force to be:

$$f_{\text{buoy}} = (0, -\alpha s + \beta(T - T_{\text{amb}}), 0) \tag{5.1}$$

Here α and β are nonnegative constants which can be set by the user to achieve different behaviour. Note that f_{buoy} reduces to zero force where there's no smoke and the temperature is at the ambient level.

We typically store T and s in the grid cell centers (the same as pressure). To add the buoyancy force to the v component of velocity in its offset MAC grid location, we thus need to average the T and s values at the neighbouring centers, e.g. $T_{i,j+1/2,k} = (T_{i,j,k} + T_{i,j+1,k})/2$.

We also have to detail how T and s are evolved. They are of course advected around with the fluid, but in reality they also slowly diffuse. However, just as we dropped the viscosity term in the momentum equation because the unavoidable numerical dissipation mimics it anyhow, we need not explicitly model heat and smoke diffusion: they'll happen in our numerical methods whether we want them or not. Thus our new evolution equations are simply:

$$\frac{DT}{Dt} = 0 \qquad \frac{Ds}{Dt} = 0 \tag{5.2}$$

When we advect velocity in our simulation, we will also advect temperature and smoke concentration.

Boundary conditions for smoke are reasonably straightforward. For velocity and pressure, we may have solid boundaries for any objects that the smoke is supposed to move around (or out of, if we set a nonzero boundary velocity), and at the edges of the grid that aren't enclosed we have a free surface ($p = 0$). We can extend the temperature field

inside objects just by taking the actual temperature of the object (and in this way, we naturally get convection off hot objects), and outside the domain as the ambient background temperature T_{amb}. The smoke concentration doesn't exist inside objects, except if it is supposed to be pouring out of one (with a nonzero boundary velocity), in which case we can set a source value for smoke inside the object. Otherwise we should extrapolate the smoke concentration into the object, i.e. when asked for smoke inside an object we interpolate from the closest values outside.

The trouble with this model is that, when simulated on a typical coarse grid, the numerical dissipation from the simple semi-Lagrangian advection scheme we use is excessive. The interesting features—small vortices, sharp divides between smoky and clear air—simply vanish far faster than desired. Thus the rest of this chapter details two methods for helping to counter numerical dissipation.

5.1 Vorticity Confinement

To counter the problem of vortices artificially disappearing too fast, we add a **"vorticity confinement"** force which boosts the strength of vortices in the flow, keeping it lively and interesting for much longer.

We first need to know how to measure vortices. The quantity of interest is called, appropriately enough, **"vorticity"**, and is defined as:

$$\vec{\omega} = \nabla \times \vec{u} \tag{5.3}$$

If you need to, refer to Appendix A to brush up on the curl differential operator appearing here. You can verify that for the velocity field of a rigid rotation, for example, the vorticity is exactly twice the angular velocity, lending credence to the idea it measures rotational features. A vortex, loosely speaking, is a peak in the vorticity field, a place that's spinning faster than all the fluid nearby.

Knowing that the numerical advection is going to artificially dissipate vorticity, we want to construct a force field which can partially compensate by increasing the vorticity. This is what we will call vorticity confinement: it's a force which tries to confine small vortices from spreading out and diminishing. The trick is to figure out the direction this force should take to increase the rotation of a vortex.

Our first step after finding the vorticity is to find where the vortices are, that is the local maxes of vorticity. We'll construct unit vectors $\vec{N}$ that point to these simply by normalizing the gradient of $|\vec{\omega}|$:

$$\vec{N} = \frac{\nabla|\vec{\omega}|}{\|\nabla|\vec{\omega}|\|} \tag{5.4}$$

Now $\vec{N}$ points towards the center of rotation of a vortex, and $\vec{\omega}$ itself points along the axis of rotation, so to get a force vector that increases the rotation, we just take a cross-product:

$$f_{\text{conf}} = \epsilon \Delta x (\vec{N} \times \vec{\omega}) \tag{5.5}$$

The ϵ here is a parameter that can be adjusted to control the effect of vorticity confinement. We throw in the Δx term to be physically consistent: as we refine the grid and Δx tends to zero, the erroneous numerical dissipation in the velocity advection also tends to zero so our fix should too.

Let's step through numerically implementing this. We begin by averaging velocities from the MAC grid to the cell

centers (as discussed in chapter 2), and then use central derivatives to approximate the vorticity:[1]

$$\vec{\omega}_{i,j,k} = \left(\frac{w_{i,j+1,k} - w_{i,j-1,k}}{2\Delta x} - \frac{v_{i,j,k+1} - v_{i,j,k-1}}{2\Delta x}, \right. \\ \frac{u_{i,j,k+1} - u_{i,j,k-1}}{2\Delta x} - \frac{w_{i+1,j,k} - w_{i-1,j,k}}{2\Delta x}, \\ \left. \frac{v_{i+1,j,k} - v_{i-1,j,k}}{2\Delta x} - \frac{u_{i,j+1,k} - u_{i,j-1,k}}{2\Delta x} \right) \tag{5.6}$$

The gradient of $|\vec{\omega}|$ is similarly estimated with central differences at the grid cell centers, for use in defining $\vec{N}$:

$$\nabla |\vec{\omega}|_{i,j,k} = \left(\frac{|\vec{\omega}|_{i+1,j,k} - |\vec{\omega}|_{i-1,j,k}}{2\Delta x}, \quad \frac{|\vec{\omega}|_{i,j+1,k} - |\vec{\omega}|_{i,j-1,k}}{2\Delta x}, \quad \frac{|\vec{\omega}|_{i,j,k+1} - |\vec{\omega}|_{i,j,k-1}}{2\Delta x} \right) \tag{5.7}$$

When we normalize this to get N, we should of course guard against a divide-by-zero by using, for example:

$$\vec{N}_{i,j,k} = \frac{\nabla |\vec{\omega}|_{i,j,k}}{\|\nabla |\vec{\omega}|_{i,j,k}\| + 10^{-20}} \tag{5.8}$$

Finally, we take the cross-product to get f_{conf} at the grid cell centers; we can take the appropriate averages to apply this to the different components of velocity on the MAC grid.

We can simplify this to two dimensions by thinking of the two-dimensional velocity field as $(u, v, 0)$. This gives a vorticity field $\vec{\omega} = (0, 0, \omega)$, which we can also interpret as just a scalar ω.

Vorticity is an extremely important fluid quantity, and by taking the curl of the momentum equation one can derive the **"vorticity equation"** which shows how vorticity evolves in time. In fact, many numerical methods for fluid flow have been devised based on this equation (especially in two dimensions, where the vorticity equation is particularly simple). Generally speaking, these vorticity methods accurately track the location, strength, and axis of vortices with vortex particles. Unfortunately, especially in three dimensions, it tends to be difficult to accurately model boundary interactions and other important physics with a pure vortex particle approach. However, we will point out the recent work of Selle et al. [SRF05] which demonstrates how to augment the fluid simulation approach in these notes by tracking vortex particles which feed into a more accurate version of vorticity confinement.

5.2 Sharper Interpolation

While vorticity confinement does wonders for countering numerical dissipation in the velocity field, it doesn't apply to the temperature and smoke concentrations. Using the standard bilinear/trilinear interpolation in semi-Lagrangian advection quickly smears out sharp features in these fields, which for large scale smoke (where true physical diffusion is negligible) looks distinctly wrong.

One partial solution to this problem is to use more accurate interpolation. Fedkiw et al. [FSJ01] suggest a particular choice of cubic interpolant designed to be accurate yet stable, which we will go through in one dimension. This can be used to bootstrap up to two or three dimensions in the same way linear interpolation serves as the basis for

[1] Note: the null-space problem we discussed earlier isn't particularly alarming here: we just will lack the ability to "see" and boost the very smallest vortices. We still get the benefit of boosting slightly larger ones.

bilinear or trilinear interpolation: we interpolate in one dimension along one axis first, then interpolate between the interpolated values along the next axis, and so forth.

The underlying scheme is the Catmull-Rom spline. Given values q_{i-1}, q_i, q_{i+1}, and q_{i+2}, we want to estimate q at a point x between x_i and x_{i+1}. To simplify life, we'll conceptually translate and scale things so that $x_i = 0$ and $x_{i+1} = 1$, i.e. so that the x here is the fraction along the interval we want. We estimate $q(x)$ with a cubic polynomial on the interval that passes through the points q_i and q_{i+1}, and whose slopes at the endpoints match the central difference approximations from the data. That is, we want the slopes to be:

$$d_i = \frac{q_{i+1} - q_{i-1}}{2} \tag{5.9}$$

$$d_{i+1} = \frac{q_{i+2} - q_i}{2} \tag{5.10}$$

The Hermite cubic that satisfies these four conditions is:

$$q(x) \approx q_i + d_i x + (3\Delta q - 2d_i - d_{i+1})x^2 + (-2\Delta q + d_i + d_{i+1})x^3 \tag{5.11}$$

where $\Delta q = q_{i+1} - q_i$. (Note that the coefficients presented in [FSJ01] have a typo, a missing 2 in the cubic term—which perhaps serves as a warning against blindly copying formulas from papers without first deriving them oneself!)

The danger with the Catmull-Rom interpolant is that it can over or undershoot the data. For example, if $q_i < q_{i+1}$ but the slope $d_{i+1} < 0$, the maximum of the cubic occurs somewhere in the middle and is greater than q_{i+1}. If our semi-Lagrangian scheme happened to sample there, it could create a new value for q that was larger than any old value of q. If this keeps happening, q could blow up exponentially: not a good thing at all! To prevent this, we limit the cubic by restricting the slopes to agree with the sign of Δq. That is, we set d_i or d_{i+1} to zero if it has sign different from Δq. Then the interpolant is guaranteed to be monotonic, and we cannot go unstable.

Chapter 6

Water and Level Sets

Our model for water is simply a fluid with a free surface that moves with the fluid velocity. We will include the vorticity confinement force from the previous chapter on smoke to counteract numerical dissipation in the velocity advection. What's new is how to keep track of where the water is.

6.1 Marker Particles

The simplest approach, used for example in the original Marker-and-Cell paper [HW65], is to use **"marker particles"**. Marker particles are simply particles used to mark where the fluid is: any grid cell containing a marker particle is marked fluid, and the empty cells are air.

We begin by sampling the fluid volume with several particles per grid cell. A fairly effective method is to create eight particles in a randomly jittered $2 \times 2 \times 2$ arrangement (or four particles in 2×2 in two dimensions).

At the start of each time step we move the particles through the divergence-free velocity field, each following the simple motion

$$\frac{d\vec{x}}{dt} = \vec{u}(\vec{x}) \tag{6.1}$$

At least a second-order Runge-Kutta method, modified Euler, should be used as Forward Euler behaves miserably for rotational motion. (It can be instructive to try other schemes, such as leapfrog:

$$\vec{x}^{n+1} = \vec{x}^{n-1} + 2\Delta t \vec{u}^n(\vec{x}^n) \tag{6.2}$$

which is guaranteed to trace rotational motion perfectly within its stability limit.) If large time steps in the fluid simulation are being used, then it may be preferable to subdivide the time interval and take multiple steps in moving the particles. In simulations involving solid objects, it is also advisable to detect whether particles have penetrated a solid object and move them out to the fluid region if so.

If you're on your toes you might notice something odd here. These numerical integrators need to sample the fluid velocity along the trajectory of the particle. But if the water is splashing into a new region of space—i.e. the particles are moving outside of the current fluid volume—the fluid velocity might not be defined. The answer is to extrapolate the current fluid velocity out into the rest of the grid (air and solids) before moving marker particles. Indeed, this

will also be necessary for the semi-Lagrangian method to be able to work to determine new velocities in the new fluid regions. We'll return to the question of exactly how to extrapolate velocity in section 6.3: in a nutshell, we'll choose the velocity at a point outside the fluid volume to be the fluid velocity at the closest point on the fluid.

After the particles have been moved, we label each grid cell that contains a marker particle as fluid and the rest as empty (or solid). This information is then used in the other steps of the fluid simulation, such as the pressure projection.

Marker particles are extremely simple and robust, but they do have one glaring flaw: it's difficult to get accurate information about the water surface out of them. They can be rendered with a blobby implicit surface, but the result tends to be bumpy even for water that is supposed to be flat. Similarly if the accurate free surface boundary condition method from the last part of chapter 4 is to be used, we need good information about where the free surface cuts through grid lines: blobbies can give a possible answer again, but it is noisy. This motivates turning to an alternative method for capturing the location of the water: level sets.

6.2 Level Sets

Recall that an implicit surface is one defined as the set of zeros of a function. Letting the function be $\phi(\vec{x})$, the surface is:

$$\{\vec{x} \quad | \quad \phi(\vec{x}) = 0\} \tag{6.3}$$

We will use the convention that the interior (water in our case) is the set where $\phi(\vec{x}) < 0$ and the exterior (air) is the set where $\phi(\vec{x}) > 0$. The chief advantage of implicit surfaces as compared to meshes is that there is no connectivity to worry about, making operations such as topology changes (e.g. two water drops merging together, or a sheet breaking apart into droplets) trivial.

A **"level set"** is simply an implicit surface whose function ϕ is just represented with values sampled on a grid: to be precise, we will assume they are sampled at the grid cell centers. The surface can then be manipulated simply by modifying values on the grid: in particular, level sets are perfectly suited for using PDE's to evolve surfaces. We refer readers to the book by Osher and Fedkiw [OF02] for a detailed study of the numerics and applications of level sets.

Level sets are attractive for water simulation since they can very easily model smooth surfaces (unlike marker particles) while still trivially handling topology changes (unlike meshes), and further easily provide information such as whether a point $\vec{x}$ is inside the fluid: we simply interpolate ϕ at $\vec{x}$ from nearby grid points and check if the sign is negative. They are not without their own problems, however, which we will return to at the end of this section.

Note that if two functions ϕ_1 and ϕ_2 are zero at the same locations, then they represent the same implicit surface even if their nonzero values are different. Thus we have a lot of freedom to choose what our level set should be, and the question becomes: what's the most convenient function?

For good behaviour in numerical methods we desire a function that is reliable—that is, numerical errors that perturb our evaluation shouldn't change the apparent location of the surface very much. In other words, if we jitter $\vec{x}$ by ϵ, we would like $\phi(\vec{x})$ to vary by ϵ. This is obviously a condition on the gradient of ϕ:

$$\|\nabla\phi(\vec{x})\| = 1 \tag{6.4}$$

It turns out we can meet this condition, and get a whole lot more features to boot, by using the **signed distance** function for ϕ.

6.2.1 Signed Distance

The signed distance function of a closed surface is simply the distance to the closest point on the surface, with positive sign if we are outside and negative sign if we are inside. That is, $|\phi(\vec{x})|$ tells us how close we are to the surface, and $\text{sign}(\phi(\vec{x}))$ tells us if we are on the inside or the outside. Obviously this fits the bill for a function defining the surface implicitly: the distance to the surface is zero if and only if we are on the surface.

Suppose now that ϕ is signed distance. At some point $\vec{x}$ inside the surface, let $\hat{n}$ be the unit length direction to the closest point on the surface. Notice that $\phi(\vec{x} + \epsilon\hat{n})$ must be $\phi(\vec{x}) + \epsilon$: if I move along this direction $\hat{n}$, my closest point on the surface doesn't change, and my distance to it changes by exactly how much I move. Therefore the directional derivative of ϕ in this direction is 1:

$$\nabla\phi \cdot \hat{n} = 1 \tag{6.5}$$

Also notice that if I move in any other direction, ϕ cannot change any faster—the fastest way to increase or decrease my distance to the surface is obviously to move along the direction to the closest point. Thus the gradient of ϕ, which is the direction of steepest ascent, must in fact be the direction $\hat{n}$ to the closest point on the surface!

$$\hat{n} = \nabla\phi \tag{6.6}$$

Exactly the same result holds outside the surface, and in the limit as we approach the surface, we just get the outward point surface normal. Putting the two previous equations together, we also see $\|\nabla\phi\| = 1$ as we desired. Another nice feature that falls out of this is that for any point $\vec{x}$, the closest point on the surface is located at $\vec{x} - \phi(\vec{x})\nabla\phi(\vec{x})$.

Typically we will start the simulation with a known signed distance function (either precomputed and loaded in, or an analytic function such as $\|\vec{x} - \vec{c}\| - r$ for a sphere centered on $\vec{c}$ with radius r). However, if we are only given the implicit surface with an arbitrary function on a grid, or given the surface as a mesh, we need to calculate signed distance.

6.2.2 Calculating Signed Distance

The next question is how to calculate the grid point values of signed distance for our level set. There are two general approaches, PDE methods that numerically approximate $\|\nabla\phi\| = 1$ (technically known as the Eikonal equation), and geometric methods that instead calculate distances to the surface. There are merits to both approaches, but we will focus on the latter as they're very robust and accurate but also can be simple and quite fast. In particular, we base our method on algorithm 4 in [Tsa02]. For a recent review of many other algorithms, see [JBS06].

The algorithm is given in figure 6.1. It efficiently propagates information about the surface out to grid points far away, without requiring expensive[1] geometric searches for every single grid point. There are several details to work out: finding the closest points for grid points near the surface, what order to loop over grid points, and determining inside/outside.

[1] Expensive either in terms of CPU time for brute force methods, or programmer time for sophisticated fast methods!

- Find the closest points on the surface for the nearby grid points, setting their signed distance values accordingly. Set the other grid points as unknown.
- Loop over the unknown grid points (i, j, k) in a chosen order:
  - Find all neighbouring grid points who have a known signed distance and closest surface point.
  - Find the distances from $\vec{x}_{i,j,k}$ to those surface points. If $\vec{x}_{i,j,k}$ is closer than a neighbour, mark the neighbour as unknown again.
  - Take the minimum of the distances, determine if (i, j, k) is inside or outside, and set its signed distance value accordingly.

Figure 6.1: A geometry-based signed distance construction algorithm.

Finding Closest Points

If our input is an implicit surface already sampled on the grid (but presumably not with signed distance) then we can immediately identify which grid points are near the surface: they are those with a neighbour of a different sign. This could be the case if we use marker particles and simply wrap a blobby implicit surface around them—we will still want to compute signed distance in this case to facilitate extrapolating the velocity field. An alternative for marker particles is to simply initialize cells containing particles to -1 and empty cells to $+1$; this can give stair-step artifacts though so it's preferably to smooth this ± 1 field out with weighted averages before calculating signed distance.

The simplest approximation in this case (grid values already given) is to estimate the surface points along the lines between a point and its nearest neighbours by finding where the linear interpolant is zero. Then take the closest of all these candidate points. This has the advantage of being guaranteed to find reasonable estimates of points on the surface, though they may well not be the closest points, and so accuracy can be less than desired. A simple way of improving the accuracy is testing additional points surrounding the grid point, in directions other than the grid points, with interpolated values. In practice this is the simplest and most robust approach.

A more sophisticated attack is to estimate the direction to the closest point with the gradient, evaluated with a finite difference, and then do a line search for a zero of the interpolated implicit function along this direction. This is considerably more accurate, but also less robust in the face of general input: we have to make sure the direction is good (actually does point towards the surface) and worry about convergence of our line search.

A third alternative is to first approximate the surface in each grid cell using marching cubes or variants, and then calculate the exact closest point on the polygonal surface.

This brings us to the other case: if our geometry is given in the form of a triangle mesh. In this case we need to use some form of culling to reduce the number of triangles to be tested to a manageable number. For example, we can initialize a list of candidate triangles for each grid point to be empty. Then for each triangle, add it to the lists of the grid points whose cells it (or its bounding box) intersects. For each grid point with a nonempty list, we find the closest point on the triangles in the list. If the distance is d, we further check any additional triangles from lists of gridpoints whose cells are within distance d. Finally we need to determine whether the grid point is inside or outside, which is covered below.

Loop Order

There are two suggestions given in [Tsa02], one based on the Fast Marching Method [Set96, Tsi95] and one based on the Fast Sweeping Method [Zha05].

The Fast Marching Method is based on the realization that grid points should get information about the distance to the surface from points that are closer, not the other way around. We thus want to loop over the grid points going from the closest to the furthest. This can be facilitated by storing unknown grid points in a priority queue (typically implemented as a heap) keyed by the currest estimate of their distance. We initialize the heap with the neighbours of the known grid points, with their distance values and closest points estimated from those known grid points. We select the minimum and remove it from the priority queue, set it as known, and then update the distance and closest point information of its unknown neighbours (possibly adding them to or moving them up in the priority queue). And then do that again, and again, until the priority queue is empty. This runs in $O(n \log n)$ time for n unknown grid points.

The Fast Sweeping Method approach takes a different message from the fact that information propagates out from closer points to further points. For any grid point, in the end its closest point information is going to come to it from one particular direction in the grid—e.g. from $(i+1, j, k)$, or maybe from $(i, j-1, k)$, and so on. To ensure that the information can propagate in the right direction, we thus sweep through the grid points in all possible loop orders: i ascending or descending, j ascending or descending, k ascending or descending. There are eight combinations in three dimensions, four in two dimensions. Each time we sweep through the grid, we include the grid point itself in the list of its neighbours (if the grid point has had a closest point and distance set from a previous sweep—otherwise not). For more accuracy, we can repeat the sweeps again; in practice two times through the sweeps gives good results, though more iterations are possible.

The benefit of Fast Sweeping over Fast Marching is that it is $O(n)$, requires no extra data structures beyond grids, and thus is extremely simple to implement. However, the Fast Marching Method has a signficant advantage in that it permits **"narrow band"** methods: that is, if we don't need the signed distance function more than, say, five grid cells away from the surface, we can stop the algorithm once we hit that distance. For example, in our fluid code if we choose the time step based on the five-grid-cell heuristic (3.10) we shouldn't ever go beyond five grid cells from the current surface. In this case, Fast Marching only takes $O(m \log m)$ time, where m is the number of grid cells set—typically much smaller than the number of grid cells in the whole grid. The remaining grid points can have their ϕ values set to $\pm 5\Delta x$ in this example.

Inside/Outside Tests

If our input is an implicit surface whose function is already sampled on the grid, then the sign of these samples instantly tells us inside/outside: we simply have to copy the sign over to our distance computation. The trickier case is if the input is a triangle mesh. If the mesh is watertight, then the parity of the number of intersections along a ray cast out from the grid point tells us inside/outside (an odd number indicates we were inside, an even, outside). However, CG models often come in the form of triangle soup, with holes, self-intersections, and the like. In this case, we refer to the robust test developed by Houston et al. [HBW03a, HNB$^+$06].

6.2.3 Boundary Conditions

We really are only interested in the free surface of the water, not its boundary with the solid. If we were to try and use the signed distance from the entire fluid surface, both free and solid, we run into two problems. One is that we find we can't avoid having a meniscus-like rounding of the water near the curve where solid, air and water meet: interpolating the level set function automatically rounds off sharp edges. We can't get a truly flat water surface as a result. The other problem is that the water level set surface might not perfectly match up with the solid surface with which it's in contact, resulting in unsightly air pockets appearing in the inevitable gaps when we render.

To fix this, we compute signed distance in the water and air only relative to the water-air free surface, ignoring solid boundaries. We then extrapolate the resulting ϕ into the solid. If done carefully enough this can enable impressive simulations of surface tension interactions: see [WMT05]. A simpler approach is simply to extrapolate ϕ out as a constant, as is done with velocity below (section 6.3), then recompute the ϕ signed distance within the solid. Note that this requires having a signed distance function for the solid, ψ, which will guide the extrapolation of ϕ. The solid signed distance function ψ is zero at the solid surface, negative inside the solid, and positive in both the air and water.

6.2.4 Moving the Level Set

The most critical part of the simulation is actually moving the surface represented by the level set. The free surface, i.e. the points where $\phi = 0$, should be advected with the fluid velocity: we can simply advect the entire level set to capture this. That is, we solve:

$$\frac{D\phi}{Dt} = 0 \tag{6.7}$$

Unfortunately, advecting a signed distance field doesn't in general preserve the signed distance property. Thus we need to periodically (every time step, or possibly every few time steps) recalculate signed distance as we have discussed above.

However, the worst problem is numerical dissipation. Any small features in the surface—ripples, droplets, thin sheets, etc.—show up as sharp high frequency profiles in the signed distance field. When we use semi-Lagrangian advection with trilinear interpolation, we smooth out these sharp profiles, causing the features to simply disappear! To a lesser extent the same thing happens when we recalculate signed distance. Even highly sophisticated fifth-order accurate schemes for solving level set motion run into the same problem, though not quite as badly.

The fundamental issue is that as the level set surface moves, we keep resampling the signed distance function on the fixed grid. Similar to the sampling theorem for signal processing, we cannot reliably represent small features (on the order of a grid cell thick or smaller) with this sampling. The level set represention tends to automatically eliminate these features as they move through the grid, no matter how formally accurate the advection scheme is.[2] To fix this, we thus need to augment or replace our grid representation of the level set. Several methods have been proposed to do this, mostly relying on including marker particles in the simulation.

[2]This automatic dissipation of sharp features is actually considered a blessing in some applications, but this is not one of them!

6.3 Velocity Extrapolation

Our fluid simulation loop looks like this:

- Make sure the fluid/air signed distance function ϕ^n is signed distance if necessary.
- Extrapolate the divergence-free velocity field from the last time step into the air region, to get an extended velocity field $\vec{u}^n$.
- Add body forces such as gravity and vorticity confinement to get an interim velocity field..
- Advect the interim velocity field and ϕ in the divergence-free $\vec{u}^n$ velocity field.
- Solve for and apply pressure to get a divergence-free velocity $\vec{u}^{n+1}$ in the new fluid region.

We now will address the second step, the extrapolation of velocity out into the air.

The simplest scheme, proposed in [EMF02], is to extrapolate out as a constant. That is, the velocity at a point in the air should be the same as the velocity at the closest point on the surface. All the points lying between this one and that closest point all have the same closest point of course, thus they should all have the same extrapolated value. This means that the directional derivative of the extrapolated velocity should be zero in the direction $\nabla\phi$:

$$\nabla\phi \cdot \nabla u = 0 \qquad \nabla\phi \cdot \nabla v = 0 \qquad \nabla\phi \cdot \nabla w = 0 \tag{6.8}$$

We've thus changed the extrapolation condition into a simple first order linear PDE. Higher order extrapolation can be done in the same way—see [Asl04].

Information about each component of velocity (or any other quantity we might want to extrapolate) should come from grid points nearer to the surface, i.e. with smaller ϕ values. Keeping that in mind, we can construct one-sided finite difference approximations to (6.8) where we only use closer neighbours. This forms a sparse linear system which, due to the condition on only using values from grid points with smaller values of ϕ, must have an adjacency structure equivalent to a directed acyclic graph. That is, if we order the unknowns by increasing ϕ values we get a lower triangular matrix which can be immediately solved by forward substition—this is in fact equivalent to doing the Fast Marching Method. For more speed an $O(n)$ topological sort of the unknowns can be used instead of a full sort based on ϕ.

6.4 Surface Tension

Our final subject is adding surface tension forces for small-scale water phenomena. For a fuller treatment including modeling of the moving contact line (where solid, air and water all meet), see [WMT05].

The physical chemistry of surface tension is conceptually simple. Water molecules are more attracted to other water molecules than air molecules, and vice versa. Thus water molecules near the surface tend to be pulled in towards the rest of the water molecules and vice versa. In a way they are seeking to minimize the area exposed to the other fluid, bunching up around their own type. The simplest linear model of surface tension can in fact be phrased in terms of a potential energy equal to a surface tension coefficient γ times the surface area between the two fluids (γ for water and air at normal conditions is approximately $0.073 J/m^2$). The force seeks to minimize the surface area.

The surface area of the fluid is simply the integral of 1 on the boundary:

$$A = \iint_{\partial\Omega} 1 \tag{6.9}$$

Remembering our signed distance properties, this is the same as

$$A = \iint_{\partial\Omega} \nabla\phi \cdot \hat{n} \tag{6.10}$$

Now we use the divergence theorem in reverse to turn this into a volume integral:

$$A = \iiint_{\Omega} \nabla \cdot \nabla\phi \tag{6.11}$$

Consider a virtual infinitesimal displacement of the surface, δx. This changes the volume integral by adding or subtracting infinitesimal amounts of the integrand along the boundary. The result infinitesimal change in surface area is:

$$\delta A = \iint_{\partial\Omega} (\nabla \cdot \nabla\phi)\delta x \cdot \hat{n} \tag{6.12}$$

Thus the variational derivative of surface area is $(\nabla \cdot \nabla\phi)\hat{n}$. Our surface tension is proportional to this, and since it is in the normal direction we can think of it in terms of a pressure jump at the air-water interface (pressure only applies in the normal direction). Since air pressure is zero in our free surface model, we have that the pressure at the surface of the water is:

$$p = \gamma \nabla \cdot \nabla\phi \tag{6.13}$$

It turns out that that $\kappa = \nabla \cdot \nabla\phi$ is termed the **"mean curvature"** of the surface, a well-studied geometric quantity that measures how curved a surface is.

The only change we need to make in our pressure solve is to change the ghost pressures in the air so that we linearly interpolate to $\gamma\kappa$ at the point where $\phi = 0$, rather than interpolating to zero. The mean curvature κ can easily be estimated at that point by using the standard central difference for the Laplacian, on trilinearly interpolated values of ϕ.

Part III

Real-Time Fluids

Chapter 7

Real-Time Simulations

With the computational power of next generation game consoles and PC's more and more physical effects – so far only seen in feature films – are now entering the realm of real-time applications such as 3-d computer games. There is, however, still a big gap between pre-computing a physical effect for many hours on a farm of workstations and simulating the effect in real-time, typically at $40 - 60$ frames per second on a single console. In order to make a simulation algorithm suitable for the application in interactive games, it needs have certain characteristics:

- **Cheap to compute**: A 3-d computer game must run at a constant rate of $40 - 60$ frames per second. From the total time available for a single frame, physically-based simulation only gets a fraction besides other tasks such as rendering or character animation. So the less computation power the algorithm consumes the better. Recently, the physics simulation task has been moved away from the CPU and onto the GPU (NVidia) or into a specially dedicated Physics Processing Unit (PPU) (AGEIA). Still, the faster an algorithm the more objects can be simulated in real-time. So low computational complexity remains one of the most important constraints.

- **Low memory consumption**: While in off-line computations, people simply buy as much memory as they need to simulate the desired effect, the memory on a game console or single PC is limited and needs to be shared among different tasks very similar to the computation time. Therefore memory efficiency of the method has to get special attention, too.

- **Stability**: In off-line simulations adaptive time-stepping can be used in situations where stability problems arise. In contrast, a game runs at a fixed frame rate. The simulation method must, therefore, be stable for the given time step size no matter what happens. External forces and the movement of boundaries can get almost arbitrarily high. In certain games, the characters move at over 100 mph for instance. A real-time simulation method needs to be able to cope with all those situations and must remain stable under all circumstances.

- **Plausibility**: Of course it is not possible to reduce the computational and spatial complexity so drastically and increase the stability so significantly without some trade off with the quality of the result. Therefore, what we require from a real-time simulation method is visual *plausibility* and "wow"-effects, not necessarily scenes that are undistinguishable from the real world.

7.1 Real-Time Water

The constraints described above point out the fact that solving the full 3-d Navier-Stokes equations is still not a practical way of simulating water in games. For the simulation of water in real-time, three main simplified methods have become popular in recent years.

- **Procedural Water** A procedural method animates a physical effect directly instead of simulating the cause of it [FR86, HNC02]. A common way to procedurally generate the surface of a lake or ocean, for instance, is to superimpose sine waves of a variety of amplitudes and directions. There are no limits to creativity when it comes to devising procedural animation techniques. Everything goes as long as the method achieves the desired visual effect. An advantage of procedural animation is its controllability, an important feature in games. The main disadvantage of simulating water procedurally is the difficulty of getting the interaction of the water with the boundary or with immersed bodies right.
- **Heightfield Approximations** If all we are interested in is the animation of the two dimensional surface of a lake or ocean, it would be an overkill to simulate the entire three dimensional body of water. In this case, only the surface is represented as a two dimensional function or heightfield and animated using the 2-d wave equation. This simplification can make the simulation orders of magnitude faster. The downside of this approach is that a function or heightfield can only represent one height value at each location on the plane. This make the simulation of breaking waves impossible. We will look into heightfield approximations in more detail in chapter 8.
- **Particle Systems** The other extreme is a small amount of water, a splashing fluid, a jet of water, small puddles or runnels. In that case, a third simplification method would be a good candidate, namely a particle system. The combination of particle-based fluids with heightfield approximations can generate an additional set of interesting effects. Particle-based methods are discussed in more detail in chapter 9.

Chapter 8

Heightfield Approximations

If we want to simulate the surface of a larger body of water such as a lake or an ocean, representing and simulating the water as a heightfield is probably the best choice. We start with what we call the *hello world* of heightfield fluids, i.e. the most simple variant of it. This simple algorithm has become quite popular in the game and demo scene because it is almost trivial to implement and yet yields amazingly realistic results. Let us assume we have a discrete array of heights `u[i,j]` of size $0 \leq i < N$ and $0 \leq j < M$ and a second array `v[i,j]` of the same size. Then the algorithm reads as follows:

```
forall i,j do u[i,j] = u0[i,j]; v[i,j] = 0;
loop
    forall i,j do v[i,j] += (u[i-1,j] + u[i+1,j] + u[i,j-1] + u[i,j+1])/4
                            - u[i,j];
    forall i,j do v[i,j] *= 0.99;
    forall i,j do u[i,j] += v[i,j];
endloop
```

The array `u[i,j]` is initialized with some interesting initial height values `u0[i,j]` other than a constant. A flat initial heightfield will remain flat throughout the simulation. The second array `v[i,j]` stores the vertical velocities of the heights. We initialize the velocities with zero although they could be initialized differently. Inside the simulation loop the velocities are first updated, then they are dampened and finally, the heights are updated using the new velocities. As the last action inside the simulation loop, the actual heights `u[i,j]` should be visualized somehow. Besides the initial conditions in the first line, boundary conditions are also needed. The first line inside the simulation loop accesses height values outside of the boundaries of `u[i,j]`. The simplest possible boundary conditions just mirror the boundary of the height array. This means we define

```
u[-1,j] = u[0,j],
u[N,j]  = u[N-1,j],
u[i,-1] = u[i,0],
u[i,M]  = u[i,M-1].
```

This definition reflects waves at the boundaries of the array.

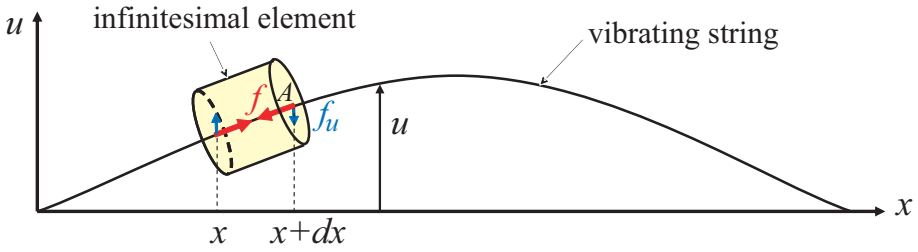


Figure 8.1: Applying Newton's second law of motion to an infinitesimal element of a string yields the 1-d wave equation $u_{tt} = c^2 u_{xx}$.

Having such a simple procedure with all the amazing results it generates is great. However, it would be nice to know where exactly this magic comes from and how this algorithm could be tuned further. This is why we will now give the mathematical background of the method.

In mathematics and physics, substances and objects are typically represented as continua first and only later, for the numerical solution of their governing equations, discretized using meshes or regular grids [Jef03]. We, thus, represent the water surface first as a continuous two-plus-one-dimensional function

$$u : (x, y), t \rightarrow u(x, y, t), \tag{8.1}$$

which represents the height of the fluid u at a every location (x, y) in the plane and at every time t. Second, we need a governing equation which tells us how the height field evolves in time given the initial heightfield $u(x, y, 0) = u_0(x, y)$. As in the full three dimensional approach, we resort to Newton's second law of motion $f = ma$ but this time adapt it to the special setting of heightfields. Because deriving the governing equation in one dimension is simpler, we first look at a one dimensional string and then generalize the situation to two dimensional membranes and water surfaces.

Let us consider the one dimensional string depicted in Fig. 8.1. We describe its deviation from the x-axis with the continuous function $u(x, t)$ which tells us how far an element at position x is moved along the u-axis at time t. We now look more closely at an infinitesimal element of the string between positions x and $x + dx$. Let us assume that the string is under constant stress σ. Then, the force acting normal to the cross section A is $f = \sigma A$. At the two positions x and $x + dx$ the surface normals of the element have opposite signs and the same is true for the internal forces. Since we want to know how these forces change the function $u(x, t)$, we are only interested in their components along the u-axis, i.e. perpendicular to the x-axis. For small deviations, these components can be approximated by

$$f_u \approx u_x \sigma A, \tag{8.2}$$

where u_x is the derivative of u with respect to x while f_u is just the component of f along the u-axis. We are now ready to formulate Newton's second law of motion for the infinitesimal string element:

$$(\rho A dx)\, u_{tt} = \sigma A u_x|_{x+dx} - \sigma A u_x|_x \tag{8.3}$$

On the left side we have the mass $\rho A dx$ times the acceleration u_{tt} which is equal to the sum of the forces acting on the two cross sections at $x + dx$ and x. By dividing both sides by $A dx$ this equation simplifies to

$$\rho u_{tt} = \sigma u_{xx}. \tag{8.4}$$

Now, if we substitute $c = \sqrt{\sigma/\rho}$ we get the one dimensional wave equation

$$u_{tt} = c^2 u_{xx}. \tag{8.5}$$

By substitution one can easily verify that $u(x,t) = a \cdot f(x+ct) + b \cdot f(x-ct)$ is a solution of the wave equation for arbitrary functions $f()$. This general solution shows that particular solutions of the wave equation are waves travelling at speed c in both directions. The scalar c, thus, represents the speed of sound or the wave speed within the given material. Intuitively, the wave equation tells us that material in positively curved regions is accelerated upwards while material in negatively curved regions is accelerated downwards. The wave equation naturally generalizes to higher dimensions. For heightfield fluids, we are particularly interested in the two-dimensional case

$$u_{tt} = c^2(u_{xx} + u_{yy}), \tag{8.6}$$

which is the specific two-dimensional form of the more general equation $u_{tt} = c^2\Delta u$ or $u_{tt} = c^2\nabla^2 u$. This is a second order partial differential equation (PDE) telling us how the two-plus-one-dimensional heightfield $u(x,y,t)$ evolves in time. To solve it, we first replace it by two first order PDEs in time:

$$u_t = v \tag{8.7}$$

$$v_t = c^2(u_{xx} + u_{yy}), \tag{8.8}$$

We, then, discretize this system using a regular spatial grid of spacing h and a constant time step Δh, yielding

$$v^{t+1}[i,j] = v^t[i,j] + \Delta t c^2 \left(u[i+1,j] + u[i-1,j] + u[i,j+1] + u[i,j-1] - 4u[i,j]\right)/h^2 \tag{8.9}$$

$$u^{t+1}[i,j] = u^t[i,j] + \Delta t v^{t+1}[i,j] \tag{8.10}$$

We made the integration scheme more stable than a pure explicit Euler step by switching the order of the update statements and using the new velocity $v^{t+1}[i,j]$ for the computation of the new positions, which is sometimes called a semi-implicit Euler step.

Now we are almost where we started with some important differences to the *hello world* version of heightfield fluids. This version does not dampen the velocity. Dampening is important and can greatly increase the stability of the simulation. On the other hand, dampening typically removes the interesting high frequency details of an animation. We also have the meaningful parameters h, Δt and c to control the simulation. Since we used explicit Euler integration as the discretization of the time integration, our algorithm is only *conditionally* stable. In particular, the *Courant-Friedrichs-Lewy* stability condition holds which says that information must not travel more than one grid cell per time step, i.e. $c \cdot \Delta t < h$.

Also, spatial boundary conditions must be given. We already saw one possible such condition, namely the mirror or reflection condition. There, we just replicated the height values across boundaries with the result that boundaries act like walls which reflect waves. Another possibility are the so-called *periodic boundary conditions*. Here we imagine the entire simulation domain to be replicated in space like tiles. When a value beyond left side of the domain is needed, we access the cell on the very right, i.e. perform a wrap around. This way, waves which leave the simulation domain on one side reenter it on the other.

We will close this chapter about heightfield fluids with some comments on how to implement the interaction with other types of objects and more general boundaries.

- The simulation domain does not need to be rectangular. Mirror or periodic boundary conditions can be applied whenever a cell is encountered that is marked to be a boundary cell. Thus, by marking arbitrary cells as boundary cells, arbitrary domains can be simulated.

- One way to implement the interaction of heightfield fluids with particle systems is to absorb particles whenever they touch the top of a water column. In that case, a certain amount of water is added to the column and the particle is deleted. [OH95]
- Interaction of heightfields with rigid bodies can be simulated by letting rigid bodies push down water columns. The water columns below the rigid bodies need to push the bodies upward. They can do this by applying a force proportional to their represented area times the pressure of the water below the body. The pressure can be computed via the difference in height of the column and the average column height.

Chapter 9

Smoothed Particle Hydrodynamics

9.1 Simple Particle Systems

As mentioned above, for splashes, spray or water jets, particles are probably the best choice. In those simple cases it is often not even necessary to simulate the interaction of particles with themselves. We call a particle system without particle-particle interaction a *simple* particle system. Such a system can be implemented very efficiently which means that a large number of particles can be simulated in real-time. All we need is a set of N particles $0 \leq i < N$ with masses m_i, positions $\mathbf{x}_i$, velocities $\mathbf{v}_i$ and accumulated external forces $\mathbf{f}_i$. These particles are either created and initialized with meaningful positions and velocites before the simulation starts or they are generated during the simulation by *emitters*. An emitter typically generates the particles with a certain rate [particle/s] and a certain velocity distribution around a principal main direction. To make sure that particles are not only generated but also die after a certain while, they are given lifetimes. When their lifetime gets close to their maximum lifetime, they are faded out smoothly. The lifetimes can also be used to color the particles appropriately. Since there are no particle-particle interactions, only per-particle forces exists and the governing equation is a set of decoupled ordinary differential equations

$$\dot{\mathbf{x}}_i = \mathbf{v}_i \tag{9.1}$$

$$\dot{\mathbf{v}}_i = \mathbf{f}_i / m_i \tag{9.2}$$

which can be efficiently integrated using an explicit Euler integration scheme for instance.

9.2 Particle-Particle Interactions

If we want to simulate small bodies of water with particles, it is quite essential that the particles feel each other. Otherwise, they all aggregate at a single spot in a corner or along an edge. In general, an interaction force for particles has the form

$$\mathbf{f}(\mathbf{x}_i, \mathbf{x}_j) = F(|\mathbf{x}_i - \mathbf{x}_j|) \cdot \frac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|}, \tag{9.3}$$

where $F()$ is the magnitude of the force depending only on the distance between the particles. The force acts along the connection between the particles, otherwise it would introduce a torque. A popular choice is the *Lennard-Jones*

force, often used in molecular dynamics simulations

$$\mathbf{f}(\mathbf{x}_i, \mathbf{x}_j) = \left(\frac{k_1}{|\mathbf{x}_i - \mathbf{x}_j|^m} - \frac{k_2}{|\mathbf{x}_i - \mathbf{x}_j|^n} \right) \cdot \frac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|}, \tag{9.4}$$

with k_1, k_2, m and n control parameters. Popular choices are $k_1 = k_2 = k$, $m = 4$ and $n = 2$.

With N particles, there are $O(N^2)$ possible interactions which we have to evaluate. This quadratic complexity can be the show stopper. Imagine we were able to simulate ten thousand particles without interactions in real time. This means we would now have to evaluate hundred million interactions also in real-time. To avoid quadratic complexity one introduces a cutoff distance d beyond which the particles do not feel each other anymore. In order to keep the simulation stable it is important that the function $F()$ is not just cut off to zero at distance d but that it is C^0 and C^1 continuous at d as well. Now at each time step, the particles are filled into a regular grid with cell spacing d. After this, potential interaction partners for a given particle only need to be searched for in the same or in adjacent cells. If the particles are evenly distributed and the number of particles per cell is bound by a constant, the interactions can be evaluated in linear time.

9.3 SPH

The next question we ask is, can we do better than using Lennard-Jones interaction forces? Is it possible to use the Navier-Stokes equations and solve them on the particles? The answer is yes [MCG03]. The first problem we have to solve is to generate smooth, continuous fields from quantities that are only given on the particles, i.e. at discrete locations in space. The Smoothed Particle Hydrodynamics method, originally devised for the simulation of stars [Mon92], solves this problem. At its core, it defines a way to smooth discretely sampled attribute fields. This is done via so-called *smoothing kernels* $W(r)$. The kernel defines a scalar weighting function in the vicinity of the position $\mathbf{x}_i$ of particle i via $W(|\mathbf{x}-\mathbf{x}_i|)$. Defined this way, the kernel is symmetric around the particle because it only depends on the distance to the particle. The kernel function also needs to be normalized meaning that $\int W(|\mathbf{x} - \mathbf{x}_i|)d\mathbf{x} = 1$. A popular choice is the *poly6* kernel

$$W_{\text{poly6}}(r) = \frac{315}{64\pi d^9} \begin{cases} (d^2 - r^2)^3 & 0 \leq r \leq d \\ 0 & \text{otherwise,} \end{cases} \tag{9.5}$$

because r only appears squared and no square root has to be evaluated. We now have all the ingredients to compute a smooth density field from the individual positions and masses of the particles:

$$\rho(\mathbf{x}) = \sum_j m_j W(|\mathbf{x} - \mathbf{x}_j|). \tag{9.6}$$

The density of particle i is then simply $\rho_i = \rho(\mathbf{x}_i)$. Now it becomes apparent why the kernels need to be normalized. With this restriction, the total mass of the system computed as the integral of the density field yields

$$\int \rho(\mathbf{x})d\mathbf{x} = \sum_j \left(m_j \int W(|\mathbf{x} - \mathbf{x}_j|)d\mathbf{x} \right) = \sum_j m_j \tag{9.7}$$

Having the density values of the individual particles, we can now compute smoothed fields A_s of arbitrary attributes A_i of the particles as

$$A_s(\mathbf{x}) = \sum_j m_j \frac{A_j}{\rho_j} W(|\mathbf{x} - \mathbf{x}_j|). \tag{9.8}$$

A nice property of this formulation is that the gradient of such a field can easily be computed by replacing the kernel by the gradient of the kernel

$$\nabla A_s(\mathbf{x}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(|\mathbf{x} - \mathbf{x}_j|). \tag{9.9}$$

In the Eulerian (grid based) formulation, fluids are described by a velocity field $\mathbf{v}$, a density field ρ and a pressure field p. The evolution of these quantities over time is given by two equations. The first equation assures conservation of mass

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0, \tag{9.10}$$

while the Navier-Stokes equation formulates conservation of momentum

$$\rho \left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} \right) = -\nabla p + \rho \mathbf{g} + \mu \nabla^2 \mathbf{v}, \tag{9.11}$$

where $\mathbf{g}$ is an external body force and μ the viscosity of the fluid. The use of particles instead of a stationary grid simplifies these two equations substantially. First, because the number of particles is constant and each particle has a constant mass, mass conservation is guaranteed and the first equation can be omitted completely. Second, the expression $\partial \mathbf{v}/\partial t + \mathbf{v} \cdot \nabla \mathbf{v}$ on the left hand side of the Navier-Stokes equation can be replaced by the substantial derivative $D\mathbf{v}/Dt$. Since the particles move with the fluid, the substantial derivative of the velocity field is simply the time derivative of the velocity of the particles meaning that the convective term $\mathbf{v} \cdot \nabla \mathbf{v}$ is not needed for particle systems.

There are three body forces (unit $[N/m^3]$) left on the right hand side of the Navier-Stokes equation modeling pressure ($-\nabla p$), external forces ($\rho \mathbf{g}$) and viscosity ($\mu \nabla^2 \mathbf{v}$). The sum of these body forces $\mathbf{f} = -\nabla p + \rho \mathbf{g} + \mu \nabla^2 \mathbf{v}$ determines the change of momentum $\rho \frac{\partial \mathbf{v}}{\partial t}$ of the particles on the left hand side. For the acceleration of particle i we, thus, get:

$$\mathbf{a}_i = \frac{\partial \mathbf{v}_i}{\partial t} = \frac{\mathbf{f}_i}{\rho_i}, \tag{9.12}$$

where $\mathbf{v}_i$ is the velocity of particle i and $\mathbf{f}_i$ and ρ_i are the body force and the density field evaluated at the location of particle i, repectively. We will now see how the body forces can be evaluated using SPH.

9.3.1 Pressure

Application of the SPH rule described in Eqn. 9.9 to the pressure term $-\nabla p$ yields

$$\mathbf{f}_i^{\text{pressure}} = -\nabla p(\mathbf{x}_i) = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(|\mathbf{x}_i - \mathbf{x}_j|). \tag{9.13}$$

Unfortunately, this force is not symmetric as can be seen when only two particles interact. Since the gradient of the kernel is zero at its center, particle i only uses the pressure of particle j to compute its pressure force and vice versa. Because the pressures at the locations of the two particles are not equal in general, the pressure forces will not be symmetric. Different ways of symmetrization of Eqn. 9.13 have been proposed in the literature. Here is a very simple solution which is stable and fast to compute:

$$\mathbf{f}_i^{\text{pressure}} = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(|\mathbf{x}_i - \mathbf{x}_j|). \tag{9.14}$$

Since particles only carry the three quantities mass, position and velocity, the pressure at particle locations has to be evaluated first. This is done in two steps. Eqn.9.6 yields the density at the location of the particle. Then, the pressure can be computed via the ideal gas state equation

$$p = k(\rho - \rho_0), \tag{9.15}$$

where k is a gas constant that depends on the temperature and ρ_0 is the environmental pressure. Since pressure forces depend on the gradient of the pressure field, the offset mathematically has no effect on pressure forces. However, the offset does influence the gradient of a field smoothed by SPH and makes the simulation numerically more stable.

However, incompressibility is not enforced strictly as in the Eulerian case. Pressure forces are only generated after the fact, when density variations have already formed. This is clearly a penalty method and yields a bouncy behavior of the fluid. To avoid this effect, one can predict the densities and compute pressure forces on predicted densities or estimate the divergence of the velocity field using SPH and then solving the Poisson equation on the particles [PTB+03].

9.3.2 Viscosity

Application of the SPH rule to the viscosity term $\mu\nabla^2\mathbf{v}$ again yields asymmetric forces

$$\mathbf{f}_i^{\text{viscosity}} = \mu\nabla^2\mathbf{v}(\mathbf{x}_i) = \mu\sum_j m_j\frac{\mathbf{v}_j}{\rho_j}\nabla^2 W(|\mathbf{x}_i - \mathbf{x}_j|). \tag{9.16}$$

because the velocity field varies from particle to particle. Since viscosity forces are only dependent on velocity differences and not on absolute velocities, there is a natural way to symmetrize the viscosity forces by using velocity differences:

$$\mathbf{f}_i^{\text{viscosity}} = \mu\sum_j m_j\frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j}\nabla^2 W(|\mathbf{x}_i - \mathbf{x}_j|). \tag{9.17}$$

A possible interpretation of Eqn. 9.17 is to look at the neighbors of particle i from i's own moving frame of reference. Then particle i is accelerated in the direction of the relative speed of its environment.

9.3.3 External Forces

External forces such as gravity, collision forces and forces caused by user interaction are applied directly to the particles as in the case of simple particle systems without the use of SPH.

9.4 Rendering

There are many ways to render fluids which are defined via particles. One of the simplest and fastest methods is to use sprites. If a smoothing filter is applied to the depth buffer before the sprites are drawn, the granularity of the particles can be hidden to a certain degree. Another way of rendering particle fluids is to draw an iso surface of the density field and triangulate it using the marching cubes algorithm.

Appendix A

Background

A.1 Vector Calculus

The following three differential operators are fundamental to vector calculus: the gradient ∇, the divergence $\nabla\cdot$, and the curl $\nabla\times$. They occasionally are written in equations as **grad**, **div**, and **curl** instead.

A.1.1 Gradient

The gradient simply takes all the spatial partial derivatives of the function, returning a vector. In two dimensions:

$$\nabla f(x,y) = \left(\frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}\right)$$

and in three dimensions:

$$\nabla f(x,y,z) = \left(\frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z}\right)$$

It can sometimes be helpful to think of the gradient operator as a symbolic vector, e.g. in three dimensions:

$$\nabla = \left(\frac{\partial}{\partial x}, \quad \frac{\partial}{\partial y}, \quad \frac{\partial}{\partial z}\right)$$

The gradient is often used to approximate a function locally:

$$f(\vec{x} + \Delta\vec{x}) \approx f(\vec{x}) + \nabla f(\vec{x}) \cdot \Delta\vec{x}$$

In a related vein we can evaluate the **"directional derivative"** of the function, that is how fast it is changing when looking along a particular vector direction, using the gradient. For example, if the direction is $\hat{n}$:

$$\frac{\partial f}{\partial n} = \nabla f \cdot \hat{n}$$

Occasionally we will take the gradient of a vector valued function, which results in a matrix (sometimes called the **"Jacobian"**). For example, in three dimensions:

$$\nabla \vec{f} = \nabla(f, g, h) = \begin{pmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial z} \\ \frac{\partial g}{\partial x} & \frac{\partial g}{\partial y} & \frac{\partial g}{\partial z} \\ \frac{\partial h}{\partial x} & \frac{\partial h}{\partial y} & \frac{\partial h}{\partial z} \end{pmatrix}$$

Note that each **row** is the gradient of one component of the function. One way to remember that it's the rows and not the columns is that it should work with the approximation:

$$\vec{f}(\vec{x} + \Delta\vec{x}) \approx \vec{f}(\vec{x}) + \nabla\vec{f}(\vec{x})\Delta\vec{x}$$

The matrix-vector product is just computing the dot-product of each row of the matrix with the vector, and so each row should be a gradient of the function:

$$\nabla(f, g, h) = \begin{pmatrix} \nabla f \\ \nabla g \\ \nabla h \end{pmatrix}$$

An alternative notation for the gradient that is sometimes used is:

$$\nabla f = \frac{\partial f}{\partial \vec{x}}$$

Using a vector in the denominator of the partial derivative indicates we're taking derivatives with respect to each component of $\vec{x}$.

A.1.2 Divergence

The divergence operator only is applied to vector fields, and measures how much the vectors are converging or diverging at any point. In two dimensions it is:

$$\nabla \cdot \vec{u} = \nabla \cdot (u, v) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}$$

and in three dimensions:

$$\nabla \cdot \vec{u} = \nabla \cdot (u, v, w) = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z}$$

Note the input is a vector and the output is a scalar.

The notation $\nabla\cdot$ is explained by thinking of it as symbolically taking a dot-product between the gradient operator and the vector field, e.g. in three dimensions:

$$\begin{aligned} \nabla \cdot \vec{u} &= \left(\frac{\partial}{\partial x}, \quad \frac{\partial}{\partial y}, \quad \frac{\partial}{\partial z}\right) \cdot (u, v, w) \\ &= \frac{\partial}{\partial x}u + \frac{\partial}{\partial y}v + \frac{\partial}{\partial z}w \end{aligned}$$

A.1.3 Curl

The curl operator measures how much a vector field is rotating around any point. In three dimensions this is a vector:

$$\nabla \times \vec{u} = \nabla \times (u, v, w) = \left(\frac{\partial w}{\partial y} - \frac{\partial v}{\partial z}, \quad \frac{\partial u}{\partial z} - \frac{\partial w}{\partial x}, \quad \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y} \right)$$

We can reduce this formula to two dimensions in two ways. The curl of a two dimensional vector field results in a scalar, the third component of the expression above, as if we were looking at the three dimensional vector field $(u, v, 0)$:

$$\nabla \times \vec{u} = \nabla \times (u, v) = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}$$

The curl of a two dimensional scalar field results in a vector field, as if we were looking at the three dimensional field $(0, 0, w)$:

$$\nabla \times w = \left(\frac{\partial w}{\partial y}, \quad -\frac{\partial w}{\partial x} \right)$$

The simple way to remember these formulas is that the curl is taking a symbolic cross-product between the gradient operator and the function. For example, in three dimensions:

$$\begin{aligned} \nabla \times \vec{u} &= \left(\frac{\partial}{\partial x}, \quad \frac{\partial}{\partial y}, \quad \frac{\partial}{\partial z} \right) \times (u, v, w) \\ &= \left(\frac{\partial}{\partial y} w - \frac{\partial}{\partial z} v, \quad pdzu - \frac{\partial}{\partial x} w, \quad pdxv - \frac{\partial}{\partial y} u \right) \end{aligned}$$

The curl is a way of measuring how fast (and in three dimensions along what axis) a vector field is rotating locally. If you imagine putting a little paddlewheel in the flow and letting it be spun, then the curl is twice the angular velocity of the wheel. You can check this by taking the curl of the velocity field representating a rigid rotation.

A vector field whose curl is zero is called curl-free, or **"irrotational"** for obvious reasons.

A.1.4 Laplacian

The Laplacian is usually formed as the divergence of the gradient (as it repeatedly appears in fluid dynamics). Sometimes it is written as ∇^2 or Δ, but since these symbols are occasionally used for other purposes, I will stick to writing it as $\nabla \cdot \nabla$. In two dimensions:

$$\nabla \cdot \nabla f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

and in three dimensions:

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

The Laplacian can also be applied to vector or even matrix fields, and the result is simply the Laplacian of each component.

Incidentally, the partial differential equation $\nabla \cdot \nabla f = 0$ is called Laplace's equation, and if the right-hand side is replaced by something nonzero, $\nabla \cdot \nabla f = q$ we call it the Poisson equation. More generally people can multiply the gradient by a scalar field a (such as $1/\rho$), like $\nabla \cdot (a \nabla f) = q$ and still call it a Poisson problem.

A.1.5 Differential Identities

There are several identities based on the fact that changing the order of mixed partial derivatives doesn't change the result (assuming reasonable smoothness), e.g.

$$\frac{\partial}{\partial x}\frac{\partial}{\partial y}f = \frac{\partial}{\partial y}\frac{\partial}{\partial x}f$$

Armed with this, it's simple to show that for any smooth function:

$$\nabla \cdot (\nabla \times \vec{u}) \equiv 0$$
$$\nabla \times (\nabla f) \equiv 0$$

Another identity that shows up in vorticity calculations is:

$$\nabla \times (\nabla \times \vec{u}) \equiv \nabla (\nabla \cdot \vec{u}) - \nabla \cdot \nabla \vec{u}$$

The Helmholtz or Hodge decomposition is the result that any smooth vector field $\vec{u}$ can be written as the sum of a divergence-free part and a curl-free part. In fact, referring back to the first two identities above, the divergence-free part can be written as the curl of something and the curl-free part can be written as the gradient of something else. In three dimensions:

$$\vec{u} = \nabla \times \vec{\psi} - \nabla\phi$$

where $\vec{\psi}$ is a vector-valued potential function and ϕ is a scalar potential function. In two dimensions this reduces to ψ being a scalar potential function as well:

$$\vec{u} = \nabla \times \psi - \nabla\phi$$

This decomposition is highly relevant to incompressible fluid flow, since we can interpret the pressure projection step as decomposing the intermediate velocity field $\vec{u}^{n+1}$ into a divergence-free part and something else which we throw away, just keeping the divergence-free part. When we express a divergence-free velocity field as the curl of a potential ψ, we call ψ the **"stream function"**.

Some more useful identities are generalizations of the product rule:

$$\nabla(fg) = (\nabla f)g + f\nabla g$$
$$\nabla \cdot (f\vec{u}) = (\nabla f) \cdot \vec{u} + f\nabla \cdot \vec{u}$$

A.1.6 Integral Identities

The Fundamental Theorem of Calculus (that the integral of a derivative is the original function evaluated at the limits) can be generalized to multiple dimensions in a variety of ways.

The most common generalization is the Divergence Theorem discovered by Gauss:

$$\iiint_{\Omega} \nabla \cdot \vec{u} = \iint_{\partial\Omega} \vec{u} \cdot \hat{n}$$

That is, the volume integral of the divergence of a vector field $\vec{u}$ is the boundary integral of $\vec{u}$ dotted with the unit outward normal $\hat{n}$. This actually is true in any dimension (replacing volume with area or length or hypervolume as appropriate). This provides our intuition of the divergence measuring how fast a velocity field is expanding or compressing: the boundary integral above measures the net speed of fluid entering or exiting the volume.

Stokes' Theorem applies to the integral of a curl. Suppose we have a bounded surface S with normal $\hat{n}$, and with boundary curve Γ whose tangent vector is τ. Then

$$\iint_S (\nabla \times \vec{u}) \cdot \hat{n} = \int_\Gamma \vec{u} \cdot \tau$$

This can obviously be restricted to two dimensions with $\hat{n} = (0, 0, 1)$. The curve integral is called the **"circulation"** in the context of a fluid velocity field.

We can also integrate a gradient:

$$\iiint_\Omega \nabla f = \iint_{\partial\Omega} f\hat{n}$$

Some of the most useful identities of all are ones called **"integration by parts"**, which is what we get when we combine integration identities based on the Fundamental Theorem of Calculus with the product rule for derivatives. They essentially let us move a differential operator from one factor in a product to the other. Here are some of the most useful:

$$\begin{aligned}
\iiint_\Omega (\nabla f)g &= \iint_{\partial\Omega} fg\hat{n} - \iiint_\Omega f(\nabla g) \\
\iiint_\Omega f\nabla \cdot \vec{u} &= \iint_{\partial\Omega} f\vec{u} \cdot \hat{n} - \iiint_\Omega (\nabla f) \cdot \vec{u} \\
\iiint_\Omega (\nabla f) \cdot \vec{u} &= \iint_{\partial\Omega} f\vec{u} \cdot \hat{n} - \iiint_\Omega f\nabla \cdot \vec{u}
\end{aligned}$$

Replacing $\vec{u}$ by ∇g in the last gives us one of Green's identities:

$$\begin{aligned}
\iiint_\Omega (\nabla f) \cdot (\nabla g) &= \iint_{\partial\Omega} f\nabla g \cdot \hat{n} - \iiint_\Omega f\nabla \cdot \nabla g \\
&= \iint_{\partial\Omega} g\nabla f \cdot \hat{n} - \iiint_\Omega g\nabla \cdot \nabla f
\end{aligned}$$

A.1.7 Basic Tensor Notation

When you get into two or three derivatives in multiple dimensions it can get very confusing if you stick to using the ∇ symbols. An alternative is to use tensor notation, which looks a little less friendly but makes it trivial to keep everything straight. Advanced differential geometry is almost impossible to do without this notation. We'll present a simplified version that is adequate for most of fluid dynamics.

The basic idea is to label the separate components of a vector with subscript indices 1, 2, and in three dimensions, 3. Usually we'll use variables i, j, k, etc. for these indices. **Note**: this can get very confusing if you also are thinking of discretizing on a grid—if you want to avoid that confusion, it's often a good idea to only use Greek letters for your tensor indices, e.g. α, β, γ instead.

So the gradient of a function is $(\partial f/\partial x_1, \partial f/\partial x_2, \partial f/\partial x_3)$. This is still a bit longwinded, so we instead use the generic $\partial f/\partial x_i$ without specifying what i is: it's a "free" index.

We could then write the divergence, for example as:

$$\sum_i \frac{\partial u_i}{\partial x_i} \tag{A.1}$$

This brings us to the **"Einstein summation convention"**. It's tedious to have to write the sum symbol Σ again and again. Thus we just won't bother writing it: instead, we will assume that in any expression that contains the index i twice, there is an implicit sum over i in front of it. If we don't want a sum, we use different indices, like i and j. For example, the dot-product of two vectors $\vec{u}$ and $\hat{n}$ can be written very succintly as:

$$u_i n_i \tag{A.2}$$

Note that by expression I mean a single term or a product—it does not include addition. So this:

$$u_i + r_i \tag{A.3}$$

is a vector, $\vec{u} + \vec{r}$, not a scalar sum.

Einstein notation makes it very simple to write a matrix-vector product, such as $A\vec{x}$:

$$A_{ij} x_j \tag{A.4}$$

Note the free index in this expression is j: this is telling us the j'th component of the result. This is also an introduction to second-order tensors, which really are a fancy name for matrices: they have two indices instead of the one for a vector (which can be called a first-order tensor). We can write matrix multiplication just as easily: the product AB is

$$A_{ij} B_{jk} \tag{A.5}$$

with free indices i and k: this is the i, k entry of the result. Similarly, the outer-product matrix of vectors $\vec{u}$ and $\hat{n}$ is:

$$u_i n_j \tag{A.6}$$

Other useful symbols for tensor expressions are the Kronecker delta δ_{ij} and the Levi-Civita symbol ϵ_{ijk}. The Kronecker delta is δ_{ij}, which is actually just the identity matrix in disguise: $\delta_{ij} x_j = x_i$. The Levi-Civita symbol has three indices, making it a third-order tensor (kind of like a three dimensional version of a matrix!). It is zero if any of the indices are repeated, +1 if (i, j, k) is just a rotation of $(1, 2, 3)$, and -1 if (i, j, k) is a rotation of $(3, 2, 1)$. What this boils down to is that we can write a cross-product using it: $\vec{r} \times \vec{u}$ is just

$$\epsilon_{ijk} r_j u_k \tag{A.7}$$

which is a vector with free index i.

Putting all this together, we can translate the definitions, identities and theorems from before into very compact notation. Further more, just by keeping our indices straight we won't have to puzzle over what an expression like

$\nabla\nabla\vec{u}\cdot\hat{n}\times\nabla f$ might actually mean. Here are some of the translations that you can check:

$$(\nabla f)_i = \frac{\partial f}{\partial x_i}$$
$$\nabla = \frac{\partial}{\partial x_i}$$
$$f(x_i + \Delta x_i) \approx f(x_i) + \frac{\partial f}{\partial x_i}\Delta x_i$$
$$\frac{\partial f}{\partial n} = \frac{\partial f}{\partial x_i}n_i$$
$$(\nabla\vec{f})_{ij} = \frac{\partial f_i}{\partial x_j}$$
$$\nabla\cdot\vec{u} = \frac{\partial u_i}{\partial x_i}$$
$$(\nabla\times\vec{u})_i = \epsilon_{ijk}\frac{\partial u_k}{\partial x_j}$$
$$\nabla\cdot\nabla f = \frac{\partial^2 f}{\partial x_i \partial x_i}$$
$$\frac{\partial}{\partial x_i}\epsilon_{ijk}\frac{\partial u_k}{\partial x_j} = 0$$
$$\epsilon_{ijk}\frac{\partial}{\partial x_j}\frac{\partial f}{\partial x_k} = 0$$

The different versions of the product rule for differentiation, in tensor notation, all just fall out of the regular single variable calculus rule. For example:

$$\frac{\partial}{\partial x_i}(fg) = \frac{\partial f}{\partial x_i}g + f\frac{\partial g}{\partial x_i}$$
$$\frac{\partial}{\partial x_i}(fu_i) = \frac{\partial f}{\partial x_i}u_i + f\frac{\partial u_i}{\partial x_i}$$

The integral identities also simplify. For example:

$$\iiint_\Omega \frac{\partial u_i}{\partial x_i} = \iint_{\partial\Omega} u_i n_i$$
$$\iiint_\Omega \frac{\partial f}{\partial x_i} = \iint_{\partial\Omega} f n_i$$
$$\iiint_\Omega \frac{\partial f}{\partial x_i}g = \iint_{\partial\Omega} f g n_i - \iiint_\Omega f\frac{\partial g}{\partial x_i}$$

A.2 Numerical Methods

These notes concentrate on methods based on finite differences, which themselves boil down simply to applications of Taylor series.

Assuming a function f has at least k smooth derivatives, then

$$f(x+\Delta x) = f(x) + \frac{\partial f}{\partial x}(x)\Delta x + \frac{1}{2}\frac{\partial^2 f}{\partial x^2}(x)\Delta x^2 + \frac{1}{6}\frac{\partial^3 f}{\partial x^3}(x)\Delta x^3 + \cdots + \frac{1}{(k-1)!}\frac{\partial^{k-1} f}{\partial x^{k-1}}(x)\Delta x^{k-1} + R_k \quad \text{(A.8)}$$

The remainder term R_k can be expressed in several ways, for example:

$$R_k = \int_x^{x+\Delta x} \frac{1}{k!}\frac{\partial^k f}{\partial x^k}(s)s^{k-1}ds \quad \text{(A.9)}$$

$$R_k = \frac{1}{k!}\frac{\partial^k f}{\partial x^k}(s)\Delta x^k \quad \text{for some } s \in [x, x+\Delta x] \quad \text{(A.10)}$$

$$R_k = O(\Delta x^k) \quad \text{(A.11)}$$

Note that Δx could be negative, in which case the second form of the remainder uses the interval $[x+\Delta x, x]$. We'll generally stick with the last form, using the simple $O()$ notation, but do remember that the hidden constant is related to the k'th derivative of f—and if f isn't particularly smooth, that could be huge and the Taylor series isn't particularly useful.

A.2.1 Finite Differences in Space

Using Taylor series, we can estimate derivatives of a function from discrete samples. For example, using the Taylor series from above at $f(x+\Delta x)$, we see:

$$\frac{f(x+\Delta x) - f(x)}{\Delta x} = \frac{\partial f}{\partial x}(x) + O(\Delta x) \quad \text{(A.12)}$$

The remainder term, indicated by $O(\Delta x)$, is called the local truncation error. Reassuringly, if we reduce the spacing Δx towards zero, the error (assuming f is smooth) reduces asymptotically at roughly the same rate. Because the error only varies linearly with Δx, this is a **"first order accurate"** finite difference approximation to the derivative at x. It's also **"one-sided"**: we're using information at x and to the right $x+\Delta x$, but not information from the left.

Typically f is stored as a sequence of discrete samples on a grid; we would say f_i as short-hand for $f(x_i)$, and with $\Delta x = x_{i+1} - x_i$ the above first order one-sided finite difference is $(f_{i+1} - f_i)/\Delta x$.

However, it turns out this exact same expression is actually a more accurate estimate of the derivative at the midpoint $x + \frac{1}{2}\Delta x$. Using the Taylor series for $f(x)$ and $f(x+\Delta x)$ with respect to the midpoint, it's not hard to show that:

$$\frac{f(x+\Delta x) - f(x)}{\Delta x} = \frac{\partial f}{\partial x}(x + \frac{1}{2}\Delta x) + O(\Delta x^2) \quad \text{(A.13)}$$

or in short-hand:

$$\frac{f_{i+1} - f_i}{\Delta x} = \left(\frac{\partial f}{\partial x}\right)_{i+1/2} + O(\Delta x^2) \quad \text{(A.14)}$$

This is now **"second order accurate"**, since the exponent in the error term is a two: as the grid spacing Δx goes to zero, the error goes down asymptotically as the square—again, assuming f is smooth enough. (If f was only Lipschitz continuous, say, so it doesn't have a second derivative, then this approximation may still only be first order accurate.) It's also a **"centered"** approximation, a.k.a. a **"central finite difference"**, since the samples from the left

and right of the approximation point are equally weighted. In general central differences can be more accurate for the same number of sample points.

One-sided second order differences, or even higher order accurate finite differences (where the truncation error has a higher exponent), may be constructed without too much difficulty from writing out more Taylor series. An alternative conceptual approach is to figure out weights that give the exact answer for polynomials up to a certain degree (i.e. functions that are in fact equal to their truncated Taylor series with remainder term of zero).

Higher derivatives can also be estimated based on Taylor series, or by taking finite differences of finite differences. For example, writing out the Taylor series for f_{i+1} and f_{i-1}:

$$f_{i+1} = f_i + \left(\frac{\partial f}{\partial x}\right)_i \Delta x + \frac{1}{2}\left(\frac{\partial^2 f}{\partial x^2}\right)_i \Delta x^2 + \frac{1}{6}\left(\frac{\partial^3 f}{\partial x^3}\right)_i \Delta x^3 + O(\Delta x^4) \tag{A.15}$$

$$f_{i-1} = f_i - \left(\frac{\partial f}{\partial x}\right)_i \Delta x + \frac{1}{2}\left(\frac{\partial^2 f}{\partial x^2}\right)_i \Delta x^2 - \frac{1}{6}\left(\frac{\partial^3 f}{\partial x^3}\right)_i \Delta x^3 + O(\Delta x^4) \tag{A.16}$$

$$\tag{A.17}$$

we see that the following is a good estimate of the second derivative:

$$\frac{f_{i+1} - 2f_i + f_{i-1}}{\Delta x^2} = \left(\frac{\partial^2 f}{\partial x^2}\right)_i + O(\Delta x^2) \tag{A.18}$$

This is the standard second order accurate central difference. An alternative derivation is to construct it with the previous central finite difference formula twice:

$$\left(\frac{\partial^2 f}{\partial x^2}\right)_i \approx \frac{1}{\Delta x}\left[\left(\frac{\partial f}{\partial x}\right)_{i+1/2} - \left(\frac{\partial f}{\partial x}\right)_{i-1/2}\right] \tag{A.19}$$

$$\approx \frac{1}{\Delta x^2}\left[(f_{i+1} - f_i) - (f_i - f_{i-1})\right] \tag{A.20}$$

However, this derivation makes it a little more difficult to see that the method is indeed second order accurate.

A.2.2 Time Integration

Finite differences can also be constructed in time to estimate the time derivative of a function. Typically, however, instead of estimating the derivative from values of the function, we use our knowledge of what the true derivative is (given by an equation of motion, for example) to solve for a future time value of the function. This is **"time integration"**: solving a differential equation in time, one time step at a time.

Appendix B

Derivations

B.1 The Pressure Problem as a Minimization

Here we go through a **"calculus of variations"** argument illustrated by Batty and Bridson [BB06] that the pressure problem:

$$\begin{aligned} \nabla \cdot \frac{\Delta t}{\rho} \nabla p &= \nabla \cdot \vec{u} && \text{inside } \Omega \\ p &= 0 && \text{on } F \\ \frac{\Delta t}{\rho} \nabla p \cdot \hat{n} &= \vec{u} \cdot \hat{n} - \vec{u}_{\text{solid}} \cdot \hat{n} && \text{on } S \end{aligned} \tag{B.1}$$

is equivalent to the following energy minimization problem introduced in chapter 4:

$$\min_{\substack{p \\ p=0 \text{ on } F}} \quad \iiint_{\Omega} \tfrac{1}{2}\rho \left\| \vec{u} - \frac{\Delta t}{\rho} \nabla p \right\|^2 - \iiint_{\Omega} \tfrac{1}{2}\rho \left\| \vec{u} \right\|^2 + \iint_{S} p\hat{n} \cdot \Delta t \vec{u}_{\text{solid}} \tag{B.2}$$

Here Ω is the fluid region, and its boundary is partitioned into S, the part in contact with solids, and F, the free surface. The functional being minimized is the total work done by pressure, measured as the change in kinetic energy of the fluid plus the work done on the solid by the fluid pressure over the next time step, which accounts for all work done by the pressure. As we said, in incompressible flow there is no "elastic" potential energy, thus the pressure is fully inelastic and dissipates as much energy as possible, hence the minimization. Also as we said before, the kinetic energy of the intermediate velocity field is a constant, hence we can just drop it in the minimization.

The calculus of variations is a way to solve problems such as this, minimizing an integral with respect to a **function**, as opposed to just a finite set of variables. We'll walk through the standard method of reducing the minimization to a PDE.

Suppose that p was the minimum of (B.2). Let q be an arbitrary nonzero function in Ω that is zero on the free surface too. Then for any scalar ϵ, the function $p + \epsilon q$ is also zero on the free surface. We can define a regular single-variable function $g(\epsilon)$ as:

$$g(\epsilon) = \iiint_{\Omega} \tfrac{1}{2}\rho \left\| \vec{u} - \frac{\Delta t}{\rho} \nabla (p + \epsilon q) \right\|^2 + \iint_{S} (p + \epsilon q)\hat{n} \cdot \Delta t \vec{u}_{\text{solid}} \tag{B.3}$$

Let's expand this out to see it's just a quadratic in ϵ:

$$\begin{aligned}
g(\epsilon) &= \iiint_\Omega \tfrac{1}{2}\rho \left(\frac{\Delta t}{\rho}\nabla(p+\epsilon q) - \vec{u}\right)\cdot\left(\frac{\Delta t}{\rho}\nabla(p+\epsilon q) - \vec{u}\right) + \iint_S (p+\epsilon q)\hat{n}\cdot\Delta t\vec{u}_{\text{solid}} \\
&= \iiint_\Omega \tfrac{1}{2}\rho \left(\frac{\Delta t}{\rho}\nabla p - \vec{u}\right)\cdot\left(\frac{\Delta t}{\rho}\nabla p - \vec{u}\right) + \epsilon\rho\left(\frac{\Delta t}{\rho}\nabla q\right)\cdot\left(\frac{\Delta t}{\rho}\nabla p - \vec{u}\right) + \tfrac{1}{2}\epsilon^2\rho\left(\frac{\Delta t}{\rho}\nabla q\right)\cdot\left(\frac{\Delta t}{\rho}\nabla q\right) \\
&\quad + \iint_S p\hat{n}\cdot\Delta t\vec{u}_{\text{solid}} + \epsilon\iint_S q\hat{n}\cdot\Delta t\vec{u}_{\text{solid}} \\
&= \left[\iiint_\Omega \tfrac{1}{2}\rho\left\|\frac{\Delta t}{\rho}\nabla p - \vec{u}\right\|^2 + \iint_S p\hat{n}\cdot\Delta t\vec{u}_{\text{solid}}\right] \\
&\quad + \epsilon\left[\iiint_\Omega \tfrac{1}{2}\rho\left(\frac{\Delta t}{\rho}\nabla q\right)\cdot\left(\frac{\Delta t}{\rho}\nabla p - \vec{u}\right) + \iint_S q\hat{n}\cdot\Delta t\vec{u}_{\text{solid}}\right] \\
&\quad + \epsilon^2\left[\iiint_\Omega \tfrac{1}{2}\rho\left\|\frac{\Delta t}{\rho}\nabla q\right\|^2\right]
\end{aligned} \tag{B.4}$$

Since we assumed p was the minimum, $\epsilon = 0$ is the minimum of $g(\epsilon)$, thus $g'(0) = 0$. In other words, the coefficient of the linear term in the quadratic must be zero:

$$\begin{aligned}
&\iiint_\Omega \rho\left(\frac{\Delta t}{\rho}\nabla q\right)\cdot\left(\frac{\Delta t}{\rho}\nabla p - \vec{u}\right) + \iint_S q\hat{n}\cdot\Delta t\vec{u}_{\text{solid}} = 0 \\
&\quad\Rightarrow\quad \iiint_\Omega \nabla q\cdot\left(\frac{\Delta t}{\rho}\nabla p - \vec{u}\right) + \iint_S q\hat{n}\cdot\vec{u}_{\text{solid}} = 0
\end{aligned} \tag{B.5}$$

Integrating by parts transforms this into:

$$-\iiint_\Omega q\nabla\cdot\left(\frac{\Delta t}{\rho}\nabla p - \vec{u}\right) + \iint_{F\cup S} q\left(\frac{\Delta t}{\rho}\nabla p - \vec{u}\right)\cdot\hat{n} + \iint_S q\hat{n}\cdot\vec{u}_{\text{solid}} = 0 \tag{B.6}$$

Now, q is zero on the F part of the boundary, so we're left with:

$$-\iiint_\Omega q\nabla\cdot\left(\frac{\Delta t}{\rho}\nabla p - \vec{u}\right) + \iint_S q\left(\frac{\Delta t}{\rho}\nabla p - \vec{u} + \vec{u}_{\text{solid}}\right)\cdot\hat{n} \tag{B.7}$$

And finally we observe that we showed this to be true for an arbitrary function q. Since these integrals always evaluate to zero no matter what q we stick in, then whatever is multiplying q must actually be zero. And that's precisely the original PDE (B.1)!

Bibliography

[Asl04] Tariq D. Aslam. A partial differential equation approach to multidimensional extrapolation. *J. Comp. Phys.*, 193:349–355, 2004.

[Bat67] G. K. Batchelor. *An Introduction to Fluid Dynamics*. Cambridge University Press, 1967.

[BB06] Christopher Batty and Robert Bridson. Accurate irregular boundaries in fluid simulation. In preparation, 2006.

[Ben92] D. Benson. Computational methods in Lagrangian and Eulerian hydrocodes. *Comput. Meth. in Appl. Mech. and Eng.*, 99:235–394, 1992.

[BFA02] R. Bridson, R. Fedkiw, and J. Anderson. Robust treatment of collisions, contact and friction for cloth animation. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 21:594–603, 2002.

[BR86] J. U. Brackbill and H. M. Ruppel. FLIP: a method for adaptively zoned, particle-in-cell calculuations of fluid flows in two dimensions. *J. Comp. Phys.*, 65:314–343, 1986.

[CGFO06] N. Chentanez, T. G. Goktekin, B. E. Feldman, and J. F. O'Brien. Simultaneous coupling of fluids and deformable bodies. In *Proc. ACM SIGGRAPH/Eurographics Symp. on Comput. Anim. (to appear)*, 2006.

[CMT04] Mark Carlson, Peter J. Mucha, and Greg Turk. Rigid fluid: animating the interplay between rigid bodies and fluid. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 23:377–384, 2004.

[Cor05] Richard Corbett. Point-based level sets and progress towards unorganised particle based fluids. Master's thesis, UBC Dept. Computer Science, 2005.

[EFFM02] D. Enright, R. Fedkiw, J. Ferziger, and I. Mitchell. A hybrid particle level set method for improved interface capturing. *J. Comp. Phys.*, 183:83–116, 2002.

[EMF02] Douglas Enright, Stephen Marschner, and Ronald Fedkiw. Animation and rendering of complex water surfaces. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 21(3):736–744, 2002.

[Fed02] R. Fedkiw. Coupling an Eulerian fluid calculation to a Lagrangian solid calculation with the ghost fluid method. *J. Comput. Phys.*, 175(1):200–224, 2002.

[FF01] Nick Foster and Ronald Fedkiw. Practical animation of liquids. In *Proc. SIGGRAPH*, pages 23–30, 2001.

[FR86] A. Fournier and W. T. Reeves. A simple model of ocean waves. In *Proc. SIGGRAPH*, pages 75–84, 1986.

[FSJ01] R. Fedkiw, J. Stam, and H. Jensen. Visual simulation of smoke. In *Proc. SIGGRAPH*, pages 15–22, 2001.

[GFCK02] F. Gibou, R. Fedkiw, L.-T. Cheng, and M. Kang. A second-order-accurate symmetric discretization of the Poisson equation on irregular domains. *J. Comp. Phys.*, 176:205–227, 2002.

[GHD03] Olivier Génevaux, Arash Habibi, and Jean-Michel Dischler. Simulating fluid-solid interaction. In *Graphics Interface*, pages 31–38, 2003.

[GSLF05] E. Guendelman, A. Selle, F. Losasso, and R. Fedkiw. Coupling water and smoke to thin deformable and rigid shells. *ACM Trans. Graph. (SIGGRAPH Proc.)*, 24(3):973–981, 2005.

[Gue06] E. Guendelman. *Physically-Based Simulation of Solids and Solid-Fluid Coupling*. PhD thesis, Stanford University, June 2006.

[HAC74] C. W. Hirt, A. A. Amsden, and J. L. Cook. An arbitrary Lagrangian-Eulerian computing method for all flow speeds. *J. Comput. Phys.*, 14(3):227–253, 1974.

[Har63] F. H. Harlow. The particle-in-cell method for numerical solution of problems in fluid dynamics. In *Experimental arithmetic, high-speed computations and mathematics*, 1963.

[HBW03a] Ben Houston, Chris Bond, and Mark Wiebe. A unified approach for modeling complex occlusions in fluid simulations. In *ACM SIGGRAPH Technical Sketches*, 2003.

[HBW03b] Ben Houston, Chris Bond, and Mark Wiebe. A unified approach for modeling complex occlusions in fluid simulations. In *SIGGRAPH 2003 Sketches & Applications*. ACM Press, 2003.

[HK05] Jeong-Mo Hong and Chang-Hun Kim. Discontinuous fluids. *ACM Trans. Graph. (Proc. SIGGRAPH)*, 24:915–920, 2005.

[HNB+06] BEN HOUSTON, MICHAEL B. NIELSEN, CHRISTOPHER BATTY, OLA NILSSON, and KEN MUSETH. Hierarchical rle level set: A compact and versatile deformable surface representation. *ACM Trans. Graph.*, 2006.

[HNC02] D. Hinsinger, F. Neyret, and M.P. Cani. Interactive animation of ocean waves. In *Proc. ACM SIGGRAPH/Eurographics Symp. Comp. Anim.*, pages 161–166, 2002.

[HW65] F. Harlow and J. Welch. Numerical Calculation of Time-Dependent Viscous Incompressible Flow of Fluid with Free Surface. *Phys. Fluids*, 8:2182–2189, 1965.

[JBS06] M. Jones, A. Bærentzen, and M. Sramek. 3D distance fields: A survey of techniques and applications. *IEEE Trans. Vis. Comp. Graphics*, 2006.

[Jef03] A. Jeffrey. *Applied Partial Differential Equations*. Academic Press, 2003.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proc. ACM SIGGRAPH/Eurographics Symp. Comp. Anim.*, pages 154–159, 2003.

[MI05] R. Mittal and G. Iaccarino. Immersed boundary methods. *Annu. Rev. Fluid Mech.*, 37:239–261, 2005.

[Mon92] J. J. Monaghan. Smoothed particle hydrodynamics. *Annu. Rev. Astron. Astrophys.*, 30:543–574, 1992.

[MST+04] M. Müller, S. Schirm, M. Teschner, B. Heidelberger, and M. Gross. Interaction of fluids with deformable solids. *J. Comput. Anim. and Virt. Worlds*, 15(3–4):159–171, 2004.

[NFJ02] Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. Physically based modeling and animation of fire. *ACM Trans. Graph. (Proc. SIGGRAPH)*, pages 721–728, 2002.

[OF02] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit Surfaces*. Springer-Verlag, 2002. New York, NY.

[OH95] J. O'Brien and J. Hodgins. Dynamic simulation of splashing fluids. In *Computer Animation*, pages 198–205, 1995.

[PTB+03] Simon Premoze, Tolga Tasdizen, James Bigler, Aaron Lefohn, and Ross Whitaker. Particle–based simulation of fluids. In *Comp. Graph. Forum (Eurographics Proc.)*, volume 22, pages 401–410, 2003.

[REN+04] N. Rasmussen, D. Enright, D. Nguyen, S. Marino, N. Sumner, W. Geiger, S. Hoon, and R. Fedkiw. Directable photorealistic liquids. In *Proc. of the 2004 ACM SIGGRAPH/Eurographics Symp. on Comput. Anim.*, pages 193–202, 2004.

[Set96] J. Sethian. A fast marching level set method for monotonically advancing fronts. *Proc. Natl. Acad. Sci.*, 93:1591–1595, 1996.

[SRF05] Andrew Selle, Nick Rasmussen, and Ronald Fedkiw. A vortex particle method for smoke, water and explosions. *ACM Trans. Graph. (Proc. SIGGRAPH)*, pages 910–914, 2005.

[Sta99] Jos Stam. Stable fluids. In *Proc. SIGGRAPH*, pages 121–128, 1999.

[TRS05] D. Tam, R. Radovitzky, and R. Samtaney. An algorithm for modeling the interaction of a flexible rod with a two-dimensional high-speed flow. *Int. J. Num. Meth. Eng. (in press)*, 2005.

[Tsa02] Yen-Hsi Richard Tsai. Rapid and accurate computation of the distance function using grids. *J. Comput. Phys.*, 178(1):175–195, 2002.

[Tsi95] J. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Trans. on Automatic Control*, 40:1528–1538, 1995.

[UMRK01] H. S. Udaykumar, R. Mittal, P. Rampunggoon, and A. Khanna. A sharp interface Cartesian grid method for simulating flows with complex moving boundaries. *J. Comput. Phys.*, 174(1):345–380, 2001.

[WMT05] Huamin Wang, Peter J. Mucha, and Greg Turk. Water drops on surfaces. *ACM Trans. Graph. (Proc. SIGGRAPH)*, pages 921–929, 2005.

[Yu05] Zhaosheng Yu. A DLM/FD method for fluid/flexible-body interactions. *J. Comput. Phys.*, 207(1):1–27, 2005.

[ZB05] Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Trans. Graph. (Proc. SIGGRAPH)*, pages 965–972, 2005.

[ZH04] Q. Zhang and T. Hisada. Studies of the strong coupling and weak coupling methods in FSI analysis. *Int. J. Num. Meth. Eng.*, 60(12):2013–2029, 2004.

[Zha05] Hongkai Zhao. A fast sweeping method for Eikonal equations. *Math. Comp.*, 74:603–627, 2005.