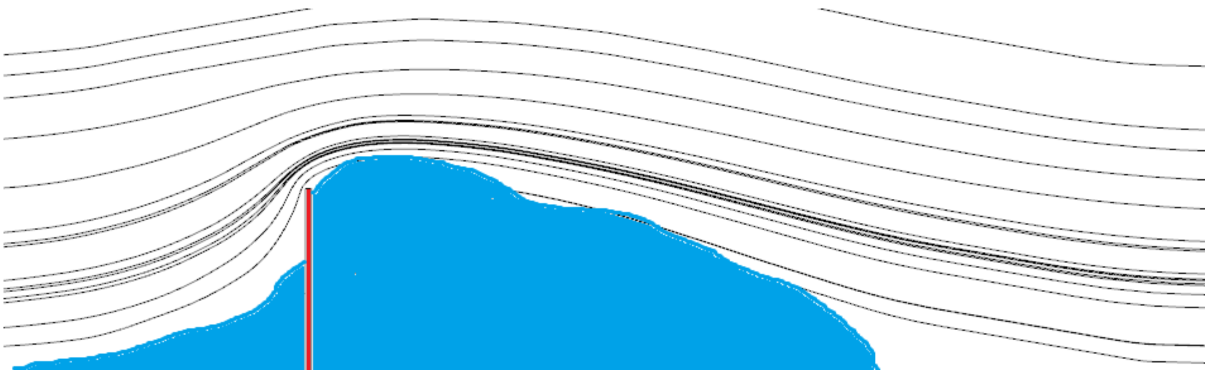
Particle-based simulation of snow drifting in an Eulerian wind field

Master's thesis in computer science - 2011



By Thomas Alexander Grønneløv[1] - August 2011

[1]TAG@Greenleaf.Dk

Abstract

In this work we simulate snow drift formation, as it is affected by a non-trivial wind field. This could be to predict the drift formation in residential areas around houses and other obstacles, or it could be to optimize the placing of snow drift fences in the landscape. The granularity of the snow flow is abstracted away by viewing it as a non-Newtonian fluid flow. This is done using a rheological material model, which is discretized using the Smoothed Particle Hydrodynamics (SPH) method.

Simultaneously an Eulerian fluid flow field, representing the wind, is embedded in the domain, and simulated using the Finite Volume Method.

To properly handle the interaction between the two methods, at their common interfaces, we have developed a method, which lets the snow influence the wind field, and which lets the wind field exert forces upon the snow.

There exist regions where snowfall, and the winds are commonly very strong. At such regions it is very beneficial to be able to quickly determine if a new building may end up changing the wind patterns, in such as way that extra snow will be deposited in undesirable locations. Simulation may additionally assist in the optimal placement of snow fences.

We have no knowledge of snow drifting being simulated using SPH previously. While models of flowing avalanches have often dealt with SPH, it appears to be a new method for drifting snow.

We have collected or derived the the relevant material characteristics of snow, and implemented a model with a more solid physical foundation than what is commonly seen in computer graphics, dealing with the visualization of wind driven snow scenes. At the same time it is a more general model than what is generally seen in snow engineering.

Our implementation of the snow simulator has been implemented in a massively parallel fashion, and programmed in CUDA, in order for it to run on a GPU.

Contents

1 Motivation

Late in the winter 2010, the Danish island of Bornholm was hit by a severe snow storm. Over a the course of a few days more than 1.4 meters of snow fell. At the time of this snow fall, a strong wind was blowing. This combination resulted in snow drifts which commonly measured more than 3 meters in height and which at several locations were as high as 6 meters. All roads were closed and even in the cities, huge drifts were blocking the streets for days.

Looking at images of this situation, it was obvious that though much snow had fallen, and though high winds were blowing, there were great variations in snow depths and drift sizes inside the cities, and on the country roads. At certain locations regular snow traps were seen, and here the drifts reached the 6 meters in height. When such a drift was located across a road, it was a big problem, as seen in Figure 1 and Figure 2. When it was away from houses and roads, it bothered no one.

Looking back, in Denmark in the 1980's and earlier, it was common practice to place snow fences (section 12.9.4) of the type seen in Figure 3 across fields to stop blowing snow from moving farther downwind, and potentially blocking a road. That practice was later suspended, apparently due to a long series of winters with little or no snow.

The placement of structures, such as houses relative to dominant wind directions during the winter, could make a difference in how strongly affected a small city would be by a snow storm. Also the optimal placement of snow fences could mitigate the effect of snow drifting at the country side and let the roads be less affected.

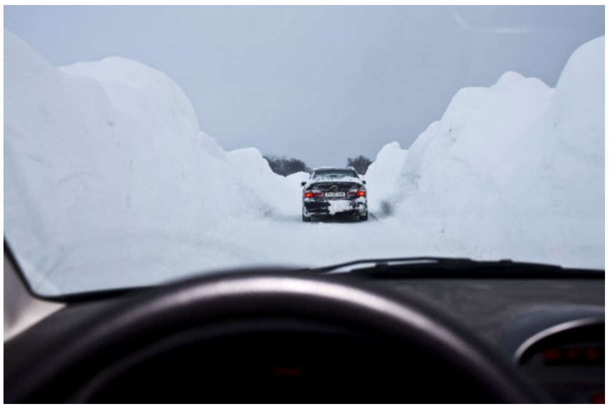


Figure 1: A road has finally been opened. Notice the walls of snow at the side of the road.[TV2 Bornholm]

To the best of our knowledge, drifting snow is not considered in house placement today, and the location of snow fences, when used, is based purely on rules of thumb such as described in [Tabler, 1991]. We therefore see a use for computer software, which can simulate the movement of snow, as it is affected by the wind, and as the wind is affected by the snow, in the presence of structures such as buildings and snow fences. This could help in the planning phase of combating drifting snow.

2 Introduction

This thesis consist of three major parts. First we consider snow as just that, snow. We look at how it is formed, its different types and how the snow moves. After a brief material mechanics introduction, we look at the material properties of snow - how it relates

Figure 2: Regular snowplows, large trucks with front mounted plows, were unable to go through the meter high drifts and excavation machines had to literally dig their way through, one shovel load at a time.[TV2 Bornholm]

Figure 3: A snow fence placed on a field to stop blowing snow from moving father downwind.[expo-net.dk]

to external stress and when it deforms or ”breaks”. Following this, we look at how the flow of snow can be modeled using rheology[2], and how the snow is both a solid and a flowing liquid, depending on the circumstances.

After this general snow investigation, we introduce the Smoothed Particle Hydrodynamics method, which is especially well suited to simulate highly deformable materials, such as wind blown, and flowing snow. The model is a general model, and will be introduced as such. Based on the knowledge we now have about snow, and a method suitable for modeling deformable materials on a computer, we describe a particle based snow simulation method, and we verify its correct behavior in relation to our current understanding of how snow behaves.

The second part will be dealing with the simulation of moving air, by first introducing computational fluid dynamics, namely the Navier Stokes equations, and methods for solving them. We will also briefly look at the concept of turbulence. After introducing the basics, we will look at a method of discretization for solving the fluid dynamics equations on a computer. This is the Finite Volume Method which is described in general terms. Using the fluid dynamics knowledge provided in the previous, we will continue on to develop a FVM fluid simulation, and validate it by comparison with textbook results, and results from commercially available simulation software.

The third and final part deals with coupling the particle based simulation of snow with the mesh based simulation of air move-

[2]The description of flow of ”solids”

ment. We will describe a simple solution to the wind-affects-snow and snow-affects-wind problem. This coupling will then be tested to see if the resulting behavior of wind blown snow is realistic, which is the eventual goal of this project.

Following the main sections, we have a general review and conclusion, which is followed by an appendix. The appendix contains some relevant material which did not fit into the general flow of the three previous parts of this thesis.

We have attempted to draw a sharp line between theory and method. The theory sections will introduce the relevant theory in very general terms, while the method will focus on what we have done to realize the methods described, in the the context of our snow simulation problem. This does make the theory sections very general, but at the same time it allows the theoretical strong reader to skip to the method, while at the same time allowing the theory sections to be usable in other contexts than snow simulation.

This text attempts to provide some intuition for the reader unfamiliar with the subject. Rather than simply stating a list of equations, we have tried to explain what those equations mean, why that is so and how to relate to this. This has contributed to making the text slightly longer than the 100 pages used as a rule of thumb, for this size of project, but the hope is that the reader, already understanding these things, can read that text quickly, and that the reader, who does not already know it, will have an easier time understanding the details, when helped along with thorough explanations - thereby also reading the text quickly.

2.1 The reader

The reader is expected to have a basic understanding of vector calculus, and the concept of vector fields. The general idea behind partial differential equations, and how they can be solved numerically, should not be too foreign, but detailed knowledge is not a requirement.

Apart from this basic mathematical understanding, the reader should have a level of understanding of Newtonian physics and fluid dynamics at least at the level of an advanced high school graduate.

Prior knowledge of parallel programming concepts, and especially General Purpose GPU, GPGPU, programming, is not a requirement, as it is introduced briefly in the appendix, and is generally used at a somewhat basic level.

Subjects, not covered in the above mentioned prerequisite, will generally be explained in the text, or references will be given to easily accessible sources of information.

2.2 Symbols and units

The symbols used in this work are listed in table 2.2. Along with the symbol, the name, the SI unit and the first use in this text, will be listed.

Symbol	Name	Unit	First use
ρ	Density	kg/m^3	4.1
μ	Viscosity	$Pa \cdot s$	4.2
μ^{υ}	Bulk viscosity	Pa	10.1
ε	Strain	1	5.1
$\dot{\varepsilon}$	Strain rate	s^{-1}	5.1.2
γ	Shear strain	1	5.1
$\dot{\gamma}$	Shear strain rate	s^{-1}	5.4
σ	Stress	Pa	5.2
τ	Shear stress	Pa	5.2
τ_Y	Yield shear stress	Pa	6.9
E	Young's modulus	Pa	5.2
Re	Reynolds number	1	10.2
G	Shear modulus	Pa	5
υ	Poisson's ratio	1	5.2
p	Pressure	Pa	6.11
t	Time	s	-
Δt	Time step length	s	8.7
u_*	Friction velocity	m/s	4.3.2
u	Velocity	m/s	-
c	Speed of sound	m/s	6.10
a	Acceleration	m/s^2	-
F	Force	N	-
f	Force density	N/m^3	4.3.2
g	Gravitational acceleration	m/s^2	-
C_D	Coefficient of drag	1	4.2
D	Particle diameter	mm	4.2
ν	Kinematic viscosity	m^2/s	4.2
T	Temperature	C	-
ε	Integration error estimate	$Variable$	9.6
u	Displacement	m	5.1

Figure 4: A list of commonly symbols used in this text. The symbol, its name and unit, as well as first use, when at all relevant, is seen here.

3 Previous work

In the field of computer generated imagery, the creation, and rendering, of snow has been of interest for some time. Particle-based simulation of snow seems to come naturally here, likely because it has the added benefit of generating actual falling snow flakes, which can be visualized while snow is still in the process of settling. In the simple end of the scale, we have falling snow which deposits on any surface blocking the vertical fall. At the other end we have detailed fluid simulation of the wind field, as it carries the snow around the scene, before finally depositing it.

In [Fearing, 2000] falling snow is simulated with quite realistically looking results. Snow is simulated as individual snow particles, each representing a single snow flake. When the particle land on geometry, that geometry is expanded upwards and colored white in order to mimic the increased snow depth. The resulting snow surface is often too coarse to properly represent the true surface, since the extruded geometry is not refined enough. Wind is not represented here, and the snow particles are in no way physically accurate, with any sort of real material properties.

In [Saltvik et al., 2007] The Graphics Processing Unit is used to simulate snow. Again as particles. The scene is a mountain region of several square kilometers. The wind field is calculated and solved, and the affect of the wind on the snow is taken into account, but the snow does not likewise affect the wind. The particles have no material properties other than drag and when they impact with the ground, they are not moved by the wind. instead a height map of snow is manipulated.

In [Moeslund et al., 2005] wind and snow are coupled for visualization. Individual snow flakes are modeled and rendered, and the snow is falling, and is carried by the wind to form drifts. The snow flakes are only simulated individually as they fall, and upon impact with obstacles, or the ground, they contribute to the raising of a triangle mesh covering the existing geometry. The snows effect on the wind does not seem to be taken into account in this work.

In [Feldman and O'Brien, 2002] snow is not represented by individual particles, but rather as a snow density inside cells in a regular grid, which covers the entire scene. The wind field is simulated, and the snow density is convected by the wind. Depending on the wind speed, and the presence of obstacles, snow may be deposited, or continue to be carried with the wind. When snow is deposited on an obstacle, or on snow already having been deposited, the slope of the snow layer is checked to see if the snow should slide into a lower neighboring location or not. The final deposition of snow can then be used to generate a snow height field which can be rendered, or intermediate snow fields can be used to dynamically change the scene geometry used when calculating the wind field.

The previous works have been focused on generating realistically looking images of snow in a scene, and the snow behavior has been the result of hand tuning with arbitrary values to define snow which has the correct look. Other work is aimed at the engineering field where accurate predictions of snow depths and snow types are required in order to issue avalanche warnings, or acquire foreknowledge of later melt-water flooding. A brief selection of such engineering tools is presented in the following.

3.1 Existing simulation tools

While several simulators have been developed as research projects, as referenced throughout this paper, it seems evident that very few snow drift simulators have been released to the public. There are simulators which have been put to real use, but they are certainly not of a type which is suitable for small computer systems. Instead they are built for clusters and intended for simulations running for days ”Typical run time is hours to days” [Lehning et al., 2006].

The tool SNOSIM [USArmyCorpsOfEngineers, 1987] was developed for the US army's engineering corps for prediction of snow accumulation, and melting, at a particular location, the Monongahela River Basin. The army's interest in that particular location was because it was responsible for the control of two general purpose reservoirs in that area. The model is seemingly very detailed, but at the same time, it is tailor made for that single location.

Another tool is Alpine3D [Lehning et al., 2006] which is designed to predict snow accumulation for the Alpine region in Europe. Predicting snow accumulation lets the authorities issue early avalanche warnings, and it gives them a tool to perform virtual measurements of the snow depth in different areas, where physical measurements could be difficult, due to limited access. This more detailed snow knowledge lets them trigger avalanches, at a time of their own choosing, before civilians are permitted in the dangerous areas.

Other tools exist for different detail levels, but it seems they all have that in common, that they are tailor made for a very specific task, at a very specific region. It is also evident that the simulators are at a very different scale from what we are attempting in our project, since the domain is entire river basins, large areas of the Mongolian plains or large mountain regions.

The smaller scales, of a few buildings or a field with a couple of snow fences, is generally not simulated directly for snow accumulation. The focus is more on the behavior of wind, and how it directly affects structures and people in the vicinity. The current state of affairs is that small scale snow drifting is estimated based on human experience and guides such as [Tabler, 1991] in which the experience is converted into a limited number of ”rules of thumb”.

A Norwegian simulator SNOW-SIM [Bang et al., 1994] deals with snow drifting around buildings, but not as a tool for building design, but rather as a research tool to further investigate the wind-snow relationship and understand the detailed behavior.

Validation of our implementation is therefore primarily based on relevant literature, containing actual field measurements, as well as those ”rules of thumb”. The results from other academic, not publicly available, simulators are also used for comparison.

We contacted two engineers from DTU Structural Engineering (Holger Koss, Associate Professor, and Christos T. Georgakis, Associate Professor) neither of whom could mention a snow simulation tool in actual everyday use. Again there were mention of active research, but no obvious product names. Instead wind tunnels are being used, and for this purpose the DTU/FORCE technology wind tunnel is currently being refitted so it can be used for tests involving ice and snow [Georgakis, 2010].

Part I

Snow

4 Snow

When water vapor crystallizes in a cold atmosphere, ice crystals form. These crystals are initially very small and light, and can be carried by even the smallest updrafts. Eventually the crystals grow to a size where the gravitational pull will force them towards the ground.

If the air temperature is below freezing or close to it, the crystals will stay intact until they land on the ground. If, on the other hand, the lower layers of air are warmer, the crystals may melt and fall as rain. For more details on the creation of snow particles we refer the reader to [Cresseri and Jommi, 2005] and [Piskova et al., 2008].

While snow is a very complex substance, it can generally be described relatively well, under certain (many) assumptions, as an viscoplastic material [Cresseri and Jommi, 2005].

4.1 Shape of snow particles

Depending on the ambient temperature where the crystal formation is taking place, a couple of basic initial forms will be created. They are plates, needles, columns and dendrites [Piskova et al., 2008] Figure 5.

Obviously the different shapes and sizes will have different aerodynamic, and mechanical properties, but as the snow falls, melts and refreezes and eventually impacts some solid surface, and is compressed by gravity of blown by the wind, the individual snow particles tend towards a common shape which is a granular material [Cresseri and Jommi, 2005]. This granular material consists of small round(ish) corns of ice with varying density ρ depending on grain sizes. The density, mass per unit volume, of the ice in the snow is equal to that of ordinary ice, but the presence of air in between crystals make the snow density much lower.

We will therefore in this project consider snow particles as small spheres of ice, when dealing with individual particles. When dealing with a collection of particles, we will consider the snow as a granular flowing material which can be approximated by a continuum as described in [Cresseri and Jommi, 2005] and section 7.

4.2 The forces acting upon snow particles

In [Zhang and Huang, 2008] the primary forces acting on a particle are given as F_g being the gravitational pull downwards, and F_D which is the drag when the particle moves relative to the wind. Additionally there exist aerodynamic lift, Magnus forces, when the particle rotates, and electrical forces. Only the first two will be considered in this project since it has been shown that while the last three forces do exist, their effect is generally two orders of magnitude smaller than the first two[Zhang and Huang, 2008].

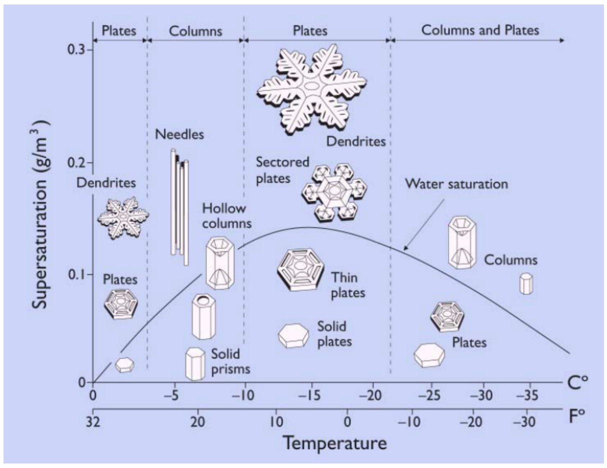


Figure 5: Depending on the ambient temperature, and supersaturation, different shapes and sized of snow particles will be created. Figure from [Piskova et al., 2008]

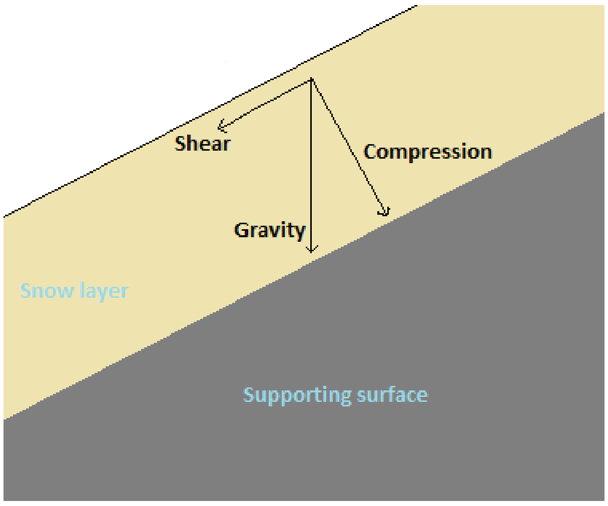


Figure 6: A snow layer, drawn in yellow, rests on an inclined surface in gray. The gravitational acceleration pulls the snow vertically down, but given the slope of the surface, this force can be split into a shearing and a compressive force.

The resulting vertical force F_g from gravitational pull, and buoyancy from the surrounding air, is defined in [Zhang and Huang, 2008] as

$$F_g = \frac{1}{6}\pi D_P^3(\rho_P - \rho_{air})g \tag{1}$$

where D_p is the particle diameter, g is the gravitational acceleration, which is set to $9.82m/s$, ρ_P is the density of the particle, which is that of ice $900kg/m^3$, and ρ_{air} is the density of the air.

As seen in Figure 4.2, the vertical gravitational acceleration force will result in both a force normal to the plane of the surface, a compressive force, and a force tangential to the plane of the surface, which is a shearing force. More about those two forces in section 5.

The drag is

$$F_d = \frac{1}{8}C_D\rho\pi D_P^2u^2 \tag{2}$$

where C_D is the coefficient of drag in air, ρ_{air} is the density of air and u is the relative velocity between the particle and the air.

The coefficient of drag is given by the following relation, which is commonly used for spherical particles of various small sizes [MIT, 2006, Nishimura and Sugiura, 1998].

$$C_D = \frac{24}{Re_p} + \frac{6}{1+\sqrt{Re_p}} + 0.4 \tag{3}$$

Here Re_p is the particle Reynolds number which can be further defined as

$$Re_p = \frac{D_p}{\nu} u \tag{4}$$

where ν is the kinematic viscosity of the air. A particle Reynolds number describes how air (or some other fluid) flows around the particle. For low Reynolds numbers Stokes law can be used to calculate the drag while high Reynolds numbers require a model taking turbulence into account.

We will use the particle size of 1mm to somewhat match the common density of new snow at $80 kg/m^3$ and corresponding diameter described in [Zhang and Huang, 2008] for new snow. The density of the individual grain of snow is naturally that of ice and not that of the new fallen snow, which contains much trapped air.

4.3 The different transport modes of snow

There are three forms of wind induced snow transportation [Zhang and Huang, 2008]. They are suspension, saltation and creeping. Snow may also slide on surfaces, such as inclined roofs, but this is primarily a gravity induced motion.

Initially the snow is suspended in the air and is carried with the air, as it itself moves. After the snow has fallen, it may be moved by creeping, which is when snow particles are moved by the wind shear, but the particles remain on the surface. They slide or roll on top of the other snow particles.

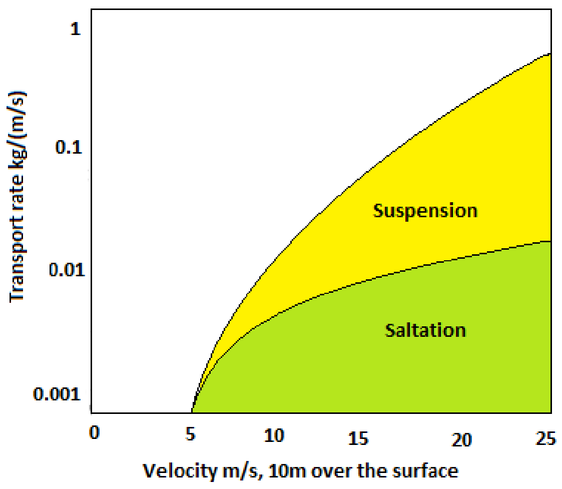


Figure 7: Distribution of snow transport in the two primary modes given wind velocity. Note how transport starts at 5m/s and initially only consist of saltation, as the wind speed increases, suspension will become a more equal partner in snow movement. Figure reworked from [Lieberherr and Parlange, 2010]

Saltation is a more energetic form of shear driven movement whereby the particles are ejected into the air by the wind, or other impacting particles, and travels briefly with the air before falling back to the snow surface again. When impacting, the particle may rebound, and it may knock other particles into the air and cause a steady flow of bouncing particles near the surface. The primary form of snow transportation is saltation [Meller, 1975, MIT, 2006] which is depicted in Figure 7.

The snow can transition from any of the three modes into any of the other ones.

4.3.1 Suspension

Snow particles generally have a very high drag to mass ratio. The individual flakes, or corns, have a large surface area relative to their mass, so they are greatly influenced by changes in wind. Suspension is not the snow transport which moves the most mass, but it is the method by which the snow travels the greatest distances. Additionally, it is the method of transport which is initially in effect and it should be considered in some detail.

When snow is suspended in air, it has the lowest density it will ever have. According to [Meller, 1975] the density of a snow filled air volume, in even the strongest blizzard, is so close to that of pure air, that suspended snow for all intents and purposes can be considered to have the viscosity and density of the air. Given that the density is very close to that of air, the suspended snow is not considered to be affecting the motion of the air itself. The air, however, has a great influence on the snow [Nishimura and Sugiura, 1998].

A particles susceptibility [MIT, 2006] describes how much it is affected by the wind, relative to gravity. The higher the susceptibility, the easier it is carried with the wind, and prevented from falling straight to the ground. This value of susceptibility (to wind transport) is simply the relationship between the force imparted by the wind F_{wind} and the force from gravity $F_{gravity}$

$$susceptibility = \frac{F_{wind}}{F_{gravity}} \tag{5}$$

The actual movement of the suspended particles is defined by the forces working on it F_g and F_d section 4.2 and the motion of the air. Close to the ground, and several meters up, the air tends to be quite turbulent which means that the wind movement is chaotic. This in turn results in a chaotic and seemingly random drag force on the particles, which in turn gives the particle a chaotic motion.

4.3.2 Saltation

Initially there will be no saltation when a slow wind blows over the snow surface. Then at a certain limiting velocity, the wind shear will set particles in motion, and the saltation is initiated in a cascade like fashion [MIT, 2006] until an equilibrium point when the mass of saltating particles remains constant. What this limiting velocity is, depends on the particle Reynolds number Re_p, but no definite value seems to be given in the literature.

The abruptness of this cascade depends largely on how well sorted the particles are. If they are of identical size, then, when first started, saltation quickly reaches the equilibrium. Very heterogeneous particle collections, on the other hand, builds up from the initial start more slowly, since some particle will be more prone to saltation than others. This in turn means that with changing wind speeds, the saltation will also be changing, but with a delay.

According to [MIT, 2006] it is not completely understood what forces initiate and drives saltation, but consensus is that it is initiated by aerodynamic drag and lift, and then driven by the same forces plus particles impacting each other, when returning to the surface. Additionally, aerodynamic lift has less of an effect on saltation than the impact from other particles.

The saltation starts at, what is called the aerodynamic threshold, and it stops at impact threshold. The two domains overlap in a large hysteresis region, so there is a large span of air velocities where the particles may equally well be be at rest and saltation.

While suspended snow has little influence on the air, saltating particles have a strong combined effect which acts as an increased roughness of the surface. Every time a particle is lifted into the wind, it extracts momentum from the air which is in part deposited into the surface as impact heat. The literature does not mention this increased heat having any effect, but the drag on the air is considered in for example [Nishimura and Sugiura, 1998, MIT, 2006, Zhang and Huang, 2008].

According to [Zhang and Huang, 2008], the drag on the air in a saltating snow layer can be described as

$$f_x = -\frac{d}{dy}\left(\rho k^2 y^2 \left[\frac{du}{dy}\right]^2\right) \tag{6}$$

where f_x is the horizontal force density, force per unit volume, k is the dimensionless Von Karman's constant of 0.41, y is the hight above the surface and u is the horizontal wind speed. Note that while obviously any relative movement between snow and wind will result in drag, we are considering a wind blowing tangential to the surface of the snow, which means that the wind velocity normal to the surface is negligible - when not considering turbulence.

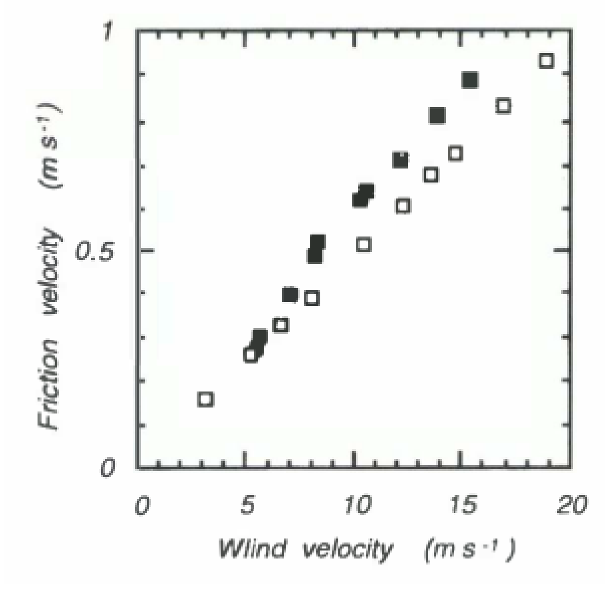


Figure 8: Friction velocity over a snow layer based on wind speed. Figure from [Nishimura and Sugiura, 1998]

According to [MIT, 2006] the particles takeoff speed is around the same value as the friction velocity, and the takeoff angle is around 21° to 25°. In experiments [MIT, 2006] this speed has been measured to $0.5m/s$ for a wind speed of $10m/s$ and it has been reported that for wind speeds below $20m/s$ the relationship between wind speed and friction velocity is somewhat linear

$$u_* = \frac{u}{20} \tag{7}$$

The term friction velocity, also called shear velocity, using the symbol u_*, describes the relationship between fluid induced shear

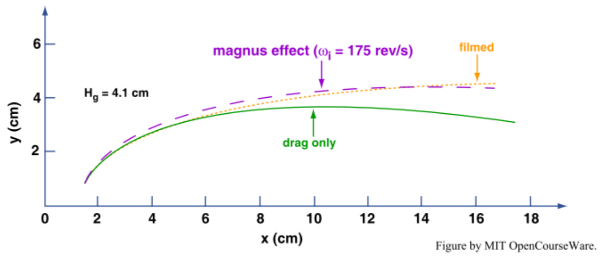


Figure 9: Trajectory of particles during saltation[MIT, 2006]

stress τ at a wall, and the fluid density ρ, in the form of a velocity (8). The higher the density, the less friction velocity is required to account for the shear stress.

$$u_* \equiv \sqrt{\frac{\tau}{\rho}} \tag{8}$$

According to [Zhang and Huang, 2008] the number of ejected particles per unit area and unit time can be modeled as

$$N = \alpha e^{\frac{u_*}{\beta}} + \lambda \tag{9}$$

where $\alpha = 0.083m$, $\beta = 0.19m/s$ and $\lambda = 0.00052m$.

It is not clear what size snow particles the expression for ejected particles is valid for. It appears that it is for a particle mix with an average particle diameter $D = 0.5mm$ and no other diameters are explicitly mentioned. We therefore make the assumption that is it valid for all common snow particle sizes, and use it as it is given.

We now rewrite it to give the mass ejected from the surface per unit area and unit time, here assuming a particle diameter, as the one they report, and snow as being spherical ice balls.

$$m = \left(\frac{4}{3}\pi\left[\frac{D}{2}\right]^3 900kg/m^3\right)(\alpha e^{\frac{u_*}{\beta}} + \lambda) \tag{10}$$

It should emphasized that this relation is at best a rough estimate and that it requires a great deal of calibration to be accurate. This is due to the assumption of entirely loose snow and, on the fact that while friction velocity seems to depend linearly on wind speed, this is an average. The higher the wind speed, the more turbulence and the more turbulence the larger variance of surface wind speed and thereby friction velocity. What this means is that while the average friction velocity may develop linearly, the variance does not, and it is those occasional higher than average friction velocities which rip up particles.

Performing the described calculations for a friction velocity of $0.5m/s$ we get 3 particles ejected per second per square meter. Note that this ejection rate is for entirely loose snow and that no relationship is given between density and ejection rate.

4.3.3 Creep

While suspended snow is important initially, when the snow falls, and while saltating snow is important when describing how snow is primarily transported after falling, creep is not important. It is commonly not modeled, since it accounts for so low a fraction of the total snow transport. We briefly touch upon its creeping properties in more details in section 6.1.

4.3.4 Sublimation

Sublimation is the evaporation of the ice in the snow directly into vapor form, without first being liquid water.

In the model [Mott et al., 2010] they initially ignore sublimation, claiming that it is irrelevant, while they later conclude that it is in fact quite important, if the final accumulated snow mass is to be used. In [Lieberherr and Parlange, 2010] they state that the effect is in fact large, but only on suspended snow.

While this should probably be implemented in a model dealing with snow deposits, the trouble with sublimation, and probably the reason it is generally *not* included in snow models, is that there are next to no details about how the mass loss through sublimation is to be calculated. It is clear that it is wind speed related as well as it is dependent on the humidity of the air and the temperature of both the snow and the air. What is not clear is exactly how those parameters are related.

We considered to add a mass loss factor to the snow, which should represent sublimation. The factor should relate to the relative speed of the air and the snow particles. That was dropped because it would at best be a hack, and because there would be no clear cut way to test if the model was accurate or not, given the lack of details about sublimation. For that reason, we will recognize its relative importance, but not implement it in this project.

4.3.5 Sliding

Gravity induced sliding is commonly not modeled, with the exception of avalanches. The mass of snow, which can accumulate on a sloped surface, does however depend on sliding, so if this is relevant to the model, then sliding should be included. In considering sliding, the static and dynamic friction between the snow and the surface, on which it rests, should be known. This is not modeled in this project.

5 Introduction to material properties

This section serves as a brief introduction to material strength mechanics. We will touch upon strain and stress and the related concepts. We briefly look at what it is, and how it is described. In order to relate strain and stress we also look at Young's elastic modulus, shear modulus and Poisson's ratio. We will then look very briefly at friction and viscosity. Any reader familiar with those topics can safely skip this section.

5.1 Strain

Strain is a dimensionless number. It is defined as the *relative* displacement between particles inside an object. While a rigid body motion - a non deforming translation or rotation - displaces particles in an object, the relative displacement is zero.

Strain can therefore be described as change in displacement over some direction at a point.

If a 1D object, aligned with the x-axis, with the initial length L, is stretched to the new length l by keeping one end fixed while the other is moved away, then it is strained, and the strain is positive, while a similar compression would be negative strain.

The different parts of the object have been strained equally, though they have not been displaced equally. The fixed end has not moved, and its displacement u is zero. The other end has been displaced $l - L$. This means that displacement u changes along the x-axis.

This can be be described though a continuous displacement function $x = X + u(X)$ defining a continuous displacement field u where x is the new position of the part of the object previously located at X.

Strain can then be defined as $\varepsilon = \frac{\partial u}{\partial x}$ meaning that it is dependent on how much the displacement changes along the spatial dimensions. That happens to be the x-axis here. In the simple example, we assume that this rate of change of displacement over the x-axis is a constant and that there is not one particular section of the object which stretches more than the rest.

In three dimensions the strain cannot be described using a single number. Not only is there normal strain along x,y and z, but there is shearing strain as well. This is described through a 3x3 strain tensor.

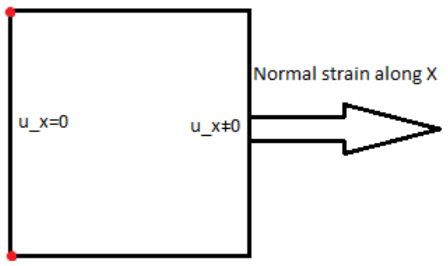


Figure 10: Normal strain. The leftmost edge is fixed, and the rightmost edge is moved to the right along the X-axis. The fixed edge is the X-plane since its normal vector is the X-vector

$$\varepsilon = \begin{bmatrix} \varepsilon_{xx} & \varepsilon_{xy} & \varepsilon_{xz} \\ \varepsilon_{yx} & \varepsilon_{yy} & \varepsilon_{yz} \\ \varepsilon_{zx} & \varepsilon_{zy} & \varepsilon_{zz} \end{bmatrix} \tag{11}$$

The different components of the strain tensor are normal strain and shear strain. Normal strain is a strain which acts normal to the plane which it acts on, as seen in 2D in Figure 10. Shear strain is a strain which acts parallel to the plane which it acts on as seen in Figure 11. Normal strain is compression or stretch along some axis while shear strain is more of a skewing strain, commonly arising from two surfaces grinding against each other. Strain caused by friction is shear strain.

The notation will be that strain acting on plane α along the vector β is written as $\varepsilon_{\alpha\beta}$ By α-plane is meant the plane which has α as its normal vector.

5.1.1 Calculating strain

There are several possible ways of calculating strain, which differ in how they well

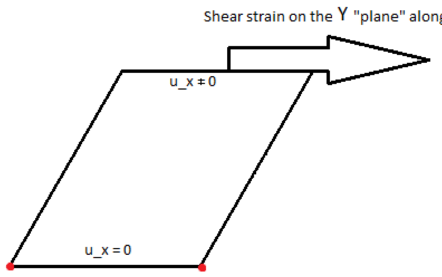


Figure 11: Shear strain. The bottom edge is fixed and the top edge is moved to the right along the X-axis. The fixed edge is the Y-plane since its normal vector is the Y-vector

they handle large deformations. That is, if strain is to be determined given an original and a deformed material configuration, and the two are very different, then special methods such as the Green Lagrangian strain tensor[Bonet and Wood, 2008], also known as the large strain tensor, may have to be used.

For small deformations, or for large deformations taken in many in small steps, the direct definition of strain as $\frac{\partial u}{\partial x}$ can be used. That is commonly refereed to as the small strain tensor or the Cauchy strain tensor [Müller et al., 2008]. The reason that is is only for small strain is that large strain will rarely consist of a purely linear displacement through the entire strain.

A rigid body rotation is an example where it will fail. This is illustrated in Figure 12 and Figure 13, where a square is rotated clockwise. In Figure 12 it is *assumed* that the motion of the corners does not change during the ”deformation” and their initial motion is extrapolated linearly into the future. The result is a deformed, much larger, square. In Figure 13, this linear behavior is not assumed and instead the true circular motion is used. This results in zero deformation and thereby zero strain.

Cauchy strain will calculate strain as

$$\varepsilon_C = \frac{1}{2}(\nabla u + \nabla u^T) \tag{12}$$

Note that while this tensor is for small strain, it can be used for large strain in a time discrete physical simulation, *if* that large strain consist of small strain in each individual simulation step and the material is not elastic so that it has stress trying to return it to some initial configuration. This is what is done in [Paiva et al., 2009, Hosseini et al., 2007], and this is what we will be doing.

5.1.2 Strain rate

While strain is relative deformation, strain rate is relative deformation, or strain, over time $\frac{\partial \varepsilon}{\partial t}$. Where strain is calculated from displacement differentiated over space, strain *rate* is calculated from velocity differentiated over space, where velocity itself is displacement differentiated over time. A common symbol for strain rate is $\dot{\varepsilon}$ and the unit is s^{-1}.

We can calculate the strain rate at any point from the gradient of the velocity field.

$$\nabla v = \begin{bmatrix} \frac{\partial v_x}{\partial x} & \frac{\partial v_x}{\partial y} & \frac{\partial v_x}{\partial z} \\ \frac{\partial v_y}{\partial x} & \frac{\partial v_y}{\partial y} & \frac{\partial v_y}{\partial z} \\ \frac{\partial v_z}{\partial x} & \frac{\partial v_z}{\partial y} & \frac{\partial v_z}{\partial z} \end{bmatrix} \tag{13}$$

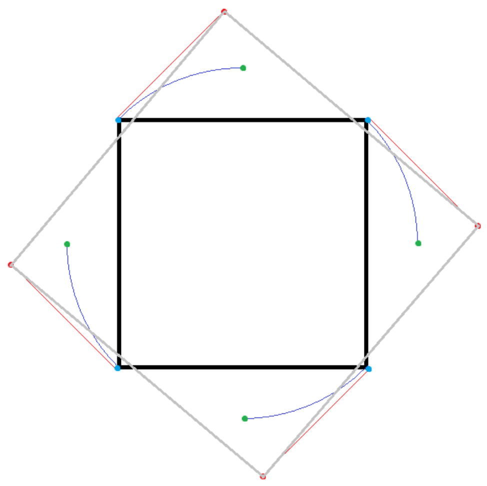


Figure 12: Large strain from rigid body rotation. The black square rotates and the red lines are linear extrapolations of the corners movement, while the blue lines are the true circular motion of the corners. Note how the ”rotated” figure, with linear extrapolations of the corners make the square grow to a much larger size - indicating large normal strain.

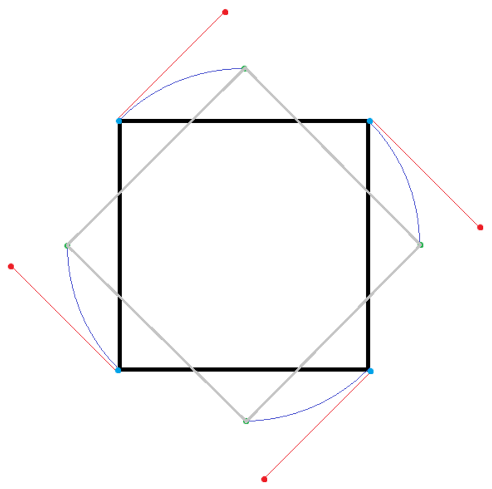


Figure 13: Large strain from rigid body rotation. The black square rotates and the red lines are linear extrapolations of the corners movement, while the blue lines are the true circular motion of the corners. Note how the rotated figure, using the true circular motion of the corners, maintain the shape and size of the original indicating zero strain.

Given the gradient, we can calculate the strain rate

$$\dot{\varepsilon} = \frac{1}{2}(\nabla v + \nabla v^T) \tag{14}$$

5.2 Stress

Stress σ is the force per unit area inside a deformable object. It is similar to strain in that it is also defined by a $3x3$ tensor relating its values to the coordinate system. It is however here describing force per area, or pressure, and has the unit of Pascal which in turn is N/m^2.

$$\sigma = \begin{bmatrix} \sigma xx & \sigma xy & \sigma xz \\ \sigma yx & \sigma yy & \sigma yz \\ \sigma zx & \sigma zy & \sigma zz \end{bmatrix} \tag{15}$$

If we consider an object subjected only to normal strain, then the relationship between strain and stress is defined by a material constant known as Young's elastic modulus designated E below.

$$\sigma_{\alpha\alpha} = E\varepsilon_{\alpha\alpha} \tag{16}$$

where the subscript $\alpha\alpha$ designates the diagonal, which is the normal components of stress and strain.

For more complicated deformation with both normal and shear strain in 3D, Young's modulus is not enough to fully describe the relationship. Here an elasticity tensor D can instead be used

$$\sigma = D\varepsilon \tag{17}$$

The elasticity tensor, also known as stiffness tensor, D encapsulates several material properties which can be described through Young's modulus E and Poisson's ratio ν [Müller et al., 2008]. In 3D it can be written as`Dette forenkles med poisson=0`

$$\begin{array}{c} \mathtt{xx} \\ \mathtt{yy} \\ \mathtt{zz} \\ \mathtt{xy} \\ \mathtt{yz} \\ \mathtt{zx} \end{array} = \frac{E}{(1+\nu)(1-2\nu)} \begin{bmatrix} 1-\nu & \nu & \nu & 0 & 0 & 0 \\ 0 & 1-\nu & \nu & \nu & 0 & 0 \\ 0 & 0 & 1-\nu & \nu & \nu & 0 \\ 0 & 0 & 0 & 1-2\nu & 0 & 0 \\ 0 & 0 & 0 & 0 & 1-2\nu & 0 \\ 0 & 0 & 0 & 0 & 0 & 1-2\nu \end{bmatrix} \begin{array}{c} \mathtt{xx} \\ \mathtt{yy} \\ \mathtt{zz} \\ \mathtt{xy} \\ \mathtt{yz} \\ \mathtt{zx} \end{array}$$

In the previous, we used the two material constants, Young's modulus E and Poisson's ratio ν. Young's modulus describes the stress strain relationship between normal strain and normal stress, much the same way as Hooks spring constant. Poisson's ratio, on the other hand, [Bonet and Wood, 2008] introduces the relationship between how much an object is compressed, or stretched along one axis, and how much it bulges out or thins in the plane orthogonal to that axis.

$$\nu = -\frac{d\varepsilon_{transversal}}{d\varepsilon_{axial}} \tag{18}$$

An entirely incompressible material will have a Poisson's ratio of 0.5, which will maintain volume through deformation. A material which do not bulge out when compressed, will have a ratio of 0. One example of such an entirely compressible material is cork, though that is naturally not true for an *arbitrarily high* compression.

If an incompressible material, with Poisson's ratio of 0.5, is subjected to compression along the y axis, it will bulge out in the xz-plane.

$$\nu = -\frac{d\varepsilon_{xz}}{d\varepsilon_y}$$
$$d\varepsilon_{xz} = -\nu d\varepsilon_y \qquad (19)$$

Shear stress Another common material constant is the shear modulus G, which is related to Young's modulus E and Poisson's ratio ν.

$$G = \frac{E}{2(1+\nu)} \qquad (20)$$

Looking at the definition of shear strain section 5.1, it is clear that if a material is subjected to normal strain, and bulges out to the sides due to a non zero Poisson's ratio, then some shearing strain will result, and it's those shearing forces which causes the sideways movement of the material. At the same time, we see that if a material is sheared, then individual particles inside the material will be pulled away from each other similar to normal strain. Depending on the problem being solved, the shear modulus or Young's modulus may be more appropriate to use. Both of which should be in play when calculating certain material properties such as the speed of shock waves in section 6.10

5.3 Friction

Friction forces are forces which oppose the movement of a material sliding over another material. The friction between two sliding objects will dissipate kinetic energy into heat, and eventually cause the sliding action to stop. There are two main variations of friction which are static friction and dynamic friction, or kinetic friction. Static friction is the friction force resisting the initiation of relative movement, while dynamic friction is the force opposing current relative movement.

The friction force depends on the normal force F_n, which is the force pushing the two sliding surfaces together, and the friction quotient μ which can be either the static quotient μ_s or the dynamic quotient μ_k. Commonly the normal force is related directly to the mass of the sliding object and the gravitational constant $F_n = mg$, but other forces than gravity may be pushing the sliding surfaces together.

The magnitude of the friction force is calculated as

$$|F| \leq \mu F_n \qquad (21)$$

5.4 Viscosity

Viscosity can be considered internal friction in a fluid object. As the particles in a fluid moves relative to each other, there is friction between them. This friction is described through a viscosity parameter μ. Viscosity also translates relative movement into heat, eventually causing movement to cease.

Where "normal" friction did not depend on the speed of the relative motion, viscosity forces do. The faster the movement, the larger the viscosity forces, resisting that movement, will be.

Relative movement in an incompressible, or nearly incompressible, fluid will only take the form of shearing strain γ and the resulting viscosity forces will therefore be shearing stress τ. The relationship between shear stress τ, shear strain rate $\dot{\gamma}$ and viscosity μ is written as

$$\tau = \mu\dot{\gamma} \tag{22}$$

6 Mechanical properties of snow

The mechanical properties of snow are not easily defined, and no complete material description currently exist. According to [Meller, 1975] and [Piskova et al., 2008] most of the general measurements were performed early on during the 1940's and 1950's and after this, research tended to focus on either the fluid like properties of snow during avalanches, or the motion of snow due to wind forces.

In summary, most of the known material properties of snow are limited to a very specific type of snow at a very specific temperature range and a very specific snow density. For this reason, snow simulations tend to make a lot of assumptions about the type of snow falling and the environment into which it falls. The common approach is to assume grain like snow corns, variable density and constant temperature [Cresseri and Jommi, 2005].

When simulating snow drifting, the trend is to focus on the wind-induced motion of the snow [Mott et al., 2010] and not on the mechanical behavior of snow itself. We also realize the relative importance of the two subjects, but still touch on the mechanical behavior since we feel it is relevant for a complete description.

A model implementing both behaviors will also be more versatile, and can be used for a broader range of modeling tasks ranging from falling snow, over drifting snow to sliding or melting snow with what this brings of trouble.

We will touch on the material properties of snow here to make it clearer what behavior is to be expected, and to motivate our choices in the development of our SPH model.

6.1 A few simple observations about snow

When falling, the snow consists of individual snow particles, snow flakes, which generally do not bond together when colliding. For snow to bond quickly the temperature must be very close to $0°C$ [Piskova et al., 2008, Lieberherr and Parlange, 2010].

This means that snow-fall consists of individual irregularly shaped flakes, which will trap a great deal of air between them when they settle. This in turn results in a very low density.

While the individual snow particle has the density of ice, the collection of loosely packed particles can have densities as low as 10 kg/m^3. This type of snow is generally called ”powder snow” [Piskova et al., 2008]. The common density for a solid bonded layer of snow is around

300 kg/m^3 [Meller, 1975], while the density of newly settled snow is lower at 100 kg/m^3. The porosity p of snow can be easily described by the following expression given in [Piskova et al., 2008].

$$p = \frac{\rho_{ice} - \rho_{snow}}{\rho_{ice}} \tag{23}$$

where p is the porosity, ρ_{ice} is the density of ice and ρ_{snow} is the density of the snow. We see that a porosity of zero is snow in the form of solid ice.

For the standard density of 300 kg/m^3 for new, but settled, snow, this would result in a porosity of $(900kg/m^3 - 300kg/m^3)/900kg/m^3 = 0.17$ This means that in such a snow cover, 83% of the volume is trapped air.

The warmer the snow is, the softer it is. The lower the density is, the softer it is [Cresseri and Jommi, 2005]. This indicates that the mechanical strength constants such as shear and bulk modulus are highly temperature and density dependent. Keep in mind that while the individual snow flake has a structural strength independent of the overall snow layers density, we are viewing the collection of particles as a continuum, and here the porosity plays a big role.

The next observation about snow behavior is that when it is compressed, it stays compressed. In other words, the deformation of snow is dominantly plastic [Cresseri and Jommi, 2005], though it has a very small elastic part. The higher the density, the greater the elastic part.

The snow behavior is often modeled as a spring and dash-pot system where the spring described the elastic part and the dash pot the plastic part, but even this coupled model does not capture the complex and nonlinear behavior [Piskova et al., 2008]. We also note that snow is more resistant to compressive stress than to tensile (pulling) stress, which can be seen by different strength constants for the different types of stress. Models such as [Cresseri and Jommi, 2005] describes this by an viscoplastic model, which seems to capture the general dynamics. In [Meller, 1975] snow is, for low pressures and densities, which is what we will be dealing with, described as a plastic solid.

In general, snow can range in type from large very soft, almost melted and glued together, particles to very small individual grains which do not stick together. It may also be tightly packed and forming a rigid, but very brittle, coherent structure.

When snow is left on its own accord, not influenced by any outside stress, other than gravity, it will slowly collapse and settle at a rate of $1cm$ to $30cm$ a day, depending on ambient temperature and the hight of the snow layer [Piskova et al., 2008]. This is caused both by creep due to gravity and by a change in the bonds between individual ice grains due to chemical water transport. Over time ice bridges will form and grow between initially un-bonded neighbor ice grains [Cresseri and Jommi, 2005]. This also has the effect that settled snow will grow stronger and stronger, but if the snow is the exited, the ice bridges break and the snow will become, and remain, week until the bonds are reformed. This makes the strength of snow very strain history dependent.

Creep under loading, external or internal, is given in [Meller, 1975], but we do not model creep explicitly since we are more concerned with the short term effects of drifting snow, whereas creeping (in the form of snow collapse) is a long term effect.

6.2 Density

The density of snow can vary enormously from a very light powder like substance to highly compressed ice. Density of one of the most relevant material parameters of snow, since most other parameters tend to depend highly on it.

From [Piskova et al., 2008] we have the following densities for different types of snow.

Snow type	Density ρ
Powder snow	10-50 kg/m^3
Hangdog by wind	100-200 kg/m^3
Firn dry	200-400 kg/m^3
Firn light	400-600 kg/m^3
Firn	300-800 kg/m^3
Ice	800-900 kg/m^3

In[Zhang and Huang, 2008] the particle size and density for three types of snow is given as

Type	Particle diameter	Density
New Snow	$\leq$ 1mm	80 kg/m^3
Fine snow	$\leq$ 0.5mm	180 kg/m^3
Old fine snow	0.5mm..1mm	230 kg/m^3

Considering that we are dealing with snow drifting, the snow will generally be low density new snow. We should therefore define a rest density for snow that has just fallen and let it evolve from here as described later in section 9.1. New snow, in the heavy end, has a density of around $\rho = 100kg/m^3$. We chose this somewhat heavy snow to let the snow quickly form the smallest of neighbor bonding, without explicitly modeling the compression of the snow due to wind forces actually smashing the crystals into each other and bonding them. Do keep in mind though, that this is just the initial rest density.

6.3 Young's modulus

Young's modulus depends on density and temperature [Meller, 1975, Cresseri and Jommi, 2005]. Other materials such as steel are also temperature dependent, but what makes this different for snow is that it is *very* temperature sensitive inside the normal range of temperatures, while steel for common use, disregarding fires, does not change drastically.

For this reason we will limit ourselves to simulations with constant temperatures. Most measurements are for snow which is dry and coherent, not slush and not fine powder.

It should be noted that the moduli for coherent snow is higher than for loose granular snow. According to [Meller, 1975] the material strength of snow increases exponentially with time as the individual grains bond closer and closer together. The loose snows strength is approximately 40% of that seen after settling for 3 weeks. This needs to be taken into account if modeling snow behavior over long periods of time, or when modeling snow fall and drifting on top of a previous and older layer of snow. It is, however, not something we will be investigating any further.

The density/modulus relationship is not linear [Meller, 1975]. It is somewhat linear in the logarithmic plot however. According to [Piskova et al., 2008], it is a valid assumption that Young's modulus is exponential (logarithmic linear) for common snow, while large density variations, as seen in avalanches, should not make this assumption. Given how Young's modulus depends on density, it is obvious that initially snow compresses very easily while further compression quickly becomes increasingly difficult.

Based on graphs of material strength testing shown in [Meller, 1975, Piskova et al., 2008] we have modeled Young's modulus as an exponential function of density for snow at $T = -20°C$ (24) which is shown graphically in Figure 14.

$$E(\rho) = 187300e^{0.0149\rho} \qquad (24)$$

but only for the common density ranges from around $\rho = 100kg/m^3$ to $\rho = 500kg/m^3$.

In the previous we talked about Young's modulus for compression, but snow is not as strong in tension, stretching, as in compression. This could let us write two values for Young's modulus, but given that tension strength has not been measured for new snow, but only for coherent, already strongly bonded, snow, we will use a tension strength of zero.

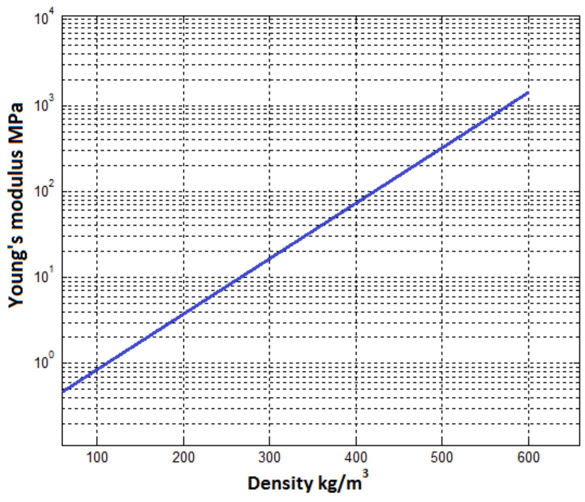


Figure 14: Young's modulus given the snow's density. Note that the modulus axis is non-linear. The plot represents our function (24)

6.4 Shear modulus

The shear modulus depends on density as strongly as Young's modulus, and a plot of the relation can be seen in Figure 15. As for Young's modulus, we base our function for shear modulus (25) on graphs from [Meller, 1975, Piskova et al., 2008]. A plot of our function is shown in Figure 15.

$$G(\rho) = 1000e^{0.0143\rho} \qquad (25)$$

which, as was the case for Young's modulus, is only valid for the common density ranges from around $\rho = 100kg/m^3$ to $\rho = 500kg/m^3$.

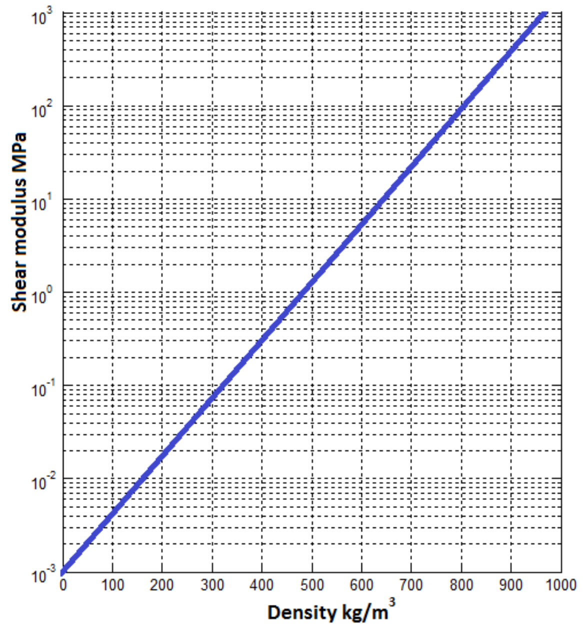


Figure 15: Shear modulus based on snow density. The modulus axis is non-linear. The plot represents our function (25)

6.5 Friction

Friction forces between snow and some surface of another material are not implemented. As shall be seen in section 9, friction between snow and some boundary will be calculated as friction between snow and snow through virtual particles.

6.6 Viscosity

According to [Meller, 1975], not only does viscous stress depend on the viscosity and the strain rate, but the viscosity itself depends on the strain rate. This is not a property which we need to model terribly accurately in our project, considering that the strain rate will generally be quite low.

From [Meller, 1975] we have an approximate relationship between viscosity and density for snow, which shows that over the range of densities $200kg/m^3 \leq \rho \leq 600kg/m^3$, the dynamic viscosity increases linearly in a logarithmic plot shown in that work.

The increase is approximately 1000 times. We have attempted to find a function which expresses this relationship as a viscosity multiplier m_μ based on an initial viscosity at an initial density of $\rho = 260kg/m^3$. This gives us

$$m_\mu(\rho) = e^{\frac{rho - 260kg/m^3}{49}} \tag{26}$$

This multiplier is 1 for $\rho = 260kg/m^3$ and 1000 for $\rho = 600kg/m^3$ and thereby matches the plot from [Meller, 1975].

The values are, however, unrelated to strain rate.

From [Nishimura and Maeno, 1989], we have some details about viscosity depending on flow rate for an avalanche at a constant density. The trouble is that the expression from that work makes the assumption that the shear strain rate in the snow is given by the linear flow velocity over the supporting surface. While this is true for *rolling* avalanches, it is not true for sliding snow drifts. We need viscosity in terms of shear strain rate.

We have chosen to use an ”avalanche viscosity” for a very slowly moving avalanche with $u = 5m/s$ an $\rho = 260kg/m^3$ to get *some* base viscosity. The viscosity model from [Nishimura and Maeno, 1989] is

$$\mu = \mu_\infty + (\mu_0 - \mu_\infty)e^{-\frac{u}{u_\mu}} \qquad (27)$$

where μ_∞ and μ_0 are empirical friction constants called viscosity at infinity and viscosity at zero. Their value is $\mu_\infty = 0.1$ and $\mu_0 = 0.5$. The flow velocity of the snow is u, and u_μ is a calibration constant set to $u_\mu = 50m/s$.

That gives us a usable base viscosity of $0.4619\ Pa \cdot s$. That approximately corresponds to motor oil at a low temperature. Obviously snow does not behave like oil, and it will only have this viscosity when already moving, and the snow is much lighter as well. More about the snows rigid behavior in section 7.

If we now, from [Meller, 1975], take the development of dynamic viscosity density changes, and from [Nishimura and Maeno, 1989] take value of dynamic viscosity for moving snow at a fixed density and fixed velocity, we can combine the two into a new expression for dynamic viscosity of snow given a density.

$$\mu(\rho) = e^{\frac{rho - 260kg/m^3}{49}} 0.4619 Pa \cdot s \qquad (28)$$

It should be somewhat obvious that viscosity is a highly non linear function in every sense of the word, and it should be noted, that while snow falls with a rest density of $\rho = 100kg/m^3$ the viscosity is defined based on a higher density $\rho = 260kg/m^3$.

This means that the viscosity multiplier will be around 0.04, and that the viscosity is very much lower. Newly fallen snow does intuitively have a low viscosity, but given that all research into snow viscosity is focused on avalanches, which have a higher density, it is unclear if we can really extrapolate the samples down into the region of snow that light.

6.7 Poisson's ratio

Poisson's ratio can ordinarily, for simpler materials, which are homogeneous, linear and isotropic, be calculated from Young's modulus and shear modulus through the following relation

$$\upsilon = \frac{E}{2G} - 1 \qquad (29)$$

Homogeneous means that the material is "the same" throughout, which is not true for snow which is ice bridges and air. Snow is generally isotropic which means that its strength is the same along every axis.

An example of a non isotropic material could be wood which is strongest along the fibers. The linearity describes that the strain stress relationship is linear, which is not true for snow which is highly density dependent. This ratio can therefore not be found from the moduli we already have, but has to be measured.

Measurements of Poisson's ratio of snow is a little uncertain, but it seems to be *very* low over the normal lower range of snow densities [Meller, 1975]. We will be using the value of $\upsilon = 0$ indicating that compressing snow will make the crystal structure crush and the air filled cavities collapse, and expel the contained air. Thereby, in a sense, some material *will* bulge out the sides, but it is air, which is no longer part of the material.

6.8 Failure strain

We have, in the literature, only been able to locate values for failure strain for normal strain. For brittle materials, the failure strain is commonly a constant, and for snow it is approximately 0.002 for all densities of snow. Young's modulus is not constant for all densities, so naturally the yield stress related to this yield strain is also not a constant.

We write the following for yield stress given the density dependent Young's modulus.

$$\varepsilon_{failure} = \frac{\sigma_{\alpha\alpha}}{E} \approx 0.002 \tag{30}$$

We will interpret this as instant failure since $\varepsilon_{failure}$ is for all intents and purposes *very* small. This essentially means that the material is either a non-elastic solid or a failed material. This matches the material model in section 7 very well, as we shall see.

6.9 Yield shear stress

Considering snow at a density of $300kg/m^3$ we have shear modulus

$$G(\rho = 300kg/m^3) = 1000e^{0.0143\rho}$$
$$G = 1000e^{4.29} = 72960Pa \tag{31}$$

Using the known failure strain of 0.002, we can calculate an *estimated* shear yield stress as

$$\tau_Y = \varepsilon_{failure} G \tag{32}$$
$$\tau_Y = 0.002(72960Pa) \tag{33}$$
$$\tau_Y = 145Pa \tag{34}$$

From [NISHIMURA and MAENO, 1987] we have that value given as 40-100 Pa for moving snow and 540-1000 Pa for hard sintered snow. As usual the physical parameters for snow vary greatly depending on a multitude of factors. We will use our method of relating yield strain to stress, since it is derived in a reasonable way, and since its result does fall inside the range of values provided in the literature. Seeing that we have a plausible model for yield stress, we write it as a function of density.

$$\tau_Y = \varepsilon_{failure} G \tag{35}$$
$$\tau_Y = 0.002(1000e^{0.0143\rho}) \tag{36}$$
$$\tau_Y = 2e^{0.0143\rho} \tag{37}$$

6.10 Speed of sound, compression waves

The speed of sound in a material[3] is related to Young's modulus. The higher the modulus, the higher the speed of sound in the material. It also depends on the Poison ratio in the opposite way. The higher Poisson's ratio, the slower the speed of sound. This makes sense since a compression wave in

[3] We need the speed of compression waves in the snow for later in section 6.11.

one direction would dissipate energy in the orthogonal direction, if the Poisson's ratio is high, while it would not for a Poisson's ratio of zero.

Note that the expression "speed of sound" in snow is little counter intuitive since it does not describe how fast a sound wave actually moves through the snow, and the air inside the snow, but rather how fast a compression wave in the crystal structure moves. For that reason we will now use the term compression wave speed or just wave speed, but we keep the common symbol c with unit m/s

The relationship of wave speed, Young's modulus, density and Poisson's ratio is written as

$$c = \sqrt{\frac{E(1-\nu)}{\rho(1-\nu)(1-2\nu)}} \qquad (38)$$

where E is Young's modulus, ν is the Poisson's ratio and ρ is the material density. We see density representing the mass and thereby the inertia of the material here.

Given that we have already found ν to be very small, and that we are using the value zero, we see that, when we assume it to be zero, we have an especially simple relation between density, Young's modulus and wave speed.

$$c = \sqrt{\frac{E}{\rho}} \qquad (39)$$

The higher the elastic modulus, the higher speed of compression waves, and the higher the density, the lower the speed of compression waves. When dealing with snow, however, Young's modulus and density are also related, as described in section 6.3 and repeated here

$$E(\rho) = 187300e^{0.0149\rho} \qquad (40)$$

This lets us rewrite the wave speed as

$$c = \sqrt{\frac{187300e^{0.0149\rho}}{\rho}} \qquad (41)$$

6.11 Pressure under compression

When a material is compressed, the local pressure increases. For an elastic material, the pressure increase will make the material return to its original volume, when the external stress is removed. The relation between pressure and other parameters such as density and temperature is called an Equation Of State, abbreviated EOS, and there are a great many different versions.

A material, such as snow, which has a very small elastic region will plastically deform, and it will not return to its original volume. This indicates that pressure does not increase permanently inside the material. Still snow does resist compression.

In [Paiva et al., 2009] the equation for pressure is given as

$$p = c^2(\rho - \rho_0) \qquad (42)$$

where p is pressure, c is the speed of compression waves in the material and ρ and ρ_0

is the current density and the rest density. This EOS is widely used because it is easy to relate to an actual material through the pressure wave speed c.

We have already in section 5 introduced a relationship between strain and stress, and should now relate that description, using Young's modulus, to the EOS here using pressure, density and pressure wave speed.

If we revisit strain, and repeat the definition of engineering strain ε_e as

$$\varepsilon_e = \frac{l-L}{L} \tag{43}$$

where l is new length and L is previous length, then we could write this in terms of density ρ with ρ being the new strained density and ρ_0 being the initial density.

$$\varepsilon_e = \frac{\rho-\rho_0}{\rho_0} \tag{44}$$

This can be done since we are talking about pressure from compression, where a compression along one diagonal axis linearly corresponds to a change in volume and therefore a change in density.

$$\begin{aligned}\rho &= \frac{m}{l_x l_y l_z}\\ \Delta\rho &= \frac{m}{\frac{l_x}{\Delta} l_y l_z}\end{aligned} \tag{45}$$

where m is the mass, l_x, l_y, l_z are the dimensions in the x,y and z directions, so the volume is $V = l_x, l_y, l_z$ and Δ is the scaling along a single arbitrary axis. If for example we have $\Delta = 2$, then the x dimension will be half of the previous value, the volume will be half and the density will be double.

The stress arising from such a density based strain ε_e multiplied with Young's modulus E, is in terms of pressure measured in Pa. The same is true for the EOS which also relates change in density to pressure in Pa. If we write them as being equal, which they *should* be if the EOS is to describe the *same* stress for the *same* strain, we have

$$\begin{aligned}\varepsilon_e E &= c^2(\rho-\rho_0)\\ \frac{\rho-\rho_0}{\rho_0} &= c^2(\rho-\rho_0)\end{aligned} \tag{46}$$

If we now isolate c in the above, we should be able to see its original definition, if the equality is to hold

$$\begin{aligned}\frac{\rho-\rho_0}{\rho_0} &= c^2(\rho-\rho_0)\\ \frac{E(\rho-\rho_0)}{\rho_0(\rho-\rho_0)} &= c^2\\ \frac{E}{\rho_0} &= c^2\\ c &= \sqrt{\frac{E}{\rho_0}}\end{aligned} \tag{47}$$

which is exactly the definition of pressure wave velocity c in a material given Young's modulus E and density ρ from section 6.10, and used throughout the literature.

While we have seen that EOS used often enough, we have never before seen anyone take the time to validate it, by thinking it through, as we just did. After this we feel fairly confident that the EOS will result in exactly the same stress, for a density variance, given a sound speed, as we would have seen using the small strain tensor and Young's modulus.

Please do note that we are talking about the small strain measure, engineering strain, here and that it is normal strain. Shearing is handled through viscosity, and yield stress as described in section 7.

This EOS therefore only describe a subset of the material behavior of snow, namely how much the material resists being compressed along any one axis.

To make this point clear, we will now rewrite the EOS as

$$p = \frac{E}{\rho_0}(\rho - \rho_0) \tag{48}$$

Note that if $\rho < \rho_0$ we will have negative pressure indicating a stretching of the material. In section 6.3 we pointed out that new snow under stretch has zero strength. This means that we should have

$$p = \begin{cases} \frac{E}{\rho_0}(\rho - \rho_0) & ; \rho \geq \rho_0 \\ 0 & ; otherwise \end{cases} \tag{49}$$

6.12 Summary

We will assume snow to be a completely plastic material which resists deformation, but which has no elastic properties. We assume perfect collapse of the crystals and use a Poisson's ratio of zero. We use the following models for Young's modulus E, and shear modulus G, as functions of density ρ. These will be based on dry coherent snow at approximately -10 degrees Celsius. The elastic moduli are only relevant for the "inelastic" snow because of how we calculate pressure wave speed and yield strength.

While temperature is assumed to be constant, we do allow for a changing density ρ, and since a change in density very much changes the material properties of snow, they are given as functions of density.

The complete summary is shown in table 6.12.

7 Rheology and granular flow

While the primary behavior of snow is described through its individual snow flakes interaction with the wind, we do need to consider how the material responds to external forces as a whole. To properly calculate how tall a snow drift will be, depends both on how well snow resists the shearing forces arising from the gravitational pull on the snow itself, as well as how well the snow resists compression.

As mentioned, snow is moved by the wind, primarily by saltation and suspension, but at the same time it is deformed by gravity. A snow drift is built by the wind, but it does not flatten out completely once the wind stops blowing. The snow is kept at its drift shape

Density dependent material properties for snow	
Initial rest density	$\rho_0 = 100kg/m^3$
Young's modulus	$E(\rho) = 187300e^{0.0149\rho}Pa$
Shear modulus	$G(\rho) = 100e^{0.0143\rho}Pa$
Poisson's ratio	$\upsilon(\rho) = 0$
Viscosity	$\mu(\rho) = e^{\frac{rho-260kg/m^3}{49}}0.4619Pa \cdot s$
Shear yield stress	$\tau_Y(\rho) = 2e^{0.0143\rho}$
Speed of sound	$c(\rho) = \sqrt{\frac{183700e^{0.0149}}{\rho}}m/s$

Figure 16: Summary of density dependent material properties used for snow. All of them were derived from graphs or point samples in cited papers.

by internal stress counteracting the external force of gravity.

Cohesion-less snow can be described as a granular flow, where there is sliding friction between the individual particles, but where those particles do not stick together.

Snow can be described as a non-Newtonian fluid using rheological material models. Rheology is for example used in describing soil movement and avalanches. It captures the material behavior in a few key material values which is the yield stress of the material, and the strain rate dependent viscosity when flowing.

7.1 Non-Newtonian flow

A Newtonian fluid is a fluid which cannot resist shear stress and always remain fluid with a constant viscosity regardless of the forces acting on it. If subject to shear stress, the fluid will continue to flow. This means that a Newtonian fluid placed over a plane, and subjected to gravity will flatten totally. This is only true to a degree since surface tension will in fact counteract this flattening, but that is only relevant on the very small scale.

A non-Newtonian fluid, on the other hand, may resist some amount of shear stress. Placed over a plane and subjected to gravity, it may form a pile of the material which collapses until the point where the shear stress, induced by gravity, is not strong enough to overtake the internal yield stress τ_Y in the material. When the material comes to rest, the slope of the pile of material is termed the friction angle, and the higher the yield strength, the steeper the slope. For snow this friction angle has been reported to be around $63°$ [Cresseri and Jommi, 2005] for snow with a density of around $300kg/m^3$.

The reason for the behavior of Newtonian fluids and non-Newtonian fluids has to do with the material properties of the fluid itself. Primarily the particle size and the particle-particle friction as well as particle-particle attraction.

When considering granular flow, the particles are at a macroscopic level, while a fluid, such as water has microscopic particles. If

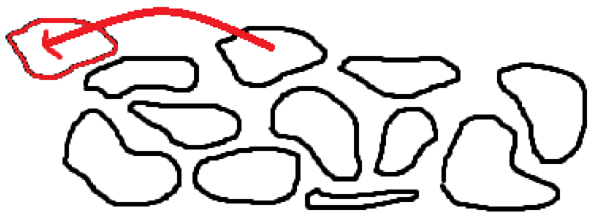


Figure 17: A particle tries to ”flow” over the other particles in a granular material

we consider a granular material, in which a single particle tries to ”flow”, as seen in Figure 17.

We note that the particles form irregular surfaces which grind towards each other when the particles move. As one particle attempts to slide to the left, it needs to rise up from its place among the other particles, slide over another particle and then click into place at a position farther to the left.

This requires energy, and it is the reason there is a certain yield strength which describes how much force must be applied to make the particles move. If the force pushing the particle is removed before the particle is moved enough to fall into a new local minimum, the particle will move back into its original position, and it will be an elastic deformation.

It is clear that, when first the particles are in motion, there is no stationary hole that the particles need to climb out of in order to move. The particles will be bouncing around, and it will take little force to continuously pull a single particle through this vibrating collection of particles.

This is the reason that the viscosity is reduced as a function of shear strain rate. If we have a stable pile of granular material and we then start to vibrate the material, it will also start to flow like a Newtonian fluid, and it will flow until it is just a single layer of particles.

A Newtonian fluid such as water will have much smaller particles. So small, in fact, that the vibration from the heat of the fluid will be enough to bounce the particles around and make it flow easily. It is clear that, as the temperature drops, the willingness to flow will be reduced and vice versa.

The result of this is that depending on the yield stress, a pile of snow will have a certain well defined slope as illustrated in Figure 18 and Figure 19. A high yield stress will result in a steeper slope. The angle of the slope against horizontal is called the friction angle, or angle of repose. A common friction angle for snow is 63° as reported in [Cresseri and Jommi, 2005].

7.2 Bingham plastic

There are a number of different rheological material models which describe a material with yield strength and viscous flow. They differ slightly in how the transition from solid to fluid takes place, and in how viscosity relates to strain rate. The more common models are the Bingham plastic, the Power law and Herschel-Buckley model [Capone, 2010, Hosseini et al., 2007] as well as the Generalized Newtonian Fluid model [Paiva et al., 2009].

What they all have in common is that they derive a viscosity from the shear strain rate and that they describe a certain yield shear

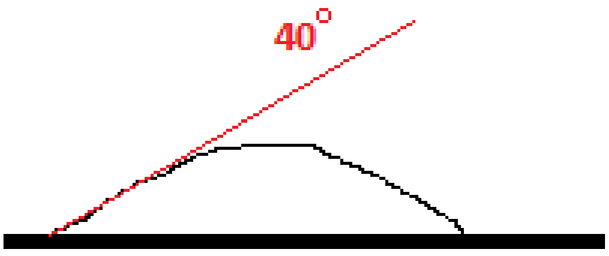


Figure 18: A pile of very loose slippery snow forming a flat pile of snow with an edge angle of 40 degrees

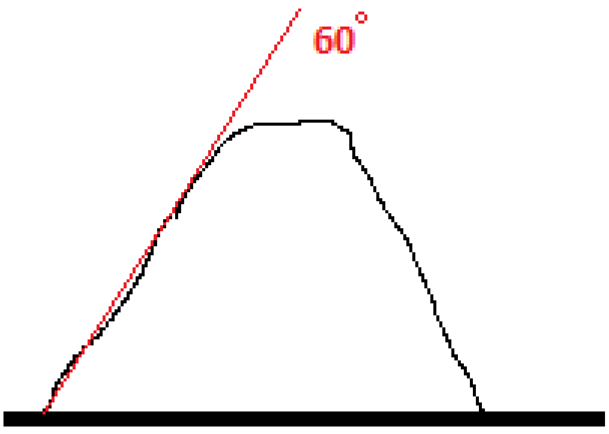


Figure 19: A pile of more compact snow forming a steep pile of snow with an edge angle of 60 degrees

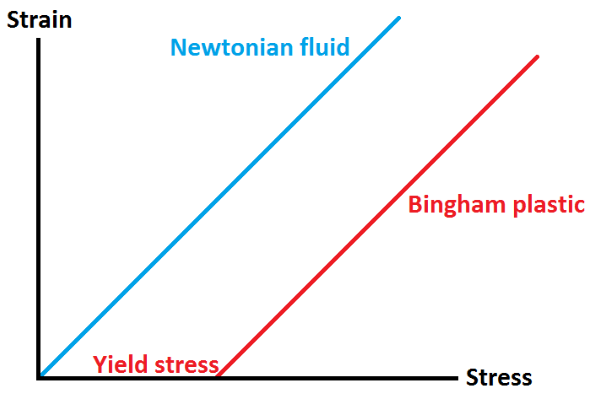


Figure 20: A Newtonian fluid and a Bingham plastic. Note how the strain first starts after a certain yield stress, and how the strain/stress relationship is then linear. The slope of the lines is equal to the viscosity of the ”fluids”

stress below which the material behaves as a solid. We will consider the Bingham Plastic model in the following.

While a Newtonian fluid has a constant stress/strain relationship, which goes through origo, zero strain and zero stress, the same is not true for a Bingham plastic. Here there is a specific yield stress which needs to be overcome before any strain occurs. After yielding, the Bingham plastic behaves like an ordinary Newtonian fluid with a linear stress strain relationship.

A principal descriptor in the Rheological model is the shear strain rate [Hosseini et al., 2007], which can be written as

$$\dot{\gamma} = \frac{1}{2}(\nabla u + \nabla u^T) \tag{50}$$

where ∇u is the gradient of the velocity field.

We also have a measure of magnitude of this shear strain rates as

$$|\dot{\gamma}| = \sqrt{\sum_{ij} \dot{\gamma}_{ij}\dot{\gamma}_{ij}} \tag{51}$$

Shear stress τ relates to shear strain rate through the following expression.

$$\tau = \mu(|\dot{\gamma}|)\dot{\gamma} \tag{52}$$

where $\mu(|\dot{\gamma})$ is a function for viscosity given the magnitude of the shear strain rate, but in our case we do not have a strain rate dependent viscosity, and therefore fall back to the expression for a Newtonian fluid, which has constant viscosity. This gives us

$$\tau = \mu_0\dot{\gamma} \tag{53}$$

We here introduce μ_0 as a value of viscosity in the fluid state, while we will be using μ as the value of viscosity in the current, solid or fluid, state.

In the Bingham plastic model, we have an entirely solid material if the shear stress is below a certain yield stress τ_Y, and a fluid if the stress is above. This is written formally as

$$|\tau| \leq \tau_Y \rightarrow \dot{\gamma} = 0 \tag{54}$$

$$|\tau| \geq \tau_Y \rightarrow \tau = \big(\frac{\tau_Y}{|\dot{\gamma}|} + 2\mu\big)\dot{\gamma} \tag{55}$$

which is simply stating that if shear stress is below yield, the stress is what it is, but the strain is zero - the material is rigid - and that if shear stress is above yield, then the material break and a new expression exist for the shear stress, and the material is now a fluid.

It is not absolutely correct to model the material as behaving in a discontinuous way, as just described. There is an elastic part where the shear stress is still below the yield stress, but where the strain is not zero. Here the material will behave as an elastic.

The shear strain possible, before enough shear stress is built up to overcome the yield stress, is however so small that it is commonly disregarded. In the case of snow we have already in section 6 observed that the failure strain is very small indeed.

One way to handle the solid part of the material may be to use two different viscosities. One very high for the ”solid” part and one lower describing the already yielded and fluid like material. The ”solid” viscosity is modeled as $\alpha = 100$ in [Paiva et al., 2009], where α is a viscosity scalar.

$$|\dot{\gamma}| \leq \frac{\tau_Y}{2\alpha\mu} \rightarrow \tau = 2\alpha\mu\dot{\gamma} \tag{56}$$

$$|\dot{\gamma}| > \frac{\tau_Y}{2\alpha\mu} \rightarrow \tau = \big(\frac{\tau_Y}{|\dot{\gamma}|} + 2\mu\big)\dot{\gamma} \tag{57}$$

What this does, is essentially seeing if the viscosity, and strain rate induced stress, is below the yield stress by rewriting the inequality $2\alpha\mu|\dot{\gamma}| \leq \tau_Y$. If it is, the material is ”solid” which means we will be using the high viscosity $\alpha\mu$.

One could argue that a highly viscous material is not the same as a rigid material, and that when modeled as viscous, the scalar should not be $\alpha = 100$, but rather $\alpha = \infty$.

While that is a valid point, snow is never really rigid. As described in section 6 snow will slowly deform under constant stress, and increase in density over time, due to creep. Because of this, it is not an entirely unrealistic assumption that rigid snow is actually a highly viscous flow. Additionally, solving differential equations with infinite stress for any strain is not feasible.

In other implementations of rheological models such as [Capone, 2010, Hosseini et al., 2007, Paiva et al., 2009] this $\alpha = 100$ has been found to be reasonable.

To use the previous equations, we need to settle on some viscosity. We have from section 6 a viscosity and we will use this for the fluid phase and $\alpha\mu$ with $\alpha = 100$ for the "solid" creeping phase.

7.3 Summary

The equations describing the rheological model in short form is given here. It is clear that the rheology in this work only deals with finding a current viscosity, high or low. This will be used in section 9.

$$\begin{aligned}
\dot{\gamma} &= \frac{1}{2}(\nabla u + \nabla u^T) \\
|\dot{\gamma}| &= \sqrt{\sum_{ij} \dot{\gamma}_{ij}\dot{\gamma}_{ij}} \\
\tau_Y(\rho) &= 2e^{0.0143\rho} \\
\mu_0(\rho) &= e^{\frac{rho-260kg/m^3}{49}} 0.4619 Pa \cdot s \\
\mu(\rho, \dot{\gamma}) &= \begin{cases} 100\mu_0 & ; |\dot{\gamma}| \leq \frac{\tau_Y}{200\mu_0} \\ \mu_0 & ; otherwise \end{cases} \qquad (58)
\end{aligned}$$

(59)

8 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics, hereafter SPH, is a so called Lagrangian method - as opposed to an Eulerian method. Where an Eulerian method defines the space in which a material moves, and does not track any part of the substance in particular, but rather keeps track of how much is located at a particular position in space, and what the properties is of the material currently at that position; a Lagrangian method deals explicitly with a part of the material and tracks this material through space and time.

A helpful analogy for an Eulerian method is a grid of weather stations located on the ground. As the weather moves across the landscape, each weather station monitors the current weather at its own position in the world. A Lagrangian method, on the other hand, would consist of a number of weather balloons which float in the air, and travel with the weather. Each balloon tracks the

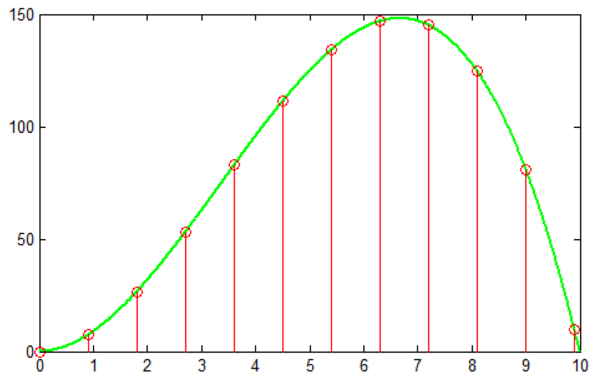


Figure 21: A continous green function and its point sample discretization

weather as it moves *with* that weather. At the same time it tracks its own position, so that it can relate its measurements to a current position over the ground.

SPH is used to discretize Partial Differential Equations PDE, in order to solve them numerically.

Physical equations, such as the Navier Stokes equations for fluid, can only be solved analytically for very specific and simple cases. For anything non-trivial, the equations must be solved numerically, which again requires a discretization of the problem.

The general PDE describes relationships between various parameters in a continuum, and it is the transformation from a continuum description to a discrete description we call ”discretization”.

While SPH is used for physical simulation in this project, it is more general than that, and in this section we will describe it in those general terms.

SPH is an interpolation method which solves the discretization problem by interpolating the (unknown) field values in between sample points termed ”particles”. While a particle is essentially just a sample point, we will hereafter use the term particle to conform to the usual SPH naming. Each particle carry with it an influence weight, commonly a mass, and various other properties, such as velocity, temperature, pressure etc. Those quantities are defined exactly only for the specific positions of the particles. In order to obtain field values in *between* particles, the attributes of the individual particles are smoothed over space and then a sum is calculated to find the field values. An example of such a smoothing of a point sample is seen in Figure 22 where the smoothing is done by a Gaussian kernel.

The interesting thing about SPH, in a physical sense, is that the sample points need not be stationary. Rather, they can be moving with the medium being sampled, and this way they, in a sense, *are* what they are sampling.

Further more, the particles have no predefined neighbor relations to other particles, as is seen in mesh based methods such as Finite Difference, Finite Volume and Finite Element etc. Neighbor relations are calculated and recalculated dynamically. This property makes SPH especially well suited for describing highly deforming materials, such as fluids.

This concept of point samples and interpolation is also used in kernel based probability density estimations [Bishop, 2006], where it is attempted to learn about an underlying probability distribution based on point samples. In the following, we will a little later

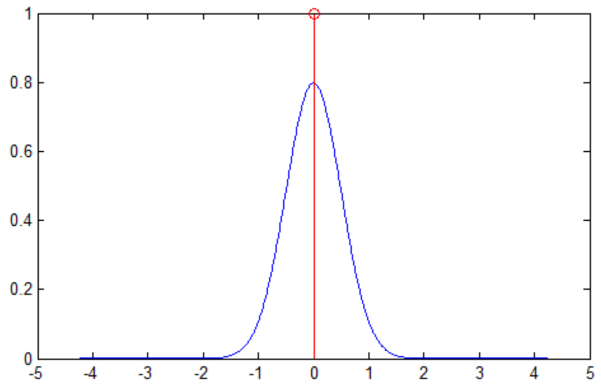


Figure 22: A point sample has a weight in its surroundings when used for interpolation. Here its interpolation weight is smoothed using a Gaussian kernel. This is done by convolving the point with the kernel. We note that the farther away the point should have interpolation influence, the lower the weight will be.

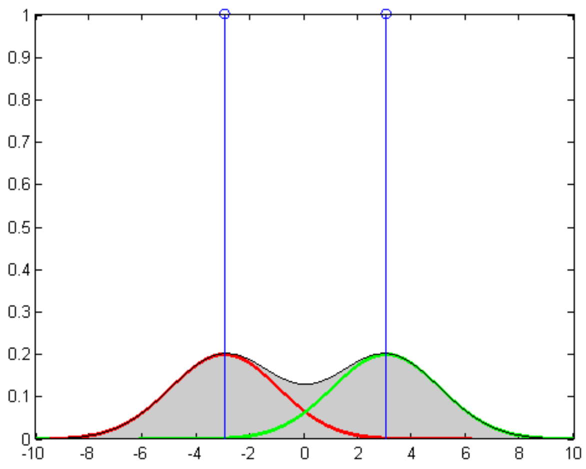


Figure 23: Two point samples are smoothed using a Gaussian. Note that the shaded area is the sum of the two smoothed weights and that this area equals one.

illustrate SPH by using it to recreate an unknown function, its derivatives and its integral, given only point samples.

Given that this is an interpolation method, the closer the points are, the more true and detailed the description can be, and the father apart the points are, the more uncertain the values are and the more the result is a smoothed representation of the underlying material.

In Figure 23 we see two points which are smoothed to a degree where they overlap. We note that each samples influence integrate to one over the domain and that the sum of samples integrate to two. More about this shortly.

Generally not all samples are created equal, and may therefore have different weight in the interpolations. When talking about kernel interpolation in probability theory, the weight generally corresponds to the number of point samples in a certain neighborhood, summed together, while in physics the weight is the mass of the material that a point sample represents.

8.1 Integral formulation

We will now briefly look at the mathematical derivation of the method. For a more in depth version, we refer the reader to [Liu and Liu, 2003].

If we, for the example, assume that we have an ”unknown” function $f(x), x \in \Re$ which we only have discrete samples of, we will attempt to reconstruct this function using SPH. We will pretend to not know the true function in the following, but is is defined as the following which is used to obtain the samples.

$$f(x) = 10x^2 - x^3 \tag{60}$$

We can rewrite this function in terms of an integral over space of the function multiplied with a point sampling operator

$$f(x) = \int_\Omega f(x')\delta(x-x')dx' \tag{61}$$

where Ω is the volume containing x and δ is the Dirac delta operator, which is defined as

$$\delta(x) = \begin{cases} 1 & ;x = 0 \\ 0 & ;x \neq 0 \end{cases} \tag{62}$$

That delta function based definition of $f(x)$ is true because for every $x' \neq x$, the inside of the integral evaluates to zero while it becomes $f(x)$ when $x' = x$. That integral definition of $f(x)$ is essentially the continuous variant of point samples, where there are an infinite number of point samples, and the function if perfectly represented. The delta function can be considered a mathematical way of obtaining a point sample of a function through multiplication.

If we now replace the delta function $\delta(x-x')$ with a special smoothing kernel $W(x-x',h)$, as seen in Figure 22, where h is a smoothing parameter defining how wide the smoothing kernel is, we can write $f(x)$ in integral form again as

$$f(x) \approx \int_\Omega f(x')W(x-x',h)dx' \tag{63}$$

The smoothing parameter h is commonly referred to as the support radius. The support radius is the maximal distance r at which the weight $W(r,h)$ is non zero. We will look more closely at that in section 8.4.

With the definition in (63) we do not have a perfect representation, but rather a smoothed one. Taking the limit of h going to zero, we will regain the actual function.

$$f(x) = \lim_{h \to 0} \int_\Omega f(x')W(x-x',h)dx' \tag{64}$$

since

$$\lim_{h \to 0} W(x-x',h) = \delta(x-x') \tag{65}$$

According to [Liu and Liu, 2003], the accuracy of SPH kernel based approximation is $O(h^2)$.

The kernel has to integrate to 1, for the above relations to be true. If it does not, it will rescale the function, and the smoothed representation would be scaled by α where $\alpha = \int_\Omega W(x-x',h)dx'$. Note that in fact the definition of the delta function ensures that it integrates to 1 as well.

$$f(x) \approx \alpha \int_\Omega f(x')W(x-x',h)dx' \tag{66}$$

While we do not get an exact representation of the function $f(x)$ using the integral representation with a smoothed kernel, we do still get an exact integral so that

$$\int_\Omega f(x) = \int_{dx} \int_{dx'} f(x')W(x-x',h)dx'dx \tag{67}$$

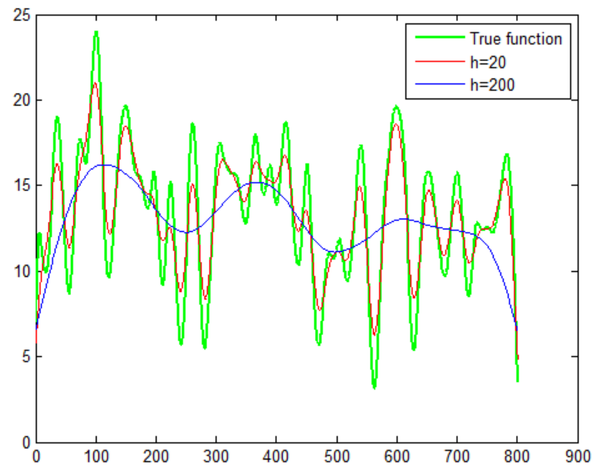


Figure 24: A noise function is smoothed using two different kernels with different support radius. It is evident that the larger the support, the more the function deviates from the true function and the more smoothed it is. The x and y values are just dimensionless real values numbers.

This means that while the smoothing methods in SPH does not guarantee the recovery of the exact function, it does, however, guarantee the conservation of the value of the integral. That is evident when considering what happens to each of the infinite point samples. It is smoothed over space, but given that the smoothing kernel always integrates to 1, the integral of a single point sample will be conserved

$$\int_\Omega f(x-x')\delta(x-x')dx' = \ldots$$
$$\int_\Omega f(x-x')W(x-x',h)dx' \tag{68}$$
$$f(x-x')\int_\Omega \delta(x-x')dx' = \ldots$$
$$f(x-x')\int_\Omega W(x-x',h)dx' \tag{69}$$
$$f(x-x')\cdot 1 = f(x-x')\cdot 1 \tag{70}$$
$$f(x-x') = f(x-x') \tag{71}$$

This is a very useful feature of SPH, when used for physical simulations, since this is an automatic conservation of the point sampled quantities, such as momentum or energy in general.

8.2 Derivatives

The spatial derivative of the smoothed function is obtained simply by replacing W with $\nabla \cdot W$. This is seen from the following rewrite, taken from [Liu and Liu, 2003], of (63) with the divergence operator $\nabla\cdot$ added to the equation. Note that for a *real* valued function, the divergence of the function is equal to the gradient - the derivative - while for vector valued functions, the divergence is the sum of partial derivatives. We use the divergence, rather than just the gradient, for reasons related to the Gauss divergence theorem, used at the end.

$$\nabla\cdot f(x) = \int_\Omega [\nabla\cdot f(x')]W(x-x',h)dx' \tag{72}$$

Differentiating f(x') multiplied with W can be done using the chain rule which gives

$$[\nabla \cdot f(x')]W(x-x',h) = \nabla \cdot [f(x')W(x-x',h)] - f(x') \cdot \nabla W(x-x',h) \quad (73)$$

Combining (72) and (73) gives

$$\nabla \cdot f(x) = \int_\Omega \nabla \cdot [f(x')W(x-x',h)]dx' - \dots \int_\Omega f(x') \cdot \nabla W(x-x',h)dx' \quad (74)$$

Gauss' divergence theorem, as explained in section 11.1, lets the first integral term be rewritten as a surface integral. The vector n is the normal at the surface boundary.

$$\nabla \cdot f(x) = \int_S f(x')W(x-x',h)] \cdot nds - \dots \int_\Omega f(x') \cdot \nabla W(x-x',h)dx' \quad (75)$$

This can be done because, for a kernel with compact support, the $W(x-x')$ term will be zero at the surface of the domain, as seen in Figure 25, if the kernel is entirely inside the domain. The surface S is here the surface, outer boundary, of the entire domain Ω and not of the smoothing function! Therefore, if the smoothing kernel is entirely within the domain, it will be zero at every point of the domains surface S - the red kernel is not touching the dark green boundary. If, on the other hand, the smoothing kernel is partially sticking out of the domain, this assumption does not hold. This is seen as the blue figure which is non-zero for part of the domain boundary S indicated with solid red.

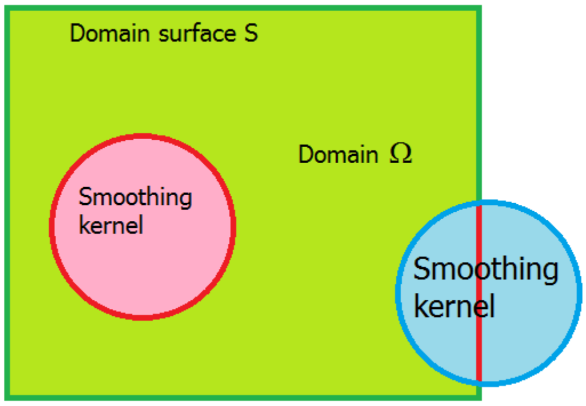


Figure 25: A domain Ω in light green having outer boundary S in dark green. A smoothing kernel in red is entirely within the domain, never touching the domain surface S. Another smoothing kernel in blue is partially outside the domain, touching the domain boundary S along the solid red line.

For this reason, a smoothing kernel *should* have compact support, meaning that it has some finite distance at which it becomes zero $W(r) = 0$ for some $r < \infty$. Otherwise we have no hope of containing the smoothing kernel inside the domain.

This lets us rewrite the expression into its final form (76).

$$\nabla \cdot f(x) = -\int_\Omega f(x') \cdot \nabla W(x-x',h)dx' \quad (76)$$

As mentioned in [Liu and Liu, 2003] artificial boundaries may influence the actual surface in a way that make the surface integral non zero, but we will be dealing with that in our boundary treatment in section 8.5.

8.3 Discrete SPH

In the integral formulation before, we saw how the function was represented through an infinite number of point samples, by integrating over the entire domain. That is still not a discrete version of the function. It is merely a version which introduces the concept of point samples and smoothing. In order to become a discrete description, the method has to rely on a finite number of samples, rather than an infinite number.

The SPH approximation given a finite number of samples is rewritten from the continuous integral form to a discrete sum over j samples as

$$\begin{aligned}
f(x) &= \int_{\Omega} f(x')W(x-x',h)dx' && (77)\\
&\approx \sum_j f(x_j)W(x-x_j,h)\Delta V_j && (78)\\
&\approx \sum_j f(x_j)W(x-x_j,h)\frac{1}{\rho_j}(\rho_j \Delta V_j) && (79)\\
&\approx \sum_j f(x_j)W(x-x_j,h)\frac{1}{\rho_j}m_j && (80)
\end{aligned}$$

Here we introduce ΔV_j as the volume of the particle and ρ_j which is the density of the particle.

Recall from earlier that every point sample had an associated weight, which was, in physical terms the particles mass. That weight distributed over a volume is then a weight density, or again in physical terms, just the density. First we rewrite the integral into a sum over j elements. To do this, we need to multiply with each samples volume. This volume is then rewritten in terms of weight and density.

This results in the general SPH method to approximate any field value A, known only from point samples, at any location r.

$$A(r) = \sum_j m_j \frac{A_j}{\rho_j} W(|r-r_j|,h) \qquad (81)$$

In order to obtain the density, through a direct summation as the above, we insert ρ_j instead of A_j and get

$$\begin{aligned}
\rho_i &= \sum_j m_j \frac{\rho_j}{\rho_j} W(|r-r_j|,h) && (82)\\
&= \sum_j m_j W(|r-r_j|,h) && (83)
\end{aligned}$$

From section 8.2 we know that the derivatives of a SPH approximation can be found by using the derivative of the kernel function rather than the derivative of the sample function. We can therefore write the derivative as

$$\nabla A(r) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(|r-r_j|,h) \qquad (84)$$

8.3.1 SPH interpolation example

We now briefly look at SPH interpolation of a function, knowing only the point sample values.

In Figure 26 we observe an attempted interpolation using a very narrow Gaussian kernel with a standard deviation of 0.5. The

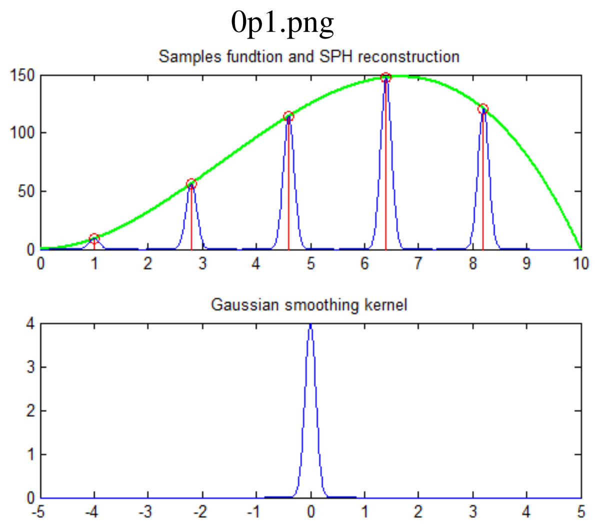


Figure 26: Gaussian kernel which have too small support with a standard deviation of only 0.5. The function is not well approximated.

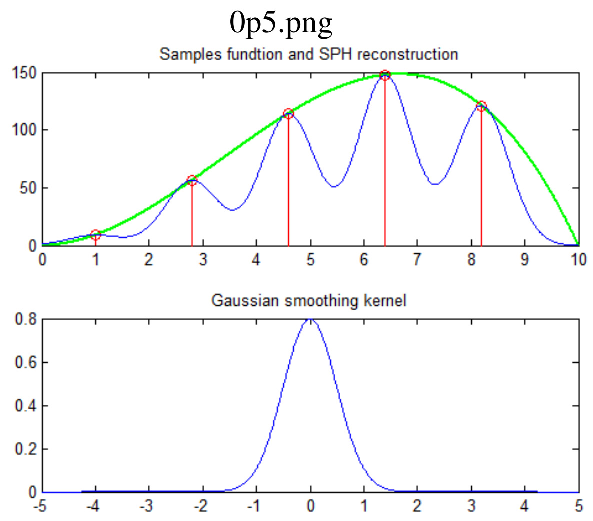


Figure 27: Standard deviation of 0.5, while the approximation does not reach zero in between samples, it is still a poor representation of the true function.

true function is shown in green and the point samples are red vertical lines with a circle indicating the sample value. It is evident that the function is not reconstructed with this kernel. Instead we see a set of individual Gaussians with a value of zero in between.

Using a slightly lager support radius, as on Figure 27, now with a standard deviation of 0.5, the result is still not reasonable, but at least now the value does not go to zero in between samples.

Expanding the support, now with a standard deviation for the Gaussian of 1, the result looks reasonable in Figure 28. The function is smooth and quite close to the true function. We note that in the highest part of the function, the SPH approximation is too low and in the lower parts of the function the SPH approximation is too high. This is due to the fact that the high point sample is affected by the neighboring lower samples and that the same effect is seen in reverse for the lower points.

If we expand the support further to a standard deviation of 4, the approximated function is becoming *too* smooth as seen in Figure 29

If we instead of changing the support radius, change the number of samples, we can try interpolating with the very narrow kernel with a standard deviation of 0.1 but now using ten times as many samples. This is seen in Figure 30 where it is clear that this is the best approximation so far. As shown earlier, as the smoothing length goes towards zero and as the number of samples goes to infinity, the approximation error goes to zero.

We now show the same approximation for the derivative of the function. In Figure 31 we see that again using a standard deviation

1.png

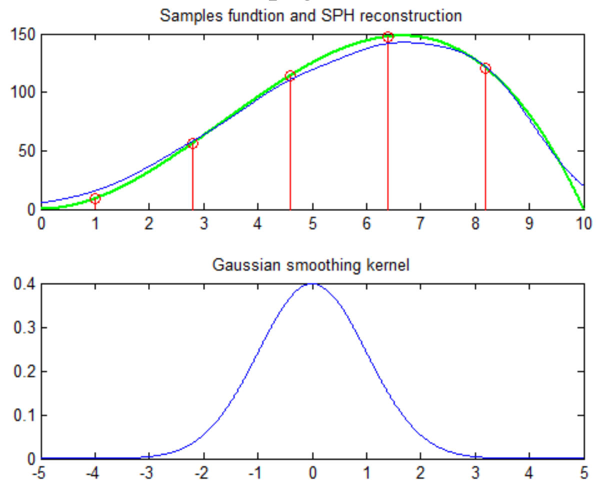


Figure 28: Standard deviation of 1 and the approximation mathes the true function very well.

4.png

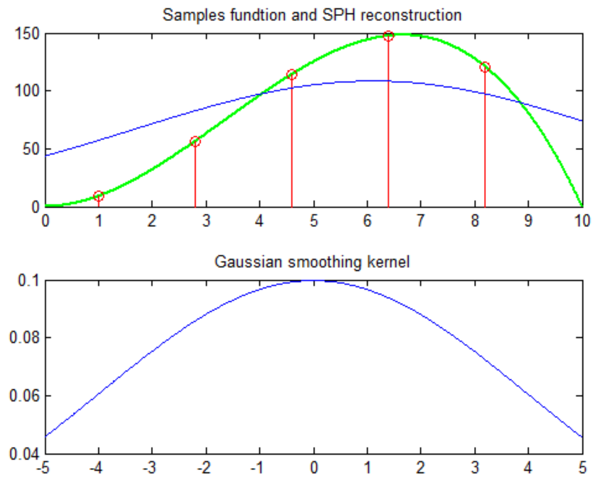


Figure 29: Standard deviation is 4 and the approximation is not much too smooth and it does not capture the true nature of the function.

0p1 50 points.png

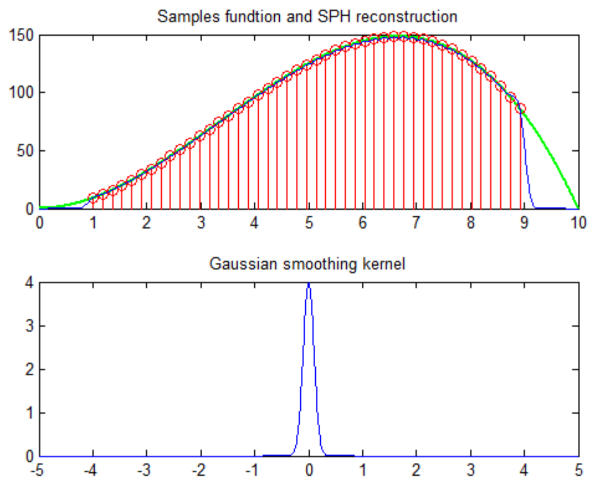


Figure 30: Standard deviation of 0.1 and 50 samples. This approximation approaches the true function since the kernel width approaches zero and the sample count goes towards infinity, at which point the approximation error becomes zero.

of 1, we get a decent approximation - only this time for the derivative of the function.

Finally we see in Figure 32 what happens if some areas have a lower sample point density. While the used standard deviation of 0.5 works somewhat reasonable for most of the domain, it fails miserably in the neighborhood of $x = 4$ where there are far between sample points. This demonstrates the relationship between smoothing kernel support radius and concentration of sample points. The farther spaced the points are, the greater the support radius needs to be. This will be addressed in more detail in section 8.7.

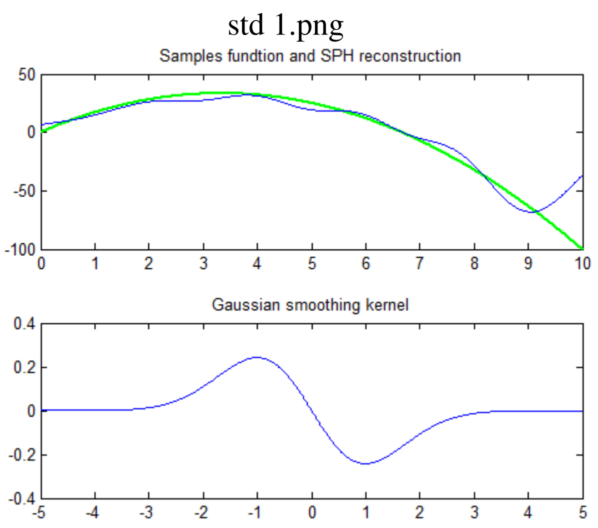


Figure 31: Standard deviation is 1 and this time we see that the first derivative of the function can be well approximated as well. Note that this plot is zoomed closer than the previous ones to better show the result.

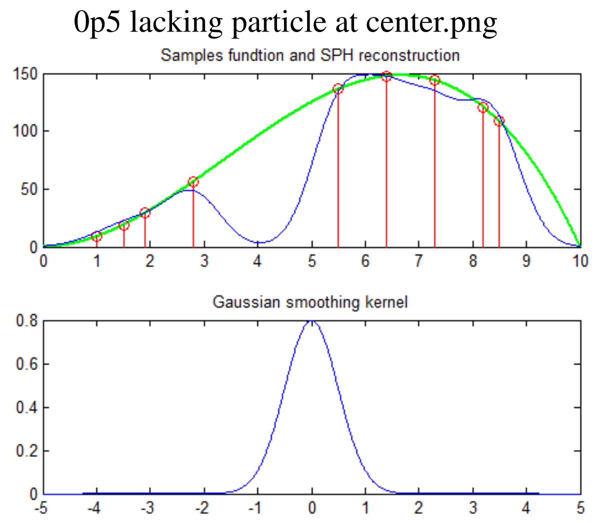


Figure 32: Std 0.5 particle missing at center. Note that SPH assumes the lack of particle to mean there is none...

8.4 Kernels

For a kernel to give a reasonable smoothing of a particle sampled field, it has to fulfill some general requirements, and in some cases it has to be designed to fit a specific purpose as described in the following. Recall that $W(r_i - r_j, h)$ means a kernel function over the distance vector between two positions r_i and r_j and with a support radius of h, meaning that $|r| \geq h \rightarrow W(r,h) = 0$.

8.4.1 Density kernel

Often density is smoothed using a function, which looks like a Gaussian. In [Monaghan, 1992] it is stated that the physically most reasonable kernel *is* a Gaussian. A Gaussian distribution is what is commonly seen in nature, and it defines a normal distribution. A Gaussian is, however, not compact since it only approaches zero as the distance from its mean increases - it never actually becomes zero. As mentioned previously in section 8.2 it really should be compact in order for the derivation of some of the SPH identities to hold true. For this reason the Gaussian is commonly replaced by an approximation that looks like the Gaussian and which is compact. Further more, a Gaussian with infinite support requires every particle to be considered a neighbor, and not just the particle close by, even though the contribution of far away particles is negligible.

One such function is known as poly6 which is defined as in (85) and this is the function we have used. A problem observed when using the $poly_6$ kernel is mainly that it is not

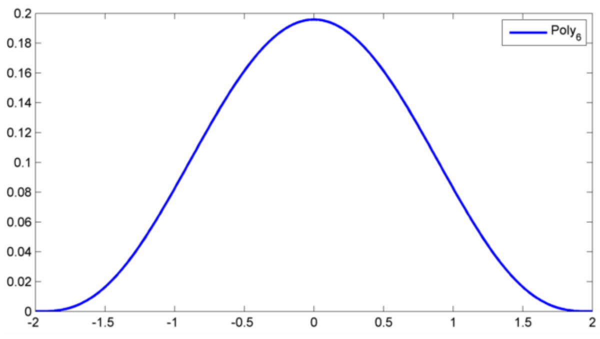


Figure 33: The $poly_6$ kernel which approximates a Gaussian inside the support radius which is here 2

infinitely differentiable as the Gaussian is.

$$W_{poly6}(r,h) = \frac{315}{64\pi h^9}\begin{cases}(h^2-||r||^2)^3 & ;0 \le ||r|| \le h \\ 0 & ;otherwise\end{cases} \tag{85}$$

The fraction in the equation is the normalization constant, used to ensure an integral of 1. It is interesting that r^2 is used since the squared distance between two points is easily calculated, while finding $|r|$ requires the use of a computationally more costly square root.

8.4.2 Kernel for field gradient forces

While the poly6 kernel is easy to evaluate, it has the same "problem" as the Gaussian which it emulates. Its first derivative approaches zero when $|r|$ approaches zero. This means that as two sample points come close together, the kernel based gradient weight will will decrease and for $|r| = 0$ it will be zero. If sample points are very close and they have different values for some parameter A, then it stands to reason that the gradient of A should be increasing towards infinity as the particle distance approaches zero. This is however not what happens if the gradient of the kernel approaches zero as the distance approaches zero.

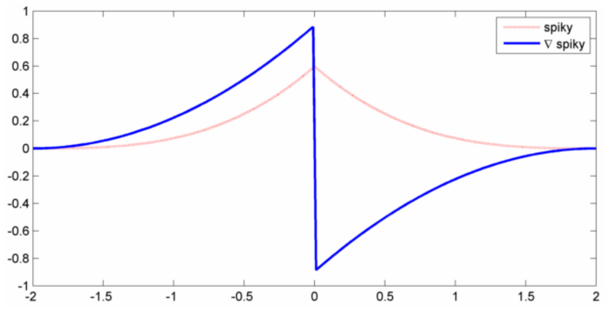


Figure 34: Spiky kernel and its gradient for a support radius h of 2

To remedy this problem the so called Desbruns spiky kernel (86) was designed, and is seen along with its gradient in Figure 34.

$$W_{spiky}(r,h) = \frac{15}{\pi h^6}\begin{cases}(h-||r||)^3 & ;0 \le ||r|| \le h \\ 0 & ;otherwise\end{cases} \tag{86}$$

This kernel is discontinuous in its first derivative and is shaped such that the gradient will increase exponentially as two sample points approach each other, and as they pass each other the sign of the gradient will immediately invert.

$$\nabla W_{spiky}(r,h) = -\frac{45}{\pi h^6}\begin{cases}\frac{r}{||r||}(h-||r||)^2 & ;0 \le ||r|| \le h \\ 0 & otherwise\end{cases} \tag{87}$$

8.5 Boundary conditions

Boundaries can be easily handled by using constraints which keep the particles from penetrating. This is commonly done, but the result is a loss of density ”over the boundary”. A particle has a point position in space, but the smoothing function has a certain size. If the point particle is positioned right along the boundary, then some of the smoothing function will be outside the boundary. This means that the mass conservation is essentially lost.

As an alternative to the simple constraint one can use virtual boundary particles [Liu and Liu, 2003]. They provide boundaries with mass conservation as well as boundaries that can exert repulsive forces using the same physics as between individual particles. Additionally it gives easy access to no slip boundaries as shall be seen.

Whenever a particle is closer to the boundary than one smoothing radius, then is is ”touching” the boundary with the smoothing radius. In those situations, a virtual mirror particle will be created on the exact opposite side of the boundary. If the real particle is close to several boundaries, it will create a boundary particles for each boundary. This can be easily done by projecting the particle through the plane of the boundary and out on the other side. The component of the velocity vector of the virtual particle normal to the boundary will also be the exact opposite of the real particle. The tangential velocity component may be the same as the real particle to give a slip boundary, while it could also be set to zero to give a no slip boundary as described in section 8.5.

Mass conservation In Figure 35, a particle is seen positioned at x=10, near a boundary at x=15. The smoothed density function of the particle is shown as a Gaussian. While the particle is inside the domain $0 \leq x \leq 15$, some of its density is lost outside the domain, seen as an orange area. If we integrate density over the entire domain we expect the result to be the total mass of all particles. This is no longer the case. To remedy this, we can create and place a virtual boundary particle on the outside of the domain in a way so it forms a mirror image of the real particle. This is seen in Figure 36 as the particle at x=20.

If we again integrate density over the domain, and include the part of the virtual particle which spills over the boundary, then we will again arrive at the real total mass of particles inside the domain. In Figure 36 we draw this integral of density, or sum of Gaussians as it is, as a green curve. We note that this green curve initially coincides with the shape of the real particles smoothing function, but as we approach the boundary the green curve (integral of density) diverges and attains a higher value. It is clear that this is due to the simple fact that what the real particle lost to outside the domain is exactly identical to what the virtual particle lost to inside the domain. Adding the virtual contribution will make the density inside the domain rise and mass has been conserved.

Impenetrable boundaries If the particle velocities, the accelerations and the time steps are such that a particle cannot pass through another particle, then obviously no particles can pass through a boundary

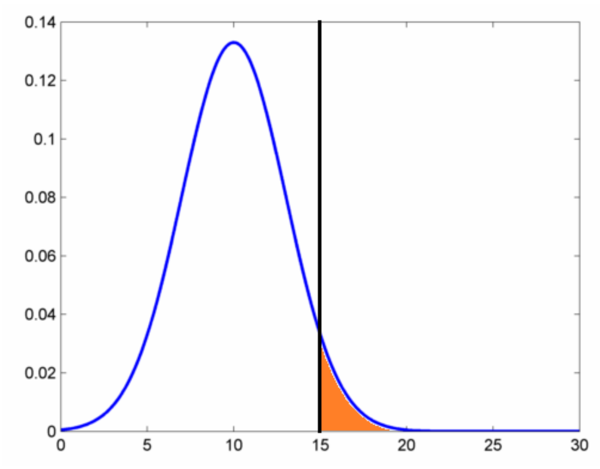


Figure 35: A particle next to a boundary is loosing some of its density outside the domain. Note the orange part which is the parth of the smoothed density which is lost.

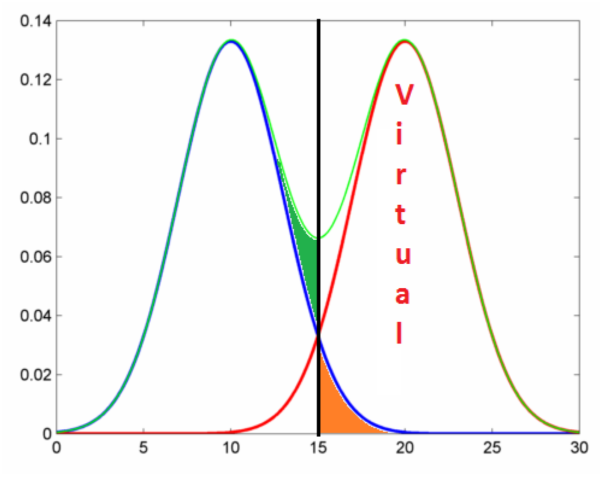


Figure 36: By adding a virtual boundary particles as a mirror image on the other side of the boundary, the sum of the two densities inside the domain will exactly equal the complete density of the real particle.

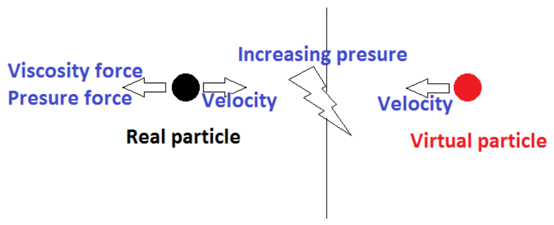


Figure 37: A particle experiences both viscosity forces and presure forces as it approaches a boundary ”guarded” by a virtual particle.

”guarded” by a mirror particle as seen in Figure 37. This will be the case if the CFL condition is uphold as described in section 9.6. As the real particles approaches the boundary, pressure forces will increase and push the particle away from the boundary and the approaching real particle will also experience velocity dampening viscosity forces due to the virtual particle moving in the opposite direction.

No slip boundary If a no slip boundary is required, it can be easily implemented by having virtual particles which have their inplane velocity set to zero. This means that if a boundary is aligned with the xy-plane, then the virtual particles should have zero velocity in the xy-plane. The velocity component along the boundary’s normal vector, the z-axis, is not constrained to be zero and will just be the opposite of the real particles normal velocity as seen in Figure 38. By having the boundary particles not move in the boundary plane, the real particles will experience frictional drag when sliding along the boundary as if they interacted with a stationary no slip boundary. Different friction quotients can be used for different boundaries

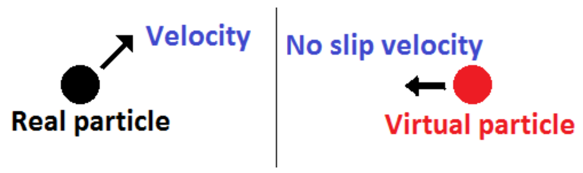


Figure 38: A virtual particle has its in-plane velocity set to zero to provide a no slip boundary

by assigning real-virtual particle pairs various friction quotients.

Difficulties with boundary particles For complex geometry, the boundary particles may be too simple a solution. Every real particle will interact with every boundary particle in range. This is not a problem for a boundary which extends over an entire plane in 3D, such as the xy-plane, but for smaller boundaries it can cause inaccuracies. In Figure 39 two particles are near the same corner. Both project a boundary particles to the inside of the corner and, in this example, both the boundary particles are places in the same place. While the real particles are not inside each others support radius, both do see the other ones boundary particle. The result is that both real particles will experience a repulsive force from the boundary which is twice what it should have been. Many other examples of this type can be found in real simulations, and it is a source for some concern and it has been addressed in some detail in [Monaghan and Kajtar, 2009].

In our solution, we recognize that inaccuracy, but it is irrelevant for us given how we handle the boundary particles as described in section 9.4. For other implementations, we note, however, that for small smoothing lengths, the problem is only relevant very near to an edge or corner, or on opposite sides of very thin objects. Only particles less than one smoothing radius away from the trouble zone will ever experience the problem and then *only* if another particle happens to on the other size of the zone - no farther than two smoothing radii away. This will realistically happen, but given than the event will be somewhat rare and that the result, should it happen, is not catastrophic, this is something most models can probably live with. It is not a problem for us though.

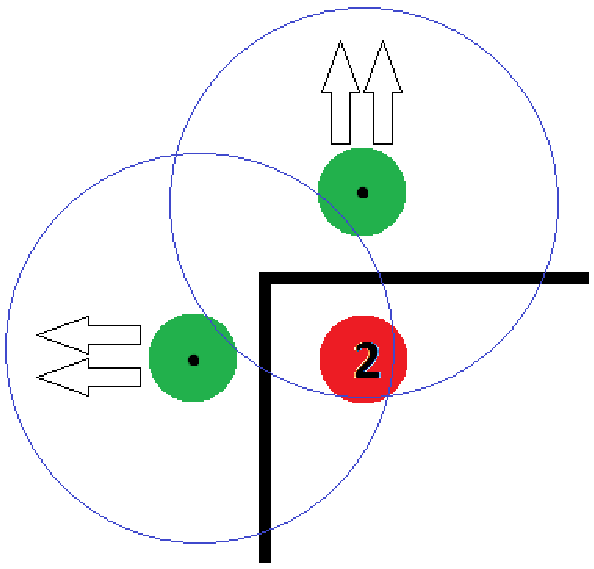


Figure 39: Two particles are near a corner and they both create a virtual particle which results in double the expected repulsive force

8.6 Surface

We may need to determine which SPH particles define the outer boundary of a SPH sampled field. This is commonly

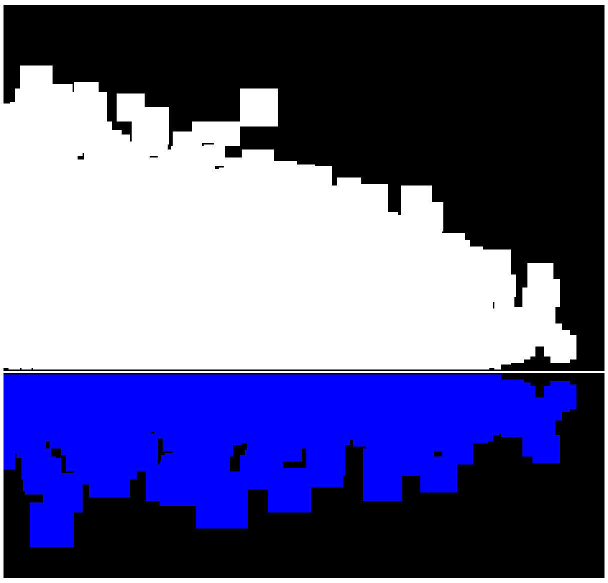


Figure 40: A snow drift over a horizontal boundary, using exaggerated smoothing lengths for better illustration. Notice how the blue boundary particles are a mirror image of the real particles closest to the boundary

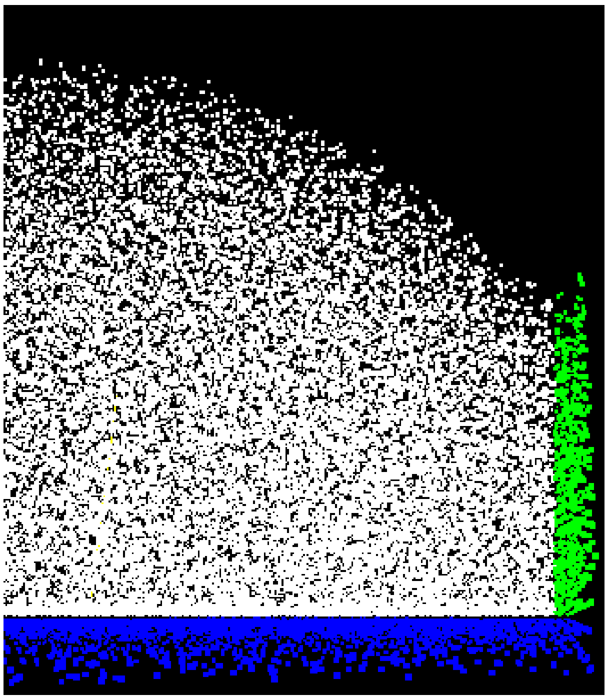


Figure 41: A snow drift over a horizontal and next to a vertical planar boundary. Note how the blue and green boundary particles mirror the real white particles. The drift is in 3D and seen slightly at an angle which is why not every particle has a visually obvious matching boundary particle.

implemented using a so-called color field [Müller et al., 2003] which is defined as

$$C(r) = \sum_j m_j \frac{1}{\rho_j} W(r - r_j, h) \qquad (88)$$

where m_j is the mass of the j'th particle, ρ_j is the density of the j'th particle, W is the smoothing kernel using the smoothing length h and the particle distance vector $r - r_j$.

We note that using this definition, the field contribution of the j'th particle, inside the particles support, is $1m_j$. Put another way, if the kernel W was used to calculate the density, then the density scaled kernel will be 1 inside the support, or formally

$$\frac{W(r - r_j, h)}{\rho} = \begin{cases} 1 & ; |r - r_j| \leq h \\ 0 & ; otherwise \end{cases} \qquad (89)$$

In Figure 42 we see a 2D field with evenly spaced particles. We observe the resulting density field as a sideways surface plot in Figure 43. Here it becomes evident that the smoothing length used was is 5, which is why the density drops to zero at a distance of 5 from the outer particles.

We now calculate the color field C as seen in Figure 44 and then the gradient of this field ∇C which is shown in Figure 45. We note that the gradient $\frac{\nabla C}{|\nabla c|}$ is the surfaces normal vector pointing into the region of higher density. If we now consider the Laplacian of the field $\nabla^2 C$ then we can calculate the actual curvature of the edge. This is seen in Figure 46.

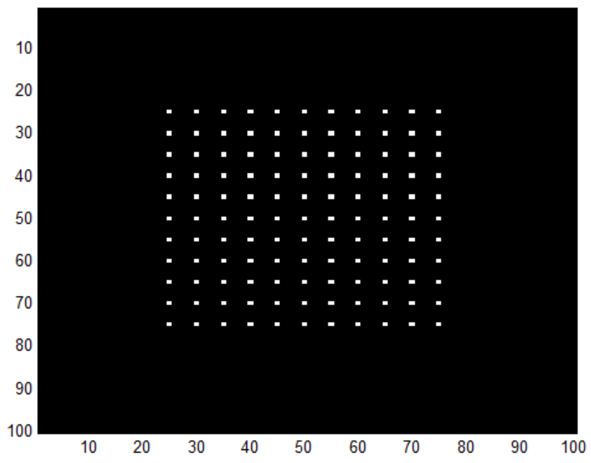


Figure 42: A group of evenly spaced particles in 2D. Each particle has the mass $m = 1kg$

We now have a method of locating areas with high curvature, which represents the edge of the sampled field, and we can find the normal vector of this surface. We will be using this later in section 9.3.

8.7 Variable spatial resolution

When a particle is left without any neighbors inside its support radius, then the field value at its position will be equal to the particles own value. This makes sense since it is an average of one value. It is, however, then a quite coarse sampling of the field, and unless the field intensity just happens to be identical to that of a smoothing kernel, it will be a bad approximation at any non zero distance away from the sample point itself. A recommended minimum number of support particles is 57 for 3D [Liu and Liu, 2003]. The values may vary slightly from model to model depending on the level of smoothness

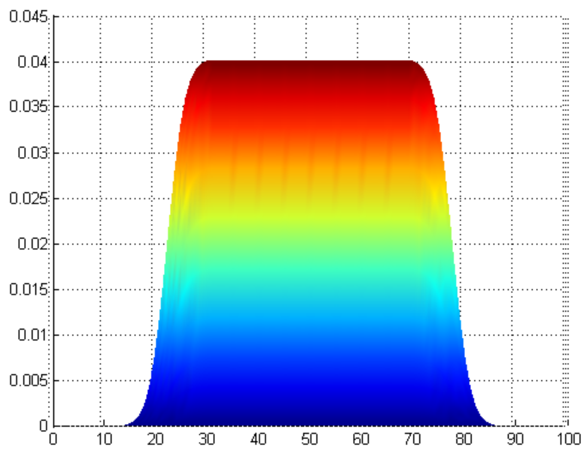


Figure 43: A density field for the particles in Figure 42 is seen as a surface plot from the side. Note how the density remains constant inside the particle field and then drops off sharply away from the particles, but still remains non zero until a distance of 5 from the outermost particles.

desired.

It is interesting to note that, if the number of support particles should be kept at a fixed value, and the particles are very tightly packed, then the smoothing length h should be very small. This would mean that the kernel would approach the delta function at which point there is no smoothing but rather an *accurate* point sampling.

In [Liu and Liu, 2003] a method of adaptively changing the smoothing length is employed.

$$h_i^n = h_i^{n-1}\frac{1}{2}\left[1+\left(\frac{N_{target}}{N_i^{n-1}}\right)^{1/3}\right] \qquad (90)$$

where h_i^n is the smoothing length for particle

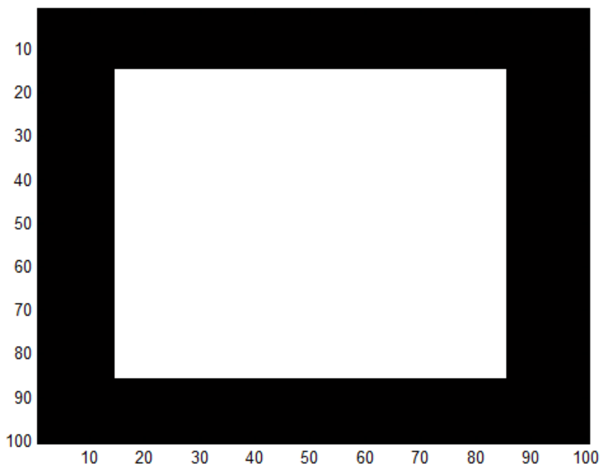


Figure 44: A density field for the particles in Figure 42 is seen as a surface plot from the side. Note how the density remains constant inside the particle field and then drops off sharply away from the particles, but still remains non zero until a distance of 5 from the outermost particles.

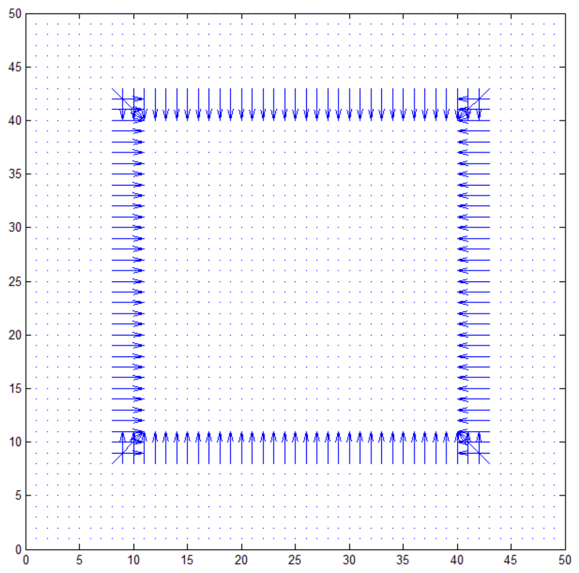


Figure 45: The gradient of the color field ∇C is seen to be the normal vector of the surface, pointing into the non-zero density area.

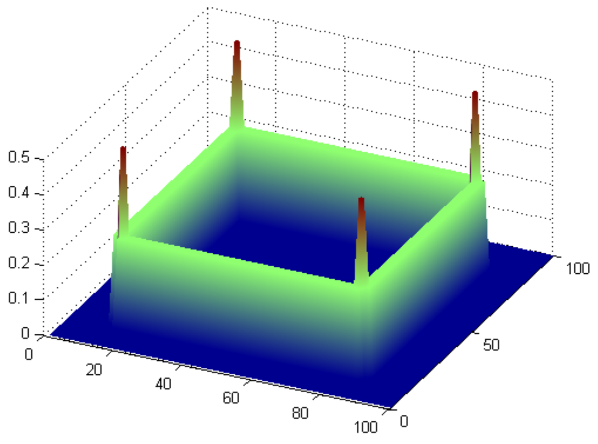


Figure 46: The Laplacian of the color field $\nabla^2 C$ is shown as a surface plot. Note how the curvature is higher in the four corners since there is a non-zero gradient along both axis.

i at time step n. N_{target} and N_i^{n-1} are the optimal number of particles in the support and the current number of particles in the support for particle i at time step n.

It is a reasonable practice to constrain the smoothing length both upwards and downwards. This is to prevent the SPH discretization from assuming that the field is defined everywhere (using a large smoothing length) when it may in fact be more localized, and to prevent very small smoothing lengths which will generally make the SPH model very sensitive to large time-steps.

Using the above relaxation method, the smoothing length can easily be adapted to give a reasonable number of particles - both for very densely packed regions and for more sparsely sampled parts of the scene. It has been brought to our attention that this will make h be a function of time as well as of space $h(x,t)$ and we may have $\frac{\partial h}{\partial t} \neq 0$.

This is not something we have seen being considered in any other work, and we will also ignore this for now. Additionally, when integrating the systems derivatives over time, as detailed in section 9.6, we are in fact not adjusting the value of h in-between full steps. This means that through one whole integration step Δt, h does *not* change. This is not a problem since SPH does not fail just because the *perfect* number of neighbors is not used, and because the change is h is expected to be very small in between steps

We will be using the above mentioned method, but there is another method which should be mentioned for completeness when dealing with adaptive spatial resolution. In works such as [Hong, 2009, Zhang et al., 2008, Adams et al., 2007] a method is investigated where not only the smoothing length is adaptive. The particles themselves are adaptive in the sense that they can split into more smaller particles in areas where high resolution is required and they can merge into larger particles where the resolution is currently higher than needed. This means that given a fixed number of computational resources, those resources can be put to use where they are most used.

Finally, it should be remembered that no smoothing length should be larger than the minimal dimension of the spatial partitioning cells. This is described in section 9.5.

9 SPH snow model

Building on the knowledge from the previous sections, we will now develop an actual snow model using the SPH method. It should be noted yet again that we are not simulating individual snow flakes, but volumes of snow. That snow may be loose wind block snow flakes or it may be tightly packed ice crystals.

9.1 Dynamic rest density

We let the rest density of the snow develop over time as the snow is compressed. This is to let snow be compressed in an inelastic fashion.

$$\rho_{rest} = \rho_{rest} + \alpha(\rho - \rho_{rest}) \qquad (91)$$

where α is a temporal smoothing factor with $0 < \alpha \leq 1$. The higher the value of α the faster the density changes. Given that snow does have a (tiny) elastic region before failure and given that particles may have some density fluctuations due to integration inaccuracies, we should let the system have a brief time during which it can recover from higher compression, before changing the rest density all the way to the current density.

Note that the above equation will also let the rest density decrease over time if, for some reason, the compressed snow particles are removed from each other again.

If the expression is used in that form for some rest density, then we will have a perfectly elastic material, which is not realistic. For that reason, we let the rest density evolve over time as

$$\frac{\partial \rho_{rest}}{\partial t} = \rho_{rest} + \beta(\rho - \rho_{rest}) \qquad (92)$$

β is a value between 0 and 1 defining how quickly the density changes. The idea is that when snow is compressed, it will resist the compression, but only initially. After being compressed to some new higher density, the snow comes to rest at this density - the yield strength has been exceeded and snow has deformed plastically. There is then no internal pressure trying to return the material to its previous density.

9.2 Particle acceleration

The total rate of change of velocity is given by

$$\frac{du_i}{dt} = a_i^{pressure} + a_i^{viscosity} + a_i^{wind} + a_i^{gravity} \qquad (93)$$

where each a-value is the acceleration resulting from that term.

9.2.1 Pressure

Objects are pushed away from areas of high*er* pressure towards areas with lower*er* pressure. Therefor knowing the pressure field p, or rather its gradient ∇p, lets us calculate the pressure force. The force density is then the negative gradient (94).

$$f_{pressure} = -\nabla p \tag{94}$$

The negative pressure gradient is calculated through direct summation.

$$-\nabla p_i = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(r_i - r_j, h) \tag{95}$$

and the particle pressure is found from the EOS as detailed in section 6.11.

We note that, if a pair particles interact, and they have different pressure, they will not experience the same pressure force, given that the gradient kernel is zero at the location of the particle in question $\nabla W(0,h) = 0$. Each particle therefore calculate a force based on only the pressure at the other particles, which in all likelihood, is not the same throughout the domain. For this reason the equation should be made symmetric [Liu and Liu, 2003] by including the current particle as well in an average pressure term.

$$-\nabla p_i = -\sum_j m_j \frac{p_i + p_j}{\rho_i + \rho_j} \nabla W(r_i - r_j, h) \tag{96}$$

That expression is the force density. To arrive at acceleration, we divide the force density with the density, or in the symmetric case, the densities

$$a_i^{pressure} = -\sum_j m_j \frac{p_i + p_j}{\rho_i^2 + \rho_j^2} \nabla W(r_i - r_j, h) \tag{97}$$

9.2.2 Viscosity

The viscosity force is given by the Laplacian of the velocity field $\nabla^2 u$, multiplied with the viscosity μ, which could be seen as the strain rate multiplied with viscosity, as previously described in section 7 where the expression is given for strain rate dependent viscosity.

$$\begin{aligned} f_i^{viscosity} &= \mu \nabla^2 u \\ f_i^{viscosity} &= \mu \sum_j m_j \frac{v_j}{\rho_j} \nabla^2 W(r_i - r_j, h) \end{aligned} \tag{98}$$

as in the case of pressure, the kernel is zero at the particle itself, meaning that the expression is not symmetric. It can be rewritten as

$$f_i^{viscosity} = \mu \sum_j m_j \frac{v_j - v_i}{\frac{1}{2}\rho_j + \frac{1}{2}\rho_i} \nabla^2 W(r_i - r_j, h) \tag{99}$$

Again we have a force density which lets us write the acceleration as

$$\begin{aligned} a_i^{viscosity} &= \frac{1}{\rho_i} \mu \nabla^2 u \\ a_i^{viscosity} &= \mu \sum_j m_j \frac{v_j - v_i}{\frac{1}{2}\rho_j + \frac{1}{2}\rho_i} \nabla^2 W(r_i - r_j, h) \end{aligned} \tag{100}$$

9.2.3 Gravity

Gravity is a constant force which pulls the particle down. In our simulation, down is defined as the direction $-y$.

Earlier we presented a force which included the buoyancy of the air. Given that this value is very small, we do not use it in the actual implementation.

$$F_i^{gravity} = m_i g \qquad (101)$$

where g is the gravitational acceleration $g = 9.82 m/s$ and m_i is the SPH particle mass.

The acceleration is then

$$a_i^{gravity} = g \qquad (102)$$

9.2.4 Wind

The wind forces are the drag forces from relative wind velocity for a particle. Note that the wind velocity is found as described later in section 13.1.

We repeat the drag equations from section 4.2.

$$\Delta u = u_{air} - u_{particle}$$
$$F_i^{drag} = \frac{1}{8} C_D \rho \pi D_P^2 \Delta u^2 \qquad (103)$$

where C_D is the coefficient of drag in air, ρ_{air} is the density of air and Δu is the relative velocity between the particle and the air.

$$C_D = \frac{24}{Re_p} + \frac{6}{1+\sqrt{Re_p}} + 0.4 \qquad (104)$$

Where Re_p is the particle Reynolds number.

$$Re_p = \frac{D_p}{\nu} u \qquad (105)$$

This gives the acceleration form drag on the particle i.

$$a_i^{drag} = \frac{1}{8 m_i} C_D \rho \pi D_P^2 \Delta u^2 \qquad (106)$$

where $C_D \rho \pi D^2$ can be calculated beforehand.

9.3 Particle ejection from surface

We need to somehow model the SPH particle ejection from the surface given a certain friction velocity. From section 4 we have

$$u_* = \frac{u}{20}$$
$$\alpha = 0.083m$$
$$\beta = 0.19m/s$$
$$\lambda = 0.00052m$$
$$m_{eject} = \left(\frac{4}{3}\pi\left[\frac{D}{2}\right]^3 900kg/m^3\right)(\alpha e^{\frac{u_*}{\beta}} + \lambda) \qquad (107)$$

where m_{ejast} is still the ejected mass per square meter per second.

It should be repeated that this ejection model is a *rough estimate* which needs calibration, but we have implemented it as follows.

Every particle identified as a surface particles, as described in section 8.6, has a probability of ejecting depending on friction velocity. As seen in section 4 the ejection is commonly 20°. We interpret this as the ejection being in the plane spanned by the surface normal vector n and the wind velocity vector u, and the ejection angle being 20° from the wind velocity vector towards the normal vector. Since the ejection velocity is equal to the friction velocity u_*, we replace the wind velocity vector by the friction velocity vector. This gives us the ejection velocity vector

$$e = \left(\cos(20°)n + \sin(20°)\frac{u_*}{|u_*|}\right)|u_*| \tag{108}$$

and knowing the mass ejected per second per square meter, and the SPH particle mass gives us the particle ejection rate $R = ejections/s$.

$$R = \frac{m_{eject}}{m} \tag{109}$$

We now need to relate this to an ejection probability P_{eject} each second. The number of SPH particles SPH_n per unit volume m^3 is derived from local density ρ and particle mass m, which is here assumed to be the same for every particle.

$$SPH_n = \frac{\rho}{m} \tag{110}$$

If we assume the particles are spaced equally in a regular cubic grid, we can get the number of particles along each axis $axis_n$ and from that the particles per unit area SPH_{nA}

$$\begin{aligned} axis_n = \sqrt[3]{SPH_n} &= \sqrt[3]{\frac{\rho}{m}} \\ SPH_{nA} = axis_n^2 &= \left(\frac{\rho}{m}\right)^{\frac{2}{3}} \end{aligned} \tag{111}$$

Now we know how many particles there are per unit surface and we know how many particles should eject every second.

This lets us write the ejection probability, for every surface particle, every time step Δ as $P^{\Delta t}_{eject}$

$$\begin{aligned} P^{\Delta t}_{eject} &= \Delta t \frac{R}{SPH_{nA}} \\ P^{\Delta t}_{eject} &= \Delta t \frac{m_{eject}}{m} / \left(\frac{\rho}{m}\right)^{\frac{2}{3}} \end{aligned} \tag{112}$$

To sum this up, we now know that if a particle is a surface particle (section 8.6), then the probability that it should eject during a time step is given by (112), and if it ejects, it will be adding the ejection velocity vector from (108) to its current velocity.

9.4 Boundaries

As detailed in section 8.5, boundaries in SPH need special treatment, and one such boundary solution could be virtual mirror particles on the other side of the boundaries. This is sometimes seen implemented [Liu and Liu, 2003] by making a pass over all particles before each step and creating actual boundary particles. While this is conceptually simple, it can result in a very large number of particles when the model has a high particle count and a large portion of those particles are resting on some boundary. There is the additional problem that if thin obstacles are contained within the scene, then one particles virtual boundary particle may be created in such a way that it affects the particles on the other side of the obstacle.

The solution we have devised for our work is to use the ordinary neighbor location from section 9.5 and expand this with conditional behavior whenever the particle, seeking neighbors, is close to a boundary.

If a particle is within one smoothing length from a boundary, then it will be interacting with that boundary. In that situation it will need to create a mirror image of itself on the other side of the boundary. If the particle has a neighbor particle which is also close to a boundary then that neighbor will need to be mirrored as well. Rather than explicitly creating these mirrored boundary particles, this takes place in the ordinary neighbor location as seen in algorithm 1.

Note that if a particle A is not a neighbor to particle B, then A is also not a *true* neighbor of the boundary mirrored B. This is snown in section 8.5 where is is seen that assuming the opposite will in fact result in errors, although they would be rare. This means that when accumulating influences from particles and their virtual boundary mirrored counterparts, we need only consider the true neighbors and their virtual versions. This makes it somewhat easy to extend the normal SPH contribution calculations with boundaries.

All our boundaries are part of obstacles. An obstacle could be a cube inside the scene, which represents an impenetrable building. The cube then consists of 6 boundaries and any particle closer to the cube than one smoothing length will be affected by it. For any convex volume, it is true that any outside point will be separated from that volume by a plane containing one of its boundary surfaces. This is illustrated in Figure 47.

A particle may be close to several boundaries, each belonging to the same obstacle as seen in Figure 48 where the upper left red particle is close to both the upper and the left boundary of the square obstacle. In that case, the particle will have to be mirrored over first one boundary (that is the blue particle) and then over the second boundary (to the green particle). For convex obstacles, this will always result in a boundary particles which is on the inside of the obstacle and which contributes with exactly what the real particle has lost to the inside of the obstacle as explained in section 8.5.

Obstacle closeness Every obstacle is surrounded by an axis aligned box which is one smoothing length larger than the contained obstacle in every dimension. It is easy to quickly find if a particle is close to an obstacle or not this way, by determining if the particle is inside the bounding box.

Boundary closeness To determine if a particle is close to a given boundary, the particles position p is projected into the boundary plane, along the planes normal vector n, and the distance d between plane and particle is calculated. A positive distance indicates that the particle is above the plane

```
Input: list of particles
Output: forces F for each particle
foreach particle P do
    F← 0
    foreach neighbor N do
        F← F+calculateForces(P,N)
        if N is near boundary then
            M← N
            foreach boundary B do
                M← mirror(M,B)
            end
            F← F+calculateForces(P,M)
        end
    end
end
```

Algorithm 1: Extended SPH force calculation for particles already determined to be close to an obstacle consisting of boundaries.

and if this distance is less than the smoothing length, then the particle should be mirrored over the plane.

$$D = \frac{|p \cdot n|}{|n|} \tag{113}$$

Concavity An obstacle must be defined as a convex volume. If concave obstacles are required, they can be constructed from a number of convex shapes.

9.4.1 Domain boundaries

While the ground and obstacles inside the domain are defined as impenetrable boundaries, the sides of the domain is not. one of the SPH models strong sides is that the sam-

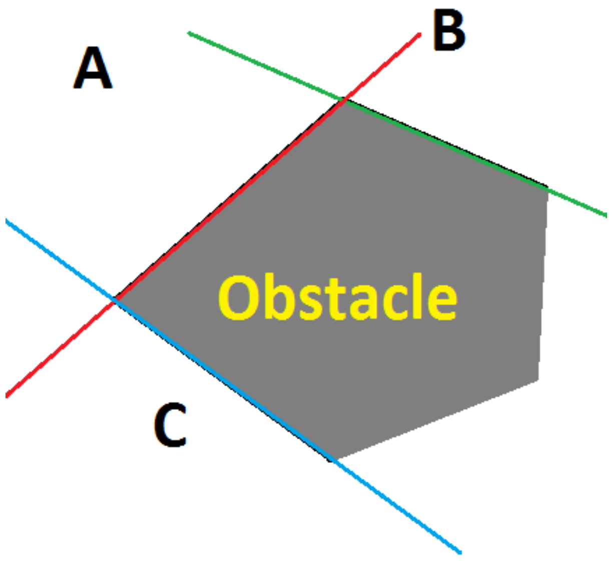


Figure 47: A convex obstacle is always separated any outside point by a single plane (line in 2D) along one of its edges. The point A is separated from the obstacle by the red plane, B by the green plane and C by the blue. Each plane runs along the edges of the obstacle.

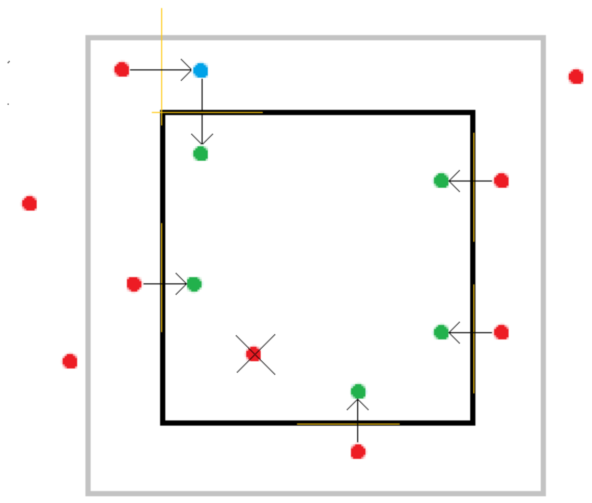


Figure 48: A square (convex) boundary is drawn in solid black and is surrounded by an inflated version, light gray, of itself which is expanded one smoothing length in every direction. True particles are rendered in red. Some are far enough from the black boundary to not be affected by it. Others are less than one smoothing length from the boundary - inside the gray expanded version - and are affected. If the particle is above one of the boundary edges, it ”spawns” a virtual boundary particle, seen in green or blue, which is the mirror image over the boundary. If that particle is also close above a boundary, it will again be mirrored into a virtual particle. Only the final green boundary particles are used. The single red particle inside the boundary is an invalid position which should never be seen.

pling resolution follows the material. Therefore we could track snow infinitely.

The wind, however, is contained inside a box $256m \times 256m \times 256m$, and therefore there is no point in tracking SPH particles as they exit the domain. Whenever a particle moves over the boundary, it will be re-initialized inside the scene at random x,y position at at a height $min \leq h < 255$ where min is a simulation parameter defining the height of obstacles and estimated snow layer.

The reason snow is not always created at the top of the domain is that the wind may blow it through one of the side boundaries before it has a chance to land on the ground. The vertical fall may simply be too large.

While it is not seen to be a problem creating a SPH particle in the zone with suspended snow, due to the very low density there, it is a problem creating articles in the region of rested snow and inside obstacles, and doing so would certainly make the simulation unstable, when particles could suddenly find themselves directly on top of one another.

9.5 Particle interaction and neighborhoods

In order to find the particle-particle forces, for each particle we need to determine which particles are inside its support radius. This is where SPH simulations tend to spend most of their time, and it is therefore important to give the matter great consideration if the simulation is to run fast. This section is therefore quite comprehensive.

The neighbor location is also a very parallel procedure. Every particle needs to locate its own personal neighbors and calculate the forces resulting from the interactions. This is an area where the massive parallelism of a GPU can be put to good use.

Brute force The naive method of finding particles, which are within the relevant neighborhood, is to calculate the distance to every other particle, and consider it a neighbor when the distance is less than the neighborhood radius. This is brute force, and for large groups of particles, it is intractable with $O(n^2)$.

Recursive tree A more common, and certainly more clever, method for fast neighbor search, implements a tree structure which is refined to a level where each leaf contains a single particle. Such a tree, being it a kd-tree, oct-tree or some other type of recursive spatial subdivision, has a depth of O(log n) and for n particles to find their neighbors we have O(n log n). These methods do, however, require that the tree is rebuilt every time the particles move relative to each other. While tree construction is also $O(nlogn)$, which is then the upper bound on the complete method with tree construction and tree search, the individual steps in the construction and search may still be complex and slow enough to make the method less than optimal.

When coding for a GPU there may arise extra complexities in an algorithm, such as tree search, which stems from the fact that the usual top down tree construction is by nature serial, and that the tree search is by nature recursive, while GPU code is neither. This has been addressed in [Garcia et al., 2008] and in [Ajmera et al., 2008] where kd-tress have been implemented on GPU, and with good results. The algorithms are however more complicated than alternative simpler methods, and we have decided to keep kd-trees in mind for later optimization, but to disregard it now.

Regular grid An alternative to a recursive spatial subdivision is to split the domain into a regular grid as seen in Figure 49. Here every cell will contain zero, one or several particles. If the cells in the grid are larger than or equal to the size of the support radius of the particles, it is simple to locate the neighbors. This is done by looking in the cell that the particle itself is inside as well as the cells adjacent to the particles cell. This will amount to 27 cells including the cell containing the particle itself. It should be noted that the only reason this is a viable solution is that the particles have a finite support radius. In astronomy, where SPH originate, gravity plays a vital role and the range of gravity is infinite. For this reason SPH in astronomy always employs an actual tree-like subdivision scheme since a regular grid would offer no improvement over brute force when considering an infinite support radius.

For a regular grid and finite support radius. the task is simplified to letting each particle determine which cell it belongs to and when looking for neighbors to look in the adjacent cells. This can easily be accomplished in parallel as described later.

9.5.1 Regular spatial grid (spatial hash)

Due to the in-feasibility of brute force for large number of particles and the complexity of spatial trees, we choose to implement a regular grid.

It is often seen, as in [Ihmsen et al., 2011, Goswami et al., 2010], that the grid is defined so that the cells are equal to the smoothing length of the particles. The smoothing length is generally small and this means that the grid is very fine. To avoid storing a huge number of cells, of which most will be empty, the grid is constructed as a 3D hash table where each grid cell will map into one unique bucket in the hash table and where each bucket will contain particles from several cells. This one to many relationship makes it possible to both have a very fine grid and to have an infinite scene. The hash index of a particle is calculated quickly using the following equation

$$H(x,y,z) = [(\frac{x}{d}p_1) xor (\frac{y}{d}p_2) xor (\frac{z}{d}p_3)] \% m \tag{114}$$

where $p_{1..3}$ are large prime numbers, m is the size of the hash table, d is the size of the grid cell and "%" is the modulus operator.

One downside of this hashing is that particles, which are located far apart in the scene, will map to the same hash bucket. This means that we lose spatial coherence, and that particles will be handled in very heterogeneous groups.

Another downside, related to the small cells, is that particles generally need only move very little before they are moving into new cells. This means that we need to update the cell location of all the particles quite often, which can be avoided when using larger cells. This is described in section 9.5.8.

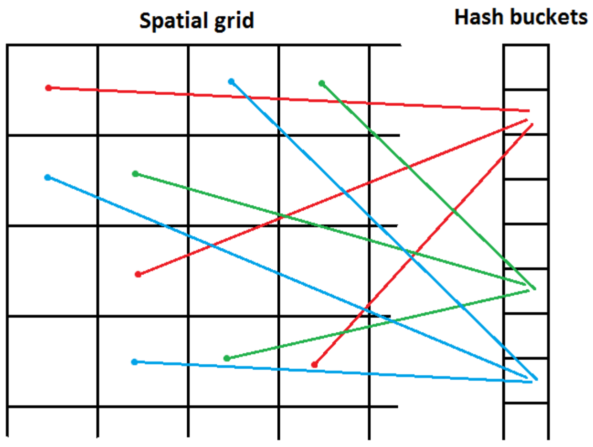


Figure 49: A regular grid mapped to a hash table. Notice how several cells map to the same bucket.

For those reasons, we use large cells. This could be considered a collision-free hash method. We can do this because we also accept that our scene is finite. This is not a real problem, considering that the air simulation using the Eulerian Finite Volume Method also requires a finite scene.

We define the cells size as the domain size divided by 256 to create a grid with 256x256x256 cells all in all. That is almost 17 million cells.

While the cells are large, we actually have quite a lot of unique cells. This is not a problem, since each cell will not take up much storage. It will only serve as a reference into the list of particles.

The benefit of this grid scheme can be seen by an example. If we have 2 million particle distributed evenly in the scene, then each particle should consider all other (2 million - 1) particles as potential neighbors and calculate the distance to them, if using the brute force method. Using the 256x256x256 grid, we have 2 million particles divided by 256^3 cells equal to 0.119 particles in each cell. Having only 27 cells to consider, that is 3.22 particles which may or may not be neighbors.

9.5.2 Construction of cell grid

In this part, we will be dealing with some GPGPU specific performance considerations. If this is unfamiliar territory, we have a brief introduction in the appendix.

A simple implementation of the regular cell grid is to have a single list in memory with an entry for each cell. This list is called the cell list CL and it has an entry for each cell in the regular grid, whether empty or full. The entry contains a reference to the first particle termed *First* and a count of the particles in the cell termed *Count*.

The particles should be stored in a sorted list called the particle list PL and this list should be sorted based on the particles cell index, so that all particles in any given cell will be stored sequentially in the list. This allows us to look in CL and see both how many particles are stored in any particular cell as well as the first particle in PL belonging to that cell. This method is widely used and seen in several other works such as [Ihmsen et al., 2011, Goswami et al., 2010, Krog, 2010, Capone, 2010].

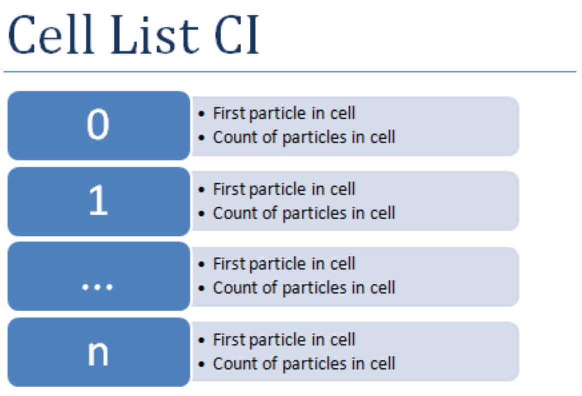


Figure 50: The Cell List CL containing n entries for n cells. Each entry containing information of the first particle in the cell as well as the number of particles in the cell

Acronym	Description
CL	Cell list, list of all cells
CI	Cell Index, index of some cell in CL
PL	Particle list, list holding all particles
PI	Particle Index, particles position in PL
P	Particle, some particle

The method works as follows. Each particle has a particle index PI which is not stored with the particle but rather derived from the particles position in the PL. Thus the particle with PI=5 will be stored at PL[5]. When updating the cell grid, the particles each recalculate the index of the cell to which they belong. This is called the Cell Index CI. The CI is stored as an attribute of the particle itself. Then the particles are sorted based on their CI. This results in a usable list of particles which has particles belonging to the same cell follow sequentially.

To update the cell grid, the CL is initialized so that each CL.Count is set to zero and each CL.First is set to some large value greater than the number of particles.

Now each particle P with index PI writes into CL[PI].First min(CL[P.CI].First, PI). This will cause all particles in a given cell to attempt to write their particle index PI into the same CL.First, but eventually only the particle with the lowest index will have its value stored. At the same time, each particle increments the counter CL.Count of its cell so that the count will show how many particles belong to the cell.

```
initialize CL
parallel foreach particle P in PL
  P.CI = calculateCellIndex(P)

sort(PL after CI)

parallel foreach particle P in PL
  atomInc(CL[P.CI].Count)
  atomMin(CL[P.CI].First, PI)
```

It should be obvious that the part of this update which takes the longest time is the sort which is $O(nlogn)$ as opposed to the other operations that are $O(n)$. The sort can utilize concurrent execution on the GPU, but not perfectly. The cell index calculation, and atomic increment and min, operations are on the other hand $O(n)$ and they *can* be perfectly parallelized. There can be overhead from atomic operations accessing the same memory location, and there are a limited number of atomic operation units, but on the Fermi architecture this is no longer considered a great problem. It should, however, be kept in mind that atomic operations accessing the same variables, should be interleaved in time so that overlap is as small as possible. According to [Goswami et al., 2010] the sorting sort can still be done relatively fast using parallel radix sort and taking advantage of the fact that the particle list will form an already almost sorted list. In our implementation we use the parallel radix sort of the Thrust library for GPGPU computing `http://code.google.com/p/thrust/`.

9.5.3 Neighbors in cell grid

To find the potential neighbor particles in the same cell as the querying particle itself it only a matter of looking up the particles own Cell Index CI and looking at the Cell List CL[CI] to find the first particle in the cell as well as the number of particles. Assuming the first particle is F and the number of particles is n, then all particles in the same cell as the querying particle is stored in the Particle List PL[F..F+n].

To find the potential neighbor particles in any of the neighboring 26 cells, the same procedure is repeated, only now we use the Cell Index of the Neighboring cells. How this is actually done depends on how the 3D structure of the cell grid is unwrapped into 1D indices, which is the subject of the next section.

9.5.4 Load balancing

In distributed computing the concept of load balancing is an important one. What this means is that for n equally powerful processing units, we desire to partition the work into n equally demanding work items. This way we do not overburden one unit while some other unit stands idle.

If it is unclear how exactly the total work is split into equal sized jobs, the simple solution is to partition the work into a much larger number of smaller jobs. A number much higher than the number of processing units. Then each processor can work on a subtask, and if it just so happens that the assigned subtask is easy to complete quickly, then the processor can take on a new small job.

This way a processor may complete a large number of tiny jobs or a single large job, and the processors will generally complete at the same time. Taking this partitioning to the extreme with an infinite number of infinitely small jobs, the processing units will complete exactly at the same time.

There is however an overhead to loading and unloading jobs which makes this intractable. Furthermore, every job has some sequential operations which prevents it from being sliced into infinitely small sub tasks.

When programming for the GPU, we have a number of multiprocessors MP which have identical performance as outlined in table 1. The on chip memory on each MP is divided into blocks of 16kB and 48kB where one is used for shared memory and one is used for L1 cache. It is user configurable which block is used for what.

We assign them work through thread blocks consisting of an integer number of threads. Each block must have the same size. How large a block can be and how many blocks any MP can load depends on how demanding it is to run the kernel code for a block. If it is a very light weight work, then the multiprocessors can load and process a large number of threads at one time. This can be partitioned into many small blocks or a single large block. Complex code will require a larger number of registers, and possibly shared memory than simple code will.

To illustrate the complexity of this, we run through a short example.

Consider a kernel where a thread requires 26 registers and no shared memory. We could load the maximal number of threads on the MP, which is 1536. This would then require 1536 threads x 26 registers/thread = 39936 registers. This requires more registers than are available, so we need to cut down on the threads. Given the register requirements, we can only load 32768 registers / 26 registers / thread = 1260 whole threads.

The question is then how we best partition 1260 threads into blocks. It could be 3 blocks of 315 threads which would fill the MP entirely by using all registers. There is a caveat though. Threads are executed in batches or 16 threads called a half warp. These threads execute in truly parallel lockstep. 315 is not a multiple of 16, so the processor should load either 19x16 threads = 304 threads or 20x16 threads = 320 threads. We only had registers to serve 315 threads, and now we have to settle for only loading 304 threads. We now have 3 blocks of 304 threads or 912 threads in total and use 912 threads x 26 registers/thread = 23712 registers out of the available 32768 registers. Now we are wasting registers, and further more, we better hope our job splits easily into blocks of 304 sub tasks.

The next question to ask is how the 304 sub tasks relate to each other. Do they access the same areas of memory, so they can cooperate around the cache and do they pass

NVidia GTX 580 Fermi architecture	
Multiprocessors	16
Max threads in block	1024
Max threads on MP	1536
Registers per block	32768
Max blocks per MP	8
Shared memory per MP	16kB/48kB
L1 cache per MP	48kB/16kB
Threads per warp	32
Threads per half warp	16

Table 1: GTX580

through the same code paths in their half warps so we avoid warp divergence?

The above example was only meant as a brief example of the complexities of optimal load balancing. Given various constraints, not every job can be easily partitioned into optimal sized blocks. For this reason, we need to consider carefully how to partition the job.

Note that the MP has 32 alus and can therefore run in true parallel for 32 threads. We only need extra threads to hide latency.

When a thread block completes execution, it will be unloaded from the MP, and a new block will be loaded and executed. This continues until all blocks have completed. An integer number of blocks can be loaded onto a multiprocessor,all blocks must have the same number of threads and the maximal simultaneous number of threads is a constraint. This means that to fully load the processor, we need to figure out a block size which will fill the processor completely with threads.

when a block completes , it is unloaded and a new one can start. A fast block is therefore ok, as is a slow block, but it is optimal if all threads are equally hard so they all complete at the same time. In fact we just need the threads in a warp to complete at the same time, but THAT is desired! therefore threads should be homogeneous inside a half warp which makes the same requirement for a block in general.

Lets assume we can have n simultaneous threads. We now seek a block size

Having decided how to split the particles into smaller sub domains is only the first piece of the puzzle. We now need to consider how the particles should be stored in memory to give the best results. While the spatial domain for the simulation is 3D, memory is 1D. This means that we need to unfold the 3D spatial partitioning grid into a 1D chain of cells. This can be done in several ways which all have their pros and cons.

We will in the following consider a 3D linear type unfolding as well as a space filling curve SFC based method.

If we assign particles to thread blocks based on their initial index, first particle to n'th particle to thread block 0 and n'th+1particle to 2n'th particle to thread block 1 and so on, we are likely to have particles with very different spatial positions in the same thread block. Different spatial positions will mean different sets of neighbors and generally very different particles. This is not optimal. If we instead assign all particles in a given spatial cell to a thread block, then we have the problem that not all cells have the same number of particles. Some will have many particles and others will have very few, or none at all. This means that we cannot assign the perfect number of particles to the thread blocks in order to obtain full occupancy.

We seek to obtain full occupancy on the streaming multiprocessors MP. For this to happen, we need to launch a precise number of threads on each MP. What that precise number is, it depends on the actual hardware being used and it depends on the complexity of the thread code being executed. To better utilize the cache, we also seek to handle particles close in space in the same thread block. Two particles which are neighbors will commonly be in the same cell and therefore search for other neighbors in the same set of 28 cells. They will also tend to share the same set of neighbor particles with a few exceptions. This means that they will want to access the same memory locations and that that memory will commonly be quickly accessible through the cached.

We will be using a Space Filling Curve SFC as described in [Goswami et al., 2010] where each domain subdivision cell has a SFC-index which indicates its location along the curve.

We use a SFC only to ensure that particles close in space are generally also close in memory. This does help in memory locality and cache utilization, but the primary goal is to be able to easily select n particles from a linear segment of particles in memory and have those particles in that linear segment be from the same spatial neighborhood.

A 3D space filling curve, as seen in Figure 51 is a special form of a winding curve which spans the entire unit cube. It does so in one continuous path without the large jumps as seen in a linear ordering in Figure 52, which is the more common form of memory layout.

There exist a number of space filling curves, which are closely related to fractals in that they can be continuously refined to an infinite resolution and when doing so they show a clear self similarity at all detail levels. The first, and probably best known is the Peano-Hilbert curve, but we choose to use the simpler Z-order Morton curve since it is very simple to implement by bit interleaving as described in section 9.5.5.

It is clear that, for the linear ordering, particles which are on the same horizontal line are somewhat close on the curve except when the line shifts to the line above or below. In contract on the Z-order curve, particles are generally close, on the curve, to both the neighbors above and below as well as to the left and right side. Additionally all particles in the same power of two block are contained in a continuous segment of the curve. It is not clearly seen that the Z-order curve is better than the linear ordering in this small example, but as the number of dimensions of the space spanned by the curve grows, and as the size of the dimensions grows,

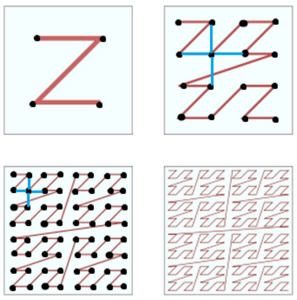


Figure 51: A particular form of space filling curve is the Z-order curve. here shown in 2D for four different detail levels. [Wikimedia commons]

the linear ordering will be increasingly bad since it prioritizes only 1D neighbor relations along a single axis, while the Z-order curve weights all three axis equally. If we consider a particle and its 4-neighborhood in 2D, as seen as blue lines in Figure 51 and Figure 52, we see that using the linear ordering, the distance to the neighbor above and below will grow as $O(\sqrt{n})$ for all particles resulting in $O(n\sqrt{n})$ while the same does not happen for the Z-order curve. For most particles it does not change the path distance to the neighbors at all that the dimensions of the domain grows. As drawn in the figures, the distances in the linear ordering grow from (1+1+4+4=10) to (1+1+7+7=16) while the Z-order curve distances grow from (1+3+2+6=12) to (1+3+2+6=12). It is clear that for very small curves in low dimensions, the Z-order curve is not as good as simple linear ordering. For 1D, the linear ordering is optimal in fact, but for higher dimensions and for longer curves the spatial locality is much better carried over to 1D memory layout with the Z-order curve.

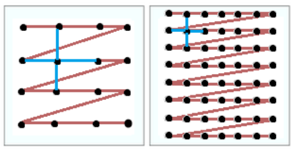


Figure 52: Linear ordering shows a number of very large jumps at the end of each line.

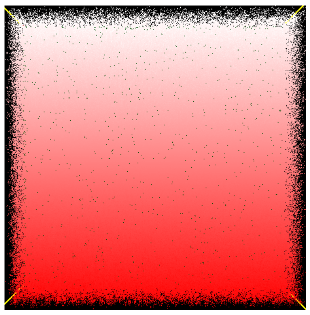


Figure 53: Particles colored according to their memory order using the method of linear ordering. Note how the particles are ordered as a linear gradient.

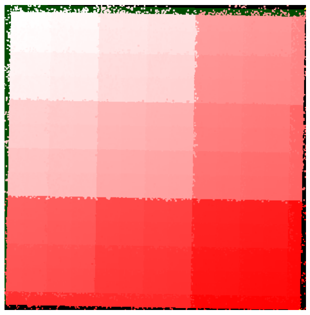


Figure 54: Particles colored according to their memory order using the SFC method. Note how the particles are ordered in square shapes which are contained in even larger shapes.

9.5.5 Calculating SFC index by bit interleaving

It is quite easy to calculate a 3D SFC index by using bit interleaving. In Figure 55 the column and the row indices of a 2D curve are printed in binary. Seeing that the curve takes one step along the columns, then one along the rows while stepping back along the columns and then along columns again before jumping back up the rows and advancing along the columns, there is a strong resemblance with how the bits in a binary number increases and decreases when counting in increments of one.

This pattern is used in calculating the curve index by letting the binary representation of the columns control every other bit in the curve index and letting the binary representation of the rows control the other bits. If we interleave the bits so that the n'th bit in the column index control the $2n't$ bit in the curve index and so every n'th bit in the row index control the $2n+1'th$ bit in the curve index, then we get the numbers seen in Figure 55. While the example is in 2D, this can easily be extended to 3D, or even higher dimensions, by still interleaving the bits so the index along each dimension controls every n'th bit in the curve index for a n dimensional curve.

To optimize the speed of this calculation, we do not extract every n'th bit in the x,y,z index in our 3D model and mix them together one by one. Instead we pre-calculate a table of indices where the bits have already be spaced for 3D use so that 1111 becomes 001001001001 as seen in table 2. This way we can quickly interleave three different indices by using two left shift operations and two OR operations. Given that shifting a binary number left two times is the same as multiplying with 2 two times, it can be written simply as

Original	Bit spaced version
00000000	000000000000000000000000
00000001	000000000000000000000001
00000010	000000000000000000001000
00000011	000000000000000000001001
00000100	000000000000000001000000
.....	
11111111	001001001001001001001001

Table 2: An example of the numbers 0,1,2,3,4 and 255 in bit spaced form

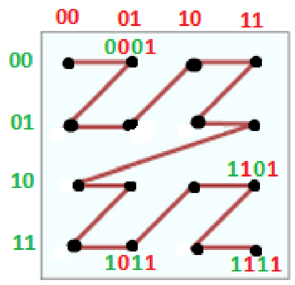


Figure 55: A Z-Curve can easily be constructed using bit interleaved indices. Notice how every other bit is taken from the column and row indices repectfully when concatenating the Z-curve index

$$SFC(x,y,z) = 4Bit(z) + 2Bit(y) + Bit(x) \tag{115}$$

where x,y and z are the integer spatial coordinates, and $Bit(x)$ is the bit spaced version of the number x, as just mentioned.

9.5.6 Variable smoothing length

Generally the cell grid should have a cell size larger than or equal to the particles support radius, in order to have simple access to potential neighbors in the grid. When using variable smoothing lengths, some particles could, however, have supports which are larger than the cells in the grid. The solution would generally be to either make sure the cells are initially large enough, to change the cell size dynamically or to take into account more than the closest 26 neighbor cells.

Neither solution is very attractive, but if the grid is laid out along a space filling curve, then this can be done more easily.

When using the SFC layout, it is trivially easy to use a cell grid with a cell size of S *as if* it was of size $(2^n)S$ for any positive n. This is due to the fact that any power of two block in the grid will consist of sequential cells as seen in Figure 56. It is evident that we can equally well consider the elements in the grid as arranged in one large cell, four smaller cells or 16 even smaller cells. This is however not a feature we will be taking advantage of in this project.

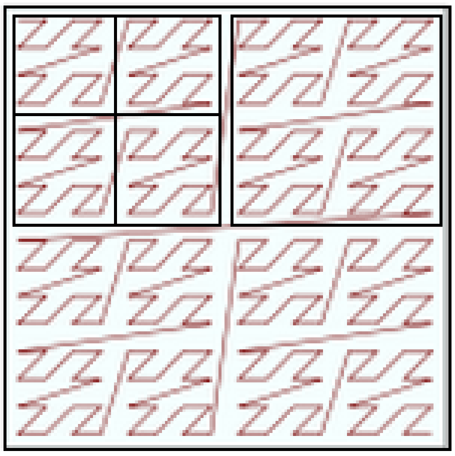


Figure 56: Note how the elements in any power of two block comes in order and how there are small distances inside any power of two block, while there are longer jumps between the end of one block and the start of the next.

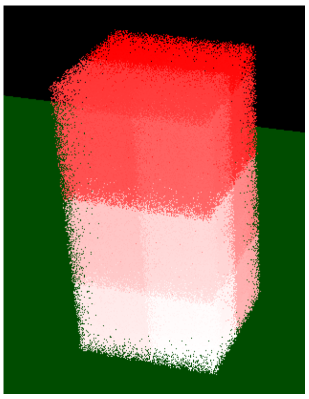


Figure 57: A 3D block of particles with color according to their memory location . Note the power of two blocks in all dimensions

9.5.7 Recalculate or store neighbors

We could generate a list of all neighbor interactions over all particles and once and for all calculate the different weight functions among every pair or neighbor particles. Afterwards, we could run through this list and distribute the interactions with one thread per interaction. This would avoid having to recalculate the weights repeatably, and for each particle, (double the work) but it would result in different threads adding forces to the same particles, which should have to be atomic operations. Those have the problem that threads may be paused while waiting for other threads to write their result to the particle.

Alternatively we could make simpler code which quickly finds the neighbors and calculate, what needs calculating.

The two approaches have been tested in [Ihmsen et al., 2011] and the conclusion was that the overhead of either dynamically adding neighbor lists or atomically writing into existing lists made the simple recalculating method just as fast.

Having to choose between two equally fast methods, where one is simpler than the other, we opt for simplicity and recalculate everything.

9.5.8 Cell update frequency

In the cell grid, we *must* recognize particles as neighbors if they SPH neighbors, but at the same time it is not imperative that we recognizes particles as non-neighbors if they are not SPH neighbors. It will be wasteful to look at potential neighbors, which are in fact not true neighbors, but it will not cause any errors.

The closest any two particles can be in the cell grid, without being in each others cell neighborhood, is $\min_(\Delta x, \Delta y, \Delta z)$ where Δx,Δy and Δz are the cell sizes along the x-axis, the y-axis and the z-axis. This is shown in 2D in Figure 58.

If we now consider two particles, which are not in each others cell neighborhood, then they will obviously not be neighbors in the SPH sense of the word either. We have already stated that a cell should always be *at least as large as the maximal SPH smoothing kernel radius.* If one or both particles now move over a cell boundary, they will be neighbors in the cell grid - they may even end up in the same cell - but it is not a certainty that they are also neighbors in the SPH way, which is defined as

$$|p_1 - p_2| \leq \max_{h_1,h_2} \tag{116}$$

where $|p_1 - p_2|$ is the distance between the particles and $\max_{h_1,h_2}$ is the maximal smoothing length of the two particles.

This is seen in Figure 59 where the purple particle is now in the red particles grid neighborhood, but not in its SPH neighborhood.

Give the minimal distance between particles outside cell grid neighborhood and given the maximal distance at which the two particles become SPH neighbors, we can derive a minimal time it will take for the two particles to go from not being cell grid neighbors to being SPH neighbors. This time will be based on the maximal particle velocity.

$$\Delta t \leq \frac{\min_{\Delta x,\Delta y,\Delta z} - \max_{p_h}}{2\max_{p_u}} \tag{117}$$

Here Δt is the upper bound on time interval between cell grid updates. The other Δ's again signify the dimensions along each coordinate axis, p_h is particle smoothing length and p_u is particle velocity.

What we state is that the minimum distance a pair of particles should move towards each other is the cell size minus the largest smoothing length. The time it takes, to move this distance, is found by dividing it by two times the maximal particle velocity. It is two times the maximal velocity since the particles may be moving towards each other with the effective relative velocity of up to two times the maximal individual velocity.

While the above time constraint could be relaxed by looking at every particle pair and finding their actual relative velocity and actual distance, it would naturally be a huge $O(n^2)$ problem which is not worth the trouble. What we are looking for is just the maximal safe update interval, which is generally much larger than the SPH time steps.

The larger the cells, the smaller the smoothing lengths and the smaller the velocities, the longer we can wait before updating the cell grid. Given that the cell grid update can be quite computational intensive, this is a most desirable property. Strangely enough, we have not seen this optimization employed by other grid based SPH implementations.

Finally, note that any non zero time step will be too long to ensure detection of particles moving *out* of each others neighborhoods. This is the wastefulness we mentioned initially.

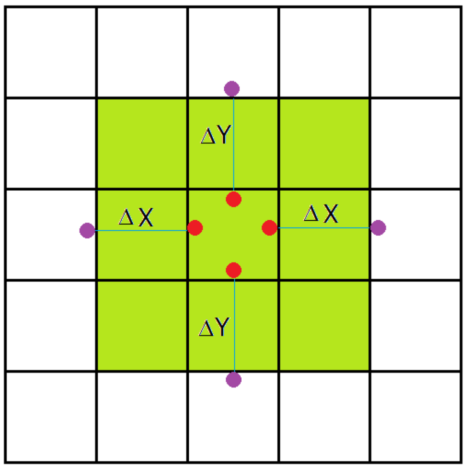


Figure 58: The neighborhood of the red particles in the center is shown in green. Four purple particles are each as close to a red particle as possible - without entering the neighborhood. In a grid with cell width Δx and cell height Δy this minimal distance is $min(\Delta x, \Delta y)$.

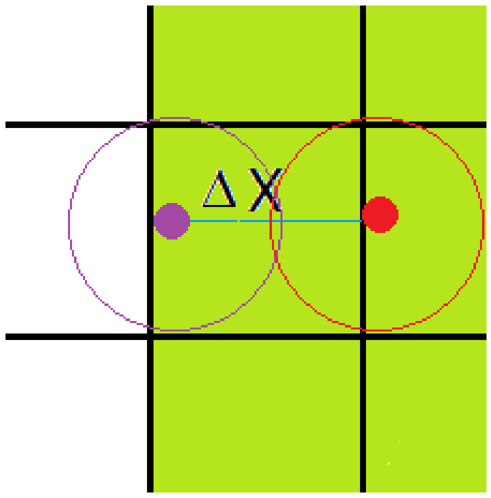


Figure 59: A purple particle has moved into a red particles cell grid neighborhood, but even though the smoothing kernels touch, neither particle is in the other particles SPH particle neighborhood.

9.6 Numerical integration

In order to advance the system forward in time, from $y(t)$ to $y(t+\Delta t)$, we work with two differential equations. One is the particle acceleration $\frac{\partial u}{\partial t}$ from (93) and the other is the particles current rest density $\frac{\partial \rho_{rest}}{\partial t}$ from (92).

We will use a somewhat expensive complex of numerical integration [Kreyszig, 2005], known as the Runge Kutta Feldberg method, RKF.

This method was chosen because it is fifth order accurate, which may allow us to use larger time steps in the presence of stiff systems and because it has a built-in support for variable time steps.

We will briefly describe the method in the following, but refer the interested reader to [Kreyszig, 2005] for further details. The method will be written in terms of time integration, though, as with other methods of numerical integration, it can integrate over all types of domains.

The method used six evaluations to integrate forward in time using both a fourth order and a fifth order method, written below as y^5_{n+1} and y^4_{n+1}. The difference between the two methods gives an estimate of the error ε

The six function evaluations are defined as $k_1..k_6$

$$\begin{aligned}
y_n &= y(t)\\
k_1 &= \Delta t f(y_n)\\
k_2 &= \Delta t f\left(y_n + \frac{k_1}{4}\right)\\
k_3 &= \Delta t f\left(y_n + \frac{3k_1}{32} + \frac{9k_2}{32}\right)\\
k_4 &= \Delta t f\left(y_n + \frac{1932k_1}{2197} - \frac{7200k_2}{2197} + \frac{7296k_3}{2197}\right)\\
k_5 &= \Delta t f\left(y_n + \frac{439k_1}{216} - 8k_2 + \frac{3680k_3}{513} - \frac{845k_4}{4104}\right)\\
k_6 &= \Delta t f\left(y_n - \frac{8k_1}{27} + 2k_2 - \frac{3544k_3}{2565} + \frac{1859k_4}{4104} - \frac{11k_5}{40}\right)
\end{aligned} \tag{118}$$

$$\begin{aligned}
y_n &= y(t)\\
y_{n+1} &= y(t+\Delta t)\\
y^5_{n+1} &= y_n + \frac{16k_1}{135} + \frac{6656k_3}{12825} + \frac{28561k_4}{56430} - \frac{9k_5}{50} + \frac{2k_6}{55}\\
y^4_{n+1} &= y_n + \frac{25k_1}{216} + \frac{1408k_3}{2565} + \frac{2197k_4}{4104} - \frac{k_5}{5}\\
\varepsilon(t+\Delta t) &= y^5_{n+1} - y^4_{n+1}
\end{aligned} \tag{119}$$

If the maximum norm of the error estimate $|\varepsilon|_\infty$ is above some threshold we half the step size and if it is below some other threshold, we double it

$$\Delta t = \begin{cases} \frac{1}{2}\Delta t & ; |\varepsilon(t+\Delta t)|_\infty > \varepsilon_{\text{upper limit}}\\ 2\Delta t & ; |\varepsilon(t+\Delta t)|_\infty < \varepsilon_{\text{lower limit}}\\ \Delta t & ; otherwise \end{cases} \tag{120}$$

Note that $\varepsilon(t+\Delta t)$ is the error at $t+\Delta t$ and that it is a vector. This means that we have an error estimate for every parameter of the system. We *can* therefore look at the sensitivity of individual properties of individual particles.

We feel that this method is ”simpler” than some other methods commonly used in interactive physics simulations, such as as Leap-Frog or the Verlet variants. This is because every parameter of the system is represented at the same points in time, which is not true for Leap-Frog, and because those parameters are given explicitly, which is not true for the Verlet methods where some derivatives are implicit and depending on the time-step size. This in turn means that re-computation is required, if the step size changes.

RKF is however more computational intensive, per step, using 6 function evaluations compared to the 2 used in simpler methods, but this can for stiff systems be offset by larger time steps, which gives a larger total simulation to computation ratio $\frac{\Delta t}{computations}$.

Time-step constraints Usually the CFL condition [Liu and Liu, 2003] would need to be calculated based on particle velocities, smoothing lengths and physical stiffness. From this CFL calculation, we would have an upper bound on the time step Δt. This is not needed when using the RKF method, where there is a clear error estimate every step.

9.7 Analysis of SPH snow

In this section we will analyze the performance and the correctness of the SPH snow model. Snow will only be affected by a constant and homogeneous wind field to focus on the SPH alone.

9.7.1 Friction angle

Friction angle is also known as angle of repose. We know from [Cresseri and Jommi, 2005] that snow with a density of $300 kg/m^3$ should have a friction angle of 63 degrees.

Hypothesis When snow is dropped in a pile, its sides will have an angle of $63°$.

Test 512.000 particles, each with a mass of 1.95 grams (total mass 1 ton) were created with a rest density of 300 kg/m^3 and and initial shape of a cube resting on the ground plane. Gravity was turned on over a period of 5 seconds and the particles were allowed to sink together and spread out. The floor was made to act like a no-slip boundary.

The slow onset of gravity is to avoid an impact-like effect when the gravity force suddenly went from 0 m/s^2 to $9.82 m/s^2$. A sharp onset of gravity tended to make the snow pile flatten more than when gravity slowly came on. Considering that friction angle has nothing to do with the shape of the pile when dropped from a height, but rather with the relationship between inter-particle friction versus shear stress from particles pushing in between each other due to gravity, it seems to be correct to use the gentle gravity.

When the floor was not a no-slip boundary, the pile also flattened more than else. This is due to the fact that only the rigidity of the snow itself was keeping it from shearing. With a no-slip boundary, that boundary aided in resisting shear stresses, which is what would be seen in the real world on any surface with a non zero friction quotient. The show can then get a ”foothold” on the ground.

The simulation was stopped when the particles came to a rest after approximately 5 seconds and the friction angle was measured on a screenshot from an an angle parallel to the ground and from a point one half snow-pile-height from the ground.

60 degrees.png

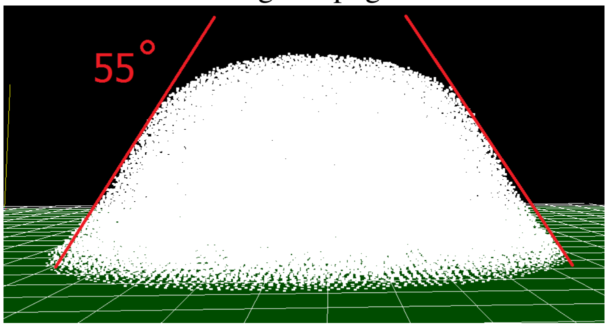


Figure 60: Pile of snow showing close to the expected 55° friction angle

Result The resulting pile and superimposed friction angle is seen in Figure 60. It is unclear how exactly the friction angle should be measured, considering that the slope is not a straight line. Therefore it was done by manually drawing a line which coincided reasonably with the side of the pile and then measuring that line. The result of 55° was considered a reasonable value, considering the high degree of uncertainty for the viscosity.

Changing the yield stress for the snow, we were able to make more rigid snow as seen in Figure 61. What that figure shows is also how arbitrary friction angle measurements can be, given that, in that example, it can range from over 80 degrees to 50 degrees. The reason the top is so rounded is due to an initial collapse of the sides. The sides are only ”supported” from one side and are therefore more likely to fall down and outwards under shear stress.

other degrees.png

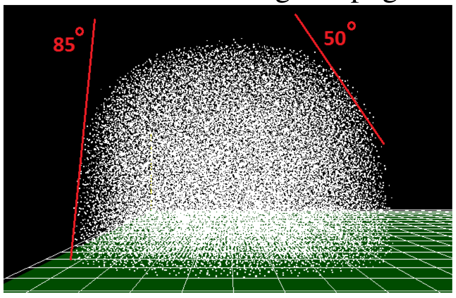


Figure 61: Pile of snow with higher friction angle due to a higher viscosity. The particles are rendered small to better visually separate them.

9.7.2 Rigidity of compressed snow

It is the assumption that the higher the density of the snow, the higher the rigidity. The literature does not state a concrete relationship factor between friction angle and density, though it does relate Young's elastic modulus and shear modulus to density and this to some extent can be related to friction angle. The test of rigidity was therefore somewhat loose and its primary purpose was to see that the density dependent rheological behavior was giving more solid snow, when compressed, and that the adaptive rest density was in fact working so that the snow would not explode outwards after having been compressed.

Hypothesis Snow will increase in density when compressed, and using adaptive rest density, this will not result in a permanent state of stress. The compressed snow will be more rigid after compression.

Test Again 512.000 particles were generated with an initial rest density of $300kg/m^3$ and a particle mass of 1.95 grams. That is approximately one ton of snow. The initial domain, in which the particles were created, was $3.33m^3$. Then the domain was reduced in volume to $1.25m^3$ which resulted in the density increasing to $800kg/m^3$, which is almost the density of ice. The domain boundaries were then expanded to several meters in either direction of the ice block - keeping the floor boundary constant.

Result The rest density of the particles was measured after initial compression, and, with some very small variance due to irregular packing of the particles, it was $800kg/m^3$. The result is seen in Figure 62. As expected, the new rest density of the particles meant that there was no stress inside the cube, and therefore no outward force made the cube explode, as would have been the case if the original rest density of $300kg/m^3$ had been maintained. The cube was now keeping its shape and remained a cube.

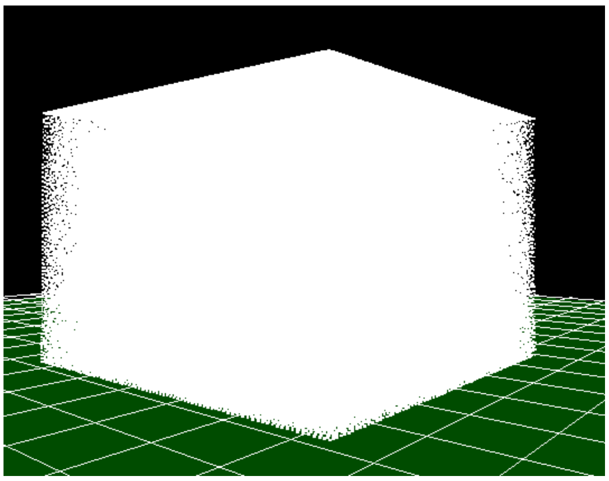


Figure 62: A very compact block of snow particles now forms an ice cube with a density of $800kg/m^3$

9.7.3 Young's modulus sanity check

We will consider a layer of snow with a depth of $L = 1m$ and a density of $\rho = 60kg/m^3$ as newly fallen snow. On the snow, we put a person wearing a pair of snow shoes. The person has a mass of $m = 80kg$ and the gravitational acceleration is $g = 9.82m/s^2$. The combined area of the snow shoes is $A = 2(0.76m \cdot 0.25m) = 0.38m^2$.

The shoe area is based on mountaineering shoes and information from Wikipedia stating

"Sizes are often given in inches, even though snowshoes are nowhere near perfectly rectangular. Mountaineering shoes can be at least 30 inches (76 cm) long by 10 inches (25 cm) wide."

We now observe how deep the person sinks. This is done by using the small strain tensor and assuming constant density during compression.

We rewrite the strain tensor to isolate the new length l from the initial length L

$$\begin{aligned} \varepsilon &= \frac{l-L}{L} \\ l &= \varepsilon L + L \end{aligned} \tag{121, 122}$$

Hypothesis Based on very basic personal snow experience, we expect the person to sink some $0.2m$ into the snow.

Results The calculations are as follows

$$\begin{aligned} E(\rho) &= 187300e^{0.0149\rho} && = 457928Pa \\ \sigma &= \frac{-g \cdot m}{A} && = -2067Pa \\ \varepsilon &= \frac{\sigma}{E} && = -0.0045 \\ l &= \varepsilon L + L && = 0.995m \end{aligned} \tag{123}$$

That was not as expected. The person sank 5mm into the snow. Snowshoes may be good, but not that good.

This unfortunately shows us that our expression for Young's modulus is not accurate. It is not entirely unexpected since the measurements of the modulus have been based on settled snow - albeit at different densities - while we are using it on loose snow here.

In the interest of curiosity, we could reverse the calculations and see what Young's modulus should have been, if we should see the $20cm$ depression. We note that 20cm of 100cm is not really a small strain any longer, so we use the Green strain tensor instead.

$$\begin{aligned} l &= L - 0.2m && = 0.8m \\ \varepsilon_G &= \frac{l^2 - L^2}{L^2} && = -0.36 \\ \sigma &= \frac{-g \cdot m}{A} && = -2067Pa \\ E &= \frac{\sigma}{\varepsilon} && = 5742Pa \end{aligned} \tag{124}$$

That value of Young's modulus is some 1.25% of the value we have been using. The reason it has not made everything else entirely wrong, is that wind blown snow is generally not very stressed. It just blows around feeling nice and loose.

It is surprising, though, that Young's modulus seems to be so far off, while the yield shear stress, derived much the same way, is so accurate. We have no obvious explanation for this yet.

Sanity check (p74) revisited

The following seven pages are from the thesis defence. At that point I had realized why the "sanity check" of Youngs modulus failed.
Please read this to get the previously missing explanation.

Person standing on 1m deep snow.
Snow density is $\rho = 80kg/m^3$.
Person mass $80kg$ (including backpack).
Shoe area $A = 0.38m^2$.
How deep will she sink?

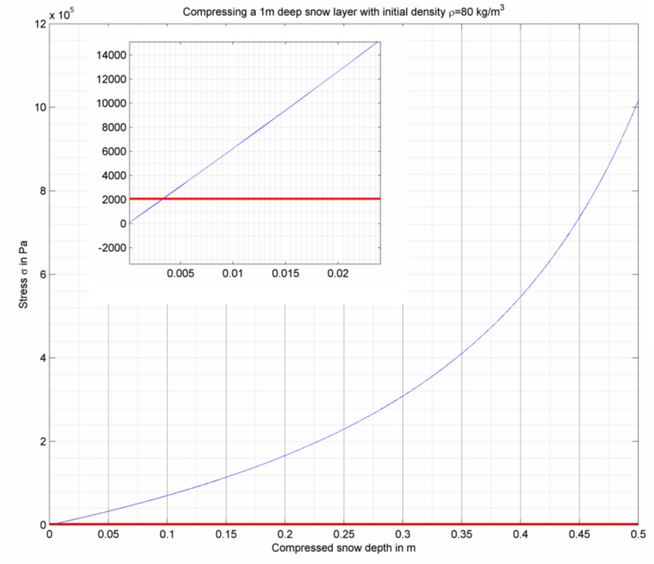
Sanity rechecked 2

Using a slightly higher density $\rho = 80 kg/m^3$ than in paper $\rho = 60 kg/m^3$.

$$\begin{aligned} E(\rho) = 187300 e^{0.149\rho} &= 616903 Pa \\ \sigma = \frac{-gm}{A} &= -2067 Pa \\ \epsilon = \frac{\sigma}{E} &= -0.00335 \\ l = \epsilon L + L &= 0.997 m \qquad (10) \\ & \qquad (11) \end{aligned}$$

We expected some $0.20m$ compression and got nothing near that.

Sanity check - stress during compression



Sanity check - what is missing

For brittle materials, the failure strain is commonly a constant, and for snow it is approximately $\epsilon = 0.002$ *for all densities of snow.*

At what compression can the resulting normal yield stress carry the person?

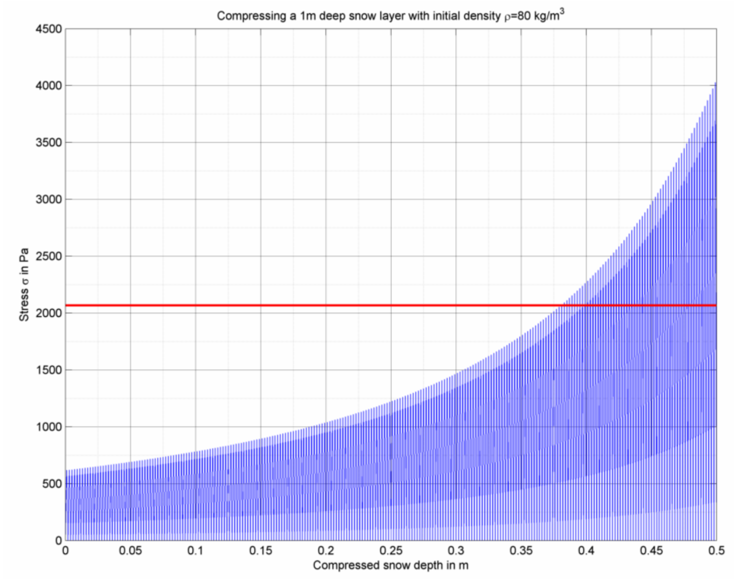
Sanity check - stress during compression WITH failure

Now assuming constant strain and total failure.

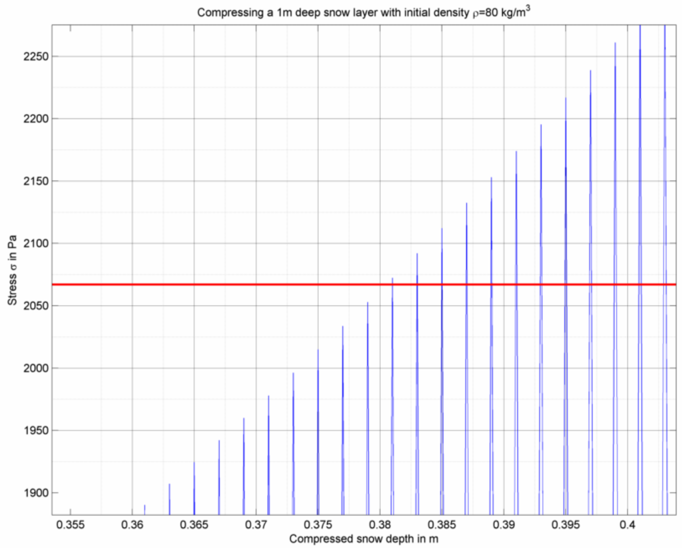
$$\begin{aligned}
E &= \frac{\sigma}{\epsilon} \\
E &= \frac{-2067}{-0.002} && = 1033500Pa \\
E(\rho) &= 187300e^{0.0149\rho} && = 1033500Pa \\
\rho &= 129kg/m^3 && \qquad (12)
\end{aligned}$$

$$sink = 1m - \left(1m\frac{80kg/m^3}{129kg/m^3}\right) = 0.379m \qquad (13)$$

Sanity check - stress during compression WITH failure



Sanity check - stress during compression WITH failure, Zoom



9.7.4 SPH Time step size

The size of the time step for the SPH implementation was in the range $0.0005s \leq \Delta t \leq 0.7s$.

The higher step sizes were when no snow had yet landed on the ground. At this time, the particle interaction was very low. The lower step sizes were when some snow was fixed at the ground and other snow fell into it at speed. Due to the high friction quotient and high relative velocity, this was to be expected.

When looking at the individual particles' error estimates ε, just before shrinking the time step, as defined in section 9.6, we note that only a few of the particles have too large errors, while the large majority is well below the threshold which can be seen in a histogram plot of ε.

9.7.5 SPH steps per second

The stepping speed of the SPH was found to scale almost linearly with the number of particles as seen in table 3.

Num particles	Step time	Step time per particle	Steps per second
512,000	6 ms	12 μs	167
1024,000	15 ms	14 μs	67
2048,000	39 ms	19 μs	26

Table 3: Time summary of SPH model

Part II

Wind simulation

10 Computational Fluid Dynamics

This section will give a quick introduction to general Newtonian fluid dynamics, with focus on the basics and the general parts. Later in section 12 it will be made more concrete.

10.1 Fluid flow

A fluid is commonly defined as a material which cannot resist shear stress. It is clearly seen in, for example, water which is highly resistant to normal stress, in that it can only be compressed with great difficulty, but which flows easily when sheared - slided against itself. The only shear strain resistance comes from the fluids viscosity, which is for fluids such as water or air, very low.

A general model for the behavior of fluids is the Navier Stokes equations for fluid flow which relate terms such as external forces, viscosity, pressure gradient and flow velocity.

The general Navier Stokes equations for compressible fluid, which is derived in [Liu and Liu, 2003] among many other places, is written as

$$\rho\left(\frac{\partial u}{\partial t}+u\cdot\nabla u\right)=\ldots$$
$$-\nabla p+\mu\nabla^2 u+\left(\frac{1}{3}\mu+\mu^{\upsilon}\right)\nabla(\nabla\cdot u)+f \qquad (125)$$

The third right hand side term $(\frac{1}{3}\mu+\mu^{\upsilon})\nabla(\nabla\cdot\upsilon)$ is describing how much the fluid resists compression depending on the bulk viscosity μ^{υ}. We note that if there *is* no compression/decompression, then the divergence $\nabla\cdot u$ will be zero which makes the entire term vanish. This leaves us with the Navier Stokes equation for *in*compressible fluid and the additional constraint of zero divergence.

While no physical substance is truly incompressible, and atmospheric air, which we will be modeling, is most certainly not, the incompressibility assumption is commonly used. As a rule of thumb, fluids flows with a Mach number Ma below 0.3, can reasonably well be considered incompressible.

$$Ma=\frac{u}{c} \qquad (126)$$

The fluid velocity is u and the speed of sound c in a fluid is described by

$$c=\sqrt{\frac{\mu^{\upsilon}}{\rho}} \qquad (127)$$

which relates directly to the speed of sound in a solid as seen in section 6.10, jot now using the bulk viscosity in place of Young's modulus.

Given that the Mach number is defined as velocity relative to the speed of sound in the fluid, and that this speed of sound is defined from the bulk viscosity, we see how Mach numbers serves as an indicator of the influence of the compression term. If the flow is close to the speed of sound, the fluid may build up locally and increase its density. Fluid can flow into a region faster than the pressure forces can push it away again, since pressure moves at the speed of sound... or rather sound moves at the speed of pressure waves.

The speed of sound in atmospheric air at $-10°C$ is $325.16m/s$ meaning that for velocities below $0.3(325.16m/s) = 97.5m/s$ we can consider the incompressible case. Even a hurricane has wind speeds below this, so in our case, incompressibility is a valid assumption.

We rewrite the equation to give rate of change of velocity for incompressible fluid.

$$\begin{aligned}\frac{\partial u}{\partial t} &= \frac{1}{\rho}\mu\nabla^2 u - \frac{1}{\rho}\nabla p - (u\cdot\nabla)u + \frac{1}{\rho}f \\ \nabla\cdot u &= 0\end{aligned} \tag{128}$$

Here the first equation defines the time derivative of the flow velocity u in terms of viscosity, advection, pressure and external forces. The second equation is the incompressibility constraint stating that the divergence of the flow should be zero. The division by density represents the inertia of the fluid, relating mass density to force density.

Note in this that f is not force, ordinarily written F and measured in Newton, but force density which is

$$f = \frac{F}{V} = \frac{N}{m^3} \tag{129}$$

10.1.1 Advection

The general advection expression is

$$u\cdot(\nabla\phi) \tag{130}$$

where the quantity ϕ is advected by a velocity field u. $u\cdot\nabla$ is sometimes referred to as the advection operator.

Advection of a quantity in a fluid is the transport of that quantity, due to the movement of the fluid.

If we look at the components in the advection operator working on ϕ, we see that it is just the dot product of the velocity vector and the gradient, spatial derivative, of ϕ. There is another word for that, which is directional derivative. The operator simply calculated the derivative of the quantity in the direction upstream, and scaled this by the magnitude of the velocity vector. Knowing if the value of ϕ is higher or lower upstream, lets us know what is coming. If the quantity increases upstream, then it will increase downstream over time, and if the flow velocity is high, this increase will come rapidly.

While the directional derivative view is easy to understand, it may sometimes be confusing that $u \cdot (\nabla u)$, as it is seen in fluid dynamics, is in fact the advection of the flow velocity itself, by that same flow, but again, if the wind speed is higher upwind, then that higher wind speed will come to us, with the wind, and the wind speed locally will increase.

10.1.2 Pressure

The pressure term

$$-\frac{1}{\rho}\nabla p \qquad (131)$$

states that there is an accelerating force from higher pressure towards lower pressure. The actual acceleration a is the force density $f = -\nabla p$ divided by the density of the fluid $a = -\frac{1}{\rho}\nabla p$. Given that the expression does not deal with macroscopic fluid volumes, the force is applied over an infinitely small surface on an infinitely small volume.

10.1.3 Viscosity

The viscosity term

$$\frac{1}{\rho}\mu\nabla^2 u \qquad (132)$$

describes the fluids resistance to internal movement.

If some parts of the fluid travel fast and some travel slow, the faster will speed up the slow, and the slow will slow down the faster. The rate at which this occurs depends on the scale of the relative movement as well as the viscosity which is the thickness of the fluid.

The Laplacian of the velocity field $\nabla^2 u$ is the divergence of the gradient of the velocities. A positive value then means that the velocity gradients point away, diverge. If the velocity gradients points away, then the surrounding velocity values must generally be larger. This in turn means that they will speed up the local velocity through sliding friction, e.g. viscous forces.

10.1.4 Fractional step method

One possible way of solving the equations is to evaluate each part on the right hand side of the time derivative $\frac{\partial u}{\partial t}$ of the velocity field, and them simply add them together.

This is the method known as the fractional step method. The problem with this method is that it disregards the constraint that the flow should be divergence free $\nabla \cdot u = 0$.

We therefore first find the new velocity field from its time derivative using some form of numerical integrations such as section 12.6 and then we fix the field to be divergence free.

10.1.5 Divergence free flow

Given a divergence full velocity field w, we seek to find the divergence *free* field u. This can be done using the Helmholtz Hodge decomposition of the field. This is based on Helmholtz's theorem, which is used as is.

Theorem 1. *Any sufficiently smooth and rapidly decaying vector field in three dimensions can be separated into the sum of an irrotational vector field and a divergence vector field.*

This lets us write the current divergent field w as a sum of an unknown divergent free field u and an unknown scalar field which we call ∇p.

$$w = u + \nabla p \tag{133}$$
$$\nabla \cdot w = \nabla \cdot u + \nabla^2 p \tag{134}$$
$$\nabla \cdot w = \nabla^2 p \tag{135}$$

Initially in (133) we write the values as described. Then we take the divergence on both sides in (134). Finally in (135) we use the fact that the field u is defined to be divergence free, meaning $\nabla \cdot u = 0$, which lets us remove it from the equation.

The final version is a Poisson equation which we will describe how to solve in section 12.7. Solving it will give us the unknown p.

The reason ∇p is used as symbol for the irrotional field is that in our case it is actually the gradient of the pressure field. The essence of the decomposition is that we find the pressure field which will exactly make the velocity field divergence free by returning to the initial version in (133) and rearranging to get an expression for the divergence free field.

$$w = u + \nabla p \tag{136}$$
$$u = w - \nabla p \tag{137}$$

This concludes the general introduction to fluid dynamics. In section 12 we will use this to model an actual fluid flow.

10.2 Turbulence

A fluid flow is generally laminar, transitional or turbulent. A common rule of thumb [Squires, 2008] is that Reynolds numbers *Re* below 2100 are laminar while numbers above 4000 are turbulent. The numbers in between are transitional with laminar and turbulent parts. In Figure 63 we see both laminar, transitional and turbulent flows.

$$Re = \frac{uL}{\nu} \tag{138}$$

where u is the mean fluid velocity relative to walls or obstacles, L is the characteristic length and ν is the kinematic viscosity. The characteristic length is somewhat vaguely defined in the literature, but it is commonly seen as the largest dimension of an object disturbing the fluid flow or the internal diameter of a pipe in which the fluid flows.

The Reynolds number describes the ratio of inertia to viscosity in the fluid. If the viscous dampening is much higher than the inertia, the higher frequency components of the flow, e.g. the turbulence, will tend to be dampened. A turbulent flow is characterized by a high degree of variation over time and locally very high pressure and velocity gradients, which means that the velocity diffuses more quickly than for laminar flow.

Given that turbulent flow has much large variation in space and time, accurate modeling of turbulence requires a very fine spatial and temporal resolution. This can be problematic for numerical solutions. From [Squires, 2008] we have a relation between *Re* and the required number of grid points. It is proportional to the Reynolds number in the $\frac{9}{4}'th$ power.

$$\eta = \left(\frac{\nu^3}{\varepsilon}\right)^{1/4}$$

$$N_x N_y N_z \approx (\frac{L}{\eta})^3 \propto Re^{9/4} \tag{139}$$

where η is the Kolomogorov length scale, ν is the kinematic viscosity and ε is the rate of energy dissipation.[Squires, 2008].

A similar calculation can be made for the temporal resolution, but we will not dive into the finer details of turbulence modeling here, but only repeat the conclusion from [Squires, 2008] which is that a very high resolution is required if the smallest eddies are to be calculated directly.

In practical fluid modeling, approximation methods are used. One of the more general methods appear to be the Large Eddy Simulation [Sagaut, 1998], LES, which was originally developed in the 1970's.

The method works by first smoothing the velocity field to arrive at a field with only the low frequency components. The finer detail turbulent motion is then approximated and added back in to the flow.

The effect is that the method explicitly calculate the scales that it can, and then approximate the scales, that the space and time grid cannot capture.

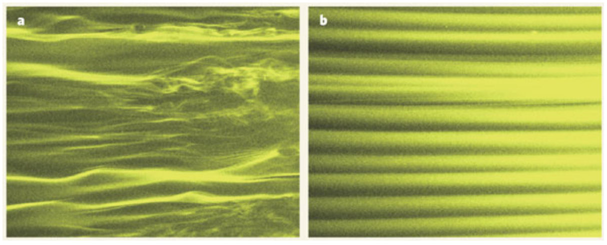


Figure 63: Two fluid flows. The leftmost shows a laminar flow transitioning into a turbulent flow, while the rightmost shows a pure laminar flow.[WikimediaCommons]

11 Finite Volume Method

The finite volume method, hereafter FVM, is used to discretize a continuum in order to solve a partial differential equation problem. It does this by partitioning the continuum into a set of sample volumes as seen in Figure 64. In the folowing, we will describe the so-called cell centered method.

Each volume, commonly referred to as cell, is bounded by boundary surfaces which either connect to a neighboring volume or which defines the outer boundary of the domain.

By observing the simple fact that change to the contents of any such volume will be due to flux over a volume boundary[4], we note that we can describe the development of a volume by the events taking place on its surfaces.

As an example, consider a fluid described by FVM. Each cell holds part of the fluid. If fluid moved out of a cell, it does this over

[4]Here considering only a material which does not itself change over time. It has a material derivative of zero.

a boundary, and by crossing a boundary between cells, it will at the same time move *into* the neighboring cell. If we know the fluid velocity for every point on the boundary, and we know the size of the boundary, then we implicitly know the amount of fluid crossing the boundary. Additionally, we know that we are never loosing any of the fluid since what leaves one cell will enter another. The exception to this is the outer boundaries where suitable boundary conditions need to be defined.

11.1 Gauss's divergence theorem

The Gauss divergence theorem (140) states that the outward flux of a volume is equal to the integral of the flux inside the volume.

This is intuitively understood if we consider exchange between two points inside the volume. One point receives and another gives. The sum is zero, and the total give-take inside the volume is zero. The only way to give or take between any two points, in a way that does not sum to zero inside the volume, is if this giving or taking is across the outer boundary. This in turn means that we can entirely ignore the flux inside the volume, which sums to zero, and only consider the flux over the boundary, as the theorem states.

In (140) the left hand side is the divergence, or the outflow, of some vector field quantity F in the volume. The right hand side is the outflow of that quantity over the surface of the volume. Here $F \cdot n$ is the dot product of the vector F and the normal vector of the surface, which is the same as the part of F actually crossing over the boundary - as opposed to moving parallel to the boundary. By integrating the part of F crossing the boundary moving outwards, over the entire surface, we have the total quantity leaving the volume - or the divergence.

$$\int_V (\nabla \cdot F) dV = \int_S F \cdot n dS \tag{140}$$

11.2 Conservation

The FVM is a conservative discretization method in that it does perfectly account for the location of any discretized quantity. Whenever something moves, it does so in between cells and what leaves one cell will enter another cell. At the outer boundaries, we will have boundary conditions which exactly state how the discretized quantity flows. This property makes FVM an attractive method for physical simulations where the partial differential equations are commonly constrained by conservation laws in order to provide unique solutions.

11.3 Boundaries

The general divergence theorem is valid for any shape of boundaries, but to make the surface integral more manageable for computation, the boundaries are defined as a piecewise continuous function. This way the integral over an arbitrary surface can be described as a sum over its pieces.

$$\int_S F \cdot n dS = \sum_e \int_{S_e} F \cdot n dS \tag{141}$$

If, further more, those pieces are simple linear functions, such as planes in 3D, then the surface integral can be easily evaluated

$$\int_S F \cdot n dS = \sum_e F_e \cdot n A_e \tag{142}$$

where A_e is the area of the the $e'th$ surface element and n is the normal vector of the element, which is a constant for a plane. The F_e is the average value over the entire face, which is explained in the next section.

11.4 Mean value theorem

The mean value theorem states that on an interval of any continuous function, or segment of a function, there will be at least one point on which the derivative of the function will be equal to the average derivative over the entire interval, as seen in (143). In the interval a to b, there will be a point α which has exactly the derivative of the average derivative over the interval.

$$f'(\alpha) = \frac{f(b) - f(a)}{b - a} \tag{143}$$

While this theorem may seem rather trivial, it does show that we can rewrite any function integral as a single mean value f_{mean} multiplied with the length of the segment $b - a$. Consider the line integral over x from a to b

$$\begin{aligned}\int_a^b f(x)dx &= (f_{mean})(b-a) \\ &= (f_{mean})c\end{aligned} \tag{144}$$

where c is the length of the segment.

This means that if we know the true mean value of the function, then integration is trivial. We can then assign that mean value to the surfaces surrounding a volume, and reduce the surface integral to a sum of simple multiplications between constant mean values and constant areas. The question is how we know this mean value.

11.4.1 Mean value by interpolation

In FVM we deal with volumes, and the surfaces bounding those volumes. We do know the integral of some quantity inside those volumes, but we do not know the exact distribution of the quantity.

Relying on the mean value theorem, we know that *some* point inside the volume will accurately represent the *exact* mean value.

Given that we do not know where that point is, or what it's value is, we have to make an educated guess. It seems reasonable to assume that the center of a volume may commonly give an appropriate representation of the mean value.

For this reason, we assign the volume center the value of the mean value. By assigning the mean value to the center point inside the volume, we can now interpolate between those known values, with known positions, in order to get approximate values for other positions as well.

Averaging scheme Following the same argument as for the volumes, we will say that a good mean value point for a surface is

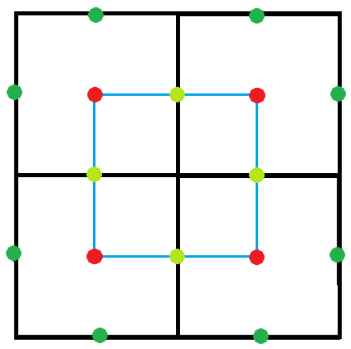


Figure 64: A mesh consisting of four cells. The mean value for each cell is assumed to be located at the cell center, and is indicated as a red dot. The blue lines represent interpolation between pairs of center mean values and the midpoint of this interpolation is represented by a green dot

the center of that surface. This is how we can now estimate the surface values, by interpolation, or averaging. This is seen in Figure 64, where we have mean values indicated in red, lines of interpolation indicated in blue and interpolated values indicated in green. The dark green values do not arise from interpolation, but are instead predefined boundary values.

In Figure 64 the mesh is particularly simple in that the cells are equally sized squares. This is in no way a requirement of the method, which supports arbitrary meshes. Given such a simple mesh, the mean value of any interior surface s separating the volumes A and B can be described through a simple average of the two volume values.

$$s_{mean} = \frac{A_{mean} + B_{mean}}{2} \tag{145}$$

Upwind scheme There an other note worthy way of estimating a value at the cell faces, when the value is transported by a flow field through the cells. It is known as the upwind scheme. We will not be using it in this work, but it is mentioned for completeness. As for the averaging method just mentioned, s_{mean} is the estimated quantity at the face, knowing the quantity in the two connected cells.

$$s_{mean} = \begin{cases} A_{mean} & ; u_e \cdot n_e \geq 0 \\ B_{mean} & ; u_e \cdot n_e < 0 \end{cases} \tag{146}$$

The n_e is the normal at the face pointing from cell A to cell B, and this time we also consider a flow field with velocity vector u_e at the face. It is this flow which transports the quantity through the cells.

Note that if the flow is moving primarily from A to B, we estimate that the quantity at A is also the quantity present at the face, given that it is constantly being moved from A to and through the face.

For more complex shapes such as Figure 65 the interpolation is much more complicated, regardless of the method used. In that example the line connecting the two center points does no longer intersect the center of the separating edge. Now other forms of averaging between known center points may be used, but we will not consider this problem here, but refer to [Versteeg and Malalasekra, 2007].

Staggered grid An alternative to the so-called ”cell centered grid”, as we have just

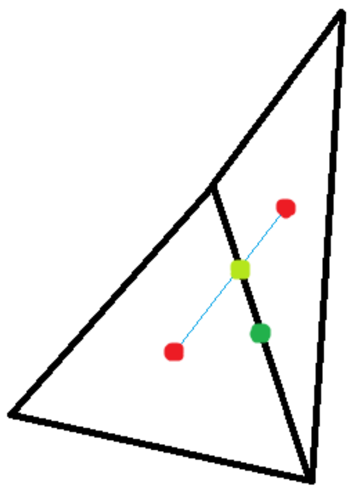


Figure 65: A two cell mesh consists of unequal triangles and the interpolation between center values no longer intersect the midpoint of the separating edge.

explained, is a staggered grid. Here, not all quantities are defined at cell centers and calculated at faces. Instead some are actually defined at the cell faces, or at the vertices joining cells. Commonly velocities are defined outside the cells that way.

While it does avoid some interpolation, and while it in some ways increase the effective resolution of the grid used in calculations of gradients and solve some potential problems known as checker boarding, we have chosen to not implement this method in this version of the simulator.

This was done to keep the expected complexity of coding low, though it would likely be worth revisiting that solution in a second iteration. For details on staggered grids in FVM we refer to [Versteeg and Malalasekra, 2007].

11.5 Boundary values

The previous boundary description deals with internal boundaries between cells in the mesh. The external boundaries are different in that their value, or their derivative, is given explicitly and not obtained through interpolation.

Where the boundary value ϕ_{ab} between cells A and B, sharing face f would otherwise be found through interpolation

$$\phi_{ab} = \frac{1}{2}(\phi_A + \phi_B)$$

it is now given explicitly for a Dirichlet boundary as $\phi_{ab} = value$ and implicitly in differential form for Neumann boundaries as

$$\frac{\partial \phi_{ab}}{\partial f_n} = value \tag{147}$$

11.6 Derivatives

Derivatives of the quantities described in the cells can commonly be found by considering cell center distances and cell center differences. While methods such Finite Difference deal exclusively with derivatives along the coordinate system axis, x,y,z, FVM is different in that derivatives are defined over boundaries instead. Knowing those derivatives lets us calculate the flux over a face. In other words, we do not deal with

$$\frac{\partial \phi}{\partial x, y, z}$$

for some quantity ϕ. Instead we consider the face derivative

$$\frac{\partial \phi}{\partial f_n}$$

for the face f with normal vector n. This can be evaluated for the cells A and B over the face f with the normal vector n away from cell A, where the cell center distance is $|B-A|$

$$\frac{\partial \phi}{\partial f_n} = \frac{\phi_B - \phi_A}{|B-A|} \tag{148}$$

This is the directional derivative over the face from A towards B. In that expression for face derivative, we have assumed that the mesh is well behaved in that the face normal is parallel to the vector between the two cell centers, or more formally

$$f_n \cdot \frac{B-A}{|B-A|} \approx 1 \tag{149}$$

where now A and B represents the coordinates of the cell centers.

11.7 Matrix formulation

Given the simple volume based description of field values as described, we can now easily write matrix equations representing values in a FVM mesh. We will show how to calculate the face velocities from cell velocities, the cell divergence from face values and the face gradient from cell values. Note that this is all we need in fluid computation given that the Laplacian is just the divergence of the gradient and we have expressions for both divergence and gradient.

One can define the normal vector for cell A as pointing out from A and the normal vector for cell B as pointing out from B, but this gives some ambiguity at their common face and to avoid having to write expressions for the different vector valued quantities at the faces as seen from both A and B, where the value is identical, with only the sign reversed, we define the normals at a face as along the positive coordinate axis.

This means that the normal vector at the face *ab*, from Figure 64, is [1,0] and not [-1,0], and the normal at *bd* is [0,1] and not [0,-1]. This is important to keep in mind in the following.

We will now write the matrix expressions for some arbitrary quantity ϕ.

Face values Using the notation shown in Figure 66, and the fact that any boundary surface mean value, is the average of the two mean values, in the volumes, separated by the boundary, we can write the surface values as

$$\frac{1}{2}\begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \phi_A \\ \phi_B \\ \phi_C \\ \phi_D \end{bmatrix} = \begin{bmatrix} \phi_{ab} \\ \phi_{bd} \\ \phi_{dc} \\ \phi_{ca} \end{bmatrix} \tag{150}$$

where capital letters ϕ_A denote cell *center* values and ϕ_{ab} is the *face* center values.

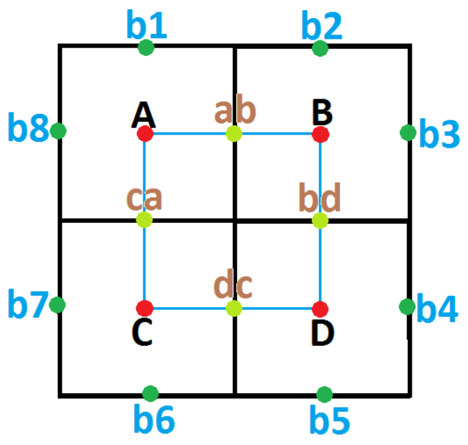


Figure 66: The same mesh of four cells, as in 64, is now labeled at points of interest. The volume centers are labeled with black capital letters, the internal boundaries with brown letters detailing the relevant volumes the boundary separates. The outer boundaries are show with blue letters.

Gradient We now write the matrix version of gradients, at the faces using the value D to represent the cell center distances which in our simple uniform mesh is a constant.

$$\frac{1}{D}\begin{bmatrix} -1 & 1 & 0 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 0 & -1 & 1 \\ 1 & 0 & -1 & 0 \end{bmatrix}\begin{bmatrix} \phi_A \\ \phi_B \\ \phi_C \\ \phi_D \end{bmatrix} = \begin{bmatrix} \nabla\phi_{ab} \\ \nabla\phi_{bd} \\ \nabla\phi_{dc} \\ \nabla\phi_{ca} \end{bmatrix} \tag{151}$$

Divergence The divergence (the *out*flux) of every volume element can be written equally ”compact”.

The flux over a boundary is driven by some flow field. It may be a fluid flow velocity with arbitrary direction, or it could be heat diffusion along the gradient of the heat. In the following we use u as flow velocity.

We split the flow velocity into its two components x,y since the example mesh is two dimensional. Remember that the normal vector of a surface is pointing along the positive coordinate axis. It further means that the y component of the flow u at the face ab is irrelevant in the divergence calculation since ab is always using a normal with zero y component.

The same is true for all other vertical boundaries and for horizontal boundaries the same can be said about the x component. For this reason, the vector values at the boundaries are only represented by either their x or their y component. In the following, A is the area of a boundary between two cells.

In this example it is a constant, but in more complex meshes, it may vary between surfaces.

Terms such as $\phi_{ab}u^x_{ab}$ means the value of ϕ at the face ab multiplied with the flow field u’s x-component at the face ab. Note the transpose of the matrix in the following.

$$A\begin{bmatrix} 1 & -1 & 0 & 0 \\ 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 1 \\ -1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 \\ -1 & 0 & 0 & 0 \end{bmatrix}^T \begin{bmatrix} \phi_{ab}u^x_{ab} \\ \phi_{bd}u^y_{bd} \\ \phi_{dc}u^x_{dc} \\ \phi_{ca}u^y_{ca} \\ \phi_{b1}u^y_{b1} \\ \phi_{b2}u^y_{b2} \\ \phi_{b3}u^x_{b3} \\ \phi_{b4}u^x_{b4} \\ \phi_{b5}u^y_{b5} \\ \phi_{b6}u^y_{b6} \\ \phi_{b7}u^x_{b7} \\ \phi_{b8}u^x_{b8} \end{bmatrix} = \begin{bmatrix} \nabla\cdot\phi_A \\ \nabla\cdot\phi_B \\ \nabla\cdot\phi_C \\ \nabla\cdot\phi_D \end{bmatrix} \tag{152}$$

Given the expression for divergence, constraints, such as the zero divergence from incompressible fluid dynamics section 10, can be written compactly and solved using a multitude of matrix solvers. If we rewrite the divergence as

$$Au = 0 \tag{153}$$

where A is the large stencil matrix and u is the face flux vector.

It quickly becomes obvious, especially looking at the large matrix in the divergence expression, that FVM can compactly[5] be written in matrix form which makes computer implementation of the model somewhat easy.

At the same time it is seen that the matrix form gives rise to some very large and very sparse matrices, if the number of FVM cells is high.

This is however not a serious problem, if one has access to numerical libraries suited to quickly deal with large sparse matrices, such as CUSPARSE [Nvidia, 2011], which we are using.

12 FVM model of wind for snow simulation

Having covered the basics of fluid simulation in section 10 and the Finite Volume Method FVM in section 11, we will now look at how it can be used to computationally model a wind field. This is done by discretization of the Navier Stokes equations, hereafter NS, for incompressible fluid flow, using the finite volume method. This problem is then solved using the fractional step method seen in section 10.

12.1 Mesh

While all the previous FVM explanations will work equally well on rectangular meshes or meshes with different cell shape, they do assume that the cells are regular in the sense that the connecting lines from cell center to cell center passes through the face centers.

We have chosen to use a uniform resolution cubic cell mesh where every cell is a cube 1m in every dimension. This is not an optimal mesh in any sense of the word, but it was somewhat simpler to implement in this first

[5]It is the notation which is compact - not the matrices.

version of the simulator. We considered using tools such as DistMesh for Matlab to create fine meshes which could conform to arbitrary geometry, but due to time constraints, this was dropped.

All simulations are made inside a cubic domain 256m in each dimension, which was also to keep the implementation initially simple. Those choices means that mesh resolution is wasted on higher areas - where there will certainly be no snow drifts - and that the mesh cannot be refined near obstacles, where a higher resolution could be usable. The 1m cell size is not *extremely* coarse though, and in [Mott et al., 2010] for example, a resolution of 0.8m is used.

12.2 Fractional step

For every cell, we have a three dimensional velocity u and a scalar pressure p. We seek to model the development of u over time and apply a pressure term which makes the fluid divergence free. We solve this using the fractional step method as in section 10.

From section 10 we have the Navier Stokes equations, which are repeated here fore convenience.

$$\begin{aligned}\frac{\partial u}{\partial t} &= \overbrace{-u\cdot\nabla u}^{advection} + \overbrace{\frac{1}{\rho}\mu\nabla^2 u}^{diffusion} - \overbrace{\frac{1}{\rho}\nabla p}^{pressure} + \overbrace{\frac{1}{\rho}f}^{forces} \\ \nabla\cdot u &= 0 \end{aligned} \tag{154}$$

When solving the NS numerically, using the fractional step method, we have to evaluate the contribution of each of the terms, advection, diffusion and pressure. We will not be using any external forces. In the following, the symbol f will instead represent a face. We will also not be transporting any thing in the fluid other than the velocity field itself. Had we *not* been using the SPH model for the snow, the snow would have been advected and diffused along with the velocity field.

12.3 Advection

The advection flux of the velocity field over a face is dependent on the face's area f_a, the velocity at the face f_u and the outward facing normal vector f_n of the face. The total advection over all faces f is then

$$advection \approx \sum_f u_e(u_e\cdot f_n)f_a \tag{155}$$

12.4 Diffusion

The diffusion flux of the velocity field over an face is again dependent on the face's area f_a, the gradient of the velocity over the face ∇f_u and the viscosity μ. The total diffusion of velocity over the face is then

$$diffusion \approx \sum_f \mu\nabla f_u f_a \tag{156}$$

Note how the diffusion is defined in term of the Laplacian ∇^2, or the divergence of the gradient $\nabla\cdot\nabla u$ of the diffused quantity, here velocity, but in FVM divergence is found by summing flux over boundaries which reduces ∇^2 to a sum of gradients over the boundaries.

The gradients themselves can be found as described in section 11.7.

12.5 Pressure

A pressure difference over an face will accelerate the velocity inside the two connected cells, parallel to the normal vector. The acceleration will be away from the higher pressure towards lower pressure.

The acceleration is given by the pressure force divided with the mass of the cell, which is the fraction before the pressure gradient.

To find the pressure contribution to the velocity change, we need the *directional derivative* of the pressure over the face.

This can be found in FVM as explained in section 11.6. We then have the total pressure force over the face.

$$pressure \approx \sum_f f_a \nabla f_p \tag{157}$$

with f_a being the area of the face.

12.6 Time integration

Knowing the time derivative of velocity $\frac{\partial u}{\partial u}$ we can advance from $u(t)$ to $u(t+\Delta t)$ using any one of a multitude of numerical integration methods.

We will be using the midpoint method, also known as RK2. It is an explicit method which is second order accurate. It is a very simple method to describe and implement, but it is not terribly stable. The time steps should be chosen to *at least* abide by the CFL condition and generally one ensures that the step size is only a fraction β of this upper limit. The upper bound for the time step can then be written as

$$\begin{aligned} 0 < \beta \leq 1 \\ \Delta t_{max} \leq \beta \frac{\Delta x}{u_{max}} \end{aligned} \tag{158}$$

where Δx is the cell size.

This condition ensures that physical "signal", e.g. material moving $\frac{u_{max}\Delta t}{step}$, will not travel faster than the numerical speed which is $\frac{\Delta x}{step}$. Had we been dealing with a compressible fluid, we would have had to consider the velocity of pressure waves moving at the speed of sound, which would have made the time step constraints even more severe.

For our purpose it is usable though since we will not be relying on a wind field with a high temporal resolution.

Given a *unknown* function $y(t)$, its initial value $y(t=0)$ and its time derivative $y'(t) = f(y(t),t)$, we seek to integrate y' Δt forward in time to find $y(t+\Delta t)$. The midpoint method does this by evaluating the current time derivative, advancing half a step forward in time using this derivative[6],

[6]This is just an explicit Euler integration step

reevaluate the derivative at this midpoint position and then take the full step using this reevaluated derivative. More formally it is

$$\begin{aligned} y_n &= y(t) \\ y_{n+1} &= y(t+\Delta t) \\ y_{n+1} &= y_n + \Delta t f\left(y_n + \frac{\Delta t}{2} f(t, y_n), t + \frac{\Delta t}{2} \right) \end{aligned} \tag{159}$$

where y_n and y_{n+1} are introduced as shorthand for y at current and next time step.

We seek to find the new velocity using the midpoint method. The time-derivative $f(y,t)$ is in our case not time dependent, e.g. it does not depend on time - only on the current state of the system at any one time $y(t)$. It can therefore be written as $y'(t) = f(y)$. The midpoint method is then simply

$$\begin{aligned} f(t) &= \frac{\partial u}{\partial t} \\ u_{n+1} &= u_n + \Delta t f\left(u_n + \frac{\Delta t}{2} f(u_n) \right) \end{aligned} \tag{160}$$

12.7 Solve for divergence free flow

Advancing the velocity field forward in time, as describe, will in all likelihood not result in a divergence free velocity field. To enforce this constraint, we will employ the Helmholtz Hodge decomposition, as described in section 10. This will give us a scalar pressure field and a divergence free velocity field.

If we call the newly calculated, likely divergent, velocity field w and the desired divergence free velocity field u, we have

$$u = w - \frac{\Delta t}{\rho} \nabla p \tag{161}$$

where the divergence of the pressure field is an unknown found from

$$\nabla^2 p = \nabla \cdot w \tag{162}$$

which is a Poisson equation on the form $\nabla^2 \phi = f$. This equation can be solved using various solvers for linear systems. For simplicity, and as a first version, we use the Jacobi method [Heath, 2002].

The problem can be transformed into a set of linear equations using the matrix form, as described in section 11. The Laplacian operator can thus be written as a large matrix called L, and the pressure can be written as a vector p, n elements long, where n is the number of cells in the mesh. Multiplying the Laplacian operator matrix with the pressure vector gives the Laplacian vector of the pressure $Lp = \nabla^2 p$. This is shown in Figure 67.

It should be noted that the *only* non-zero elements in a matrix row, are the ones related to the Laplacian of a cell. Given that every cell is a cube with six faces shared with other cells, only six neighbors contribute to the divergence of the pressure gradient (the Laplacian of the pressure) for each cell. This means that $n-6$ positions in a matrix row is empty and that the Laplacian matrix has $n^2 - 6n$ zero elements. It is in other words a *very* sparse matrix.

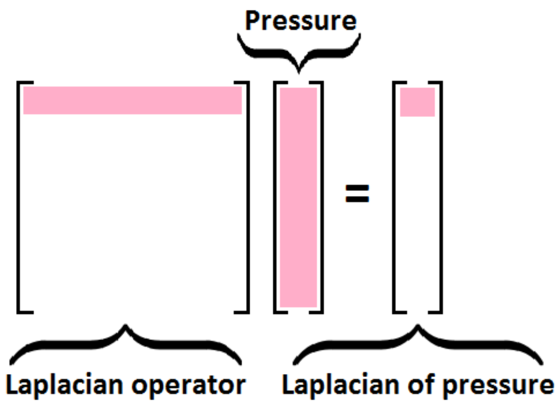


Figure 67: A matrix can be defined which when multiplied with a pressure vector will result in a vector of Laplacians ∇^2. It is seen that each value in the result vector depend only of one dot product of the pressure vector and one matrix row.

This system could now be solved directly as

$$\begin{aligned} Lp &= \nabla u \\ p &= L^{-1}u \end{aligned} \tag{163}$$

but given very large matrices, this will be very slow and it will take up a lot of space since generally the inverse of L will not be a sparse matrix, even though L is.

The Jacobi method solves matrix systems as above. It solves problems on the form $Ax = b$ for the unknown x and the known A and b, through iterative relaxation. The A matrix should be square, which is always the case for a system with n equations for n unknowns, and A should be diagonal dominant, which means that the diagonal values should be larger than all other values in a matrix column or row.

The requirement of diagonally dominance is not a problem for our L matrix, since the diagonal values will be 6 while the non-zero off diagonal values will be 1.

This is a result of how the gradients for a cell are found from difference between cell center values, and how divergence of gradients is sum of face gradients. This is because the cell itself (the one we want the gradients for) will be present in six such gradient calculations and sums, for one divergence of gradients calculation, while the neighbors will be present only once each.

To continue with the Jacobi method, solving $Ax = b$ for x, the matrix A is partitioned into a diagonal D and an off-diagonal R part which are defined as follows, using the Identity matrix I

$$\begin{aligned} A &= D + R \\ D &= AI \\ R &= A - D \end{aligned} \tag{164}$$

The unknown x value is initialized to some guess which may or may not be close to the actual x. The correct solution can be found using fewer iterations if the initial x is close to the correct value. This means than the previous value, in our case the pressure field, can be a good starting point for the next Helmholtz Hodge decomposition and projection.

While the inverse of the large sparse matrix, here A, is not easy to get, the same is not true for the diagonal matrix D where each value, in the inverse, is just the reciprocal of the non-inverted values.

$$D^{-1}_{ij} = \frac{1}{D_{ij}} \tag{165}$$

The next value of x x_{k+1}, given D, R, b and the previous value x_k is found as

$$x_{k+1} = D^{-1}(b - Rx_k) \tag{166}$$

This is repeated until the residual $r = Ax - b$ is below some threshold, or until an upper limit for the iterations is reached.

If the iterations are stopped before the residual becomes zero, then the resulting velocity field will not be perfectly divergence free. It is generally not a large problem since each new field will not be increasingly divergent.

While we could have written GPGPU code to directly handle the above matrix computations, we have opted not to. The problem is not very interesting, since it is only a very large number of multiplications, additions and subtractions. Instead we have used CUSPARSE which is a matrix/vector operations library from Nvidia.

It lets us perform fast computations of very large sparse matrices on the GPU. Details about this library can be found in [Nvidia, 2011]. It should be noted that there exist another GPGPU library, CUSP, with implemented solvers, which are better than our implementation, but we only realized that after having implemented our own solver.

12.8 Update frequency

For a fluid simulator, which should show the dynamic properties of a fluid in meticulous details, the flow should be updated as often as possibly. In our case, we will instead update the flow quite slowly. We need to describe how the air moves around obstacles, but we can very well live with a time averaged wind field which develops slowly. The wind just needs to be update when the geometry (obstacles or snow drifts) change, and this does not happen quickly. Other works [Feldman and O'Brien, 2002] describes this same prioritizing where the wind field is updated at a much lower frequency than the other elements in the domain.

This is essentially a steady state simulation which slowly adjusts to the domain changes.

Initially we let the wind field develop, based on boundary conditions. When the field is fully developed, which the literature [Versteeg and Malalasekra, 2007] says is expected after the fluid has moved ten times the length of the domain, we will start the actual simulation by adding snow. We observed this development time to be shorter in general, but it can be defined as

$$t = \frac{L}{u} \tag{167}$$

where t is the time for the field to develop, L is the distance traveled through the domain and u is the mean fluid velocity.

After development, we slow the wind update frequency to one update every 5s simulation time. This lets the wind react to snow buildup, but otherwise pretend that the field is constant.

12.9 Boundary values

The simulation domain requires various boundary values for the six sides. We need an inflow where the wind enters and we need a solid boundary at the ground which prevents wind from passing and which resists tangential fluid movement. Additionally we need open boundaries at the remaining sides. Here wind may enter or leave as it desires.

12.9.1 Inflow

The common way to model incoming air, which is supposed to have already traveled over the ground for a while, is a logarithmic high dependent velocity [MIT, 2006]. This is sometimes called "law of the wall".

$$u = \frac{u_*}{\kappa} ln \frac{y}{y_0} \tag{168}$$

where u is the mean flow velocity at height y above the ground and y_0 is the roughness length, which is the height below which the flow velocity becomes zero. This length over a snow surface is commonly set to 0.01m [Mott et al., 2010]. The constant κ is the Von Karman constant, which is set to 0.41.

Given this expression, we can define the face normal velocity component for the faces on the inflow boundary.

12.9.2 Open boundaries

The domain is limited in size to $256m$ in either direction, but while the domain is not endless, it has to be modeled that there can be inflow and outflow through the boundaries, since we are modeling an open block of atmosphere and not a closed box filled with air. In fluid dynamics an open boundary can be modeled using a Neumann boundary condition

$$\frac{\partial p}{\partial f_n} = 0 \tag{169}$$

where we state that the pressure gradient over the faces on the open boundary are zero. This means that if fluid wants to flow out through the boundary, there will be no pressure buildup which opposes this. Also, if fluid wants to move away from the boundary, into the simulated volume, no negative pressure at the boundary will resist this.

There is also no viscous shearing stress experienced by fluid moving tangential to the boundary. This is a slip boundary where it is implicitly assumed that fluid in the boundary faces moves exactly as fluid inside the boundary does. We do not use interpolation in this case, but assign the boundary face fluid velocity b_u the value of the fluid velocity C_u of the cell C connected to the boundary b face.

$$b_u = C_u \tag{170}$$

12.9.3 No-slip boundary

A no slip boundary satisfies the "law of the wall" (168) where the velocity near the obstacle approaches zero. Again, we dispense with the interpolation method for finding face velocities, and simply assign the face a velocity value. For the no-slip boundary at the ground this velocity is zero. The no-slip boundary used in later test of driven cavity flow in section 12.10.1 is assigned a non-zero velocity to emulate the sliding boundary driving the flow.

$$b_u = velocity \tag{171}$$

12.9.4 Porous boundaries

The final boundary, which is in fact an internal boundary between cells, is a porous boundary. It is used to emulate a snow fence which essentially is a wall with, commonly horizontal, slits allowing some air to pass through.

According to [Tabler, 1991], the optimal porosity is 40% to 50%. We feel that a reasonable way of modeling this is with an ordinary open face between cells, but with an "area" for flux computations which is only half its true value. This means that the flux of any quantity over the face will be half what it would otherwise be, which is the same result as if we had partitioned a boundary into a number of blocking faces intermixed with a number of open faces.

There is one difference between a true porous boundary and an open face with half the area, and that is the turbulence on the lee-way, and to a certain extent windward, side of the boundary. A solid face, with a number of large holes in it, will have a very turbulent wind field due to the wind shear arising from some air moving through the holes and sliding against other air which hides in the wind shade behind solid parts of the wall. According to [Tabler, 1991] this turbulence is a large contributing factor to the effectiveness of snow fences, and it is one of the reasons the fences are not solid.

Given that we do not model turbulence explicitly, and given that we will certainly not be using a mesh fine enough to directly model the holes in the fence, this is an effect we cannot hope to catch in the current implementation.

Therefore we use

$$\begin{aligned} 0 \leq \omega \leq 1 \\ f_A = \omega f_A \end{aligned} \tag{172}$$

for face f with area f_A and porosity factor ω, where a porosity of 0 means it is solid and a factor of 1 means it is just an open boundary.

12.10 Analysis of FVM wind

We will now evaluate the behavior of the fluid simulator used for the wind simulation. This will in part be done by comparison with reference results from the literature, and in part by comparing with results from our own simulations using the multiphysics simulator Comsol `www.comsol.dk`.

Our implementation is not able to render the fluid flow on its own, so the velocity field was written to a file and then imported into Matlab. Here it was converted into usable form for a stream line rendering, which is what is shown in the figures in this section.

12.10.1 Shear-Driven Cavity flow

One of the standard benchmarks in CFD is the shear-driven cavity flow problem. The problem considers incompressible flow in a square domain with a sliding upper lid. As a point of reference, we use [Erturk et al., 2004] which investigates the shear-driven, or lid driven, cavity flow in detail.

We will be using a Reynolds number of 1000, which is represented in [Erturk et al., 2004] and shown in Figure 68. A test scene scene was created for testing the driven cavity flow. The fluid domain was defined to be a 2D[7] box 10 meters long in each dimension. The top boundary was a no slip boundary moving to the right, while the three other boundaries were stationary no slip boundaries.

Kinematic viscosity υ for air at $0°C$ is $1.3310^{-5} m^2/s$ and the characteristic length is, in this case, approximately 10m. This gives a lid velocity of 0.0013 m/s.

$$Re = \frac{vL}{\upsilon} \tag{173}$$

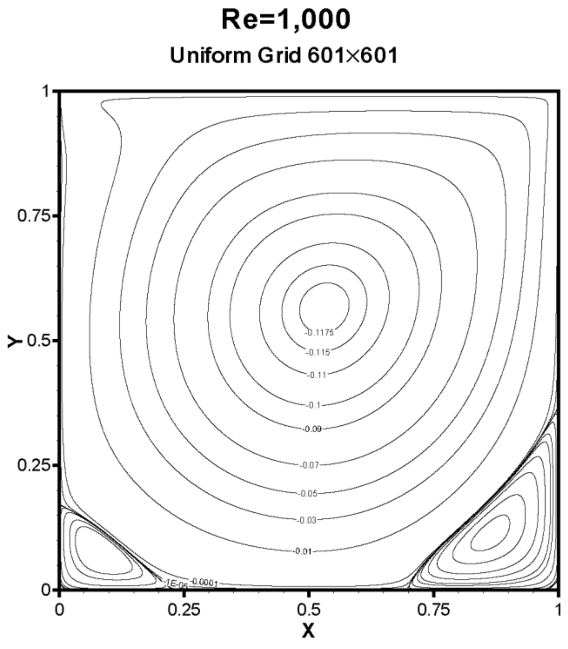


Figure 68: A 2D Shear-Driven Cavity from from [Erturk et al., 2004]. The "lid" is moving to the right of the plot. Note the large vortex which is almost centered and the two smaller vortices in the lower corners, and that the rightmost lower vortex is the larger of the two.

[7]Actually 3D, but only one cell deep and with slip boundaries on the front and back walls

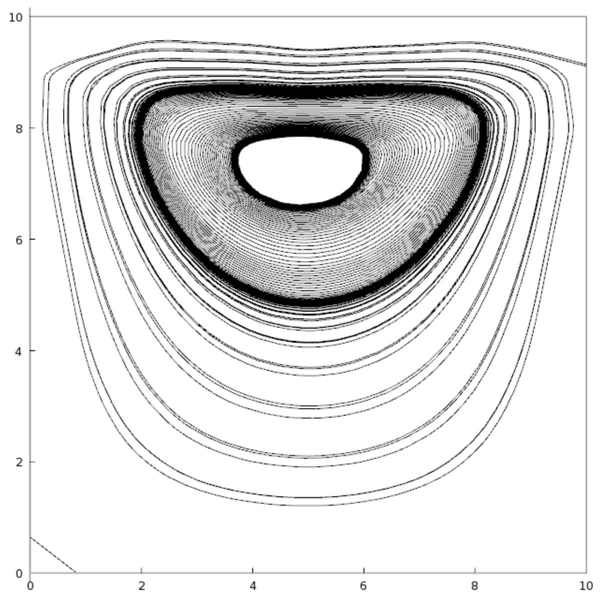


Figure 69: A driven cavity flow in a 10m by 10m 2D box using a grid resolution of 1m. This flow shows only one vortex near the top.

We test the scene with two different mesh resolutions. One on a 10x10 grid as seen in Figure 69, which makes each cell one cubic meter, and the other at 40x40 as seen in Figure 70, which makes each cell 1/64 of a cubic meter.

In Figure 69 we see that the flow is too smooth and that only the principal vortex is seen, and it is much too close to the top. This is what could have been expected for lower Reynolds numbers if the fluid had been much thicker. Obviously the grid resolution is much too small to capture the true behavior.

In Figure 70 we have a flow which resembles the reference in Figure 68 quite well as can be seen in the overlay in Figure 71. The two flows are quite similar, though not iden-

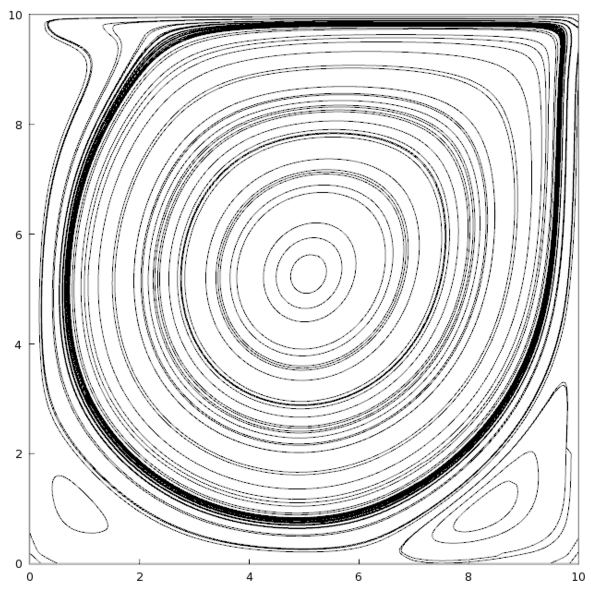


Figure 70: Same driven cavity as before but this time with a 0.25m grid resolution. This time the principal vortex is almost at the center, and there are two smaller vortices in the bottom corners, with the rightmost vortex being larger.

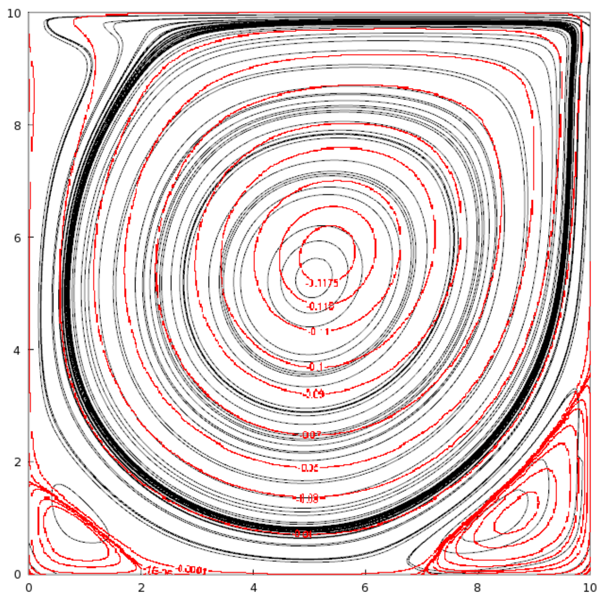


Figure 71: The reference flow [Erturk et al., 2004] is shown in red over our result, using the fine grid, in black. The two flows are quite similar, though not identical.

tical. The reference grid size was 1/600 the size of the cavity in both dimensions, while our grid was 1/40 in both dimensions. Due to implementation specific problems, we did not test with a finer grid, but considering the difference in resolution, the results are quite promising.

It should be noted that the size of the cavity was chosen to match the size of obstacles in the scene. In a full scene, we could not expect to use a finer grid resolution than 1m in each dimension, since it would be too computationally costly. Given that constraint, there would be little point in improving the test resolution far beyond that. This means that the result from Figure 69 is to be considered the result we would see in an *actual* simulation for cavities, spaces between obstacles, of average size. The finer result from Figure 70, on the other hand, shows us that the fluid simulation is working properly and that its inaccuracies are just a result of a to coarse grid.

12.10.2 Wind field around cube

A 2D scene was created which was 20m high and 80m wide. 10m from the left side was an obstacle 5m in each dimension. The wind came from the left at a velocity of 10m/s. This is a turbulent flow with a high Reynolds number as explained in section 10.2.

The reference used for comparison was this time a simulation using the multiphysics program Comsol `www.comsol.dk`. The data from that simulation was exported and rendered by Matlab, as was the data from our own simulation. This was done to make easier comparisons.

If we look at the comparison in Figure 72 we see that again our fluid flow is much too smooth compared to the reference. We have in the previous test with driven cavity shown that the fluid solver can be made accurate with a higher grid resolution and do not repeat this test here. The issues with out implementation, as before, prevent us from increasing the resolution from the default 1m in each dimension, since this test scene is somewhat larger than the one used in the cavity flow.

Though the flow lacks the turbulent look, it still gives us a small area in front of the obstacle with very low flow velocity, and

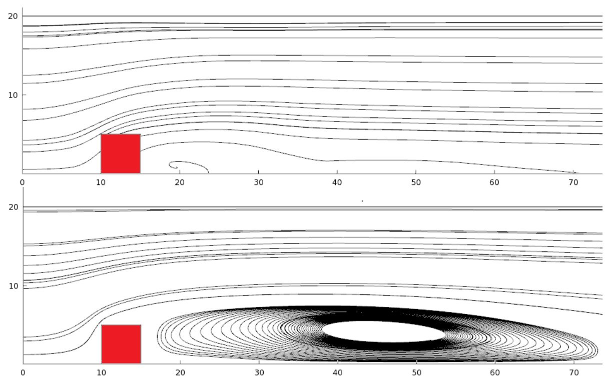


Figure 72: Streamlines around building seen from the side. The top plot is the result from our simulation using a grid 1m in each dimension. The bottom plot is from Comsol using a grid 0.5m meter in each dimension and also using a $k\varepsilon$ turbulence model with default settings.

a larger area behind the obstacle with an equally low flow velocity. This means that while the flow does not circle around on the leeway side and return, the flow does slow down enough for snow deposition to occur.

12.10.3 Wind field around snow fence

In Figure 73 we see the streamlines near a snow fence, which are remarkable more willing to move towards the ground behind the obstacle, than what we saw for the solid building. This is most likely a direct result of the flux across the snow fence faces not being zero. The air velocity directly behind the fence is much lower than above the fence, but it is not quite as low as behind a solid cube as seen in Figure 72. The turbulence behind the fence is not seen, but that was also not expected, as explained in section 12.9.4.

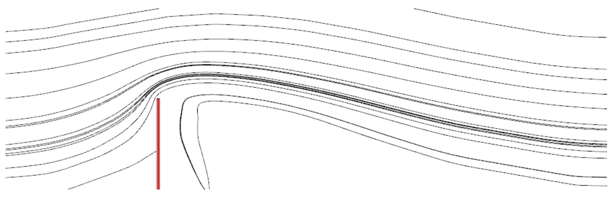


Figure 73: A 3 meter high 50% blocking snow fence was simulated and the velocity field vectors of the wind, was exported and rendered with Matlab.

Part III

Coupled model

13 Coupled snow and wind

While the snow particles are obviously strongly affected by the wind while in suspension and saltation, they are more resistant to the wind once they have become stationary. Stationary snow, such as a snow drift, strongly influences the flow of air since it obviously block and divert air. According to [Mott et al., 2010] this effect is quite significant.

We have previously, in section 4, described how the wind is experiencing drag from saltating particles near the surface. This is an element which we have not included in the final implementation, due to time constraints.

13.1 Wind speed pseudo particles

In our finite volume model, a cell centered one, velocity vectors are defined at the midpoint of the cells. If extra artificial SPH particles are placed at the cell center, they can directly be given a pseudo velocity equal to the velocity vector of the face. The velocity of the particle is termed ”pseudo” because the particles will never actually move, and they will never interact with other pseudo particles, and only interact with real particles through their pseudo velocity. They will remain fixed at the same position in the mesh throughout the simulation. The real benefit is that the velocity can be taken seamlessly from a FVM description to a SPH description and the SPH model can work with the FVM velocities as if they were originally SPH velocities.

We have chosen a smoothing length h of the virtual particles which is slightly larger than the diagonal radius of the cells.

$$h = \frac{cellSize}{\sin 45^\circ} \tag{174}$$

This ensures that a particle moving from one cell to the next, through the corner, will not jump from the influence of one wind particle to the next. When the smoothing length is at least slightly larger than the diagonal radius of the cells, the particle will not leave the influence of the first particle before entering the influence of the second. This smoothing is essentially the reason that we do not directly use the FVM cell center flow velocities, seeing that this would give a discontinuous wind drag effect, and if smoothing was required anyway, SPH smoothing seems reasonable, and easy to implement. We have, however, not tested if this non smoothed drag would truly cause problems, or if the problem is hypothetical.

In this wind influence, the smoothing length of the real particles is of no consequence. Only that of the wind particles is relevant. As explained in section 9.5, every SPH particle will look for interactions in its own cell and the 26 surrounding cells. This means that it will already be looking for particles in the cells containing the wind particles, which means that if it only recognizes wind

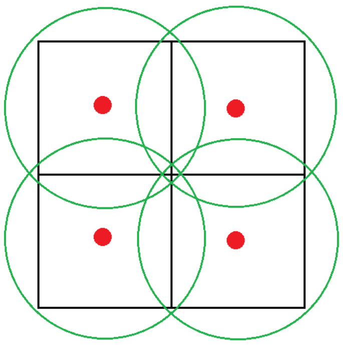


Figure 74: Every volume in the FVM mesh has a special wind speed SPH particle which ”radiates” a wind velocity field outwards to neighboring SPH particles. Note how, in 2D, every position on the inside of the ”grid” will be in the neighborhood or at least two particles. This will have the effect of smoothly interpolating the wind velocities throughout the domain.

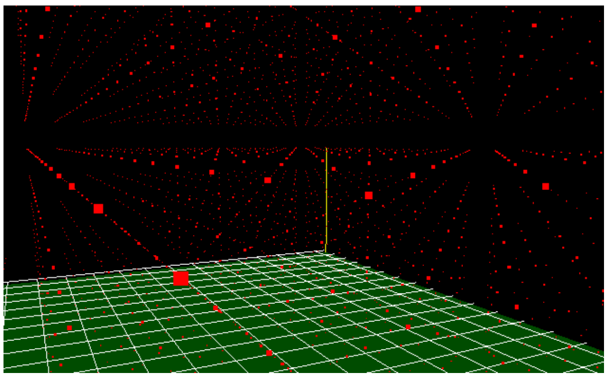


Figure 75: Empty scene with the artificial wind particles plotted as red. Only every 10th particle in each direction is rendered.

particles for what they are, it can ignore its own smoothing length and find the wind contribution from the pseudo particles.

13.1.1 Snow-controlled wind velocity

When a snow drift exist, it affects the wind field by preventing the wind from blowing through the drift. The surface of the drift forms a new boundary with a zero velocity on the inside of the drift and a no-slip boundary on the outside. This could be handled by dynamically altering the mesh, so it forms a boundary against the snow drift.

Rather than altering the mesh, whenever snow moves in the drift, we will use the same pseudo particles as before. This time the snow will affect the velocity stored in the particles. This influence will depend on the local snow density around the pseudo particles.

$$u_{pseudo} = \begin{cases} \frac{1+cos(\frac{\rho_{pseudo}\pi}{100kg/m^3})}{2} u_{FVM} & , \quad \rho < 100kg/m^3 \\ 0 & , \quad \text{otherwise} \end{cases} \tag{175}$$

The cosine function is used to ensure a smooth function. Here u_{pseudo} is the pseudo particles velocity and u_{FVM} is the FVM cell velocity, which will be scaled towards zero as density approaches $100kg/m^3$.

Accumulated snow will not let the wind pass through it. When the initial terrain is a rough one, then the accumulation of snow will smooth the terrain so it in previously rough areas will become quite smooth. In other areas it may have been primarily smooth before and now it can have a large snow drift

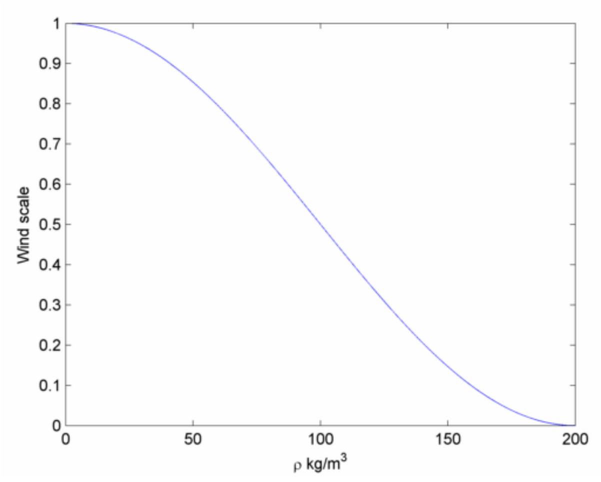


Figure 76: The scaling function relating FVM velocity to pseudo SPH particle velocity depending on local snow density

deflecting the wind. It is therefore quite important to consider how accumulated snow and the wind field interacts.

We do not consider the situation when a snow drift actually move as a whole, which theoretically could happen if a drift suddenly slides down a slope or collapses for some reason. In that case it would be incorrect to model the snow-air boundary as having zero velocity just because the snow density is high there.

13.2 Analysis of coupled model

In this section we will investigate how the coupled snow and wind model behaves under various test conditions.

13.2.1 Dune shape

The behavior of snow drifts and sand dunes are governed by the same forces [Andreotti et al., 2002, MIT, 2006, Zhang and Huang, 2008]. The detailed difference lines in the friction angle of the material being blown by the wind and in the susceptibility of the particles to wind shearing forces, which in turn depends on the particle mass and shape.

We can therefore not directly compare a sand dune under certain wind speeds to a snow drift under the same wind speeds, and expect a perfect match. We can however look at the general shape of the dunes and drifts, and evaluate the snow drifts based on that comparison.

From [Andreotti et al., 2002] we have a detailed description of dune formation and some illustrations useful for comparison.

Dunes and drifts have a gentle slope towards the wind, a sharp ridge and a steep side away from the wind. The cause of this is the fact that particles will easily blow up over the slope on the wind side and then drop into the leeway side where it will not be disturbed by the wind. Not only does the wind not blow the particles away from the drift, it actually swirls around and returns blowing the particles closer into the drift or dune.

This is illustrated in Figure 77 and a photograph of this shape is seen in Figure 78. For snow, this can in some cases result in a ridge which is not only sharp, it actually forms an overhang like a breaking wave. This is possible when wind speed is constant over a longer period, the snow is very light and the

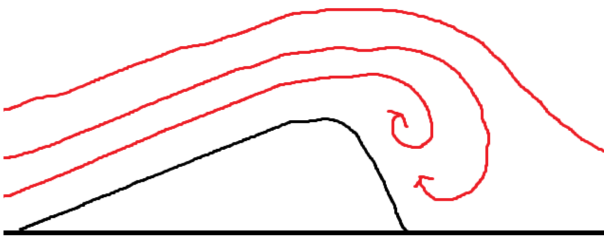


Figure 77: Wind blows particles up the leading edge of the drift and then swirls around over the top. This results in a steeper leeway side with particles in effect being blown into the drift against the general wind direction.

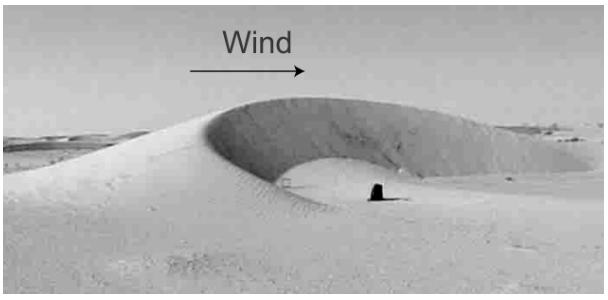


Figure 78: A sand dune shows a gentle slope towards the wind, a sharp ridge and a steep side away from the wind. Figure from [Andreotti et al., 2002]

temperature is not so low that the snow can't bind together and form tight bonds.

This is not something we have been able to see in our simulations. Probably mainly because we model even the rigid phase of snow as a low viscosity fluid as described in section 7, and because we do not have the swirling flow of the wind.

Ridged shape of snow drift When we let wind in the coupled model affect a symmetric pile of snow, the expected effect is seen.

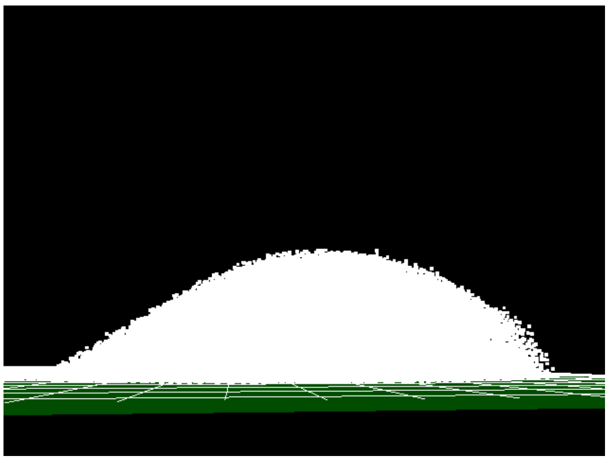


Figure 79: Snow drift with wind from the left side. The wind speed is low to medium and the leeward side is quite shallow and the top of the drift is very smooth.

In Figure 79 the wind comes from the left and it somewhat low with a velocity of 5 m/s. In Figure 80 the wind is from the left and this time the wind speed is 15 m/s. It is seen that the sharpness of the drift depends on wind speed which is to be expected.

The faster the wind, the more saltation, and the farther the particles move before coming to rest again. It is however unclear what causes the somewhat linear segments on the leeway side of the drift in Figure 80. Sometimes this is observed and sometimes it is not. It does not seem to be dependent on grid size or drift location.

Horned shape of snow drift Apart from the ridge of the snow and the difference in slope of the drifts sides, we expect it to show a curved shape with the curve pointing towards the wind. This is seen in Figure 81 as well as in Figure 78. The shape is caused

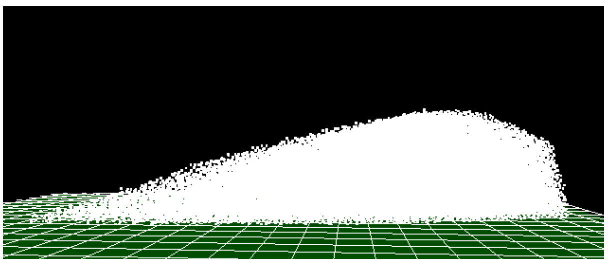


Figure 80: Snow drift with wind from the left side. The wind speed is medium to high and the leeward side is quite steep

by the higher wind speeds along the sides of the drift. Here the wind picks up particles and carry them behind and back into the drift, just as seen in Figure 77. Only, this time the motion is in the horizontal plane.

We saw this shape in out simulations. A symmetric pile of snow would blow into a ridged shape, showing a horizontal curve with the horns of the curve away from the wind. There was one slight problem though.

As the wind kept blowing, the shape tended to become more and more elongated. In Figure 82 we see a curved shape after 100 minutes of simulation with a wind speed of 10 m/s, while Figure 83 shows the same after 200 minutes. The shapes generally tended to be quite stretched. It appears to be the result of a too coarse wind field which does not adequately capture the turbulent swivel of snow at the leeway side of the drift. This theory is supported by tests with larger and smaller grid resolutions, where the drifts tended to be less stretched when the resolution was increased. Another supporting observation is that the inner side of the curved drift was *very* smooth compared to the inside in Figure 81. At the reference image from [Andreotti et al., 2002] the same smoothness is not seen. Instead it has a more noisy turbulent look.

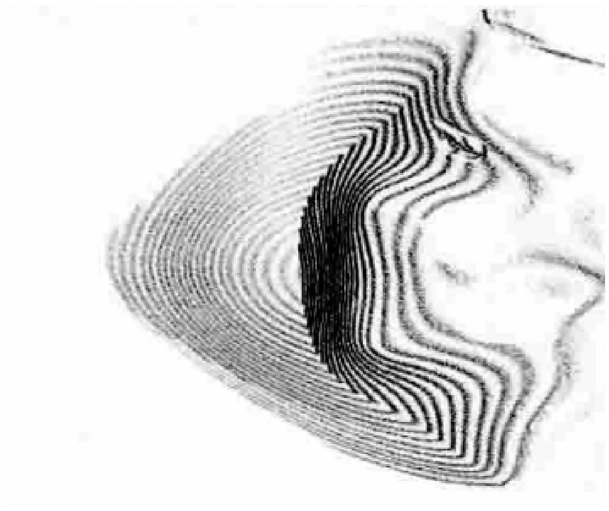


Figure 81: A dune shows a curved shape away from the wind as the sides of the dune is affected more by wind shearing than the inner portions are. [Andreotti et al., 2002]

Snow drift near building In Figure 84 we see a snow drift building near a building. The drift shows the relevant characteristics such as small drift on the windward side and a larger drift on the leeway side. Looking at the streamlines, it is not surprising that snow is deposited where it is, but we note that the slope on the leeway side is not entirely as expected. The drift does not show the expected shallow slope. Instead the snow forms a large clump of snow behind the building, and as time goes on, more snow than expected [Tabler, 1991] will form on the leeway side. This is due the the problem of the missing loop back of the streamlines as evident in Figure 72. This lack of a definite end to the wind means that the low velocity field behind the building is very much larger

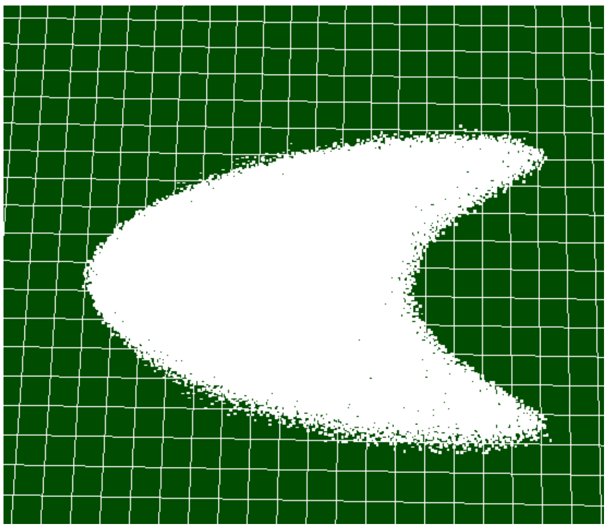


Figure 82: A snow drift curving away from the wind.

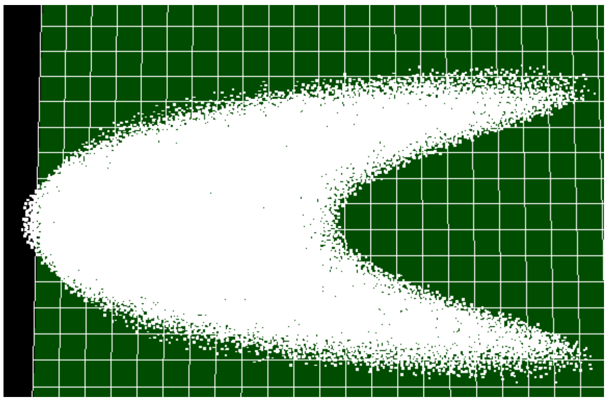


Figure 83: A snow drift curving away from the wind.

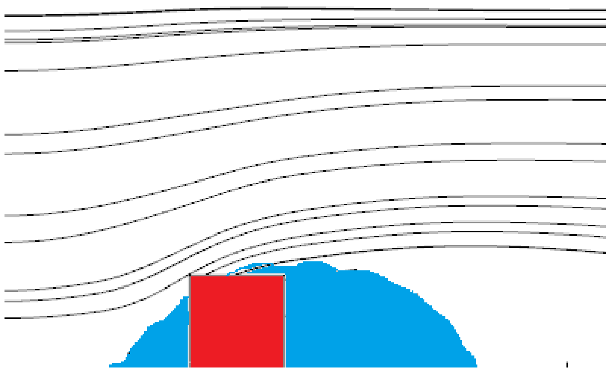


Figure 84: A scene was simulated with a single 5m high building. A 2D slice of the velocity field was exported and rendered in Matlab. The snow density was also exported and rendered as an overlay in Matlab for densities above 300 kg/m^3. The building and the snow was then hand colored to show their individual locations. The ”lumpy” look of the snow was due to somewhat large particles. Note how some streamlines pass slightly through the snow, due to the smoothing nature of the wind particles.

than it should be and snow will willingly deposit in this protective bubble of low velocity wind.

Movement of snow drift Finally we considered the motion of the drift, as a whole, with constant wind. We expected the drift to travel downwind and we expected the snow particles to leap frog over each other and take turns being at the wind side of the drift, blowing over the top and sides and being on the leeward side.

This was seen to some extent, but the problem with an elongated drift, as previously described, meant that while the drift did move, it also stretched out a little too much. The result was that the drift stretched out approximately 1 meter for every meter its center of mass moved. According to [Andreotti et al., 2002] this is not the expected behavior. When the curved, ridged drift is formed by a certain wind speed, it should not continue to change shape, but only move with the wind.

We attempted this snow drift movement at different mesh resolutions and different wind speeds. As expected, the higher wind speeds resulted in more elongated drifts, and the finer mesh reduced the artificial stretching of the drift. We were, however, unable to refine the mesh below cells of $0.5^3 m$ since the application had no support for different mesh resolutions in the same scene. This means that while we could observe a difference in error between mesh resolutions, we could not eliminate it, or even reduce it to the point where it was insignificant.

Conclusion The conclusion of the above is that we did model realistic shape of a snow drift, and that this was done by considering how the wind affects the snow and how the snow affects the wind.

At the same time, we seem to need a higher resolution of the FVM mesh - or we need to model turbulence.

It is not exactly clear why this is not a problem for the vertical plane turbulence, but it seems reasonable that it has to do with the force of gravity which pulls particles down behind the drift - turbulence or no turbulence, while the same is not true for motion in the horizontal plane.

Other wind driven snow simulations such as [Feldman and O'Brien, 2002] do not include turbulence modeling, and still do not report on the artifacts we see. It is our assumption that this is due to shorter time spans being observed, since their work is only concerned with modeling drifting long enough for them to form and generate interesting graphics. It is only a guess though.

In support of our assumption, that turbulence modeling is the missing element, we refer to [Moeslund et al., 2005], as well as our own test in section 12.10, where parameter studies clearly show that larger cells in the wind simulation results in a smoother wind field.

13.2.2 Effect of snow fences

From [Tabler, 1991] we have some reference material detailing the shape and size of a snow drift around a snow fence as seen in Figure 85.

We simulate a 50% blocking snow fence and observe the deposited snow in Figure 86. The immediate observation is that the difference in the flow, where the flux over the snow fence boundary is not zero, helps the results. The general shape is quite similar to the reference.

One thing, that we are still missing, is the expected dip in snow height just behind the fence. It is unclear what the cause of this is in the real world, but we speculate that it could have to do with the turbulent zone just behind the fence. A higher than normal shear velocity, and resulting erosion, could be expected here. Given that we do not

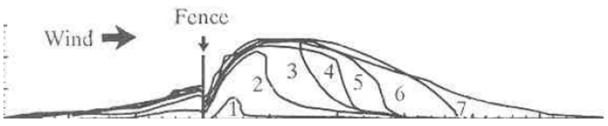


Figure 85: A reference model of a 40% or 50% blocking snow fence and the resulting snow drift [Tabler, 1991]

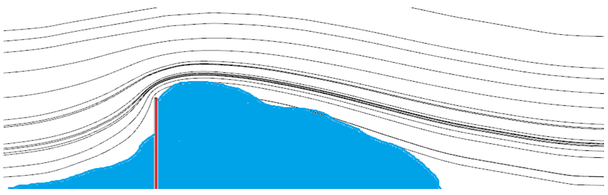


Figure 86: A 50% blocking snow fence was simulated and the density of the deposited snow, as well as the velocity field vectors of the wind, was exported and rendered with Matlab. Then it was hand colored to show snow and fence.

model this turbulence, we feel that this is a likely explanation.

Part IV

Software implementation

14 Implementation

Initially we considered making a sequential implementation on CPU and a parallel version on GPU. At first, work was done developing both. It quickly turned out that an optimal CPU implementation would be quite different from an optimal GPU implementation, and that it was quite time consuming to both.

Additionally, if the motive for making this dual implementation was to later make performance comparisons, we would end up comparing different algorithms with different optimizations on different platforms, which could still show a difference, but not in a fair way, considering that we from the start had a stronger emphasis on implementing for GPU.

Another reason for implementing a CPU version, could be easier testing of the understanding, and correctness, of the physical systems and discretization models, before porting it the GPU, where development is generally somewhat harder. It quickly turned out that the CUDA implementation came quite easy. The largest problems tended to be with the general application framework, which was designed to let us switch between the serial and the parallel simulators at runtime. Therefore we dropped the serial version entirely and focused on the CUDA implementation.

The test application was implemented using a variation of C#, C++ and CUDA C. A graphical user interface was implemented in C#. This GUI called two different .net classlibraries. One implemented a managed C++ class which could render the scene through OpenGL, and another implemented a managed C++ class which in turn called unmanaged CUDA C code to deal with the actual calculations. This somewhat roundabout design was used in order to let the GUI development be as easy as possible using C# while still separating the general SPH and FVM models into a reusable classlibrary. Initially the implementation proved a little troublesome, but after the first boiler plate development, the differentiated design proved to be quite easy to extend as the project went on.

14.1 Scene setup

There were plans for a simple and intuitive way of defining the scene through configuration files, but in the end this was dropped. The consequence of this was that whenever a new simulation should be performed, the setup should be hard-coded in the application, which was quite time consuming and prone to error. It also forced us to make somewhat simple scene designs with few obstacles, though the simulator should not have had any trouble with a vastly more complex scene with several arbitrarily spaced obstacles.

14.2 Visualization

The visualization was in no way the priority, and this means that SPH snow particles were rendered as simple square point sprites.

The ground plane was rendered as a green surface which was divided into smaller squares, for easier analysis, with white lines. The four corners of the rectangular domain were rendered as yellow lines.

In order to render the particles, the actual particle data was being copied from the main graphics card (Nvidia GeForce GTX580), where the CUDA computation took place, to a much slower Nvidia GeForce 8400GS. This copying took the path around system memory and not the more direct device-device path, as has recently been possibly with CUDA 4.0. This data copying was a bottleneck, but only for the rendering itself, since the rendering could be interleaved with the calculations without adversely affecting simulation performance.

To show the wind behavior, which was not rendered at all in the test application, we exported data from the FVM cells to a file, which was then read into Matlab for post processing and rendering.

14.3 CUDA results and parallelism

We refer the reader to section A for a brief CUDA optimization introduction if the subject is foreign.

Most of the tasks in this simulator are highly parallel. We partition the work into groups and consider how parallel each is. Following this, we analyze the parallelism and data access patterns of a specific portion of the SPH code with a focus on neighbor retrieval.

Primary tasks

- Cell grid update - every particle will in parallel write itself to its cell and then a parallel radix sort sorted the particles. The parallelism is 1 for the cell writes, disregarding potential atomic conflicts, and very close to 1 for the radix sort.
- Particle integration over time - every particle locates its neighbors, handles boundaries, and calculates forces before integrating using RKF. The parallelism is 1.
- Update wind field - CUSPARSE was used for the matrix solutions and it is unclear exactly what level of parallelism was reached. We estimate it to be at least 0.9.
- Visualization - the particles were read back from the GPU and then rendered by copying them to another graphics card. This has a parallelism of 0.

considering that we were rendering one frame 60 times each second and that we took 66 physics steps each second[8], and that rendering was interleaved with calculation, the conclusion is that this application has a parallelism very close to 1. This in turn indicates that it is of a type which greatly benefits from a massively parallel implementation as explained in section A.2.1.

[8]In the case with one million particles

CUDA profiling results The results are obtained using NVidia Visual Profiler and NSight. The updateDensityByDirectSummationKernel kernel is the main focus of profiling in this section since it the first place the neighbor location takes place.

In its current form, the kernel uses 37 registers per thread and it is launched using a block size of 256. The grid size is then the number of particles divided by 256.

In the tables, time is the kernel time it takes to locate all particles neighbors and calculate density using a SPH kernel. Divergent branches is per particle in order to compare over the different particle numbers. IPC is instructions executed per clock and serialization is the percentage of memory access which had to be serialized.

The conclusion, when observing the two tables, is that is does make a huge difference if the particles are sorted - and more so for large particle counts. The cache hit ratio was close to double up when the particles were sorted, and if the same sorting method had been used on the cells themselves, and not just on the particles, then the effect would likely have been even greater. Not only did the spatial coherence of particles in a block ensure that they accessed the same memory, it also helped ensure that the particles went along the same path through the code and had the same number of neighbors in the same cells. This is seen by the fewer divergent branches in the SFC version of the code. The resulting number of instructions per clock is therefore also higher for SFC and the serialization is lower.

If we consider the last table showing the row by row sorted particles for one particle count, it is clear that not only should particles be sorted - they should be sorted in a way that preserves spatial locality. While row by row sorting does yield significantly better results than no sorting, it is not as effective as SFC.

Occupancy The occupancy is a sensitive subject right now. Each thread uses 37 registers as the code is written now. This means that only 32768 registers / 37 registers per thread = 885 threads can be running concurrently. Using a block size of 256 threads, this is 3 blocks out of the maximum of 8. Currently 885 threads out of 1536 are running which gives an occupancy of 57%. In order to hide the memory latency, it would have been better with more threads. Especially since the code is not very instruction heavy and therefore cannot hide it by computation.

If we for the moment assume that the code is of such a nature that 37 registers is fair, then there is not much we can do about it as is clearly seen in the image from NSight. If the register count could be taken down to 18, the occupancy would be fine, and all 48 warps could be running concurrently.

15 Conclusion

We have made a physically sound derivation of the relevant properties of wind blown snow. This work was, however, based on a very fragmented knowledge base, and the resulting properties need adjustment and validation using more detailed measurements, if they exist at all. In the very first piece

Not sorted					
Particles	Time	Divergent branches	L1 hit	IPC	Serialization
512k	22ms	0.13	40%	0.11	0.71
1024k	67ms	0.19	41%	0.09	0.80
2048k	225ms	0.29	46%	0.07	0.81

Table 4: Particles stays at their initial memory location and are not moved by sorting.

SFC sorted					
Particles	Time	Divergent branches	L1 hit	IPC	Serialization
512k	6ms	0.13	73%	0.53	0.50
1024k	15ms	0.17	74%	0.48	0.51
2048k	39ms	0.23	73%	0.49	0.57

Table 5: Particles are sorted according to the SFC index of the cell they are inside.

Row by row sorted					
Particles	Time	Divergent branches	L1 hit	IPC	Serialization
2048k	53ms	0.25	67%	0.32	0.63

Table 6: Particles are sorted according to the index of the cell they are in, but that cell index is calculated first by x, then y and finally z. A 3D scanline path.

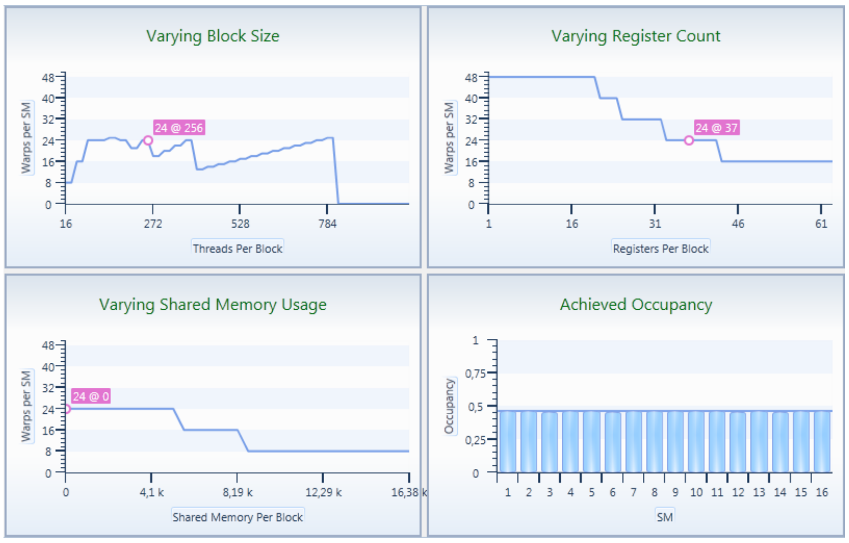


Figure 87: Occupancy information from NSight

of snow literature [Meller, 1975], the author made it *very* clear that snow was an extremely difficult material, which was highly non linear in every sense of the word. He pointed out that it depends on a multitude of properties and assumptions... he was not kidding.

Through the introduction of material strength mechanics, rheology and fluid dynamics, as well as a comprehensive description of two relevant discretization methods, we have simulated the behavior of wind blown snow. A simple coupling was designed to connect the two different discretization models, and we observed that both the isolated snow, the isolated wind and the coupled model, behaved reasonable, though not entirely correctly. When the model diverged from real world observations, the reasons for this were discussed and possible solutions were pointed out.

It seems that our solution may be computationally more expensive than other methods mentioned in section 3, where both snow and air is simulated in a Eulerian model, but the particle based method, that we have presented, has the benefit that it is able to take the simulation from initial snow fall, through drifting to later sliding and melting snow. It is, in our view, a more complete and general model.

We do note, however, that other, much simpler, methods such as [Feldman and O'Brien, 2002] can make realistic snow deposition calculations on a small scale. The basis of that work is mostly hand tuning and tweaking and not the physical description of snow, but the real question is if that is truly a problem.

While snow is a very complex substance, and it can hardly be said enough, the process of blowing snow in the wind and letting it deposit in areas where the shearing force of the wind is low, is not really that dependent on the entire material description of snow.

If, on the other hand, we desire a physically accurate model which can take snow from initial crystallization in the atmosphere, through wind transportation and deposition, and then continue to affect the wind field, and later just sit in the scene showing creeping collapse and hardening, or perhaps melting and flowing as water, then a more general model such as ours should be used. Snow may build up on slopes, and eventually slide down in the form of avalanches. We have not made in-depth investigations of the later behaviors, but there is nothing to prevent the model from being extended to support all such transitions.

The conclusion seems to be that for specific subsets of snow behavior, there exist reasonable methods which handle those quite well, but when more is required, a more complex model is needed.

We believe we have, in this work, shown how such a method could be developed, and we have taken the first small steps towards a more unified model, than what is commonly seen.

15.1 Future work

There are a number of factors which should be taken into account for further development.

15.1.1 Spatial subdivision

While the spatial subdivision by a regular grid, for neighbor retrieval, is simple to implement, it has some serious problems with variable density snow. When snow is building up and being compressed, the number of neighbors in the 27 relevant cells becomes too high.

While the simulation does not become unstable, it affects the simulation speed adversely. This is in all likelihood a problem that could be addressed by a more complex spatial subdivision such as some recursive tree structure. Alternatively the problem could be mitigated by not allowing the snow to compress. While the incompressible snow is not physically realistic, snow will generally not, quickly, pack too tightly under only the influence of the wind and gravity, unless the snow pack is very deep, in which case the deeper layers can be compressed.

15.1.2 Fine tuning and validation of material properties

Before our snow model can be trusted, the descriptive properties must be validated against a much larger, and more consistent, set of real world measurements.

15.1.3 Individual time steps and particle size

From our testing we know that not every particle is alike. Given their vastly different time-step requirements, it seems clear that the simulation would benefit from individual time-steps, as it is generally used in astronomy. It also seems evident that given an upper bound on the number of particles, variable particle sizes would allow the simulation to use the available particles in a more optimal way. The larger part of the snow mass is located deep inside the drifts and at that location there is no real need for the high resolution in neither space nor time.

15.1.4 Turbulence modeling

We noted earlier on that the snow behavior was not entirely realistic, and we argued that the cause was the lack of either direct or indirect turbulence modeling. This could be implemented either by refining the FVM mesh around the drifts and obstacles to a resolution, which would capture ”enough” of the turbulent motion, or it could be done through turbulence models such as LES. In all likelihood it should be by doing both.

15.1.5 Visualization

If the simulator is to be used as a tool to help place obstacles optimally in a snow filled wind field, it needs to be able to visualize both the wind field and the deposited snow in a more clear manner. Rendering white dots, and exporting and rendering in other applications, is much too slow and cumbersome a process.

15.1.6 Scene design

The scenes used in the simulations should be easy to define through some configuration file, where all relevant properties can be easily defined without having to change the actual program code.

Part V

Appendix

A Parallel programming on the GPU

In this section we will give a very brief introduction to the primary optimization goals one should focus on when developing a massive parallel application using CUDA. It is in no way aimed at teaching cuda programming. The focus is *only* to explain some of the focus points we have been concerned with in section 9.5 and section 14.3.

Please refer to [NVIDIA, 2011] for a more detailed CUDA introduction.

We have be using a NVidia GeForce 580 GTX with 1.5 GB of memory. This card used the new Fermi architecture which bridges many of the gaps between CPU and GPU that have previously complicated GPU coding. Of the more noteworthy we may mention a general cache system, which previously required data to be stored in textures, and recursion through a hardware stack. We have been utilizing that cache greatly in our code.

A.1 CUDA memory types

There are three primary types of memory on a device with different characteristics. They are shared memory which is located on the actual multiprocessor, and which is very fast, global memory which is located off chip and is somewhat slow and then cache memory which is essentially a part of shared memory on chip and is also very fast.

A.2 Optimization on CUDA

We will briefly mention some of the more important factors to consider when writing optimized CUDA code. Some of these are relevant for CPU coding as well.

A.2.1 Parallelization

The power of the GPU is its ability to process a very large number of threads at the same time. For this reason, a problem should lend itself willingly to parallelization by threading to be suitable for the GPU. If a problem is highly serial in nature, then a GPU implementation will often be slower than a CPU implementation. The maximal speedup by parallelizing portions of a program is defined by Amdahl's law

$$\frac{1}{1-P+\frac{P}{N}} \tag{176}$$

where P is the portion of a program which can be made parallel and N is the number of processors used.

If 10% of a program is serial in nature but 90% can be made parallel, then the maximal speedup by using an infinite number of processors is

$$\frac{1}{0.1+\frac{0.9}{\infty}} = \frac{1}{0.1} = 10 \tag{177}$$

We therefore need to focus on making as large a portion as possibly parallel if we want serious improvements.

Luckily many physical problems, certainly including SPH, are *very* parallel in nature. In SPH we can generally calculate all new properties of any one particle based only on old properties of itself and the other particles. It is hard to define exactly what value P has, but it is not far from 1.

A.2.2 Cache hit-rate

Global memory on the GPU is quite fast DDR5 ram running at high clock rate, but in relative terms it is still slow compared to the on chip memory on each multiprocessor. For that reason previous (pre-Fermi) CUDA code used the local, shared, memory on chip as a scratchpad memory or as a program controlled cache. Data to be operated on by a thread block was loaded from global to shared memory, worked on locally and the result was then written back to global memory. This was as an alternative to continuously accessing global memory for each and every single instruction. This is, if not the, then at least one of the single most important optimizations a CUDA programmer could do. Some problems, such as the common matrix multiplication example, are easily written into this form, while other problems are a little more random in their access pattern. For those problems, data could be stored in a 1D, 2D or 3D texture which had actual cache support. It was, however, a read-only cache, so when data was written back into the texture, the cache was not updated. This could complicate matters a little.

For never (as of writing spring 2011) Fermi devices, true caching can be enabled for arbitrary data and this can be handled in hardware. In other words, Fermi devices do have cache as we know it from normal CPU programming. The size of the cache can be user defined from 16kB to 48kB and is per multiprocessor.

To fully utilize a cache one still has to strive towards making memory accesses not to random. A fully random access pattern will generally miss the cache and a cache miss is slightly slower than direct access to global memory. This is therefore a priority for us; to make related data appear close in memory and to access that related data close in time as well.

A.2.3 Device occupancy

A single multiprocessor can handle an integer number of identical sized thread blocks which each consist of a number of threads. The maximal number of threads in each block vary among devices, but the GTX 580 supports 1024 threads per block. The actual number of simultaneous threads on a multiprocessor depends on the complexity of the thread code. Simple code, with low register usage, can be run in many concurrent threads, while more complex code take up more resources. To optimize the utilization of the multiprocessor, we should find the block size which packs the most threads onto the processor. If the thread code complexity is such that 1000 threads can be loaded concurrently, then we would should chose a block size N such that NxM=1000, where M is an integer number of loaded blocks. If we choose N=100, then 10 blocks M can be loaded and we fully occupy the device. If we choose N=200, then 5 blocks M can be loaded. If we choose N=150, then 6.66 blocks M should be loaded, but since M is an integer, we end up loading only 6 blocks resulting in 6 blocks * 150 threads = 900 threads totally. This is 100 threads less that we multiprocessor could actually handle.

While NxM is bounded by the resources required per thread and the resources available on the multiprocessor, we are free to choose the N which optimizes NxM. To make the partitioning of particles into equally sized thread blocks easy, and to have those threads be similar, we should ensure that threads destined for the same thread block are located in a continuous block of memory.

A.2.4 Warp divergence

When threads in a block are executed, they are divided into smaller entities called a warp. A warp on GTX 580 consist of 32 threads. Only one warp at a time is executed on a multiprocessor, meaning that while many more threads may be loaded on the processor only 32 are actually processed at any one time and the processor selects among the warps the one to execute.

This is done as a SIMD-step, Single Instruction Multiple Data, which means that the threads in a warp executes the exact same operation but on different data.

An example could be ADD(OP1,OP2,RES) where every thread performs the ADD instruction but where the operands OP1 and OP2 as well as the location of the result RES are different. This way, in one step, 32 numbers are added. For parallel code, this is often how things should be done.

There is nothing to prevent the programmer from writing code where some threads in a warp does one thing, while other threads does another. This is seen in conditional branches where not all threads follow the same path.

When this happens, some threads take one path while the other threads are paused and then the roles are reversed and the previously paused threads take the other path while the other threads are paused. With two paths of equal complexity, this will make the execution take twice as long since both paths are taken. This is known as warp divergence, and it is highly undesirably for obvious reasons.

While it is often not possible to entirely avoid divergence, it should be kept at a minimum. For this reason it is important that threads in the same warp tend to do the same things. The primary difference among particles is the number of neighbors. It is therefore optimal if all threads in the same warp have the same number of neighbors.

If a spatial partitioning is used to optimize neighbor finding, then it would also be best if the same paths through this structure is followed.

A.2.5 Summary

To summarize, we wish to have thread blocks which contain threads that are ”alike”. They should be close in space and close in memory. They should have roughly the same neighbors and preferably they should have each other as neighbors. Additionally we desire the thread blocks to allow full occupancy on the device.

Having ”alike” particles in the thread blocks will ensure that they access the same data (other particles) in the same order in memory which will optimize the cache utilization. The particles will also have a similar number of neighbors and they will travel along the same paths in spatial portioning structures.

A method of ensuring that similar particles, meaning particles close in space, are close in memory and that they are easily partitioned into thread blocks is to form a 3D space filling curve and sort the particles by an index which depends on how far along the length of the curve the particle is located. This is detailed in section 9.5.4

If the particles are sorted properly in memory, then we can take *any* continuous block of particles from memory, with any size, and be confident that the particles in that block will be similar. We are therefore free to choose the optimal block size and to define the relevant particles for any given block based on just a start and an end index. This makes it easy to obtain good occupancy.

Again this is something we can attempt to obtain by sorting the particles with a space filing curve. Particle close in space will tend to have the same number of neighbors and if a spatial partitioning is used, the particles will tend to travel along the same paths through the structure to locate the neighbors.

References

[Adams et al., 2007] Adams, B., Pauly, M., Keiser, R., and Guibas, L. J. (2007). Adaptively sampled particle fluids. *ACM Trans. Graph.*, 26.

[Ajmera et al., 2008] Ajmera, P., Goradia, R., Chandran, S., and Aluru, S. (2008). Fast, parallel, gpu-based construction of space filling curves and octrees. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, I3D '08, pages 10:1–10:1.

[Andreotti et al., 2002] Andreotti, Bruno, Claudin, Philippe, Douady, and Stéphane (2002). Selection of dune shapes and velocities. Part 1: Dynamics of sand, wind and barchans. *undefined*.

[Bang et al., 1994] Bang, B., Nielsen, A., Sundsbø, P. A., and Wiik, T. (1994). Computer simulation of wind speed, wind pressure and snow accumulation around buildings (snow-sim). *Energy and Buildings*, 21(3):235 – 243.

[Beard, 1976] Beard, K. (1976). Terminal velocity and shape of cloud and precipitation drops aloft. *undefined*.

[Bishop, 2006] Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

[Bonet and Wood, 2008] Bonet, J. and Wood, R. (2008). *Nonlinear continuum mechanics for finite element analysis*. Cambridge University Press.

[Capone, 2010] Capone, T. (2010). *SPH numerical modelling of impulse waterwaves generated by landslides*. PhD thesis, undefined.

[Cresseri and Jommi, 2005] Cresseri, S. and Jommi, C. (2005). Snow as an elastic viscoplastic bonded continuum: a modeling approach. *Rivista Italiana de Geotechnica*.

[Erturk et al., 2004] Erturk, E., Corke, T. C., and Gökçöl, C. (2004). Numerical solutions of 2-d steady incompressible driven cavity flow at high reynolds numbers. *Computing Research Repository*, cs.NA/0411.

[Fearing, 2000] Fearing, P. (2000). Computer modelling of fallen snow.

[Feldman and O'Brien, 2002] Feldman, B. E. and O'Brien, J. F. (2002). Modeling the accumulation of wind-driven snow.

[Garcia et al., 2008] Garcia, V., Debreuve, E., and Barlaud, M. (2008). Fast k nearest neighbor search using gpu. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6.

[Georgakis, 2010] Georgakis, C. (2010). The new dtu/force technology climatic wind tunnel: design, construction and calibration. *11th Italian National Conference on Wind Engineering*.

[Goswami et al., 2010] Goswami, P., Schlegel, P., Solenthaler, B., and Pajarola, R. (2010). Interactive SPH Simulation and Rendering on the GPU. In Otaduy, M. and Popovic, Z., editors, *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 1–10.

[Heath, 2002] Heath, M. (2002). *Scientific Computing - An introductionary Survey*. McGrawHill.

[Hong, 2009] Hong, W.-S. (2009). *An adaptive sampling approach to incompressible particle-based fluid*. PhD thesis, undefined, College Station, TX, USA. AAI3370710.

[Hosseini et al., 2007] Hosseini, S. M., Manzari, M. T., and Hannani, S. K. (2007). A fully explicit three-step sph algorithm for simulation of non-newtonian fluid flow. *International Journal of Numerical Methods for Heat &amp;#38; Fluid Flow*, 17(7):715–735.

[Ihmsen et al., 2011] Ihmsen, M., Akinci, N., Becker, M., and Teschner, M. (2011). A parallel sph implementation on multi-core cpus. *Comput. Graph. Forum*, 30(1):99–112.

[Kreyszig, 2005] Kreyszig, E. (2005). *Advanced Engineering Mathematics*. Wiley, 9 edition.

[Krog, 2010] Krog, Ø. E. (2010). Gpu-based Real-Time snow avalanchesimulations. Master's thesis, Norwegian University of Science and Technology.

[Lehning et al., 2006] Lehning, M., Gustafsson, D., Nguyen, and Zappa, M. (2006). Alpine3d: a detailed model of mountain surface processes and its application to snow hydrology. *Hydrological Processes*, 20(10):2111–2128.

[Lieberherr and Parlange, 2010] Lieberherr, G. and Parlange, M. (2010). Modeling snow drift in the turbulent boundary layer. *Laboratory of Environmental Fluid Mechanics*.

[Liu and Liu, 2003] Liu and Liu (2003). *Smoothed Particle hydrodynamics - a meshfree particle method*. World Scientific.

[Meller, 1975] Meller, M. (1975). A review of basic snow mechanics. *undefined*.

[MIT, 2006] MIT (2006). Movement of sediment by the wind. *undefined*.

[Müller et al., 2008] Müller, M., Stam, J., James, D., and Thürey, N. (2008). Real-time physics - class notes. *Siggraph*.

[Moeslund et al., 2005] Moeslund, T. B., Madsen, T. B., Aagaard, M., and Lerche, D. (2005). Modeling falling and accumulating snow. In *Second International Conference on Vision, Video and Graphics*. Unknown.

[Monaghan, 1992] Monaghan, J. (1992). Smoothed particle hydrodynamics. *undefined*, 30:543–574.

[Monaghan and Kajtar, 2009] Monaghan, J. J. and Kajtar, J. B. (2009). Sph particle boundary forces for arbitrary boundaries. *Computer Physics Communications*, 180:1811–1820.

[Mott et al., 2010] Mott, R., Schirmer, M., Bavay, M., and T. Grunewald, M. L. (2010). Understanding snow-transport processes shaping the mountain snow cover. *Cryosphere Discuss.*

[Müller et al., 2003] Müller, M., Charypar, D., and Gross, M. (2003). Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[NISHIMURA and MAENO, 1987] NISHIMURA, K. and MAENO, N. (1987). Experiments on snow-avalanche dynamics. *Avalanche Formation, Movement and Effec.*

[Nishimura and Maeno, 1989] Nishimura, K. and Maeno, N. (1989). Contribution of viscous forces to avalanche dynamics. *undefined.*

[Nishimura and Sugiura, 1998] Nishimura, K. and Sugiura, K. (1998). Measurements and numerical simulations of snow-particle saltation. *Anals og Glaciology.*

[Nvidia, 2011] Nvidia (2011). *CUDA CUSPARSE library.*

[NVIDIA, 2011] NVIDIA (2011). *NVIDIA CUDA Programming Guide 4.0.*

[Paiva et al., 2009] Paiva, A., Petronetto, F., Lewiner, T., and Tavares, G. (2009). Particle-based viscoplastic fluid/solid simulation. *Computer-Aided Design*, 41(4):306–314.

[Piskova et al., 2008] Piskova, M., Jarusch, S., Kopf, C., Novak, P., and Trappmann, D. (2008). An introduction to the basic properties and mechanics of snow. *Snow and avalances.*

[Sagaut, 1998] Sagaut, P. (1998). Large eddy simulation for incompressible flows.

[Saltvik et al., 2007] Saltvik, I., Elster, A. C., and Nagel, H. R. (2007). Parallel methods for real-time visualization of snow. In *Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing*, PARA'06, pages 218–227. Springer-Verlag.

[Squires, 2008] Squires, K. D. (2008). Large eddy simulations for particle-laden turbulent flows. *undefined.*

[Tabler, 1991] Tabler, R. D. (1991). Snow fence guide. *undefined.*

[USArmyCorpsOfEngineers, 1987] USArmyCorpsOfEngineers (1987). Real-time snow simulation model for the monongahela river basin. Technical report, Hydrologic Engineering Center.

[Versteeg and Malalasekra, 2007] Versteeg, H. and Malalasekra, W. (2007). *An Introduction to Computational Fluid Dynamics: The Finite Volume Method (2nd Edition) (Paperback)*. Prentice Hall; 2 edition.

[Zhang and Huang, 2008] Zhang, J. and Huang, N. (2008). Simulation of snow drift and the effects of snow particles on the wind. *Hindawi Publishing Coorporation*.

[Zhang et al., 2008] Zhang, Y., Solenthaler, B., and Pajarola, R. (2008). Adaptive sampling and rendering of fluids on the gpu. In *Proceedings Symposium on Point-Based Graphics*, pages 137–146.