Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Mathias Chunnoo

Simulating Snow as an Elastoplastic Material on the GPU

Master's thesis in Computer Science
Supervisor: Anne C. Elster
July 2022

NTNU
Norwegian University of
Science and Technology

Mathias Chunnoo

Simulating Snow as an Elastoplastic Material on the GPU

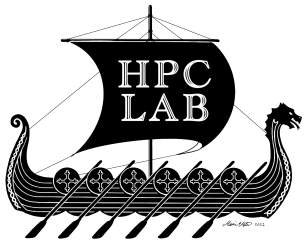
Master's thesis in Computer Science
Supervisor: Anne C. Elster
July 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

NTNU
Norwegian University of
Science and Technology

Mathias Chunnoo

Simulating Snow as an Elastoplastic Material on the GPU



Master's thesis in Computer Science
Supervisor: Professor Anne C. Elster
June 2022

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Abstract

The last couple of years, especially during the recent pandemic, Norway has seen a drastic increase in people trying out ski touring for the first time. With this increase in the number of inexperienced tourers traversing snowy mountains, there is an encompassed risk of them being exposed to or triggering avalanches. It is therefore very important to understand better how to avoid the dangers of avalanches. As the most dangerous type of avalanche is the slab avalanche, this thesis investigates the possibility of simulating such avalanches.

Snow is, however, an exceptionally complex material whose physical properties range from dusty cold snow to dense ice depending on its temperature and pressure. In addition, it is typically built up in layers that reflect the conditions at the time of the snow fall, but also the overall current weather conditions. However, recent works have shown promising results by modeling snow as an elastoplastic material. One of the numerical techniques used is the SPH (Smoothed Particle Hydrodynamics) method, which has the ability to model a wide variety of snow properties.

With the end of Moore's law becoming a reality, computers have had to improve in other ways. This has pushed the vendors of GPUs (Graphical Processing Units) to prioritize their use for general computational tasks. These computer chips contain hardware dedicated to performing highly parallel tasks. SPH simulations are inherently computationally intensive, but can benefit from a high degree of parallelism. Utilizing the power of GPUs is therefore worth exploring and this thesis thus focus on how to model slab avalanches using SPH on GPUs.

The thesis's contributions include a fairly detailed summary of the numerical methods used, including the SPH and how it is used in snow simulations, as well as a serial implementation that, while not meeting the ambitions of simulating full scale avalanches, shows an ability to model certain aspects found in slab avalanches such as crack formations. In addition, the thesis demonstrates how a parallel implementation on the GPU, which is integrated into the HPC-Lab snow simulator, while remaining computationally expensive, does result in great performance benefits. The results includes timing and graphical outputs for three types of implicit stress solved for both acceleration and velocity as well as explicit stress.

Sammendrag

De siste årene, spesielt under pandemien, har Norge sett en drastisk vekst i folk som prøver topptur for første gang. Med denne økningen i uerfarne turgåere i snøbelagte fjell, er det en medfølgende fare for utsettelse for og utløsning av snøskred. Det er derfor veldig viktig å bedre forstå hvordan man unngår farene ved snøskred. Ettersom den mest skadelige typen av snøskred er flakskred, undersøker denne avhandlingen mulighetene for å simulere slike skred.

Snø er derimot et ekstremt komplisert materiale, med fysiske egenskaper som omfatter alt fra kald støvete snø til hard is, gitt dens temperatur og trykk. I tillegg til dette er den vanligvis bygd opp i lag som reflekterer forholdene da snøen falt, i tillegg til aktuelle vær forhold. Nylig arbeid har derimot vist lovende resultater ved å modellere snø som et elastoplastisk materiale. En av de numeriske teknikkene som er brukt er SPH (Smoothed Particle Hydrodynamics) metoden som har muligheten til å modellere et vidt utvalg av snøegenskaper.

Ettersom slutten av Moores lov har tredd i kraft, har man måttet finne nye måter å øke kraften til datamaskiner. Dette har presset grafikkortleverandører til å prioritere grafikkorts bruk innen generelle beregninger. Disse kortene inneholder maskinvare som er dedikert til å utføre store mengder parallelle oppgaver. SPH simulasjoner er i seg selv beregningsmessig intensive, men de kan dra nytte av en stor grad av parallellisering. Bruken av grafikkort er derfor verdt å utforske og denne avhandlingen fokuserer på hvordan man kan modellere flakskred med SPH på grafikkort.

Denne avhandlingens bidrag inkluderer en meget detaljert oversikt over numeriske metoder som er brukt, spesielt SPH og hvordan det er brukt for snø simulering, i tillegg til en seriell implementasjon som, selv om den ikke når ambisjonene om å simulere flakskred i sin helhet, viser en evne til å modellere enkelte aspekter man finner i flakskred, slik som sprekkdannelse. I tillegg viser denne avhandlingen at en parallell implementasjon på grafikkort, integrert i HPC-Labbens snøsimulator, selv om den forblir beregningsmessig intensiv, oppnår gode ytelses forbedringer. Resultatene inkluderer tidtaking og grafiske visualiseringer av tre typer implisitt stress, hvorav det er løst for båre akselerasjon og fart, i tillegg til eksplisitt stress.

Acknowledgements

I, the author of this thesis, would like to thank my advisor Dr. Anne C. Elster for giving me such an inspiring task and for the ability to use the HPC-Lab, my family for their support and them buying me food during the heftiest evenings, the guys I've been living with for their company and for going skiing with me throughout the semester, and lastly the people at the HPC-Lab for being such great work partners.

Contents

Figures

Tables

Algorithms

Chapter 1

Introduction

Snow is an extremely complex material with a verity of different properties. In certain situations, like in the upper layer of slab avalanches, snow is compacted such that exhibit properties reminiscent of elastoplastic materials. Elastoplastic materials has seen a lot of computational work the last decades, using methods such as SPH (Smoothed Particle Hydrodynamics) in order to simulate them. With the continued improvement of GPUs (Graphical Processing Units), they can be utilized to improve the performance of such simulations. This thesis thus attempts to implement a simulation of snow as an elastoplastic material using GPUs.

1.1 Motivation

In recent years, especially during the pandemic, Norway has seen a large increase in people trying out ski touring for the first time. This means that more people are traversing alpine terrain and consequentially exposing them selves to the danger of snow avalanches. Understanding how, when and where these avalanches occur is therefore important in order to make ski touring a safe activity.

While typical weather forecasting can give indications to whether or not an certain terrain is avalanche prone, they fail to capture small scale terrain variances, which are highly important for the formation of avalanches. This thesis will thus try to create a numerical model that can capture these smaller scale properties of snow.

Previous work on snow simulation at the NTNU HPC-Lab has focused on various aspects aspects of snow and avalanches. Most of the work has focused on either snow fall, like the works of Saltvik *et al.* [1] and Eidissen [2], or the flow of avalanches, like the works of Krog [3] and Sandvik [4]. The works that has focused on the flow of avalanches has all simulated the avalanche motion as a fluid. This, however, fails to capture the effects that initiate avalanches and it does not enable modeling the most dangerous types of avalanches, which are slab avalanches. The work by Boge [5] tries to capture these effects by modeling the fracture dynamics of snow slabs. However, while that does capture some of the properties that initiates slab avalanches, it lacks the ability to model the movement of avalanches

in action and it remains disconnected from the other simulation models that are implemented.

The work by Gaume *et al.* [6] implements a material point method that models both the initial formation of slab avalanches and their movement through the terrain. Their model does however lack the ability to simulate snow fall and the formation of the snow covers that enables slab avalanches.

The most recent work on snow simulation on snow simulation is the work by Gissler *et al.* [7], where they simulate snow as an elastoplastic material using the SPH method. This method has the advantage of simulating individual snow particle, thus being able to model snow fall, accumulation and the properties of snow covers using only a single model. They do however not relate their work to avalanches and focuses rather on the general properties of snow.

The method by Gissler *et al.* [7] is essentially meant for longer time consuming simulations and they have thus implemented it on CPUs (Central Processing Units). SPH is however a method that can take great advantage of the parallelism enabled by modern GPUs.

This thesis thus tries to implement an SPH for simulating snow as an elastoplastic material with a focus on its ability to model snow properties that are present in slab avalanches. The thesis further tries to implement the method on GPUs in order to acquire the performance benefits from a high degree of parallelism.

1.2 Previous Work

This section describes previous work that has been conducted within the field of snow simulation. The section first looks at previous work on the NTNU HPC-Lab snow simulator before examining work related specifically to the use of SPH and simulation of elastoplastic materials.

1.2.1 The HPC-Lab Snow Simulator at NTNU

This section describes what has previously been accomplished throughout the work on the HPC-Lab snow simulator at NTNU.

Snowfall and Accumulation Saltvik *et al.* [1] laid the ground work for the HPC-Lab snow simulator when he implemented a model for snow fall and accumulation. The method modeled snow flakes as particles and the terrain was modeled as a height map, in which accumulation values where computed when snow particles intersected the terrain. In order to have a more physically correct snow fall model he implemented a wind model using the method of Stam [8] to simulate the movement of air as a fluid.

Eidissen [2] improved upon the work by Saltvik *et al.* [1] by implementing the work on GPUs using Nvidia's CUDA (Compute Unified Device Architecture). This greatly improved the performance of the simulator and paved a way for future work to use similar techniques.

Avalanches Krog [3] laid the ground work for simulating avalanches by implementing a fluid model on GPUs using SPH. This work modeled the flow of avalanches through terrain as a fluid. His implementation utilized a fast parallel neighborhood search algorithm by Green [9]. This work was not initially integrated in the HPC-Lab snow simulator, but was later integrated during the work of Sandvik [4].

As simulating avalanches as fluids does not capture the properties of slab avalanches, which are the most dangerous types of avalanches, Boge [5] implemented a method for modeling fracture dynamics in snow slab covers. This method does however only model the stresses within a slab and is, at the moment, completely disconnected from the other aspects of the HPC-Lab snow simulator.

1.2.2 Snow Simulation and Smoothed Particle Hydrodynamics

This section covers previous work done on snow simulation, with a focus on work that model snow as an elastoplastic material. The section will further look at work on SPH that has lead up to enabling its use for snow simulation.

Snow Simulation While most earlier work on snow simulation model snow as fluids or granular materials, the method by Stomakhin *et al.* [10] uses a material point method to model snow as an elastoplastic material. Their work lay much of the ground work for modeling snow as elastoplastic materials in later works. Their method is however implemented with the purpose of animating snow for the Disney movie Frozen and thus prioritizes visual results.

Gaume *et al.* [6] takes the work by Stomakhin *et al.* [10] and utilizes it to model full scale slab avalanches. They do however find that the method can not properly model a weak snow layer with fracture properties that are required to propagate large avalanches. They thus implement their own method of dynamic anticrack propagation in specialized snow layers.

The work of Gissler *et al.* [7] takes the physical models from Stomakhin *et al.* [10], but implements them using an SPH method. This enables individual snow particles to be modeled properly and enables better interactions with other materials. As snow particles can be modeled individually, their method can simulate both the properties of snow fall and accumulated snow and its properties, thus enabling a single model for the whole life cycle of snow, with the exception of snow melting into water.

Smoothed Particle Hydrodynamics Gingold and Monaghan [11] initially derived the SPH method for simulating the physical properties stars and the gravity acting between them. The method has since been widely used for simulating fluid as it enables a simulation where one only requires computationally expensive calculations in the space occupied by the fluid.

As the method has seen such a wide use, many improvements have been developed throughout various papers. Akinci *et al.* [12] developed a method for

letting fluid simulations, using SPH, interact with rigid bodies from other simulation methods. Their method replaces rigid bodies with a set of boundary particles that forms a shell around the edge of the rigid body. These boundary particles further enable forces that pushes the fluid away from the rigid body and friction between the rigid body and the fluid. This method was further improved by Band *et al.* [13], where they implemented a method that modeled the pressure around boundary particle more physically correct.

While much of the earliest work on SPH focuses on its use for fluid simulations, more recent work uses it to model more complex and solid materials. Peer *et al.* [14] uses SPH to simulate elastic solid in which internal forces due to stress are captured in the method. The snow model of Gissler *et al.* [7] utilizes much of the work by Peer *et al.* [14], but also integrates a method for modeling plastic behavior, thus enabling the simulation of elastoplastic materials using SPH.

1.3 Contribution

This section describes what this thesis has contributed to the continued work on snow simulation at the NTNU HPC-Lab.

In-Depth Background Research Understanding how to implement SPH as well as the physical properties of elastoplastic materials are both complex and challenging tasks. A great amount of time and work has been put down in order to more fully understand them. The amalgamation of this research has resulted in Chapter 2. This chapter will hopefully serve as a decent starting point and reference for those those interested in how to develop SPH simulations, especially, those that are interested in simulating the properties of snow, including the properties apparent in avalanche simulations.

Implementations This thesis has resulted in two complete implementations of snow simulated as an elastoplastic material using SPH.

The first implementation [15] is a serial implementation implemented on CPUs. It it thus not very high performant. It does, however, implement nine different methods for simulating snow, three different stress decouplings and three different ways of solving for the forces resulting from stress. As there is a lack of open source implementation of complete elastoplastic SPH simulations, this implementation will hopefully be a great reference for understanding the implementation details at a general level.

The second implementation is a parallel implementation utilizing GPUs. This implementation is integrated into the existing HPC-Lab snow simulator, but does require some more work to work in conjunction with previously implemented methods, like terrain and wind. This implementation is thus a starting point for simulating complete full scale slab avalanches in the HPC-Lab snow simulator.

1.4 Outline

This section outlines the content of the rest of this thesis by describing what is contained withing each chapter and section.

Chapter 2 This chapter goes into detail on all the background research done during the work on this thesis.

Section 2.1 introduces the chapter by describing what snow is and its properties, followed by a description of various types of avalanches.

Section 2.2 describes what partial differential equations are, which is the mathematical basis of numerical simulations, followed by a description of certain methods for solving them.

Section 2.3 begins by a derivation of SPH and some of its discretizations, before describing a process for using it in practice and ends with a description of boundary handling.

Section 2.4 gives an introduction to the physics behind elastic materials and the mathematical equations that arise from it, in addition to describing how one can modify these equations in order to model plastic properties.

Section 2.5 ends this chapter by describing what GPUs are and how they operate with a focus on the Compute Unified Device Architecture by Nvidia.

Chapter 3 This chapter goes further into detail on how snow can be modeled as an elastoplastic material using SPH.

Section 3.1 describes a method quite similar toe the method by Gissler *et al.* [7] and all the mathematical equations that the method requires.

Section 3.2 describes eight alternative solvers that slightly differs in various ways from the method described in the preceding section.

Chapter 4 This chapter describes the details of the implementations that where implemented during the work on this thesis.

Section 4.1 describes in detail the serial implementation implemented on central processing units.

Section 4.2 describes in detail the parallel implementation which utilizes GPUs.

Chapter 5 This chapter presents the various results acquired from running the simulation implementations.

Section 5.1 presents the results from the serial implementation. It first presents some general result, before presenting results with a focus on the simulations ability to model slab avalanches.

Section 5.2 presents results from running the parallel implementation.These results have a higher focus on the performance of the implementation.

Chapter 6 This chapter concludes this thesis by reciting what has been accomplished and paving a way for future work on snow simulation.

Section 6.1 describes what has been accomplished in terms of both work and results.

Section 6.2 points to certain aspects of snow simulation that was not accomplished during the work on this thesis, but would be interesting to see more work towards.

Chapter 2

Background

This chapter goes into detail on the various background material that has been used during the work on this thesis. The chapter starts off by describing snow and its various properties, with a special focus on how it relates to avalanches and the reasons for their occurrences. Then the chapter begins the mathematical background of the thesis by first describing partial differential equations and various methods of solving them, as they the fundamental mathematical background with which numerical simulations are built on, before going in depth into the derivation and use of smoothed particles hydrodynamics, which is the numerical framework this thesis uses for its simulations. The mathematical sections are ended with a look at the physics of elastoplastic material, and lastly the chapter describes what graphical processing units are.

2.1 Snow

Snow is a very complex material with a variety of different characteristics depending on its construction. All forms of snow are essentially a mass consisting of snow crystals of various sizes and various configurations. When the not strongly bound to each other, they are free to move around and the snow will exhibit properties similar to granular materials like sand or properties similar to fluids like water, depending on the sizes of the snow crystals. However, when the snow crystals are bound to each other, the snow acts more like an elastoplastic material, being able deform or form cracks and break up into solid chunks.

2.1.1 Snowfall

Snow typically develop when moisture in the air condenses into water droplets, which are then frozen and turned into snow crystals, or snow flakes. As they fall from the sky, their paths are influences by the wind. To which degree they are affected by wind is highly dependent on the sizes and weight of the snow flakes, which again is dependent on temperatures and humidity in the air.

2.1.2 Accumulation

When snow land on the ground, it accumulates and form a snow cover. As external effects like temperature changes the snow covers properties, more snow may fall on top of it, creating a new layer with other properties. Various environmental effects affect the development of these layers. Warm temperatures make them melt into a more sludgy substance and colder temperatures may freeze these layers into ice sheets, or enable growth of larger snow crystals. Pressure from newer snow layers compresses the layers below, making them more dense. When snow is compressed it tends to harden, making it exhibit properties that are more similar to solids.

Grain Forms The shape of snow crystals can vastly affect the properties of snow. However, inspecting snow as such a microscopic level is not always feasible. Thus, snow is typically categorized into a set of grain forms, as can be seen in Table 2.1, which includes density data from Geldsetzer and Jamieson [16], Calonne *et al.* [17] and Glen [18]. The last two rows of the table are not typically considered grain forms as they are typically not considered when evaluating avalanche danger, but they are added to show a wider variety of properties snow can exhibit.

Table 2.1: Grain forms and their approximate density.

Grain Form	**Approximate Density Range** $\left[\frac{kg}{m^3}\right]$
Precipitation Particles	60-180
Decomposing and Fragmented Particles	80-240
Rounded Grains	170-420
Faceted Crystals	125-315
Depth Hoar	200-270
Melt Forms	470-550
Melt-Freeze Crust	270-330
Wet Grains	45-310
Firn	400-830
Glacier Ice	830-920

When snow layers on the ground experience temperature and pressure changes, either from layers above or from external forces like wind, their composition of grains changes [19]. Understanding how the snow changes due to external effects is highly important when evaluating the danger of avalanches but is not relevant for this thesis.

2.1.3 Avalanches

Snow avalanches are a very typical phenomenon in alpine areas and one can often see traces of when visiting the mountains of Norway. Some avalanches occur naturally due to buildup of certain types of snow or due melting, which gives it less

strength making it unable to hold its own wight. However, in certain situations, the snow can lay stable when undisturbed, but external forces like humans can trigger avalanches. This can pose a great danger for people touring in the area and understanding when these situations occur and avoiding them is a key to making mountaineering safe. This section will thus describe four types of avalanches and their properties [20, 21].

Powder Avalanches Powder avalanches occur when great amounts of snow has fallen and its been undisturbed by wind. These avalanches typically only occur in terrain steeper than 50° [22] and can be triggered both naturally and by external forces. While these avalanches occur right after snow fall, the snow can stabilize quite quickly after the snow fall and thus not form these types of avalanches. As the newly fallen snow consists of smaller non-compacted particles, powder avalanches behaves quite similar to fluids or granular materials when they flow down mountain sides.

Wet Snow Avalanches Wet snow avalanches occurs when the snow is highly saturated with water, such as after heavy rain fall. This makes the snow both more malleable and heavier, such that it can not hold its own wight. Due to the presence of water in the snow, these avalanches behaves very similar to fluids and they typically occur in terrain with a steepness of only 5° to 25°. As the addition of water due to rain fall happens over a short period of time, these avalanches usually occur naturally. Due to the density of this type of snow, wet snow avalanches can cause more environmental damage compared to powder avalanches.

Slab Avalanches Slab avalanches are the most dangerous avalanches for people touring the mountain sides, as their causes are hard to spot and their consistency can cause great amounts of damage. They form when snow is transported by wind and compacted into a solid layer with a density of around $400\frac{kg}{m^3}$ and the layer directly below is a weaker layer with lower density, typically consisting of rounded grains or faceted crystals. As the above layer is solid, these avalanches does not typically trigger naturally. However, when an external force, like a person, compresses the snow, is can form fractures in the weak layer which propagates and breaks off massive chunks from the solid layer which slides down the mountain side. These avalanches typically occurs in terrain with a steepness of 30° to 45°. As the snow in these avalanches have a high density and compactness it exhibit properties quite similar to elastoplastic materials.

Gliding Avalanches Gliding avalanches behaves similar to slab avalanches, but does not contain a weak layer and instead only consists of a solid layer on top of the ground. These avalanches typically occur naturally at the end of the winter, when the snow heats up and the friction between the snow and the ground can not hold the weight of the snow. The process that forms these avalanches is slow

and forms large cracks in the snow layer, which makes them easily noticeable and avoidable.

2.2 Partial Differential Equations

Most physical problems can be expressed as a relationship between how things are and how they change. This is the exact purpose of differential equations, which are equations expressing the relationship between functions and their derivatives. As stated by Alan Turing: "Science is a differential equation. Religion is a boundary condition." [23]

Most problems does however require quite complex function, thus Partial Differential Equations, or PDEs, are used. PDEs are equations which expresses relationships between multivariable functions and their derivatives. An example of a PDE is the Poisson equation in three dimensions

$$\nabla^2 u = \frac{\partial^2 u^2}{\partial x^2} + \frac{\partial^2 u^2}{\partial y^2} + \frac{\partial^2 u^2}{\partial z^2} = f, \tag{2.1}$$

where u is a multivariable function and the sum of its second derivatives with respect to the three spacial directions, x, y and z, is equal to some function f. A solution to this PDE is any function u that satisfies this equation.

While some PDEs can be solved analytically, giving results as mathematical functions. Solving more complex PDEs analytically, like the PDEs arising from the physics of elastoplastic materials, is usually unfeasible. Thus they are typically solved numerically, giving an approximation of a solution to the PDE. This section will hence explain some ways of solving PDEs numerically.

2.2.1 Explicit Methods

Explicit methods are methods for solving time dependent problems by expressing an equation for a later point in time only given the current time of the problems system. Given the current state of the problems system S_t and one wants to find the state of the system $S_{t+\Delta t}$ at a later point in time, an explicit method would use a function F such that

$$S_{t+\Delta t} = F(S_t). \tag{2.2}$$

The advantage of explicit methods is that every advancement in time only requires one to compute the result of the function F once, which can be highly performant. However, explicit methods have the disadvantage that the accuracy of the solution is highly dependent on the size of the time advancement Δt.

2.2.2 Implicit Methods

While explicit methods expresses an equation for the problems system $S_{t+\Delta t}$ at a later point in time given the system S_t at the current time, implicit methods

instead expresses a requirement function G that has to hold for the two times of the system such that

$$G(S_t, S_{t+\Delta t}) = 0. \tag{2.3}$$

Thus, solving the problem amounts to solving Equation (2.3) for $S_{t+\Delta t}$.

The advantage of implicit methods is that one can achieve a higher precision with larger time advancements compared to explicit methods. However, they have the disadvantage that they are difficult to formulate and may have a higher computational cost for a singe time advancement.

2.2.3 Iterative Methods for Solving Systems of Linear Equations

When numerically solving partial differential equations with implicit methods, the problem tend to result in a system of linear equations

$$A\mathbf{x} = \mathbf{b}. \tag{2.4}$$

Solving the problem thus equates to solving the resulting system of linear equations. Many methods for solving such systems have been developed for various situations.

Systems of linear equations (2.4) can be solved directly, using for instance Gaussian elimination. However, for most practical uses, when matrices are large and sparse, using iterative methods tend to be more efficient. Iterative methods starts of with an initial guess for the vector $\mathbf{x}$ and iteratively updates the vector until it converges to some correct solution.

Jacobi Method The most basic iterative method is the Jacobi method, which splits the matrix A into a diagonal matrix D, a strictly lower-triangular matrix L and a strictly upper-triangular matrix U such that $A = D + L + U$. Then $\mathbf{x}$ is iteratively improved using the formula

$$\mathbf{x}^{k+1} = D^{-1}\left(\mathbf{b} - (L+U)\mathbf{x}^k\right). \tag{2.5}$$

The simplicity of these matrices makes computing the matrix vector products quite easy. Thus the above equation can be written in its element form as

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^k\right), \tag{2.6}$$

where x_i and b_i is the i-th element of its respective vector and a_{ij} is the j-th element of the i-th row of the matrix A.

Gauss-Seidel Method The Gauss-Seidel method improves upon the aforementioned Jacobi Method by realizing that if $\mathbf{x}$ is updated element wise from the first element to the last element, then already calculated elements can be used in the current iteration to speed up the process, giving the formula

$$\mathbf{x}^{k+1} = D^{-1}\left(\mathbf{b} - L\mathbf{x}^{k+1} - U\mathbf{x}^{k}\right). \tag{2.7}$$

This method can also be expressed in element form as

$$x_i^{k+1} = \frac{1}{a_{ii}}\left(b_i - \sum_{j=1}^{i-1} a_{ij}x_j^{k+1} - \sum_{j=i+1}^{n} a_{ij}x_j^{k}\right). \tag{2.8}$$

Over and Under Relaxation As mentioned above, iterative methods iteratively improves $\mathbf{x}$ until it converges to a solution. This can also be tough of as taking a step from $\mathbf{x}$ towards the solution. Equation (2.7) can be rewritten to emphasize this as

$$\mathbf{x}^{k+1} = \mathbf{x}^{k} + \left(D^{-1}\left(\mathbf{b} - L\mathbf{x}^{k+1} - U\mathbf{x}^{k}\right) - \mathbf{x}^{k}\right), \tag{2.9}$$

where $\left(D^{-1}\left(\mathbf{b} - L\mathbf{x}^{k+1} - U\mathbf{x}^{k}\right) - \mathbf{x}^{k}\right)$ is the step in the direction towards the solution. Using this formulation, one could change the size of the step by multiplying the step length by some factor ω, giving

$$\begin{aligned}\mathbf{x}^{k+1} &= \mathbf{x}^{k} + \omega\left(D^{-1}\left(\mathbf{b} - L\mathbf{x}^{k+1} - U\mathbf{x}^{k}\right) - \mathbf{x}^{k},\right)\\ &= (1-\omega)\mathbf{x}^{k} + \omega D^{-1}\left(\mathbf{b} - L\mathbf{x}^{k+1} - U\mathbf{x}^{k}\right).\end{aligned} \tag{2.10}$$

When $1 < \omega < 2$ this method is called successive over relaxation, or SOR, which typically converges to the solution faster than the Gauss-Seidel method. When $0 < \omega < 1$, this method is called successive under relaxation, which does converge slower, but may find solutions in situations where Gauss-Seidel does not converge.

Bi-Conjugate Gradient Stabilized Method As mentioned above, many iterative methods has been developed. Gissler *et al.* [7] uses one such method called Bi-Conjugate Gradient Stabilized Method, or BiCGSTAB, hence it will be described here. This thesis will however not delve into this methods derivation or workings. BiCGSTAB have been implemented many times, but one implementation that has been widely used, is the implementation by Nuentsa-Wakam and Guennebaud [24] for the Eigen library, thus this thesis will try to stick to their method of implementation as seen in Algorithm 2.1.

Algorithm 2.1: Bi-CGSTAB as implemented by the Eigen library, without restarts and preconditioning.

```
function BiCGSTAB(A, x, b, ϵ, i_max)
    r = b − Ax
    r_0 = r
    if b^2 == 0
        x = 0
        return x
    ρ = 1
    α = 1
    ω = 1
    v = 0
    p = 0
    i = 0
    while r^2 > ϵ^2 b^2 and i < i_max
        ρ_old = ρ
        ρ = r_0 · r
        β = ρα/ρ_old ω
        p = r + β(p − ωv)
        v = Ap
        α = ρ/r_0 · v
        s = r − αv
        t = As
        if t^2 > 0
            ω = t · s/t^2
        else
            ω = 0
        x += αp + ωs
        r = s − ωt
        i += 1
    error = √(r^2/b^2)
    return x
```

Matrix-Free Methods As seen in Algorithm 2.1, there are three places where one is required to multiply the matrix A with some vector. Sometimes it is however unfeasible to explicitly express the matrix A for a linear system. In those cases, one can use a matrix-free approach. The idea behind matrix-free methods is to replace the matrix product $A\boldsymbol{x}$ with some function $A(\boldsymbol{x})$ which gives the same result as the theoretical matrix product would have given. Thus the iterative method can call that function instead of computing a matrix product.

2.3 Smoothed Particle Hydrodynamics

Smoothed particle hydrodynamics, or SPH for short, is essentially a numerical method for solving partial differential equations. The method was first introduced by Gingold and Monaghan [11], when they used it to model astrophysical bodies. The method has been further developed quite a bit since their work, and is today used for simulating a all kinds of continuum materials, like fluids or solids.

SPH differs from typical numerical methods by using a Lagrangian discretization of the problem domain. In other words, the continuum media in question is modeled as a set of point masses. Each of these points also contain the values of all relevant continuous fields at their positions. These points are in SPH called particles, as they contain more than just position information. This kind of discretization has the advantage that the problem domain can be localized to withing continuum media, needing no calculations outside the media.

2.3.1 Derivation of Smoothed Particle Hydrodynamics

Discretization of Scalar Fields The goal of SPH is to give approximations of values of relevant fields at various positions within these fields. Monaghan [25] begins his derivation of SPH form the mathematical identity

$$f(\mathbf{x}) = \int_D f(\mathbf{x}')\delta(\mathbf{x}-\mathbf{x}')d\mathbf{x}', \tag{2.11}$$

where f is any continuous scalar function defined over some n-dimensional vector space $\mathbb{R}^n$, such that $\mathbf{x} \in \mathbb{R}^n$, D is the domain of the scalar function and δ is the Dirac delta function, defined as

$$\delta(\mathbf{x}) = \begin{cases} \infty, & \mathbf{x} = \mathbf{0} \\ 0, & \mathbf{x} \neq \mathbf{0} \end{cases}, \qquad \int_{\mathbb{R}^n} \delta(\mathbf{x})d\mathbf{x} = 1. \tag{2.12}$$

As the Dirac delta function lends itself badly to discretization, Monaghan [25] uses a generalized continuous function W with the properties

$$\lim_{h\to 0} W(\mathbf{x},h) = \delta(\mathbf{x}), \quad \int_{\mathbb{R}^n} W(\mathbf{x},h)d\mathbf{x} = 1, \tag{2.13}$$

where the function W is called a smoothing kernel and h is known as its smoothing radius. Using Taylor series expansion it can then be shown that when the smoothing kernel is symmetric, meaning it is invariant to the direction of its first argument, then

$$f(\mathbf{x}) \approx \int_D f(\mathbf{x}')W(\mathbf{x}-\mathbf{x}',h)d\mathbf{x}'. \tag{2.14}$$

This integral can then be discretized to give an approximation of any scalar function value at any position within the SPH domain. As an integral can be discretized as a sum of products of volumes and their respective function values, this integral is discretized as

$$f(\mathbf{x}) \approx \sum_i V_i f_i W(\mathbf{x} - \mathbf{x}_i, h), \tag{2.15}$$

where the sum is over all particles, V_i is the volume of particle i, f_i is the scalar functions value at the position of particle i and $\mathbf{x}_i$ is the position of particle i.

Expressing the Smoothing Kernel The Dirac delta function δ can be said to be equal to a Gaussian distribution with zero variance, hence it would be reasonable to use a Gaussian distribution with a variance of h as the smoothing kernel. One should however note that, in order to not violate the second property of Equation (2.13), one has to integrate over the whole SPH domain. However, in practice, as the contribution from particles far away from the point of integration $\mathbf{x}$ is negligible, one would want to limit the required domain of integration in order to improve performance. The smoothing kernel is therefore typically expressed as a compact function which closely resembles a Gaussian distribution. Due to computations mentioned later in this chapter, it should be noted that the smoothing kernel is required to have a well defined continuous first derivative.

There are numerous choices for the smoothing kernel that adhere to the mentioned requirements, like the polynomial kernel used by previous work on the HPC-Lab Snow simulator [4], which is created specifically for high computational efficiency. This thesis does however use the more widely adopted cubic spline kernel as it is used by most of the relevant works on elastoplasticity. The cubic spline kernel is defined as

$$W(\mathbf{x}, h) = \sigma_n \begin{cases} 1 - \frac{3}{2}q^2 + \frac{3}{4}q^3, & 0 \le q \le 1 \\ \frac{1}{4}(2-q)^3, & 1 \le q \le 2 : \quad q = \dfrac{\|\mathbf{x}\|}{h}, \\ 0, & q \ge 2 \end{cases} \tag{2.16}$$

where σ_n is a normalization factor for the kernel.

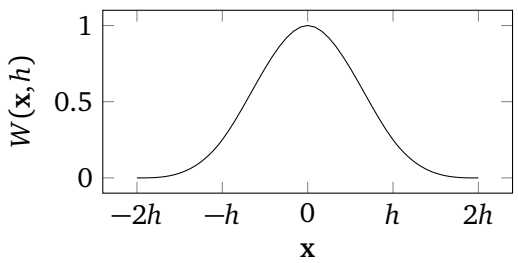


Figure 2.1: Cubic spline smoothing kernel with normalization factor set to one.

Normalization is required because the integral of the smoothing kernel over its whole domain is dependent on both the value of h and the dimensionality of the vector space where $\mathbf{x}$ is defined. Thus the normalization factor is set such that

$$\int_{\mathbb{R}^n} W(\mathbf{x},h)d\mathbf{x} = 1. \tag{2.17}$$

As the smoothing kernel is symmetric, the normalization factor for the smoothing kernel, over a two dimensional vector space, can be calculated as the reciprocal of the integral of the smoothing kernel over a circle with radius $2h$, which amounts to

$$\begin{aligned}\sigma_2 &= \left(\int_0^h 2\pi r\left(1-\frac{3}{2}\left(\frac{r}{h}\right)^2+\frac{3}{4}\left(\frac{r}{h}\right)^3\right)dr + \int_h^{2h} 2\pi r\left(\frac{1}{4}\left(2\frac{r}{h}\right)^3\right)dr\right)^{-1} \\ &= \frac{10}{7\pi h^2}.\end{aligned} \tag{2.18}$$

Similarly the normalization factor for the smoothing kernel over a three dimensional vector space can be calculated as the reciprocal of the integral of the smoothing kernel over a sphere with radius $2h$, which becomes

$$\begin{aligned}\sigma_3 &= \left(\int_0^h 4\pi r^2\left(1-\frac{3}{2}\left(\frac{r}{h}\right)^2+\frac{3}{4}\left(\frac{r}{h}\right)^3\right)dr + \int_h^{2h} 4\pi r^2\left(\frac{1}{4}\left(2-\frac{r}{h}\right)^3\right)dr\right)^{-1} \\ &= \frac{1}{\pi h^3}.\end{aligned} \tag{2.19}$$

Gradients of Scalar Fields Many uses of SPH requires computation of gradients at points within scalar fields. Given a scalar field f, its gradient at some position $\mathbf{x}$ can easily be extrapolated from Equation (2.15) as

$$\nabla f(\mathbf{x}) \approx \nabla\left(\sum_i V_i f_i W(\mathbf{x}-\mathbf{x}_i,h)\right) = \sum_i V_i f_i \nabla W(\mathbf{x}-\mathbf{x}_i,h). \tag{2.20}$$

This formulation depends on the gradient of the smoothing kernel, which comes up often enough in SPH computations that it should be explicitly expressed. As the smoothing kernel is symmetric, its gradient amounts to the gradient of the length of $\mathbf{x}$ multiplied with the derivative of the smoothing kernel with respect to the length of $\mathbf{x}$, which in turn equates to

$$\begin{aligned}\nabla W(\mathbf{x},h) &= \frac{d\|\mathbf{x}\|}{d\mathbf{x}}\frac{dW(\mathbf{x},h)}{d\|\mathbf{x}\|} \\ &= \frac{\mathbf{x}}{\|\mathbf{x}\|}\sigma_n' \begin{cases} \frac{9}{4}q^2 - 3q, & 0 < q \leq 1 \\ -\frac{3}{4}(2-q)^2, & 1 \leq q \leq 2 : \quad q = \dfrac{\|\mathbf{x}\|}{h} \\ 0, & q \geq 2 \end{cases}\end{aligned} \tag{2.21}$$

It should be noted that this kernel formulation is not defined when $\mathbf{x} = \mathbf{0}$.

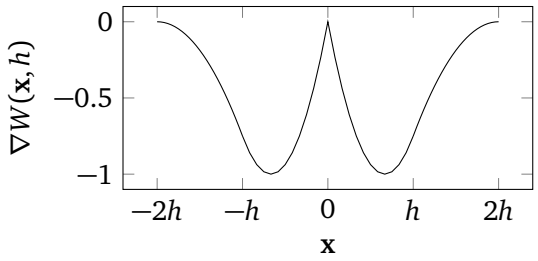


Figure 2.2: Cubic spline smoothing kernel gradient with normalization factor set to one.

Again there is a normalization factor σ_n' which is dependent on the dimensionality of the vector field. The normalization factor is computed by differentiating the smoothing kernels with their respective normalization factors, which gives

$$\sigma_2' = \frac{10}{7\pi h^3}, \tag{2.22}$$

$$\sigma_3' = \frac{1}{\pi h^4}. \tag{2.23}$$

Support Radius, Particle Neighborhood and Points of Interest The smoothing kernel as it is defined with cubic spline is, as mentioned above, a compact function. This has the property that any vector $\mathbf{x}$ with a length greater than some radius will amount to a kernel value of zero. For the cubic spline smoothing kernel, this radius is equal to twice the smoothing radius,

$$\hbar = 2h. \tag{2.24}$$

This radius $\hbar$ is called the support radius of the SPH, and should not be confused with the smoothing radius h. It should however be noted that for many other choices of smoothing kernels the support radius may be equal to the smoothing radius.

As any particle further than $\hbar$ away from some vector $\mathbf{x}$ will not contribute to the computation of field values at $\mathbf{x}$, Equation (2.15) can be rewritten with this in mind, which gives

$$f(\mathbf{x}) \approx \sum_{i:\|\mathbf{x}-\mathbf{x}_i\|\leq\hbar} V_i f_i W(\mathbf{x}-\mathbf{x}_i, h). \tag{2.25}$$

These particles within radius $\hbar$ are referred to as the particle neighborhood around $\mathbf{x}$. See Figure 2.3.

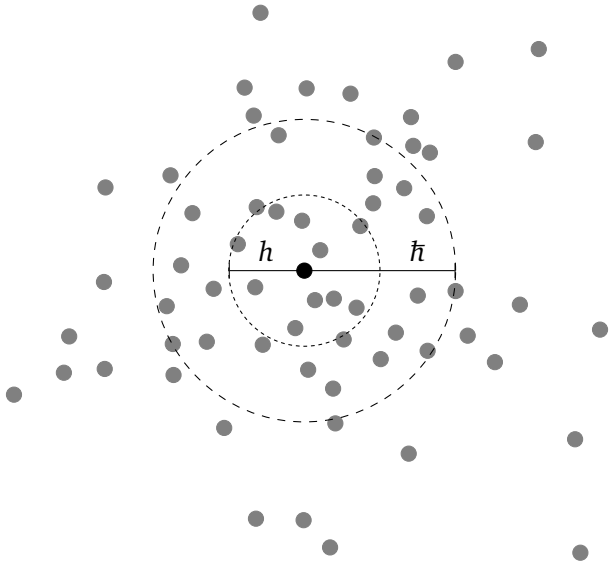


Figure 2.3: Example of particle neighborhood with radius equal to the support radius $\hbar$ which is twice the smoothing radius h.

Similarly, the kernel gradient at positions outside the neighborhood will be zero, in addition to this the kernel is not defined for the position zero, thus Equation (2.20) becomes

$$\nabla f(\mathbf{x}) \approx \sum_{i:0<\|\mathbf{x}-\mathbf{x}_i\|\leq\hbar} V_i f_i \nabla W(\mathbf{x}-\mathbf{x}_i, h). \tag{2.26}$$

Equation (2.25) shows approximation of f at the position $\mathbf{x}$. However, for most use cases of SPH, one is only interested in field values at particle positions. Thus, for the rest of this chapter, SPH discretization formulations will be rewritten to approximate the field value f_i at the position $\mathbf{x}_i$ of particle i. Furthermore will the summation over the neighborhood be rewritten as $\sum_j$, where j are particle neighbors of i, including i itself. Lastly, the smoothing kernel will be rewritten to only mention the respective particles whose positions are used, such that $W(\mathbf{x}_i - \mathbf{x}_j, h) = W_{ij}$. Rewriting Equation (2.25) with this in mind, it becomes

$$f_i \approx \sum_j V_j f_j W_{ij}. \tag{2.27}$$

The kernel gradient, Equation (2.26), can be rewritten in the same manner. Although, as the kernel gradient is undefined for zero, the particle i can not be included in its own neighborhood, this will however be excluded from the notation when summarizing over kernel gradients. Thus the kernel gradient is expressed as

$$\nabla f_i \approx \sum_{j:j\neq i} V_j f_j \nabla W_{ij} = \sum_j V_j f_j \nabla W_{ij}. \tag{2.28}$$

Differentiating Vector Fields The gradient of a vector field equates to a matrix where each row is equal to the gradient of the respective vector component, thus Equation (2.28) can be used to compute each row of the resulting matrix. Compacting this into one equation gives

$$\nabla \mathbf{f}_i \approx \sum_j V_j \mathbf{f}_j \otimes \nabla W_{ij}, \tag{2.29}$$

where $\mathbf{f}$ is an n-dimensional vector field and $\otimes$ is the outer product which can be expressed in matrix form as $\mathbf{v} \otimes \mathbf{u} = \mathbf{v}\mathbf{u}^\top$.

On the other hand, the divergence of a vector field equates to differentiating the vector field in each cardinal direction and adding the results. The derivative of the vector field along one of the cardinal directions can be computed with Equation (2.28) by using only the component of $\mathbf{f}$ and ∇W for the respective cardinal direction. Thus the divergence of a vector field can be expressed as

$$\nabla \cdot \mathbf{f}_i \approx \sum_j V_j \mathbf{f}_j \cdot \nabla W_{ij}. \tag{2.30}$$

Alternative Representations of The Differential Operators Although the previously mentioned approximations of differential operators are easily constructed, they are not always the appropriate choice. Hence many alternative formulations have been created. Monaghan [26] discussed the error arising from the use of Equation (2.28) and derives an alternative formulation from the use of the chain rule on gradients. By multiplying the argument of a function with one, the chain rule says that $\nabla f(1 \cdot \mathbf{x}) = 1 \cdot \nabla f(\mathbf{x}) + f(\mathbf{x}) \nabla 1$. This can further be extrapolated to give the alternative formulation

$$\begin{aligned} \nabla f_i &= \nabla f_i - f_i \nabla 1 \\ &\approx \sum_j V_j f_j \nabla W_{ij} - f_i \sum_j V_j \nabla W_{ij} \\ &\approx \sum_j V_j \left(f_i - f_j\right) \nabla W_{ij} \end{aligned}. \tag{2.31}$$

Using very similar logic [27], the differential operators for vector fields can be constructed as the alternative representations

$$\nabla \mathbf{f}_i \approx \sum_j V_j \left(\mathbf{f}_j - \mathbf{f}_i\right) \otimes \nabla W_{ij}, \tag{2.32}$$

$$\nabla \cdot \mathbf{f}_i \approx \sum_j V_j \left(\mathbf{f}_j - \mathbf{f}_i\right) \cdot \nabla W_{ij}. \tag{2.33}$$

Generalization of Gradients In certain situations, like when computing the gradient of pressure, even the aforementioned alternative formulation of gradient is inappropriate. However, one can generalize the method above by using the chain rule on the gradient of a scalar field times density to some power [25], which gives the generalized identity

$$\nabla(f\rho^n) = nf\rho^{n-1}\nabla\rho + \rho^n \nabla f. \tag{2.34}$$

When computing gradients of pressure, one typically prefers a pair-wise symmetric equation. Such an expression can be derived from Equation (2.34) by setting $n = -1$, which gives

$$\nabla f_i \approx \rho_i \sum_j \rho_j V_j \left(\frac{f_i}{\rho_i^2} + \frac{f_j}{\rho_j^2}\right) \nabla W_{ij}. \tag{2.35}$$

Correcting for Rotation A problem with the aforementioned representation of gradients of vector fields , Equation (2.32), is its inability to correctly handle rotations. Bonet and Lok [28] writes about how rotations within vector fields will only be preserved if the gradient of the particle position field is equal to the identity matrix, $\nabla \mathbf{x} = I$. In order to accommodate this fact, a correction matrix L_i, for the smoothing kernel at the position of particle i, is computed such that

$$\nabla \mathbf{x}_i = \sum_j V_j \left(\mathbf{x}_j - \mathbf{x}_i\right) \otimes L_i \nabla W_{ij} = I. \tag{2.36}$$

Reordering this formulation gives the expression for the correction matrix

$$L_i = \left(\sum_j V_j \left(\mathbf{x}_j - \mathbf{x}_i\right) \otimes \nabla W_{ij}\right)^{-1}. \tag{2.37}$$

The correction matrix can furthermore be used to construct a new smoothing kernel gradient which accommodates rotations in vector fields. This corrected kernel gradient is expressed as

$$\nabla \tilde{W}_{ij} = L_i \nabla W_{ij}. \tag{2.38}$$

It should however be noted that given certain configurations of the SPH particles, the resulting matrix $\nabla \mathbf{x}$ from the gradient of the particle positions, is not invertible. Thus the tensor field governed by the correction matrices will neither be well

defined nor continuous. Peer et al. [14] solves this problem by using the Moore-Penrose pseudoinverse in the cases where the matrix is not invertible.

Divergence of the Stress Tensor Field Section 2.4.2 discusses how the internal forces in an elastic material can be computed as the divergence of the stress tensor field σ. Bonet and Lok [28] explains how to derive a discretization of this divergence, which becomes

$$\nabla \boldsymbol{\sigma} = \sum_j V_i V_i \left(\boldsymbol{\sigma}_i \nabla \tilde{W}_{ij} - \boldsymbol{\sigma}_j \nabla \tilde{W}_{ji} \right). \tag{2.39}$$

As the forces due to stress not only contribute to linear force but also shear force, the rotationally corrected gradient kernel is used in this formulation.

2.3.2 Smoothed Particle Hydrodynamics in Practice

Although the previous section shows how Smoothed Particle Hydrodynamic discretizes various mathematical concepts, using SPH for numerical simulations may still pose a big challenge. Hence, this section goes through a general setup for numerical simulations of continuum medias.

Particle Spacing and Mass The spacing between SPH particles is entirely dependent on the material one wants to simulate and the simulation resolution one wants to achieve. The spacing between individual particles will also change throughout the simulation as the particles move around. However, for further computations, one needs a reference spacing. Therefore, a particle spacing Δs, is defined such that particles placed in a Cartesian grid with distance Δs between particles in the cardinal directions, see Figure 2.4, will be exactly at rest.

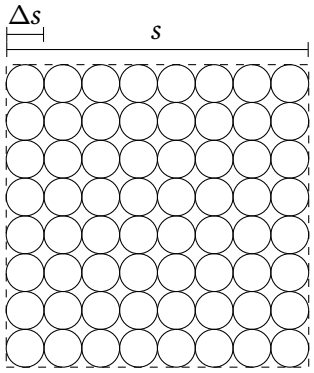


Figure 2.4: Particles placed in a square with spacing Δs and side length s.

Given a configuration where particles have been placed in a Cartesian grid such that they form an d-dimensional cube, also called a square in two dimensions,

with side length s, see Figure 2.4. These particles will take up a d-dimensional volume, or area, equal to s^d. As defined, these particles are at rest, and thus have a density equal to the rest density ρ_0 of their respective material. For instance, the rest density of water it approximately thousand kilograms, giving $\rho_0 = 1000$. Furthermore, density is defined as mass per volume, thus the mass M of the whole particles configuration can be expressed as

$$M = \frac{s^d}{\rho_0}. \tag{2.40}$$

As the distance between particles is Δs, each side of this d-dimensional cube will have $\frac{s}{\Delta s}$ particles. Thus the total number of particles n in this configuration will be

$$n = \left(\frac{s}{\Delta s}\right)^d. \tag{2.41}$$

Dividing the mass of the particle configuration among all the particles gives the particle mass m as

$$m = \frac{M}{n} = \frac{s^d \Delta s^d}{\rho_0 s^d} = \frac{\Delta s^d}{\rho_0}. \tag{2.42}$$

It should be noted that the mass of a particle is defined at the particles creation and stays constant throughout the simulation.

From the definition of density, one can also see that the rest volume V_0 of each particle is defined as

$$V_0 = \Delta s^d. \tag{2.43}$$

Support and Smoothing Radius Choosing the right value for the support radius is a crucial part of any use of SPH. Choosing a to small support radius will result in inaccurate approximations as there will be too few data points in each particles neighborhood. Choosing a too large support radius will however increase the computational cost of iterating over particle neighborhoods. Therefore one needs to find an optimal compromise between these factors. Most previous work on SPH [26] seems to agree that there is an optimal number N_d of particles within each particle neighborhood, which is dependent on the dimensionality d of the simulation domain. For two and three dimension, the agreed upon numbers are

$$N_2 \approx 20, \tag{2.44}$$

$$N_3 \approx 50. \tag{2.45}$$

The particle neighborhood around some point are all particles within a d-dimensional sphere with radius equal to the support radius $\hbar$. Thus, assuming all particles are at rest, the number of particles within the neighborhood can be

computed as the number of particle rest volumes that fit within the volume of the particle neighborhood, giving

$$N_d = \frac{\int_{\|\mathbf{x}\| \leq \hbar:\ \mathbf{x} \in \mathbb{R}^d} 1 d\mathbf{x}}{V_0}, \tag{2.46}$$

where the integral represent the volume of a d-dimensional sphere with radius $\hbar$.

Using the formula for the area of a circle and substituting in the definition of rest volume, Equation (2.43), the support radius in two dimensions becomes

$$\hbar = \Delta s \sqrt{\frac{N_2}{\pi}}. \tag{2.47}$$

Similarly, the formula for the volume of a sphere gives the support radius in three dimensions as

$$\hbar = \Delta s \sqrt[3]{\frac{3N_3}{4\pi}}. \tag{2.48}$$

As expressed in Equation (2.24), this thesis's choice of smoothing kernel gives the smoothing radius as

$$h = \frac{\hbar}{2}. \tag{2.49}$$

Computing Density and Volume While the mass of the particles stays constant throughout the simulation, the same can not be said for density and volume. The definition of density is mass per volume. However, using Equation (2.27), one can see that density can be computed independently of volume, giving

$$\rho_i = \sum_j \frac{m_j}{V_j} V_j W_{ij} = \sum_j m_j W_{ij}. \tag{2.50}$$

It should also be specified that the density ρ_i and the volume V_i are the density and volume of particle i, and should not be confused with the rest density and volume using the subscript zero.

Furthermore, the volume of a particle can be computed as

$$V_i = \frac{m_i}{\rho_i}. \tag{2.51}$$

Time Differentials, Forces and Integration Until this point, time has been left out of all formulations. Time is however an integral part of most SPH simulations. As time moves forward, the simulated continuum media moves around in space and thus the particles representing its discretization also moves around. Time will in this thesis be indicated by a superscript, thus the position of particle i at time t will be represented as

$$\mathbf{x}_i^t. \tag{2.52}$$

By differentiating the position of a particle with respect to time, one gets the velocity of the particle, expressed as

$$\mathbf{v}_i^t = \frac{d\mathbf{x}_i^t}{dt}. \tag{2.53}$$

Similarly, the velocity can be differentiated with respect to time to give the particle acceleration

$$\mathbf{a}_i^t = \frac{d\mathbf{v}_i^t}{dt}. \tag{2.54}$$

Acceleration of a particle can be computed from Newton's second law of motion. Given a force $\mathbf{F}_i^t$ acting on particle i at time t, its acceleration becomes

$$\mathbf{a}_i^t = \frac{\mathbf{F}_i^t}{m_i}. \tag{2.55}$$

When the SPH simulation advances in time, time dependent particle properties are integrated forward in time by a time step Δt. Various methods for integration exists, for instance the higher order leap-frog method used in previous SPH implementations on the HPC-Lab snow simulator [4]. However, due to the first order nature of the integration of the deformation gradient, explained in Section 2.4.2, there is no benefit to using any higher order method when simulating elastoplastic materials. This thesis chooses to use the same method as Gissler *et al.* [7], which is the Euler-Chromer method [29]. This method first integrates the velocity forward by one time step using the Euler method, giving the next time steps velocity

$$\mathbf{v}_i^{t+\Delta t} = \mathbf{v}_i^t + \Delta t \mathbf{a}_i^t. \tag{2.56}$$

Then the position is integrated forward by one time step using the newly acquired velocity, giving the new position

$$\mathbf{x}_i^{t+\Delta t} = \mathbf{x}_i^t + \Delta t \mathbf{v}_i^{t+\Delta t}. \tag{2.57}$$

2.3.3 Boundary Handling

Boundary handling tackles how SPH should handle the interaction between the simulated medium and external materials e.g. rigid bodies. Boundary handling is said to be the most difficult aspect of the SPH method [30], and a variety of methods has been developed for this.

Previous implementations of SPH in the HPC-Lab Snow Simulator [3] used a simple method for boundary handling where forces was explicitly added to particles near or within boundaries. This method is quite performant for simple boundaries, however it does come with a lot of disadvantages. More complex

boundary shapes are hard to implement accompanied by a high computational cost. This method also neglects the effect pressure variance has on the boundary interaction.

For this thesis it was instead chosen to implement boundary handling with the more widely used method of representing boundaries as a set of boundary particles. This method lends itself well to use with SPH as interactions with the boundary can use the same smoothing kernel and neighbor search as is used between other particles.

Boundary particles in general The general idea of boundary particles is to represent a volume that continuum material particles will not be able to move through. Thus, a boundary particle b has to represent some volume V_b.

Akinci *et al.* [12] derives an equation for the boundary volume by first assuming that the boundary particles has a theoretical mass m_b and a theoretical density ρ_b. Thus the volume of the boundary particle can be represented as $V_b = \frac{m_b}{\rho_b}$. By using Equation (2.50) for discretizing density and assuming that all boundary particles have the same theoretical mass, the volume discretization simplifies to

$$V_b = \frac{m_b}{\rho_b} = \frac{m_b}{\sum_k m_k W_{bk}} = \frac{1}{\sum_k W_{bk}}, \tag{2.58}$$

which is no longer dependant on the theoretical mass and density. As this equation shows, the volume of a boundary particle gets smaller as more particles are occupying the same space. This makes it such that overlapping boundary particles will cancel out and represent the same volume.

In order to compute densities of particles, one is required to know the mass of all particles interacting with the particle in question. Boundary particles does not have a specific mass, however, Akinci *et al.* [12] uses a method in which boundary particles acts as a continuum material particle at rest. Thus, a mass equivalent Ψ_b for a boundary particle b is computed as

$$\Psi_b = \rho_0 V_b. \tag{2.59}$$

Using this mass equivalent, Equation (2.50) for discretizing the density of particles, can be extended to incorporate boundary particles as

$$\rho_i = \sum_j m_j W_{ij} + \sum_k \Psi_k W_{ik}. \tag{2.60}$$

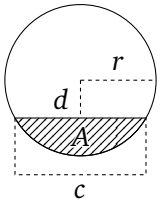
Note that boundary particles are represented as the sum over k, which is how they will be represented throughout this thesis.

Boundary Shells As mentioned above, boundary particles taking up the same space cancel out, thus one can easily fill complex structures with boundary particles without any problems. However, one typically wants to limit the number of boundary particles in order to limit their computational cost. Akinci *et al.* [12] and Band

et al. [13] thus only places a shell of boundary particles along the edge of the boundary volumes.

Although these boundary shells are only one particle thick, they represents a boundary volume completely filled with boundary particles. Therefore, their volumes has to account for these missing boundary particles.

Assuming the boundary shell is a perfectly flat line in a two dimensional simulation and a continuum material particle lies as close to the boundary as possible. The neighborhood of the continuum material particle will only contain a certain amount of boundary particles from the flat line, these boundary particles are however supposed to represent a whole minor segment, see Figure 2.5, of the circular neighborhood.



$$c = 2\sqrt{r^2 - d^2}$$

$$A = r^2 \cos^{-1}\tfrac{d}{r} - d\sqrt{r^2 - d^2}$$

Figure 2.5: Length of chord and area of minor segment distance d from circle center.

Using the same logic as in Section 2.3.2, the particle distance is Δs and the volume of a continuum material particle is Δs^2. Furthermore, the continuum material particle lies on top of the boundary particles, thus it has a distance of Δs from the line crossing the centers of the boundary particles, while it has a distance of $\frac{\Delta s}{2}$ from the actual boundary.

Thus the section of the line crossing the centers of the boundary particles which is contained within the particles neighborhood is equal to the chord, see Figure 2.5, distance Δs from the center of the neighborhood circle with radius $\hbar$. This chord has a length of c, see Figure 2.6, and a width of one particle distance, assuming it has the same volume as continuum material particles, its volume is $c\Delta s$.

The volume this boundary chord is supposed to represent is however the minor segment outside the chord with distance $\frac{\Delta s}{2}$ from the center of the particle neighborhood circle with radius $\hbar$, see Figure 2.6. This minor segment has a volume, actually area in two dimensions, of A.

Lastly the area of the chord of boundary particles is multiplied with a factor ϕ such that it equals the area of the minor segment, $\phi c\Delta s = A$. Using the formulas for the length of chords and area of minor segments, the factor in two dimensions becomes

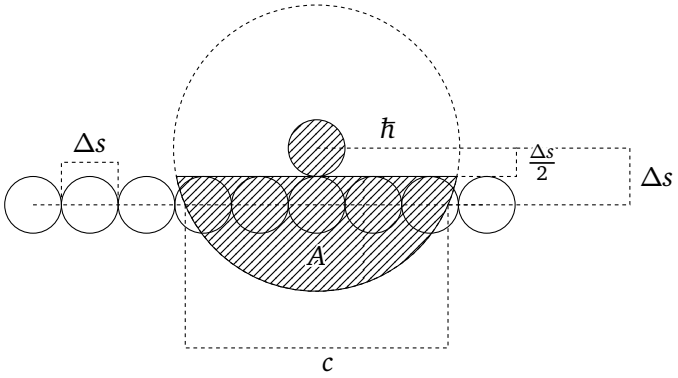


Figure 2.6: Boundary shell and the volume it represents.

$$\phi = \frac{\hbar^2 \cos^{-1} \frac{\Delta s}{2\hbar} - \frac{\Delta s}{2}\sqrt{\hbar^2 - \frac{\Delta s^2}{4}}}{2\Delta s\sqrt{\hbar^2 - \Delta s^2}}. \tag{2.61}$$

Multiplying the volume of a boundary particle, Equation (2.58), with this correction factor gives a new corrected formulation of the boundary particle volume as

$$V_b = \frac{\phi}{\sum_k W_{bk}}. \tag{2.62}$$

Boundary Shells in Three Dimensions Assuming the boundary shell is a perfectly flat plane in a three dimensional simulation and a continuum material particle lies as close to the boundary as possible. The neighborhood of the particle will only contain a certain area of boundary plane, which is however supposed to represent a spherical cap of the neighborhood sphere.

While in two dimensions a circle chord of the boundary line is contained within a particles neighborhood, in three dimensions a cross sectional disk with area A distance Δs from the neighborhoods sphere center is contained within the neighborhood. This disk has a width of one particle distance, thus, assuming the boundary particles have the same volume as continuum material particles, it will have a volume of $A\Delta s$. The area A_d of a cross sectional disk distance d from the center of a sphere with radius r can be computed as

$$A_d = \pi(r^2 - d^2). \tag{2.63}$$

In two dimensions the boundary particles within the neighborhood of a particle is supposed to represent the area of a minor segment. However, in three dimensions the boundary particles within the neighborhood of a particle is supposed to represent a spherical cap with volume V_c distance $\frac{\Delta s}{2}$ from the center of the neighborhood sphere. The volume $V_{c,d}$ of a spherical cap distance d for the center of a sphere with radius r can be computed as

$$V_{c,d} = \frac{\pi}{3}(r-d)^2(d+2r). \tag{2.64}$$

Thus, the volume of boundary particles can be multiplied by a factor ϕ such that the volume of the disk is equal to the volume of the spherical cap, $\phi A \Delta s = V_c$. This gives the formula for the volume correcting factor in three dimensions as

$$\phi = \frac{(2\hbar - \Delta s)^2(\Delta s + 4\hbar)}{24\Delta s(\hbar^2 - \Delta s^2)}. \tag{2.65}$$

Friction Friction is the idea that materials in contact with a surface experience a force in the opposite direction of their velocity. Viscosity is a property typically present in fluids, where different parts of the fluid experience a force which smooths out the velocity across the fluid such that the fluid tends to having less varying velocity. These two properties are very closely related and thus, in the case of continuum material particles interacting with boundary particles, the friction force on a particle is typically modeled as a viscous force between the particles.

The Navier-Stokes equations [8] is set of partial differential equations that are typically used when modeling fluids. This thesis will not go into details on these equations, however, they state that the acceleration due to viscous forces in a material can be expressed as

$$\mathbf{a} = \frac{\mu}{\rho}\nabla^2\mathbf{v}, \tag{2.66}$$

where μ is a factor stating how viscous the material in question is and $\nabla^2\mathbf{v}$ is the Laplacian of the velocity field, which is equal to the divergence of the gradient of the velocity field.

Various methods for discretizing the Laplacian of a vector field with SPH exists. Band *et al.* [31] states that in the case of friction, the discretization of the Laplacian has to take into account particles moving tangentially to the boundary. They thus discretize the Laplacian of the velocity of an SPH particle as

$$\nabla^2\mathbf{v}_i = 2(d+2)\sum_j V_j \frac{\mathbf{x}_i - \mathbf{x}_j}{\|\mathbf{x}_i - \mathbf{x}_j\|} \cdot \nabla W_{ij}(\mathbf{v}_i - \mathbf{v}_j), \tag{2.67}$$

where d is the dimensionality of the SPH domain. However, in the case of stationary boundaries, one can assume that $\mathbf{v}_j$ is always equal to zero. Thus the equation can be simplified by pulling the velocity $\mathbf{v}_i$ out of the summation. This assumption will be made throughout this thesis.

Using Equation (2.66) and Equation (2.67) in conjunction with Equation (2.56), one can construct an implicit formulation of the velocity of a continuum material particle after taking into account the viscous forces between it and boundary particles. This formulation becomes

$$\mathbf{v}_i^{t+\Delta t} = \mathbf{v}_i^t + \Delta t \mathbf{a}_{other,i} + \Delta t \mathbf{v}_i^{t+\Delta t} \frac{2(d+2)\mu}{\rho_i^t} \sum_k V_k \frac{\mathbf{x}_i^t - \mathbf{x}_k}{\|\mathbf{x}_i^t - \mathbf{x}_k\|} \cdot \nabla W_{ij}, \tag{2.68}$$

where, $\mathbf{a}_{other,i}$ are all other accelerations on the particle computed up until the addition of friction.

Although this is an implicit formulation, because the friction force on a particle is not dependent on other particles in the continuum material system, this implicit formulation can actually be solved explicitly as

$$\mathbf{v}_i^{t+\Delta t} = \frac{\mathbf{v}_i^t + \Delta t \mathbf{a}_{other,i}}{1 - \Delta t \frac{2(d+2)\mu}{\rho_i^t} \sum_k V_k \frac{\mathbf{x}_i^t - \mathbf{x}_k}{\|\mathbf{x}_i^t - \mathbf{x}_k\|} \cdot \nabla W_{ij}}. \tag{2.69}$$

Typically one is interested in the acceleration $\mathbf{a}_f$ due to friction as opposed to the velocity like this equation gives. The acceleration can thus be found by substituting in Equation (2.56), $\mathbf{v}_i^{t+\Delta t} = \mathbf{v}_i^t + \Delta t \mathbf{a}_i$, which gives

$$\mathbf{a}_{f,i} = \frac{\mathbf{v}_i^t + \Delta t \mathbf{a}_{other,i}}{\Delta t - \Delta t^2 \frac{2(d+2)\mu}{\rho_i^t} \sum_k V_k \frac{\mathbf{x}_i^t - \mathbf{x}_k}{\|\mathbf{x}_i^t - \mathbf{x}_k\|} \cdot \nabla W_{ij}} - \frac{\mathbf{v}_i^t + \Delta t \mathbf{a}_{other,i}}{\Delta t}, \tag{2.70}$$

where d is the dimensionality of the SPH domain and μ is a friction coefficient which states how much friction there should be between the continuum material and the boundary particles.

2.4 Elastoplastic Materials

Elastoplastic materials are materials which exhibit both elastic and plastic properties, see Figure 2.7. When the material is deformed slightly it will it will behave as an elastic material and return to its original form. However, when it is deformed beyond a certain threshold it behaves as a plastic material and keeps some of its deformation.

2.4.1 Generalized Hooke's Law for Elastic Materials

Elastic materials are materials which return to their original form after having been deformed. This operates similarly to how springs in one dimension return to their original form after being deformed. Springs follow Hooke's law

$$F = k\Delta x, \tag{2.71}$$

where a spring is compressed or stretched by a length Δx, the force F returns it to its original shape where k is a constant which depends on the springs strength.

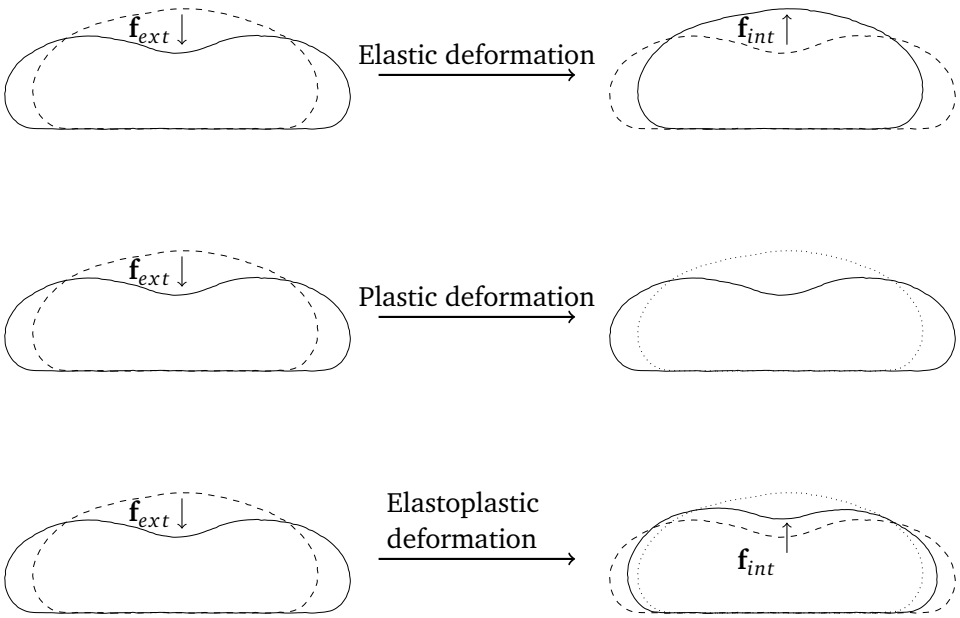


Figure 2.7: Comparison of materials deformed by an external force $\mathbf{f}_{ext}$ and their internal force $\mathbf{f}_{int}$ responding to the deformation.

When an elastic material is isotropic, independent of spacial direction, it can be thought of as if a set of infinitesimal small springs holding the material together and compressing or stretching depending on the materials deformation. However, these internal forces withing the material pull and bush in all possible directions. As an example, take a point within an elastic material, where the surrounding material is compressed along one axis and stretched along an orthogonal axis. This would result in the point experiencing two orthogonal forces, thus the forces at points within elastic materials can not be expressed as single forces.

Stress and Strain One way to understand internal forces in elastic materials is to think of an infinitesimal point mass with a corresponding infinitesimal volume. All forces acting on this point mass will thus act on its surface. In addition to this can any vector in a d-dimensional vector space can be linearly composed of d basis vectors. Thus, the internal forces acting on an infinitesimal point mass within a d-dimensional elastic material, can be uniquely expressed as an $d \times d$-matrix where each row is the force acting on a part of the surface of the point mass with a normal along one of the cardinal directions. Thus for a three dimensional elastic material, this can be expressed as

$$\boldsymbol{\sigma} = \begin{bmatrix} \frac{dF_x}{dS_x} & \frac{dF_y}{dS_x} & \frac{dF_z}{dS_x} \\ \frac{dF_x}{dS_y} & \frac{dF_y}{dS_y} & \frac{dF_z}{dS_y} \\ \frac{dF_x}{dS_z} & \frac{dF_y}{dS_z} & \frac{dF_z}{dS_z} \end{bmatrix} = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{bmatrix}, \tag{2.72}$$

where F_i is a scalar force acting in the cardinal direction i onto a surface, S_j is the area of the respective surface with normal in the cardinal direction j. The resulting matrix $\boldsymbol{\sigma}$ is called the stress in the material at the respective point. The reason for the differentiation, is due to the fact that this is the stress for an infinitesimal point mass, in other words the limit as the surface of the point mass gets infinitesimally small.

While stress represents the force of Hooke's law, Equation (2.71), one has to find an equivalent for the deformation Δx. Referring to the infinitesimal point mass volume from above, the cross-sectional length of the original volume along the cardinal direction i is expressed as U_i while the amount this length has been compressed or stretched is expressed as u_i. This deformation is called normal strain. One may also deform the volume by shearing it, which has the effect of deforming the material without changing the length of its cross-section. This can be thought of as deforming the length U_i in an orthogonal direction j, expressed as u_j. However this deformation alone would result in a rotation of the volume, hence a deformation in the direction i of the cross section along j is added to counteract this rotation. Thus, representing all deformations of a three dimensional volume results in the matrix

$$\boldsymbol{\epsilon} = \begin{bmatrix} \frac{du_x}{dU_x} & \frac{du_x}{2dU_y} + \frac{du_y}{2dU_x} & \frac{du_x}{2dU_z} + \frac{du_z}{2dU_x} \\ \frac{du_y}{2dU_x} + \frac{du_x}{2dU_y} & \frac{du_y}{dU_y} & \frac{du_y}{2dU_z} + \frac{du_z}{2dU_y} \\ \frac{du_z}{2dU_x} + \frac{du_x}{2dU_z} & \frac{du_z}{2dU_y} + \frac{du_y}{2dU_z} & \frac{du_z}{dU_z} \end{bmatrix} = \begin{bmatrix} \epsilon_{xx} & \epsilon_{xy} & \epsilon_{xz} \\ \epsilon_{yx} & \epsilon_{yy} & \epsilon_{yz} \\ \epsilon_{zx} & \epsilon_{zy} & \epsilon_{zz} \end{bmatrix}, \quad (2.73)$$

where the resulting matrix $\boldsymbol{\epsilon}$ is referred to as the strain of the material at the respective point. The reason for the division by two of the shear components is due to the symmetry of the matrix representing the same deformation twice.

With these stress and strain formulations Hooke's law, Equation (2.71), can be reformulated as the generalized Hooke's law for elastic materials

$$\boldsymbol{\sigma} = K\boldsymbol{\epsilon}, \quad (2.74)$$

where K is a constant matrix representing the strength of the material.

Young Modulus and Poisson's Ratio To use the generalized Hooke's law one has to express the strength matrix K in Equation (2.74). Isotropic elastic materials typically have different strengths depending on whether the experienced stress is normal stress or shear stress. How these strengths are defined can vary depending on usage, but for elastic materials their strengths are typically defined by their Young Modulus and their Poisson's Ratio.

The Young Modulus E, or the modulus of elasticity, is a constant describing the materials ability to withstand normal stress. It is defined as the relationship between the normal strain ϵ_{ii} and the resulting normal stress σ_{ii} for some direction i, the direction of this stress is irrelevant as the material is isotropic. The

young modulus for an elastic material in three dimensions can thus be expressed as

$$E = \frac{\sigma_{xx}}{\epsilon_{xx}} = \frac{\sigma_{yy}}{\epsilon_{yy}} = \frac{\sigma_{zz}}{\epsilon_{zz}}. \tag{2.75}$$

Poisson's Ratio ν is a constant describing how the material deforms perpendicularly to the applied force. One would for instance expect that when an elastic ball is pressed against the floor, it would be compressed heightwise while it would expand in all other directions, hence the balls Poisson's Ratio is positive.

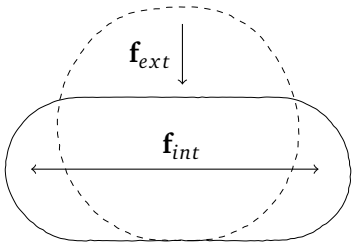


Figure 2.8: Ball pressed against the ground by an external force $\mathbf{f}_{ext}$ showing the internal forces $\mathbf{f}_{int}$ acting in the perpendicular directions due to the Poisson effect.

Poisson's Ratio is defined as the proportional relationship between the in normal strain in the direction of the acting force and the in normal strain in any of the perpendicular directions. Poisson's ratio for an elastic material in three dimensions can thus be expressed as

$$\nu = -\frac{\epsilon_{yy}}{\epsilon_{xx}} = -\frac{\epsilon_{zz}}{\epsilon_{xx}} = -\frac{\epsilon_{xx}}{\epsilon_{yy}} = -\frac{\epsilon_{zz}}{\epsilon_{yy}} = -\frac{\epsilon_{xx}}{\epsilon_{zz}} = -\frac{\epsilon_{yy}}{\epsilon_{zz}}. \tag{2.76}$$

Lamé Parameters Using the definitions of the Young Modulus ans Poisson's Ratio, a formulation of the strength in the Generalized Hooke's Law, Equation (2.74), can be derived. Deriving this formulation, for a tree dimensional material, begins by expressing the normal strains of the material.

Using Equation (2.75), one can see that given a normal stress in the x direction, the resulting strain in the same direction will become $\frac{\sigma_{xx}}{E}$. Using both Equation (2.75) and Equation (2.76), one can see that normal stress in the y direction induces the normal strain $-\frac{\nu\sigma_{yy}}{E}$ in the x direction. Similarly normal stress in the z direction induces the normal strain $-\frac{\nu\sigma_{zz}}{E}$ in the x direction. Similar logic can be used for the other cardinal direction and thus the normal strains can be expressed as

$$\begin{aligned}
\epsilon_{xx} &= \frac{\sigma_{xx}}{E} - \frac{\nu\sigma_{yy}}{E} - \frac{\nu\sigma_{zz}}{E} &&= \frac{(1+\nu)\sigma_{xx}}{E} - \frac{\nu(\sigma_{xx}+\sigma_{yy}+\sigma_{zz})}{E},\\
\epsilon_{yy} &= \frac{\sigma_{yy}}{E} - \frac{\nu\sigma_{xx}}{E} - \frac{\nu\sigma_{zz}}{E} &&= \frac{(1+\nu)\sigma_{yy}}{E} - \frac{\nu(\sigma_{xx}+\sigma_{yy}+\sigma_{zz})}{E},\\
\epsilon_{zz} &= \frac{\sigma_{zz}}{E} - \frac{\nu\sigma_{xx}}{E} - \frac{\nu\sigma_{yy}}{E} &&= \frac{(1+\nu)\sigma_{zz}}{E} - \frac{\nu(\sigma_{xx}+\sigma_{yy}+\sigma_{zz})}{E}.
\end{aligned} \tag{2.77}$$

By summarizing these normal strain definitions, one can derive an expression for the sum of the normal stresses as

$$\begin{aligned}
\epsilon_{xx}+\epsilon_{yy}+\epsilon_{zz} &= \frac{(1+\nu)(\sigma_{xx}+\sigma_{yy}+\sigma_{zz})}{E} - \frac{3\nu(\sigma_{xx}+\sigma_{yy}+\sigma_{zz})}{E}\\
&= \frac{1-2\nu}{E}(\sigma_{xx}+\sigma_{yy}+\sigma_{zz}),
\end{aligned} \tag{2.78}$$

$$\sigma_{xx}+\sigma_{yy}+\sigma_{zz} = \frac{E}{1-2\nu}(\epsilon_{xx}+\epsilon_{yy}+\epsilon_{zz}). \tag{2.79}$$

With this expression for the sum of the normal stresses, the Equations (2.77) can reformulated into equations for the normal stresses as

$$\begin{aligned}
\sigma_{xx} &= \frac{E}{1+\nu}\epsilon_{xx} + \frac{E\nu}{(1+\nu)(1-2\nu)}(\epsilon_{xx}+\epsilon_{yy}+\epsilon_{zz})\\
\sigma_{yy} &= \frac{E}{1+\nu}\epsilon_{yy} + \frac{E\nu}{(1+\nu)(1-2\nu)}(\epsilon_{xx}+\epsilon_{yy}+\epsilon_{zz})\\
\sigma_{zz} &= \frac{E}{1+\nu}\epsilon_{zz} + \frac{E\nu}{(1+\nu)(1-2\nu)}(\epsilon_{xx}+\epsilon_{yy}+\epsilon_{zz}).
\end{aligned} \tag{2.80}$$

The stress and strain independent parts of these equations can be extracted as

$$G = \frac{E}{2(1+\nu)}, \quad \lambda = \frac{E\nu}{(1+\nu)(1-2\nu)}, \tag{2.81}$$

where λ and G are called Lamé Parameters. Furthermore, by substituting in the Lamé Parameters and combining Equations (2.80) into a matrix form, the Generalized Hooke's Law, Equation (2.74), can be expressed as,

$$\boldsymbol{\sigma} = 2G\boldsymbol{\epsilon} + \lambda tr(\boldsymbol{\epsilon})I, \tag{2.82}$$

where $tr(\boldsymbol{\epsilon})$ is the trace of the strain matrix, or the sum of its diagonal, and I is the identity matrix. It should be noted that the above derivation does not account for shear tress and shear strain. It can however be shown that Equation (2.82) still holds for these components.

2.4.2 Solving for Internal Forces

In order to numerically simulate an elastic material, one is required to compute the forces that arise from the stress and strain within the material. This how this can be derived.

Elastic Deformation Gradient While the previously mentioned strain tensor, Equation (2.73), represents deformations in the material, it is preferable to represent deformations in a way that is easier to integrate and, as seen in Section 2.4.3, is able to handle plasticity. Thus, one defines the elastic deformation gradient F_E, which represents the amount a point within the material has deformed from its initial configuration.

Before defining the deformation gradient, one can define the displacement $\mathbf{u}$ of a material point, which is the amount a point has moved within the material from its original position within the material. It is defined as

$$\mathbf{u} = \mathbf{x} - \mathbf{x}_0 = \nabla_{\mathbf{x}_0}\mathbf{u} + I, \tag{2.83}$$

where $\mathbf{x}_0$ is the initial position of the point with respect to the material as a whole, thus $\mathbf{x}_0$ moves along as the whole material is moved or rotated, see Figure 2.9.

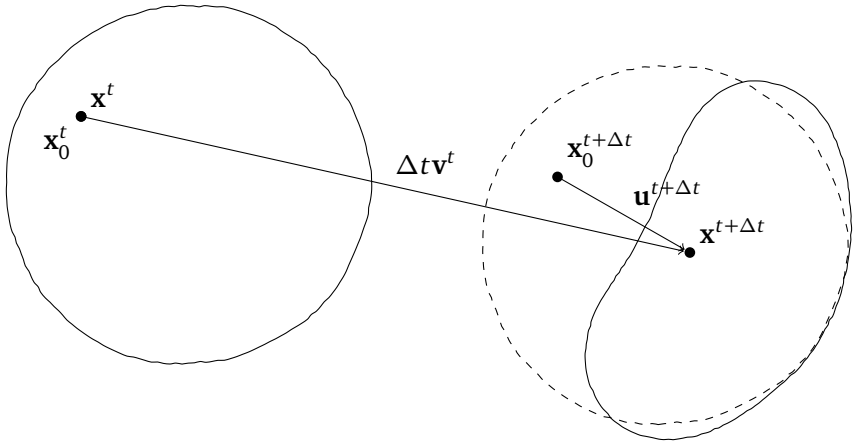


Figure 2.9: Displacement of a point within an elastic material.

The elastic deformation gradient F_E is defined as the gradient of the position of points within the deformed material with respect to their original position within the undeformed material,

$$F_E = \frac{\partial \mathbf{x}}{\partial \mathbf{x}_0} = \nabla_{\mathbf{x}_0}\mathbf{x}. \tag{2.84}$$

Furthermore, the time derivative of the deformation gradient can be expressed as

$$\frac{dF_E}{dt} = \frac{d\nabla_{\mathbf{x}_0}\mathbf{x}}{dt} = \nabla_{\mathbf{x}_0}\frac{d\mathbf{x}}{dt} = \nabla_{\mathbf{x}_0}\mathbf{v}, \tag{2.85}$$

where the time derivative of the position is equal to the velocity, as seen in Equation (2.53).

This formulation of the time derivative requires one to compute the gradient of velocity with respect to the initial shape of the material. This is however unfeasible in many situations, like in this thesis's implementation of SPH. One can however, as done by Sulsky *et al.* [32] for their particle in cell method, derive an alternative formulation of the time derivative. As the velocity of a point within the material is dependent on the position of the point, one can hence use the chain rule to express the time derivative of the deformation gradient as

$$\frac{dF_E}{dt} = \nabla_{\mathbf{x}_0}\mathbf{v} = (\nabla_{\mathbf{x}}\mathbf{v})(\nabla_{\mathbf{x}_0}\mathbf{x}) = \nabla\mathbf{v}F_E, \tag{2.86}$$

where $\nabla_{\mathbf{x}}\mathbf{v}$ is the gradient of the velocity with respect to the spacial position of material points. This is further replaced by the standard gradient operator as it is equivalent to the spacial gradient.

Using this formulation of the time derivative of the deformation gradient, an integration scheme for the deformation can be constructed as

$$F_E^{t+\Delta t} = F_E^t + \Delta t(\nabla\mathbf{v}^t)F_E^t. \tag{2.87}$$

Note that in the method by Peer *et al.* [14] for simulating elastic solids, they keep a reference to the initial particle configuration $\mathbf{x}_0$ throughout the simulation and use it to compute the deformation gradient. The methods by Stomakhin *et al.* [10] and Gissler *et al.* [7] does however not use any initial particle configuration, as the plasticity of snow makes it irrelevant, thus they instead solely rely on integration and velocity gradients to compute the deformation gradient.

Strain Since the elastic material as a whole is allowed to move freely around in space without experiencing any stress, any pure rotations of the material should not induce any strain. Given a pure rotation R, it will be canceled out when multiplying it with its transpose, as $RR^\top = R^\top R = I$. Therefore multiplying the deformation gradient with its transpose, $F_E F_E^\top$, will cancel out its rotational component. Substituting in the definition of deformation gradient given displacement, Equation (2.83), this becomes

$$F_E F_E^\top = (\nabla_{\mathbf{x}_0}u + I)(\nabla_{\mathbf{x}_0}u + I)^\top = \nabla_{\mathbf{x}_0}u + (\nabla_{\mathbf{x}_0}u)^\top + (\nabla_{\mathbf{x}_0}u)(\nabla_{\mathbf{x}_0}u)^\top + I, \tag{2.88}$$

which is typically referred to as the Cauchy-Green deformation tensor. Although this multiplication cancels out the rotational component of the deformation, it also clearly affects non-rotational deformations. One way to compensate for this is to use the Lagrangian finite strain tensor, defined as

$$\mathbf{E} = \frac{1}{2}(F_E F_E^\top - I) = \frac{1}{2}\left(\nabla_{\mathbf{x}_0} u + (\nabla_{\mathbf{x}_0} u)^\top + (\nabla_{\mathbf{x}_0} u)(\nabla_{\mathbf{x}_0} u)^\top\right). \tag{2.89}$$

This formulation is however mostly useful for larger deformations, while in a numerical simulation of continuum medias, where point masses tend toward infintessimal, all displacements are very small $\|\nabla_{\mathbf{x}_0} u\| \ll 1$. Because of this, the formulation can be simplified by assuming that $(\nabla_{\mathbf{x}_0} u)(\nabla_{\mathbf{x}_0} u)^\top \approx \mathbf{0}$. This simplified definition is called the infinitesimal strain tensor and is defined as

$$\epsilon = \frac{1}{2}\left(\nabla_{\mathbf{x}_0} u + (\nabla_{\mathbf{x}_0} u)^\top\right) = \frac{1}{2}\left(F_E + F_E^\top\right) - I. \tag{2.90}$$

Internal forces due to stress This thesis will not go into details on the derivation of forces due to stress, but will use the Cauchy momentum equation. The Cauchy momentum equation, Anderson [33], states that the internal forces due to stress acting on an infinitesimal volume in a material is equal to the divergence of the stress tensor for the same volume,

$$F_\sigma = \nabla \cdot \boldsymbol{\sigma} = \rho \mathbf{a}_\sigma. \tag{2.91}$$

This can further be transformed to give an equation for the acceleration of a point in a material due to internal stress as

$$\mathbf{a}_\sigma = \frac{1}{\rho}\nabla \cdot \boldsymbol{\sigma}. \tag{2.92}$$

This formulation can further be used to give an expression of the velocity at the next time step. In addition to the forces due to internal stress, every point in the material will also be affected by external forces like gravity, these external forces gives an acceleration $\mathbf{a}_{ext}$. Thus a formulation for the velocity at the next time step can be expressed as

$$\mathbf{v}^{t+\Delta t} = \mathbf{v}^t + \Delta t \mathbf{a}_{ext} + \frac{\Delta t}{\rho}\nabla \cdot \boldsymbol{\sigma}, \tag{2.93}$$

By substituting Equation (2.82), Equation (2.90) and Equation (2.87) into the version of Equation (2.93), the formulation becomes

$$\begin{aligned}\mathbf{v}^{t+\Delta t} = \mathbf{v}^t &+ \Delta t \mathbf{a}_{ext} \\ &+ \frac{G^t \Delta t}{\rho^t}\nabla \cdot \left(F_E^t + (F_E^t)^\top + \Delta t(\nabla \mathbf{v}^t)F_E^t + \Delta t((\nabla \mathbf{v}^t)F_E^t)^\top - 2I\right) \\ &+ \frac{\lambda^t \Delta t}{2\rho^t}\nabla \cdot tr\left(F_E^t + (F_E^t)^\top + \Delta t(\nabla \mathbf{v}^t)F_E^t + \Delta t((\nabla \mathbf{v}^t)F_E^t)^\top - 2I\right) I.\end{aligned} \tag{2.94}$$

Note how this formulation is explicit, Section 2.2.1. Expressing an impllicit formulation, Section 2.2.2, requires one to use the stress at time $\Delta t + t$, thus giving Equation (2.94) with $\nabla \mathbf{v}^t$ replaced by $\nabla \mathbf{v}^{t+\Delta t}$.

2.4.3 Handling Plasticity with Singular Value Decomposition

Elastoplastic materials exhibit plastic behaviors when experiencing stress beyond a certain threshold. More specifically, the elastic force opposing deformations in the material should be limited as to not fully return the material to its original shape when the stress is large enough. One way to do this is to limit the change of the deformation gradient. However, as with the formulation of strain, the material should be allowed freely rotate without experiencing stress. Hence any rotations in the material should be left out of the deformation gradient. To do this, the deformation gradient has to be decomposed into its rotational component and its scaling component. One way to do this is with Singular Value Decomposition [10], or SVD, which decomposes a matrix A into two pure rotation matrices U and $V^\top$ and a positive diagonal matrix Σ, such that

$$A = U\Sigma V^\top = U \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \\ 0 & 0 & \sigma_3 \end{bmatrix} V^\top, \tag{2.95}$$

where the diagonal values of Σ are the singular values of the decomposition. From this decomposition the rotational component of the matrix can be expressed as $UV^\top$, while the scaling component can be expressed as Σ.

Using this formulation of rotational and scaling components, the deformation gradient can be integrated leaving out the rotational component and limiting the amount of deformation.

First the elastic deformation gradient is integrated using Equation (2.87), giving

$$F_E^{t+\Delta t} = F_E^t + \Delta t(\nabla \mathbf{v}^t)F_E^t. \tag{2.96}$$

Note that in implicit formulatins one typically integrates the elastic deformation gradient using the velocity gradient $\nabla \mathbf{v}^{t+\Delta t}$.

Then this elastic deformation gradient is decomposed as

$$F_E^{t+\Delta t} = U_E\Sigma_E V_E^\top = U_E \begin{bmatrix} \sigma_{E,1} & 0 & 0 \\ 0 & \sigma_{E,2} & 0 \\ 0 & 0 & \sigma_{E,3} \end{bmatrix} V_E^\top, \tag{2.97}$$

where $U_E V_E^\top$ is the rotational component and Σ_E is the scaling component. The scaling component can be limited to a range between the critical compression and the critical stretch of the material, giving a new elastoplasic scaling component Σ_{EP} by clamping the singular values as

$$\Sigma_{EP} = \begin{bmatrix} \sigma_{EP,1} & 0 & 0 \\ 0 & \sigma_{EP,2} & 0 \\ 0 & 0 & \sigma_{EP,3} \end{bmatrix} : \quad \sigma_{EP,i} = clamp(\sigma_{E,i}, [1-\theta_c, 1+\theta_s]), \tag{2.98}$$

where θ_c is the critical compression of the material and θ_s is the critical stretch of the material.

Lastly the rotational component can be canceled out by multiplying with the transpose of the rotational component, giving the final integrated elastoplastic deformation gradient

$$F_{EP}^{t+\Delta t} = (U_E V_E^\top)^\top U_E \Sigma_{EP} V_E^\top = V_E \Sigma_{EP} V_E^\top. \tag{2.99}$$

One should note that the subscript of the elastoplastic deformation gradient F_{EP} is used in this section to clearify the its difference from the elastic deformation gradient F_E. However, throughtout this thesis, F_E will be used to represent all deformation gradients.

2.5 Graphics Processing Unit

Graphics processing units, or GPUs, or more commonly known as graphics cards, are a type of chip used in computers that are specifically designed to compute graphics related operations at a high speed. Historically, GPUs have been very specialized, thus only being useful for graphics related tasks. However, throughout the last two decades, GPUs have become widely more generalized and customizeable, thus enabling the use of GPUs on non-graphics related tasks, known as general purpose computing on GPUs, or GPGPU. With the widespread use of machine learning and neural networks the last decade, which benefits greatly from GPGPU, GPU vendors have been focusing even more on the GPGPU side of their products, specifically on tensor computations, which has initiated the development of tensor specific hardware like TPUs and CUDA Tensor Cores [34, 35].

2.5.1 Parallelism in General

With the end of Moore's Law [36] becoming a reality, the field of computing have seen more advancement into parallelism as a means of improving performance. This includes central processing units, CPUs, with more cores, CPUs with vectorized instructions and more use of GPGPU.

The parallelism seen in multi-core CPUs and in GPUs, operates quite differently and is therefore not always useful in the same situations. As an introduction to various methods of parallelism, this section will describe Flynn's Taxonomy [37], which is a scheme for classifying methods of computation in serial and in parallel. The class of Multiple Instructions, Single Data stream, MISD, will not be described as it it typically only used as a means of added security in highly critical systems.

Single Instruction, Single Data Stream Single Instruction, Single Data Stream, or SISD, is the workings of a typical single threaded processor. Programs that are only using SISD are called serial programs and is how the implementation

described in Section 4.1 is written. SISD works such that every clock tick, the processor performs one instruction on one set of data. An example if this it the x86 instruction `add`, which computes the sum of two integer values.

Multiple Instructions, Multiple Data Streams Multiple Instructions, Multiple Data Streams, or MIMD, is the typical workings of multi-core CPUs and programs that utilizes this are called multi threaded or parallel programs. In this case, every processor core works as its own independent SISD processor, the system as whole can thus perform multiple independent instructions on multiple sets of data. The advantage of this type of parallelism is that each processor core can perform its own independent job, such a dedicating one processor core to handling graphics and dedicating one core to physical simulations.

Single Instruction, Multiple Data Streams Single Instruction, Multiple Data Streams, or SIMD, is used by GPUs and by CPU vectorized instructions. With SIMD, the processing unit performs one instruction on multiple sets of data and will produce the same result as performing the equivalent SISD instruction multiple times. An example of SIMD is the x86 instruction `paddd`, which computes four sums of of two sets of four integer values, located consecutively in memory. `paddd` is equivalent to performing the `add` instruction four times, but only requiring one processor cycle. GPUs also works this way, such that they can perform one task multiple times on multiple sets of data. The advantage of this is that the GPUs hardware is specialized in such a way that many invocations of the same task can be performed simultaneously and thus result in a faster overall run time compared to performing the same task multiple times in series.

2.5.2 Compute Unified Device Architecture

Compute Unified Device Architecture [38], or CUDA, is Nvidia's application programming interface, API, for utilizing their GPUs for GPGPU computations. In general CUDA enables allocating to, and writing and reading from memory on the GPU and allows for defining functions in the code as GPU Kernels, which are sections of code that gets compiled to run on the GPU. In addition to this, CUDA has a system for executing these GPU Kernels from the code running on the CPU.

CUDA GPU Structure At the highest level, Nvidia GPUs contain a set of Graphics Processing Cluster, GPCs. The GPCs contain a set of Texture Processing Clusters, TPCs. The TPCs contain a set of Streaming Multi-processors, SMs. When kernels are executed they are first queued in the GPU's GigaThread Scheduler, which divides the threads needed for executing the kernel into blocks of at most 1024 threads and further decides which SM should handle each block. Each SM is further divided into SM Partitions, SPs, which each has the ability to execute at a set of threads called a warp in parallel. When the GigaThread Scheduler has decided which SMs should handle a kernel execution, the SMs themselves schedule warps

of their blocks amongst available SPs, which again schedule and issue warps for parallel execution. [39]

When executing kernels from the CPU on the GPU using CUDA, one configures a kernel grid, which is a specification of how threads should be divided into blocks. This grid configuration can be crucial for performance as GPUs contains caches at the TPC level, the SM Level and the SP Level. The GPU also has the ability to synchronize the execution of threads across blocks.

Cache, Shared Memory and Localization Caching is the idea that larger sections of memory can be loaded into a cache, which has a lower access time, such that future memory accesses withing the same section can be more performant. By localizing memory used by one thread, one can achieve great performance benefits as there is a higher chance of needed memory being cached, thus enabling lower access time for future memory uses. However, while memory localization and caching can improve performance, it is quite difficult to control, thus CUDA enables a more user controllable form of memory, called shared memory. Shared memory is the idea that one dedicates a section of the SM-wide data cache for the programmer to explicitly utilize. Thus shared memory is a highly performant memory which can be accessed by all threads across a single block.

Chapter 3

Smoothed Particle Hydrodynamics for Snow Simulation

This chapter describes how smoothed particle hydrodynamics can be utilized in order to simulate snow. It describes both certain aspects of how one would structure a general program for performing these simulations and the mathematical formulas that are required.

The chapter first describes a method that is almost identical to the method by Gissler *et al.* [7], before describing certain alternative formulations of similar methods.

3.1 An Implicit Compressible SPH Solver for Snow Simulation

This thesis is for the most part inspired entirely by the paper for an implicit compressible SPH solver for snow simulation by Gissler *et al.* [7]. Thus this chapter begins by explaining this thesis's way of implementing their method.

3.1.1 Overview

This method start off by initializing all aspects of the simulation. This includes simulation wide parameters, snow particle properties and boundary particle properties as can be seen in Algorithm 3.1.

After having initialized the simulation the simulation is run by iteratively computing the properties of the snow particles at the next time step of the simulation. During this computation, first properties independent of other particles are computed. Then the correction matrix, Equation (2.37), and accelerations $\mathbf{a}_{ext}$ due to external forces are computed for each particle. Then two separate solvers are

Algorithm 3.1: Initializing the simulation.

```
function initializeSimulation
    initialize simulation parameters
    for all snow particles i
        initialize time independent properties of i
    for all boundary particles k
        initialize all properties of k
```

used to solve for the accelerations $\mathbf{a}_\lambda$ due to pressure differences and the accelerations $\mathbf{a}_G$ due to shear stress. Finally the velocities, deformation gradients and positions are numerically integrated for each particle. The integration of the deformation gradient is however dependent on the positions of all particles and the velocity gradient at each particle, thus the integrations have to be performed separately for all particles and in their respective order. The general structure of the computation of new time steps can be seen in Algorithm 3.2.

Algorithm 3.2: Computing the next time step.

```
function computeNextTimeStep
    for all snow particles i
        compute kernel independent properties of i
    for all snow particles i
        compute L_i^t
        compute a_{ext,i}
    solve for a_λ
    solve for a_G
    for all snow particles i
        integrate v_i^t
    for all snow particles i
        compute ∇v_i^{t+Δt}
        integrate F_{E,i}^t
    for all snow particles i
        integrate x_i^t
```

3.1.2 Initialization

Initializing Simulation Parameters Most of the simulation parameters are user defined and thus need no further initialization. A couple parameters does however have be computed. All snow particles are initialized to have the same mass, therefore a simulation wide particle mass is computed from the simulation wide rest density and particle spacing using Equation (2.42) with d set to the respective dimensionality of the simulation. Lastly the support radius and smoothing radius for the smoothing kernel used by the simulation is computed using Equation (2.47) or Equation (2.48), depending on the dimensionality of the simulation, and Equation (2.49), respectively.

Initializing Snow Particle Properties Most snow particle properties are recomputed each time step and thus does not need to be initialized, but they are typically initialized to zero in the implementations. As mentioned above, the particle mass is initialized to the as the simulation wide particle mass. The other properties that has to be initialized are the ones that are integrated every time step. The deformation gradients are initialized to the identity matrix as the snow particles are assumed to be at rest at the beginning of the simulation. The positions and velocities are user defined depending on the initial particle configuration.

Initializing Boundary Particles The boundary particle properties V_b and Ψ_b, for all boundary particles b are initialized as described in Section 2.3.3. The boundary particle volumes V_b are computed using Equation (2.62), where the volume correction factor ϕ is computed using Equation (2.61) or Equation (2.65) depending on the dimensionality of the simulation. The boundary particle mass equivalents Ψ_b are computed using Equation (2.59).

This way of handling boundary particles is not the same method used by Gissler *et al.* [7]. They do however not explicitly state how they configure their boundaries, which makes it difficult to know how they made their method work. This method should however result in a boundary contribution that is approximately equal to their method.

3.1.3 Snow Particle Properties

Density, Volume Rest Density The density and volume of each particle is simply computed by Equation (2.60) and Equation (2.51), respectively.

As mentioned in Section 2.1.2, snow hardens as it is compressed, making it act more like a solid and keep its compression, thus its rest density consequently gets larger. As the deformation gradient of the snow particles handles the amount of compression experienced by the snow, its determinant represents how much each particle is compressed in reference to its rest density. Thus this method computes a new rest density for each particle i as

$$\rho_{0,i}^t = \rho_i^t |det(F_{E,i}^t)|, \tag{3.1}$$

where $|det(F_{E,i}^t)|$ is the absolute value of the determinant of the deformation gradient matrix of the respective snow particle.

Lamé Parameters The initial Lamé Parameters G_0 and λ_0 are defined by Equation (2.81), where the Young Modulus and Poisson Ratio are user defined depending on the type of snow one wants to simulate. However, as snow hardens its Young Modulus and Poisson Ratio changes. The method by Stomakhin *et al.* [10] handles this change as an exponential growth depending on a plastic hardening coefficient ξ and the amount of plastic deformation. Contrary to this method, the

method by Gissler *et al.* [7] does not use a dedicated plastic deformation gradient, thus they compute the amount of plastic deformation from the particles rest density, giving the particles Lamé Parameters as

$$G_i^t = G_0 e^{\xi \frac{\rho_{0,i}^t - \rho_0}{\rho_{0,i}^t}}, \tag{3.2}$$

$$\lambda_i^t = \lambda_0 e^{\xi \frac{\rho_{0,i}^t - \rho_0}{\rho_{0,i}^t}}, \tag{3.3}$$

where $\rho_{0,i}$ is the respective particles rest density, while ρ_0 is the simulation wide rest density.

Correction Matrix As mentioned in Section 2.3.1, to properly handle gradients and divergence in rotating vector fields, one has to compute a correction matrix. This correction matrix L_i^t is computed for every particle using Equation (2.37). One can see that this equation is dependent on the volume of all neighbor particles, thus the correction matrix is computed after all particle volumes have been computed, as can be seen in Algorithm 3.2.

3.1.4 Acceleration due to External Forces

Gravity This implementation assumes that the gravity $\mathbf{g}$ is constant across the simulation. The gravity is a user defined vector, but is typically set to earths gravity of -9.81 in the vertical direction.

Friction The acceleration due to friction between snow and boundary particles is computed using Equation (2.70), where the acceleration $\mathbf{a}_{other}$ is set to the acceleration $\mathbf{g}$ of gravity.

Combined External Forces Before the various internal forces in the snow are computed, the acceleration due to external forces are combined into one acceleration

$$\mathbf{a}_{ext,i}^t = \mathbf{g} + \mathbf{a}_{f,i}^t. \tag{3.4}$$

It should be mentioned that Gissler *et al.* [7] includes an acceleration due to adhesion as derived by Akinci *et al.* [40]. This was implemented during the work on this thesis, but was left out due to its low relevance to the types of simulations this thesis focuses on.

3.1.5 Solving for Acceleration due to Pressure Differences

While elastoplastic materials relies on stress for computing their internal forces, a lot of other SPH simulations, especially fluid simulations, relies on pressure for in-

ternal forces. In such simulations, the simulated medium will experience internal forces from areas with high pressure onto areas with lower pressure.

One of the goals of Gissler *et al.* [7] was to create a method for snow simulation that could interact with other SPH simulations, thus they create a solver which solves for accelerations arising from pressure differences in the snow. This solver can further be interlaced with other SPH solvers to compute normal forces between various materials. Only using this solver would amount to simulating an inviscid fluid, like a gas, and would mathematically equate to setting Lamé's second parameter G to zero, thus solving a simplified generalized Hooke's law, Equation (2.82), as $\boldsymbol{\sigma} = \lambda tr(\boldsymbol{\epsilon})I$.

Pressure Pressure is a scalar value defined as force per area, $p = \frac{F}{A}$. This is quite similar to the definition of normal stress in elastic materials, Equation (2.72). One can therefore say that pressure is a scalar value describing a symmetric normal stress in all directions. Using a derivation similar to Equation (2.92), one can derive that the acceleration $\mathbf{a}_\lambda$ due to pressure difference in the material is equal to gradient from high pressure to low pressure divided by density, giving

$$\mathbf{a}_{\lambda,i} = -\frac{\nabla p_i}{\rho_i}. \tag{3.5}$$

By intuition one can see that a material at rest should have zero pressure, while a material with density higher than its rest density should have positive pressure. Thus, pressure is typically expressed as a proportional relationship between the amount the materials density differs from its rest density

$$p_i = k\frac{\rho_i}{\rho_{0,i}}. \tag{3.6}$$

This equation is called an equation of state [27], where k is called the materials stiffness constant. It should be noted that most equations of state are expressed in a way that keeps the pressure positive, Gissler *et al.* [7] does however allow negative pressure.

Conservation of Mass It is widely known that in physics, energy has to be conserved. This law is commonly referred to as the Law of Conservation of Mass-Energy, on account of, when one disregard chemical reactions and sub-atomic effects, the law states that mass has to be conserved. This law thus relates to continuum materials in the sense that mass within the material can not disappear nor come into existence. This fact is modeled simply by SPH as all particles have constant mass. However, for use in the following derivation, this law is mathematically expressed as [8]

$$\frac{d\rho_i}{dt} + \rho_i \nabla \cdot \mathbf{v}_i = 0, \tag{3.7}$$

which states that mass, referred to by the particles density, moving along the velocity field into some point, should be equal to the mass moving along the velocity field out of that same point. It should noted that the derivative in the equation is typically expressed as a material derivative $\frac{D\rho}{Dt}$. The use of SPH particles does however make this distinction irrelevant.

Deriving an Implicit System for Pressure Gissler *et al.* [7] uses the formula for acceleration due to pressure differences (3.5), in conjunction with the already computed external acceleration, to derive an implicit formulation of the velocity at the next time step as

$$\mathbf{v}_i^{t+\Delta t} = \mathbf{v}_i^t + \Delta t \mathbf{a}_{ext,i}^t - \Delta t \frac{\nabla p_i^{t+\Delta t}}{\rho_i^{t+\Delta t}}, \tag{3.8}$$

where velocity changes are integrated using the Euler method (2.56).

Furthermore, Gissler *et al.* [7] expresses the mass conservation law using this implicit formulation of velocity (**??**), by first approximating

$$\rho_i^{t+\Delta t} \nabla \cdot \left(\frac{\nabla p_i^{t+\Delta t}}{\rho_i^{t+\Delta t}} \right) \approx \nabla^2 p_i^{t+\Delta t}, \tag{3.9}$$

where ∇^2 is the Laplacian operator, which is equivalent to taking the divergence of the gradient. Using this approximation, the divergence term on the left hand side of the mass conservation law (3.7) can be expressed as

$$\begin{aligned} \rho_i^{t+\Delta t} \nabla \cdot \mathbf{v}_i^{t+\Delta t} &= \rho_i^{t+\Delta t} \nabla \cdot \left(\mathbf{v}_i^t + \Delta t \mathbf{a}_{ext,i}^t - \Delta t \frac{\nabla p_i^{t+\Delta t}}{\rho_i^{t+\Delta t}} \right) \\ &= \rho_i^{t+\Delta t} \nabla \cdot \left(\mathbf{v}_i^t + \Delta t \mathbf{a}_{ext,i}^t \right) - \Delta t \nabla^2 p_i^{t+\Delta t}. \end{aligned} \tag{3.10}$$

Then the time derivative of density can be expressed using the Euler method (2.56), giving

$$\frac{d\rho_i^{t+\Delta t}}{dt} \approx \frac{\rho_i^{t+\Delta t} - \rho_i^t}{\Delta t}. \tag{3.11}$$

Substituting Equation (3.10) and Equation (3.11) into the mass conservation law (3.7) and reordering the equation, gives the equation

$$\rho_i^{t+\Delta t} - \Delta t^2 \nabla^2 p_i^{t+\Delta t} = \rho_i^t - \Delta t \rho_i^{t+\Delta t} \nabla \cdot \left(\mathbf{v}_i^t + \Delta t \mathbf{a}_{ext,i}^t \right). \tag{3.12}$$

Thereafter Gissler *et al.* [7] uses Lamé's first parameter λ as the stiffness of the material. Thus the equation of state (3.6) can be used to express density as

$$\rho_i^{t+\Delta t} = \rho_{0,i}^{t+\Delta t} \left(\frac{p_i^{t+\Delta t}}{\lambda_i^{t+\Delta t}} + 1 \right). \tag{3.13}$$

Lastly, they use the following approximations $\rho_{0,i}^{t+\Delta t} \approx \rho_{0,i}^{t}$, $\lambda_i^{t+\Delta t} \approx \lambda_i^t$ and $\rho_i^{t+\Delta t}\nabla\cdot\left(\mathbf{v}_i^t + \Delta t\mathbf{a}_{ext,i}^t\right) \approx \rho_i^t\nabla\cdot\left(\mathbf{v}_i^t + \Delta t\mathbf{a}_{ext,i}^t\right)$, which turns Equation (3.12) into the final implicit pressure equation, where $p_i^{t+\Delta t}$ is the only unknown parameter,

$$\begin{aligned}\rho_{0,i}^t\left(\frac{p_i^{t+\Delta t}}{\lambda_i^t}+1\right) - \Delta t^2\nabla^2 p_i^{t+\Delta t} &= \rho_i^t - \Delta t\rho_i^t\nabla\cdot\left(\mathbf{v}_i^t + \Delta t\mathbf{a}_{ext,i}^t\right)\\ -\frac{\rho_{0,i}^t}{\lambda_i^t}p_i^{t+\Delta t} + \Delta t^2\nabla^2 p_i^{t+\Delta t} &= \rho_{0,i}^t - \rho_i^t + \Delta t\rho_{0,i}^t\nabla\cdot\left(\mathbf{v}_i^t + \Delta t\mathbf{a}_{ext,i}^t\right).\end{aligned} \tag{3.14}$$

Expressing the Implicit Pressure Equation as a System of Equations The implicit pressure equation (3.14) is essentially a system of equations, $A\mathbf{x} = \mathbf{b}$, where $\mathbf{x}$ is a vector of all particle pressures and b is a vector of the right hand side of the equation for all particles, but expressing the matrix A is not feasible. However, as Gissler *et al.* [7] have decided to use the Jacobi method (2.5), one only has to compute the diagonal elements of the theoretical matrix and the matrix-vector product between the matrix, without its diagonal, and the pressure vector. They further simplify this by approximating the aforementioned matrix-vector product as the product between the whole matrix, including its diagonal, and the pressure vector. The matrix-vector product thus amounts to computing the left hand side of Equation (3.14).

Discretizing the Implicit Pressure Equation Before solving the system, one needs a way of computing the differentials appearing in Equation (3.14). The right hand side requires one to compute the divergence of a vector field, while the left hand side requires one to compute the Laplacian of the pressure field. As stated before, the Laplacian is equivalent to taking the divergence of the gradient. Therefore, one is required to compute divergence of vector fields and gradients of scalar fields. The divergence of a vector field can be computed using Equation (2.33). The gradient of the pressure field can be computed using Equation (2.35). However, Gissler *et al.* [7] simplifies this equation by assuming particles have the same density. Finally the boundary contribution to the pressure gradient is added, which gives the equation

$$\nabla p_i = \sum_j V_j\left(p_i + p_j\right)\nabla W_{ij} + p_i\sum_k V_k\nabla W_{ib}. \tag{3.15}$$

The last requirement before solving the system is to compute the diagonal elements a_{ii} of the system matrix. This thesis will not go into details on how this is derived, but Gissler *et al.* [7] computes it as

$$\begin{aligned} a_{ii} = &-\frac{\rho_{0,i}^t}{\lambda_i^t} - \Delta t^2 \sum_j V_i V_j \|\nabla W_{ij}\|^2 \\ &- \Delta t^2 \left(\sum_j V_j \nabla W_{ij} + \sum_k V_k \nabla W_{ik} \right) \cdot \sum_k V_k \nabla W_{ik}. \end{aligned} \tag{3.16}$$

It should be noted that in both Equation (3.15) and Equation (3.16), Gissler *et al.* [7] multiplies the boundary terms by a constant factor. This is however not needed due to the choice of boundary volume computation in this thesis.

Solving the Implicit Pressure System To solve for the pressure, Gissler *et al.* [7] uses the Jacobi Method (2.5). However, as they want their system to interact with other SPH simulations, they use a under relaxed (2.10) version of the Jacobi method, such that this solver will be slow enough that other solvers can be interlaced between its iterations. They use the relaxation factor $\omega = 0.5$. This thesis does however not interact the snow with other materials and thus does not require relaxation. The method starts off by computing properties that are independent of pressure, which includes the system matrix diagonals and the right hand side of Equation (3.14), before iteratively updating the pressure to some chosen tolerance and lastly computing the resulting acceleration using Equation (3.5). It should also be noted that, at the first time step, all pressures are initialized to zero, while for all proceeding time steps the pressure from the preceding times step is kept. In addition to this the time notation of the pressure is left out as the same particle property is used across time steps. The method can be seen in Algorithm 3.3.

Algorithm 3.3: Implicit pressure solver.

```
function pressureSolver(ω, ϵ, i_max)
    for all snow particles i
        compute ∇·(v_i^t + Δt a_ext,i^t)
    for all snow particles i
        b_i = right hand side of Equation (3.14)
        compute a_ii
    i = 0
    e = ∞
    while e² > ϵ² and i < i_max
        e = 0
        for all snow particles i
            compute ∇p_i
        for all snow particles i
            Ap_i = left hand side of Equation (3.14)
            p_i = p_i + (ω/a_ii)(b_i − Ap_i)
            e += b_i − Ap_i
        i += 1
    for all snow particles i
        compute ∇p_i
        a_λ,i = ∇p_i
```

3.1.6 Solving for Acceleration due to Shear Stress

The acceleration $\mathbf{a}_\lambda$ due to pressure differences, computed in the previous section, essentially captures acceleration due to normal stress. In other words, it handles the λ-term of the generalized Hooke's law (2.82). The next step is to solve for the acceleration $\mathbf{a}_G$ due to shear stress, arising from the G-term of the generalized Hooke's law (2.82). It should be noted that, while this will be referred to as acceleration due to shear stress, the formulation still encompasses some acceleration due to normal stress, which will be further discussed in Section 3.2.1.

Deriving an Implicit System for Acceleration due to Shear Stress To derive an implicit system for $\mathbf{a}_G$, the generalized Hooke's law (2.82) is first simplified by removing the λ-term, giving the new shear stress formulation

$$\boldsymbol{\sigma}_{G,i}^{t+\Delta t} = 2G^t \boldsymbol{\epsilon}_i^{t+\Delta t}. \tag{3.17}$$

Using the same method as used used in Section 2.4.2, in addition to assuming the acceleration $\mathbf{a}_\lambda$ is an acceleration due to external forces, an implicit velocity equation can be derived, which becomes

$$\begin{aligned}\mathbf{v}_i^{t+\Delta t} = \mathbf{v}_i^t &+ \Delta t\left(\mathbf{a}_{ext,i} + \mathbf{a}_{\lambda,i}\right) \\ &+ \frac{\Delta t}{\rho_i^t}\nabla\cdot\left(G_i^t\left(F_{E,i}^t + (F_{E,i}^t)^\top + \Delta t(\nabla\mathbf{v}_i^{t+\Delta t})F_{E,i}^t + \Delta t((\nabla\mathbf{v}_i^{t+\Delta t})F_{E,i}^t)^\top - 2I\right)\right),\end{aligned} \tag{3.18}$$

where the Lamé parameter G is included in the divergence as it may be varying across the material. This equation can be reordered such that the unknown velocity only appears on the left hand side of the equation

$$\begin{aligned}&\mathbf{v}_i^{t+\Delta t} - \frac{\Delta t^2}{\rho_i^t}\nabla\cdot\left(G_i^t\left((\nabla\mathbf{v}_i^{t+\Delta t})F_{E,i}^t + ((\nabla\mathbf{v}_i^{t+\Delta t})F_{E,i}^t)^\top\right)\right) \\ &= \mathbf{v}_i^t + \Delta t\left(\mathbf{a}_{ext,i} + \mathbf{a}_{\lambda,i}\right) + \frac{\Delta t}{\rho_i^t}\nabla\cdot\left(G_i^t\left(F_{E,i}^t + (F_{E,i}^t)^\top - 2I\right)\right).\end{aligned} \tag{3.19}$$

By substituting Equation (2.56), $\mathbf{v}_i^{t+\Delta t} = \mathbf{v}_i^t + \Delta t\mathbf{a}_i^{t+\Delta t}$, into this equation and moving all constants over to the right hand side, one arrives at the following formulation of implicit acceleration.

$$\begin{aligned}&\mathbf{a}_{G,i}^{t+\Delta t} - \frac{\Delta t^2}{\rho_i^t}\nabla\cdot\left(G_i^t\left((\nabla\mathbf{a}_{G,i}^{t+\Delta t})F_{E,i}^t + ((\nabla\mathbf{a}_{G,i}^{t+\Delta t})F_{E,i}^t)^\top\right)\right) \\ &= \frac{1}{\rho_i^t}\nabla\cdot\left(G_i^t\left(F_{E,i}^* + (F_{E,i}^*)^\top - 2I\right)\right),\end{aligned} \tag{3.20}$$

where $F_{E,i}^*$ is a new deformation gradient encompassing the current velocity and external acceleration. It is computed as

$$F^*_{E,i} = F^t_{E,i} + \Delta t \nabla \left(\mathbf{v}^t_i + \Delta t \left(\mathbf{a}_{ext,i} + \mathbf{a}_{\lambda,i} \right) \right) F^t_{E,i}. \tag{3.21}$$

A Side Note on Mistakes in Papers In version five, which seems to be the latest version, of the paper by Gissler *et al.* [7], they have mistakenly written Δt instead of Δt^2 on the left hand side of Equation (3.20). This mistake was not found until the last couple of weeks of writing this thesis and is the main reason this theses delves into alternative solvers, Section 3.2.

Rotation and Volume Corrected Gradients Solving the above equation (3.20) requires one to compute the gradient of vector fields. As explained in Section 2.3.1, computing vector field gradients in rotating materials requires the use of the corrected kernel gradient, Equation (2.38), to avoid divergence over time. However, Gissler *et al.* [7] found that this gradient formulation over estimates the volume changes. Hence they compute a new velocity gradient, which takes the gradient component due to volume changes from the uncorrected gradient and the gradient components due to rotation and shear from the corrected gradient.

First two uncorrected velocity gradients are computed as

$$\begin{aligned} \nabla \mathbf{v}'_{i,s} &= \sum_j \left(\mathbf{v}_j - \mathbf{v}_i \right) \otimes V_j \nabla W_{ij}, \\ \nabla \mathbf{v}'_{i,b} &= \sum_k (-\mathbf{v}_i) \otimes V_k \nabla W_{ik}, \end{aligned} \tag{3.22}$$

where $\nabla \mathbf{v}'_{i,s}$ is the uncorrected velocity gradient at particle i with respect to only other snow particles, while $\nabla \mathbf{v}'_{i,b}$ is the uncorrected velocity gradient with respect only to boundary particles. It should be noted that the latter one is simplified in this thesis as boundaries are stationary.

Then a rotation corrected velocity gradient is computed from the two uncorrected ones using the correction matrix, such that

$$\tilde{\nabla} \mathbf{v}_i = \nabla \mathbf{v}'_{i,s} L_i^\top + \frac{1}{d} tr \left(\nabla \mathbf{v}'_{i,b} L_i^\top \right) I, \tag{3.23}$$

where d is the number of dimensions in the simulation, tr is the trace function and I is the identity matrix.

The velocity gradient component $\mathbf{V}'_i$ coming from the change in volume is computed as the normal component of the uncorrected velocity gradient, giving

$$\mathbf{V}'_i = \frac{1}{d} tr \left(\nabla \mathbf{v}'_{i,s} + \nabla \mathbf{v}'_{i,b} \right) I. \tag{3.24}$$

The velocity gradient component $\tilde{\mathbf{R}}_i$ due to rotations in the material is computed as a normalization of the corrected velocity gradient minus its transpose, giving

$$\tilde{\mathbf{R}}_i = \frac{1}{2}\left(\tilde{\nabla}\mathbf{v}_i - (\tilde{\nabla}\mathbf{v}_i)^\top\right). \tag{3.25}$$

The velocity gradient component $\tilde{\mathbf{S}}_i$ due to shear of the material is computed as a symmetrization of the corrected velocity gradient minus its normal component, giving

$$\tilde{\mathbf{S}}_i = \frac{1}{2}\left(\tilde{\nabla}\mathbf{v}_i + (\tilde{\nabla}\mathbf{v}_i)^\top\right) - \frac{1}{d}tr\left(\tilde{\nabla}\mathbf{v}_i\right)I. \tag{3.26}$$

Lastly the volume, rotation and shear components are combined to give the fully corrected velocity gradient

$$\nabla\mathbf{v}_i = \mathbf{V}'_i + \tilde{\mathbf{R}}_i + \tilde{\mathbf{S}}_i \tag{3.27}$$

Divergence of Stress Tensor Fields As described in Section 2.3.1, the forces due to stress can be discretized as Equation (2.39). However, in this situation, one is not computing force, but instead computing an acceleration. As force is equal to mass times acceleration, one can divide Equation (2.39) by the mass m_i of particle i which gives the equation

$$\rho_i a_i = \sum_j V_j\left(\boldsymbol{\sigma}_i\nabla\tilde{W}_{ij} - \boldsymbol{\sigma}_j\nabla\tilde{W}_{ji}\right). \tag{3.28}$$

The density on the left hand side of this equation is however handled by the derivation of Equation (3.20).

Lastly, Gissler *et al.* [7] adds a stress component for all boundary particles which is equal to the normal stress of the particle in question,

$$\boldsymbol{\sigma}_{k,i} = \frac{1}{d}tr(\boldsymbol{\sigma}_i)I. \tag{3.29}$$

Thus, all divergences of stress tensors are discretized as

$$\nabla\cdot\boldsymbol{\sigma}_i = \sum_j V_j\left(\boldsymbol{\sigma}_i\nabla\tilde{W}_{ij} - \boldsymbol{\sigma}_j\nabla\tilde{W}_{ji}\right) + \boldsymbol{\sigma}_{k,i}\sum_k V_k\nabla\tilde{W}_{ik}. \tag{3.30}$$

Solving the Implicit Acceleration System Like with the implicit pressure system (3.14), the implicit acceleration system due to shear stress (3.20) essentially represents a system of equations, $A\mathbf{x} = \mathbf{b}$, Where $\mathbf{x}$ represents a vector of all particle accelerations $\mathbf{a}_{G,i}$ and $\mathbf{b}$ represents a vector of the right hand side of Equation (3.20) for all particles. It can therefore be solved using an iterative method. Gissler *et al.* [7] decides to use a matrix free version of the BiCGSTAB method, Algorithm 2.1, to solve the system. The full implementation of this implicit shear stress solver can be seen in Algorithm 3.4.

Algorithm 3.4: Implicit shear stress solver

```
function computeRightHandSide
    for all snow particles i
        compute ∇(v_i^t + Δt(a_ext,i + a_λ,i))
    for all snow particles i
        compute F*_E,i
        σ_b,i = G_i^t(F*_E,i + (F*_E,i)^⊤ − 2I)
    for all snow particles i
        compute ∇ · σ_b,i
        b_i = (1/ρ_i) ∇ · σ_b,i
    return b

//Note that x is a set of generic vectors, not the particle positions
function computeLeftHandSide(x)
    for all snow particles i
        compute ∇x_i
    for all snow particles i
        σ_x,i = G_i^t((∇x_i)F^t_E,i + ((∇x_i)F^t_E,i)^⊤)
    for all snow particles i
        compute ∇ · σ_x,i
        Ax_i = x_i − (Δt/ρ_i) ∇ · σ_x,i
    return Ax

function shearStressSolver(ϵ, i_max)
    b = computeRightHandSide()
    // See Algorithm 2.1
    a_G = BiCGSTAB(computeLeftHandSide, a_G, b, ϵ, i_max)
```

3.1.7 Integrating Time Dependent Particle Properties

Integrating Velocities The velocity of each particle is numerically integrated using Euler's method, Equation (2.56), where the acceleration of each particle is the sum of the previously computed accelerations. Thus the new velocity at the next time step is computed as

$$\mathbf{v}_i^{t+\Delta t} = \mathbf{v}_i^t + \Delta t \left(\mathbf{a}_{ext,i}^t + \mathbf{a}_{G,i}^{t+\Delta t} + \mathbf{a}_{\lambda,i}^{t+\Delta t} \right) \tag{3.31}$$

Integrating Deformation Gradients The integration of the deformation gradients is dependent on the velocity gradient. Thus, the gradient of the newly acquired velocity $\mathbf{v}_i^{t+\Delta t}$ is first computed using the rotation and volume corrected method by Gissler *et al.* [7], explained in Section 3.1.6. Then the deformation gradient $F_{E,i}^t$ is integrated with the method explained in Section 2.4.3, giving the new deformation gradient $F_{E,i}^{t+\Delta t}$

Integrating Positions The final step of the computation is to integrate the position of each particle. This method used the Euler-Chromer method of integration,

thus the positions are numerically integrated using Equation (2.57), giving the new position $\mathbf{x}_i^{t+\Delta t}$.

3.2 Alternative Solvers for Snow Simulation

This section describe certain alternative formulations of the method described in the preceding section. All these alternative formulations aim to produce the same results, but differ in the way they solve for the acceleration of snow particles.

3.2.1 Fully Decoupled Solver

When simulating snow in the way explained in Section 3.1,the implicit pressure solver is supposed to handle all accelerations due to normal stress, while the implicit shear stress solver is supposed to handle all accelerations due to shear stress. This is however not the case, as mentioned by Gissler *et al.* [7]. The shear stress solver solves Equation (3.17), $\boldsymbol{\sigma} = 2G\boldsymbol{\epsilon}$, one can see that this does in fact contribute to normal stress, as it will tend to produce a non-zero diagonal in the resulting stress matrix.

Deriving a Fully Decoupled Implicit System for Shear Stress To fully decouple the shear stress solver from the pressure solver, one has to remove all normal stress from the stress equation (3.17) solved by the shear stress solver. This equates to the equation

$$\boldsymbol{\sigma}_{G,i}^{t+\Delta t} = 2G_i^t \left(\boldsymbol{\epsilon}_i^{t+\Delta t} - \frac{1}{d} tr(\boldsymbol{\epsilon}_i^{t+\Delta t}) \right), \tag{3.32}$$

where d is the number of dimensions in the simulation and tr is the trace function.

Following the same method as in Section 3.1.6, one can derive an implicit equation for acceleration due to fully decoupled shear stress, which becomes

$$\begin{aligned} &\mathbf{a}_{G,i}^{t+\Delta t} - \frac{\Delta t^2}{\rho_i^t} \nabla \cdot \left(G_i^t \left((\nabla \mathbf{a}_{G,i}^{t+\Delta t}) F_{E,i}^t + ((\nabla \mathbf{a}_{G,i}^{t+\Delta t}) F_{E,i}^t)^\top \right) - \frac{G_i^t}{d} tr \left((\nabla \mathbf{a}_{G,i}^{t+\Delta t}) F_{E,i}^t + ((\nabla \mathbf{a}_{G,i}^{t+\Delta t}) F_{E,i}^t)^\top \right) I \right) \\ &= \frac{1}{\rho_i^t} \nabla \cdot \left(G_i^t \left(F_{E,i}^* + (F_{E,i}^*)^\top - 2I \right) - \frac{G_i^t}{d} tr \left(F_{E,i}^* + (F_{E,i}^*)^\top - 2I \right) I \right), \end{aligned} \tag{3.33}$$

where $F_{E,i}^*$ is the same deformation gradient as seen in Equation (3.21).

This equation can further be solved with the BiCGSTAB method to give acceleration $\mathbf{a}_G$, using the same approach as Algorithm 3.4.

Bulk Modulus and Material Stiffness When Gissler *et al.* [7] solved the implicit pressure system, they set the stiffness of the material equal to the first Lamé parameter λ, as the acceleration due to pressure differences is supposed to represent the normal stress of the λ-term of the generalized Hooke's law (2.82). However,

when the two solvers are fully decoupled, the acceleration from the implicit pressure solver has to represent all normal stress in the material.

As mentioned in Section 2.4.1, there are many ways to describe the strength of an elastic material, and the section further describes the Young Modulus and Poisson's Ratio. The Bulk Modulus K is one such description, which represents the materials resistance to volume change when experiencing pressure. It is defined as

$$K = \rho \frac{dp}{d\rho}. \tag{3.34}$$

This thesis will not go into detail of how other equations for the Bulk Modulus are derived, but one can derive an equation for a particles Bulk Modulus given the particles Lamé Parameters as

$$K_i = \lambda_i + \frac{2G_i}{3}. \tag{3.35}$$

This can further be used as the stiffness in Equation (3.6) to give an implicit pressure solver which solves for an acceleration given all normal stress in the material.

3.2.2 Combined Solver

Gissler *et al.* [7] uses two solvers to compute acceleration, one implicit pressure solver and one implicit shear stress solver. Their reason for doing this is so that the implicit pressure solver can interact with solvers for other materials. However, the HPC-Lab Snow Simulator is, at the moment, for the most part only interested in snow itself and hence does not need the snow to interact with other materials. For this purpose, a single combined solver can be formulated, which simultaneously solves for normal stress and shear stress.

Deriving an Implicit Equation for Acceleration due to Stress As normal stress is handled by the implicit stress solver, the implicit pressure solver, Algorithm 3.3, can be removed from the system. Then, to handle normal stress, the implicit stress solver has to be based on the entirety of the generalized Hooke's law (2.82),

$$\boldsymbol{\sigma}_i^{t+\Delta t} = 2G_i^t \boldsymbol{\epsilon}_i^{t+\Delta t} + \lambda_i^t tr(\boldsymbol{\epsilon}_i^{t+\Delta t})I. \tag{3.36}$$

Using the same derivation method as used in Section 3.1.6, one can derive an implicit equation for acceleration $\mathbf{a}_\sigma$ due to stress, giving

$$\begin{aligned}
&\mathbf{a}_{\sigma,i}^{t+\Delta t} - \frac{\Delta t^2}{\rho_i^t}\nabla\cdot\left(G_i^t\left((\nabla\mathbf{a}_{\sigma,i}^{t+\Delta t})F_{E,i}^t + ((\nabla\mathbf{a}_{\sigma,i}^{t+\Delta t})F_{E,i}^t)^\top\right) + \frac{\lambda_i^t}{2}tr\left((\nabla\mathbf{a}_{\sigma,i}^{t+\Delta t})F_{E,i}^t + ((\nabla\mathbf{a}_{\sigma,i}^{t+\Delta t})F_{E,i}^t)^\top\right)I\right) \\
&= \frac{1}{\rho_i^t}\nabla\cdot\left(G_i^t\left(F_{E,i}^* + (F_{E,i}^*)^\top - 2I\right) + \frac{\lambda_i^t}{2}tr\left(F_{E,i}^* + (F_{E,i}^*)^\top - 2I\right)I\right).
\end{aligned} \tag{3.37}$$

This equation can further be solved using the BiCGSTAB method to give the acceleration $\mathbf{a}_\sigma$, using the same approach as Algorithm 3.4.

3.2.3 Solving for Velocity

All the previously mentioned stress solvers, like the solver used by Gissler *et al.* [7], solves an equation for an acceleration due to stress. However, like in the method by Peer *et al.* [14] for simulating elastic solids, one can instead solve the equations for the resulting velocity $\mathbf{v}^{t+\Delta t}$.

Deriving an Implicit Velocity Equation The derivation of an implicit velocity equation is the exact same derivation as seen in Section 3.1.6, with the exception of not expressing the final derivative. The implicit velocity due to shear stress is thus Equation (3.19). The equation can then be solved for the velocity $\mathbf{v}^{t+\Delta t}$ using the BiCGSTAB method in the same way as seen in Algorithm 3.4.

As this method results in the velocity for the next time step, one is not required to numerically integrate the velocities. This integration can thus be removed from Algorithm 3.2.

This method of solving directly for the resulting velocity can similarly be used with the fully decoupled solver, Section 3.2.1, and the combined solver, Section 3.2.2.

3.2.4 Explicit Solver

Gissler *et al.* [7] mentions the reason for choosing an implicit solver over an explicit solver. When simulating stiff materials, the simulation has to be quite precise in order to result in a stable material. For explicit solutions this requires pretty small time steps, while for an implicit solution one can get by using larger time steps. It also turns out that for a wide variety of numerical simulations, the performance increase of having larger time steps outweighs the performance decrease of iteratively solving an implicit system.

This section will describe an explicit method for computing the combined acceleration due to stress, as seen in Section 3.2.2. However, explicit methods can be used in conjunction with the implicit pressure solver, as used in Section 3.1 and Section 3.2.1.

Deriving the Explicit Acceleration due to Stress As with the combined solver, the explicit derivation starts off from the generalized Hooke's law (2.82), with the exception that one computes the stress at the current time step. Thus the particles stress can be expressed as

$$\boldsymbol{\sigma}_i^t = 2G_i^t \boldsymbol{\epsilon}_i^t + \lambda_i^t tr(\boldsymbol{\epsilon}_i^t) I. \tag{3.38}$$

The strain of a particle can, as seen in Equation (2.90), be computed as

$$\boldsymbol{\epsilon}_i^t = \frac{1}{2}\left(F_{E,i}^t + \left(F_{E,i}^t\right)^\top\right) - I. \tag{3.39}$$

The acceleration on the particle due to this internal stress can, as seen in Equation (2.91), can be computed as

$$\mathbf{a}_{\sigma,i}^{t} = \frac{1}{\rho_i^t} \nabla \cdot \boldsymbol{\sigma}_i^t, \tag{3.40}$$

where the computation of the divergence of stress follows the same method as Equation (3.30).

Using this explicit formulation, the acceleration due to internal stress can be computed by iterating through all particles twice, as seen in Algorithm 3.5, which replaces the two implicit solvers seen in Algorithm 3.2.

Algorithm 3.5: Computing explicit acceleration due to stress

```
function computeAccelerationDueToStress
    for all snow particles i
        compute ϵ_i
        compute σ_i
    for all snow particles i
        compute ∇ · σ_i
        compute a_{σ,i}
```

Chapter 4

Implementation

This chapter describes in detail aspects of the code needed to implement a numerical simulation of snow using smoothed particle hydrodynamics.

The chapter first describes a serial implementation which has been implemented on central processing units, before describing a parallel implementation that utilizes graphics processing units.

4.1 Serial Implementation

During implementation of the methods described previously in this thesis, the methods where first implemented as a serial implementation [15], with no regards to performance. This was done to make sure the implementation was correct as the only reference used for implementation was previously written papers.

Implemented Methods In the serial implementation all nine methods mentioned throughout Chapter 3 where implemented. This includes the methods listed in Table 4.1.

Table 4.1: Implemented methods for the serial implementation

Normal and shear stress relation	Stress solver
Lamé decoupled	Implicit acceleration
Fully decoupled	Implicit acceleration
Combined	Implicit acceleration
Lamé decoupled	Implicit velocity
Fully decoupled	Implicit velocity
Combined	Implicit velocity
Lamé decoupled	Explicit acceleration
Fully decoupled	Explicit acceleration
Combined	Explicit acceleration

In order to simplify this implementation, all of the above methods where implemented as two dimensional simulations.

It should further be mentioned that all floating point values in this implementation, including vector and matrix components, are 64-bit IEEE 754 floating point values [41].

4.1.1 Neighborhood Lookup

Iterating over particle neighborhoods is an essential aspect of SPH. Although performance is not the focus of this implementation, having a neighborhood lookup with a complexity of $O(n)$ makes the implementation virtually unusable for any practical amount of particles. Thus a lookup method with an asymptotic complexity of $O(1)$ was implemented.

The neighborhood lookup method is based on splitting the simulation domain into a regular Cartesian grid and placing particles into their respective grid cells. When one wants to iterate over some particles neighborhood, it amounts to iterating through particles in the cells that could contain neighbors.

Grid Size As described in Section 2.3.1, the neighborhood of a particle in two dimensions is any particle within a circle of radius equal to the support radius $\hbar$ of the simulation. If one split the simulation domain into a grid with very large grid cells compared to the support radius, one would usually only need to look through one grid cell to find neighbors. However, in the worst case, when a particle is at the very corner of a grid cell, its neighborhood would cover four grid cell. On the other hand, a small cell size compared to the support radius would require one to look through a large amount of cells, which would have a greater performance cost. Thus the most optimal choice is the largest possible grid cell which still only requires one to look through four grid cells in the worst case. This turns out to be a grid where each cell has a side length of two times the support radius, $2\hbar$.

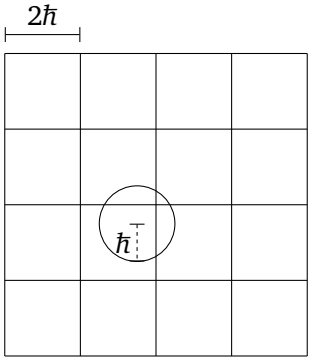


Figure 4.1: Grid with cell side length of two times support radius showing that in the worst case a neighborhood covers four grid cells.

Implementation Every particle in the simulation and their properties are first and foremost stored in an array. At the start of the simulation every particle is stored in their respective grid cell as a list of indices referring to the particle array. Before the position $\mathbf{x}_i$ of a particle is integrated to give particles position at the next time step, it is stored as the particles previous position $\mathbf{x}_i^*$. At the end of every time step, every particle is first removed from the index list of the respective grid cell for its previous position, then the particle is inserted into the index list of the respective grid cell for its new position.

It should also be noted that in this implementation, although it is not optimal, the snow particles and the boundary particles are both placed into the same grid, requiring their particle type to be checked on use.

The method for updating the particle positions within the lookup grid and the method for finding all neighbors of a particle can be seen in Algorithm 4.1.

Algorithm 4.1: Lookup table functions.

```
function update(i)
    previousCell = lookup table cell containing x_i^*
    newCell = lookup table cell containing x_i
    remove i from previousCell
    insert i into newCell

function getNeighbors(i)
    gridCorner = x_i rounded to nearest grid cell corner
    possibleNeighbors = all indices in cells touching gridCorner
    return all j from possibleNeighbors filtered by ||x_i − x_j|| ≤ h̄
```

4.1.2 Singular Value Decomposition and Pseudoinverse

Singular Value Decomposition, SVD, is used for handling plasticity when integrating the deformation gradient. This implementation is required to compute the SVD of a two by two matrix. It was chosen to follow the method by Blinn [42], which can be seen in Algorithm 4.2.

When computing the correction matrices for particles one is required to invert a matrix. As stated in Section 2.3.1, when the matrices are not inevitable, one instead computes the Moore-Penrose Psudoinverse of the matrix. As stated by Golub *et al.* [43], the Moore-Penrose Pseudoinverse of a matrix can be computed by first computing the SVD of the matrix as $U\Sigma V^\top$, then replacing the non-zero singular values of Σ with their reciprocals, giving Σ^+ and lastly combing the decomposition to give the pseudo inverse as $V\Sigma^+U^\top$. It should also be noted that when the derivative of a matrix or the singular values of a matrix are close to zero, the inverse of the matrix or the reciprocal of the singular values, might cause a floating point overflow. Thus, instead of checking whether or not values are equal to zero, one checks whether or not the absolute value is smaller than some small value ϵ. In this implementation $\epsilon = 10^{-16}$. Both the psudoinverse and safe inverse methods can be seen in Algorithm 4.3.

Algorithm 4.2: Singular value decomposition of two by two matrices.

function singularValueDecomposition(A)

$E = \frac{A_{0,0}+A_{1,1}}{2}$
$F = \frac{A_{0,0}-A_{1,1}}{2}$
$G = \frac{A_{0,1}+A_{1,0}}{2}$
$H = \frac{A_{0,1}-A_{1,0}}{2}$

$Q = \sqrt{E^2+H^2}$
$R = \sqrt{F^2+G^2}$

$a_1 = \tan^{-1}\frac{G}{F}$
$a_2 = \tan^{-1}\frac{H}{E}$

$\theta = \frac{a_2-a_1}{2}$
$\phi = \frac{a_2+a_1}{2}$

$$\Sigma = \begin{bmatrix} Q+R & 0 \\ 0 & |Q-R| \end{bmatrix}$$
$$U = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix}$$
$$V^\top = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

return U, Σ, $V^\top$

Algorithm 4.3: Moore-Penrose pseudoinverse and safe inverse of two by two matrices.

function moorePnerosePseudoinverse(A)

U, Σ, $V^\top$ = singularValueDecomposition(A);

$$\Sigma^+ = \begin{bmatrix} \textbf{if } |\Sigma_{0,0}| < \epsilon \textbf{ then } 0 \textbf{ else } \frac{1}{\Sigma_{0,0}} & 0 \\ 0 & \textbf{if } |\Sigma_{1,1}| < \epsilon \textbf{ then } 0 \textbf{ else } \frac{1}{\Sigma_{1,1}} \end{bmatrix}$$

return $V\Sigma^+U^\top$

function safeInverse(A)

$det(A) = A_{0,0}A_{1,1} - A_{0,1}A_{1,0}$

if $|det(A)| > \epsilon$
    return $\frac{1}{det(A)}\begin{bmatrix} A_{1,1} & -A_{0,1} \\ -A_{1,0} & A_{0,0} \end{bmatrix}$
else
    return moorePnerosePseudoinverse(A)

4.2 Parallel Implementation

This section describes the parallel implementation which focuses more on performance. This implementation was implemented in the HPC-Lab snow simulator and utilizes GPU programming through the CUDA platform for parallelism and improved performance.

Integration with the HPC-Lab Snow Simulator As this implementation has the same structure of other SPH implementations, large amounts of previously written code from the HPC-Lab snow simulator was reused. However, due to lack of name spacing, this implementation ran into problems with name collisions, thus all the code specific to this implementation was compiled as its own library before it was linked with the rest of the simulator.

While this implementation utilizes much of the rendering implementation that existed in the HPC-Lab snow simulator, it does not utilize any other aspects of the simulator, such as terrains. It is however implemented in such a way that integrating these aspects should pose no apparent problems.

Implemented Method Due to time limitations and mistakes that where made during development of the serial implicit methods, only the method for combined explicit stress was used in this implementation. The method was implemented as a three dimensional simulation and utilizes 32-bit IEEE 745 floating point values [41] as GPUs typically are tailored to operate on these values.

Algorithm 4.4: Parallel implementation for computing next time step.

```
function runKernel(kernel, grid)
    run with grid configuration
        kernel
    synchronize threads

function computenextTimeStep()
    particleReorderingWithPrefixSumAndCountingSort()

    runKernel(precomputeNeighbors, precomputeGrid)
    runKernel(densityVolumeAndRestDensity, generalGrid)
    runKernel(lameParameters, generalGrid)
    runKernel(correctionMatrix, matrixGrid)
    runKernel(predictedVelocity, generalGrid)
    runKernel(predictedVelocityGradient, generalGrid)
    runKernel(stress, matrixGrid)
    runKernel(accelerationDueToStress, matrixGrid)
    runKernel(integrateVelocity, generalGrid)
    runKernel(integrateDeformationgradient, matrixGrid)
    runKernel(integratePosition, matrixGrid)
```

The general structure of computation of time steps can be seen in Algorithm 4.4. All grid values seen in this algorithm will be further discussed in Section 4.2.3.

Note that predicted velocity in this context refers to the integration of velocity given only external acceleration, $\boldsymbol{v} + \Delta t \boldsymbol{a}_{ext}$.

Simulation Parameters The simulation parameters in this implementation are stored in what CUDA refers to as constant memory. This is a highly performant read-only section of memory on the GPU. As there are not many simulation parameters, their storage overhead is negligible and will thus not be further discussed in this thesis.

Particle Storage Throughout the simulation all properties required by particles are stored in separate arrays allocated on the GPU. These properties can be seen in Table 4.2 with their respective type and memory usage.

Table 4.2: Properties stored on the GPU for each particle.

Property	Type	Memory usage
`position`	`float3`	12*B*
`velocity`	`float3`	12*B*
`acceleration`	`float3`	12*B*
`mass`	`float`	4*B*
`density`	`float`	4*B*
`restDensity`	`float`	4*B*
`volume`	`float`	4*B*
`correctionMatrix`	`matrix3`	36*B*
`lambda`	`float`	4*B*
`G`	`float`	4*B*
`deformationGradient`	`matrix3`	36*B*
`predictedVelocity`	`float3`	12*B*
`predictedVelocityGradient`	`matrix3`	36*B*
`stress`	`matrix3`	36*B*
`color`	`float`	4*B*
`particleCell`	`unsigned int`	4*B*
`particleCellOffset`	`unsigned int`	4*B*
`searchFrom`	`unsigned int`	4*B*
`neighbors`	`precomputedNeighbor<128>`	2564*B*
`boundaryNeighbors`	`precomputedNeighbor<32>`	644*B*

One should note that the acceleration property is reused for both the external acceleration and the final acceleration of particles. The color property is used to color particles by their density when visualized by the simulator. The last five properties in the table will be further discussed in Section 4.2.1.

The properties of the boundary in the simulation are similarly stored and can be seen in Table 4.3.

Table 4.3: Properties stored on the GPU for each boundary particle.

Property	Type	Memory usage
`position`	`float3`	12*B*
`psi`	`float`	4*B*
`volume`	`float`	4*B*
`color`	`float`	4*B*
`particleCell`	`unsigned int`	4*B*
`particleCellOffset`	`unsigned int`	4*B*
`searchFrom`	`unsigned int`	4*B*

4.2.1 Neighborhood Lookup

The neighborhood lookup method used in this implementation is the same as used by Sandvik [4] in a previous SPH implementation on the HPC-Lab Snow Simulator, with a slight modification that allows one to search through fewer cells. The method is inspired by the work of Green [9].

The method is similar to the grid method used in the serial implementation, Section 4.1.1. However, while the serial method stores particle indices in each grid cell, and extract particle properties from a single particle array, this lends itself badly to high performance, as the random access in the particle array can not easily be cached. The parallel implementation instead first computes which cell each particle belongs to, then sorts the arrays containing particle properties such that the properties of particles belonging to the same cell lies consecutively in the memory. This way, looking up properties of particles within a cell can be done by iterating through a section of the respective array, which again can give performance advantages as the array section can be cached.

In addition to the use of this method, this implementation pre-computes kernel values between particles and their neighbors, as an added means of achieving higher performance.

Memory Usage Similarly to how the particle properties are stored in separate arrays on the GPU, the properties required by each grid cell are stored in their dedicated arrays on the GPU. The grid cells each require two properties which can be seen in Table 4.4.

Table 4.4: Properties stored on the GPU for each grid cell.

Property	Type	Memory usage
`gridCount`	`unsigned int`	4*B*
`gridOffset`	`unsigned int`	4*B*

Grid Size As explained in Section 4.1.1, the optimal grid cell size, in a two dimensional simulation, is a grid where cells have side lengths of two times the

support radius of the simulation, $2\bar{h}$, such that in the worst case one must look through four grid cells in order to find all neighbors. The same logic follows in three dimensions, where the optimal cell size length is two times the support radius, which requires one to look through eight cells in the wort case.

Particle Insertion The first step of this lookup method is first to insert particles into their respective cells. This is done by going through each particle, computing which cell they belong to and storing the index of this cell as the particle property `particleCell`, then atomically incrementing the respective cells property `gridCount` in order to keep track of how many particles are contained within the cell. Atomic increment in this case, is an increment which avoids race conditions when performed in parallel across multiple threads. Finally, the value of `gridCount` before it was incremented is set as the particles `particleCellOffset` such that every particle within the same cell has a unique value for this property.

Simple In-Place Prefix Sum In order to perform the sorting described below, one has to compute the prefix sum of the array containing the grid cell property `gridCount`. The prefix sum of a list of n numbers, is a new list of n numbers where a number with index i in the new list is the sum of the i first numbers in the original list.

Prefix sums is a well discussed subject within parallel computing [44]. One of the more common methods for computing in-place prefix sums in parallel is used for as the basis in this implementation.

Algorithm 4.5: Parallel In-place prefix sum of array of length $2t$ running on t threads where the number of threads is a power of two.

```
device function inPlacePrefixSum(array)
    t = number of threads

    for s in {2^0, 2^1, 2^2, ..., t}
        i = 2*s*(threadId + 1) - 1
        if i < 2*s
            array[i] = array[i] + array[i - s]
        synchronize threads

    for s in {t/2, ..., 2^2, 2^1, 2^0}
        i = 2*s*(threadId + 1) - 1
        if i + s < 2*s
            array[i + s] = array[i + s] + array[i]
        synchronize threads
```

The in-place prefix sum method runs $\frac{n}{2}$ threads, where n is the number of values in the input list. Then each thread is responsible for iteratively adding two and two elements together in such a way that separate threads do not interfere with each other. In order to compute the whole prefix sum without interference between threads, the threads are synchronized between every addition iteration.

With the structure implemented in this implementation, seen in Algorithm 4.5 and Figure 4.2, each thread is required to perform at most $2\log_2(n)+1$ additions.

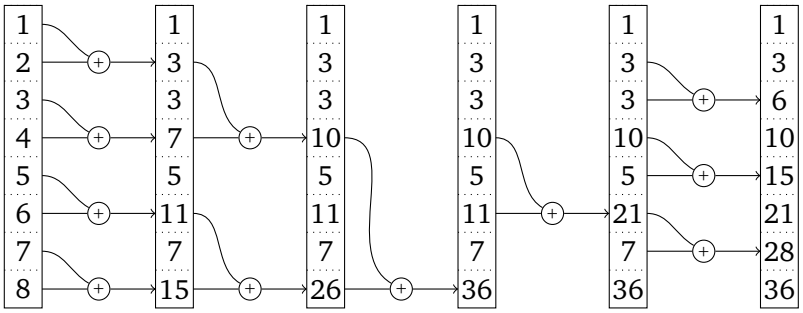


Figure 4.2: Computation of in-place prefix sum on an array with the numbers from one to eight.

Optimized Prefix Sum Although the aforementioned method is theoretically optimal, in practice one can achieve a more performant algorithm by more optimally utilizing cache or, in the case of CUDA, shared memory.

The idea behind the optimized prefix sum method is to divide the full array into, sections, or blocks, which are small enough to fit into shared memory and thus be optimally run through the in-place prefix method. The results from running the in-place prefix sum on a block is shifted one element to the right and the first element is set to zero. As the last elements is shifted out of the array, it is stored in a new array, referred to as `blockSums`. Thereafter, one computes the prefix sum of the `blockSums` array before adding these block sum prefix sums to every element of the corresponding block.

This results in a prefix sum of the whole array shifted one element over, with the first element being equal to zero. However, this only works is the block sum array fits into shared memory. If the block sum array does not fit into shared memory, one can recursively use this algorithm to compute the prefix sum of the block sum array until one has to compute the prefix sum of an array that fits into shared memory. In this implementation the method was implemented with a maximum recursive depth of three in order to more easily account for memory usage.

This optimized prefix sum method can be seen in Algorithm 4.6 and an example of its use can be seen in Figure 4.3.

Algorithm 4.6: Optimized parallel prefix sum shifted one element right.

```
device function blockPrefixSum(input, output, blockSums, inputSize)
    prefixSums = shared array of length two times threads per block

    t = number of threads per block
    i = threadId + 2 * blockId * t
    j = i + t

    prefixSums[threadId] = if i < inputSize then input[i] else 0
    prefixSums[threadId + t] = if j < inputSize then input[j] else 0

    synchronize threads
    inPlacePrefixSum(prefixSums)

    if i + 1 < inputSize
        output[i + 1] = prefixSums[threadId]
    if j + 1 < inputSize
        output[j + 1] =
            if threadId == t - 1 then 0 else prefixSums[j + t]
    if threadId == 0
        output[0] = 0
        if blockSums != undefined
            blockSums[blockId] = prefixSums[2*t - 1]

device function combineBlocks(output, blocksPrefixSum, inputSize)
    t = number of threads per block
    i = threadId + 2*blockId*t

    if i < inputSize
        output[i] += blocksPrefixSum[blockId]
    if i + t < inputSize
        output[i + t] += blocksPrefixSum[blockId]

function prefixSum(input, output, threadsPerBlock)
    inputSize = length of input
    blockSize = 2*threadsPerBlock
    blockCount = ⌊inputSize/blockSize⌋ + 1

    if blockCount == 1
        run on blockCount blocks with threadsPerBlock threads
            blockPrefixSum(input, output, undefined, inputSize)
    else
        blockSums = array of length blockCount
        blocksPrefixSum = array of length blockCount

        run on blockCount blocks with threadsPerBlock threads
            blockPrefixSum(input, output, blockSums, inputSize)

        prefixSum(blockSums, blocksPrefixSums, threadsPerBlock)

        run on blockCount blocks with threadsPerBlock threads
            combineBlocks(output, blocksPrefixSum, inputSize)
```

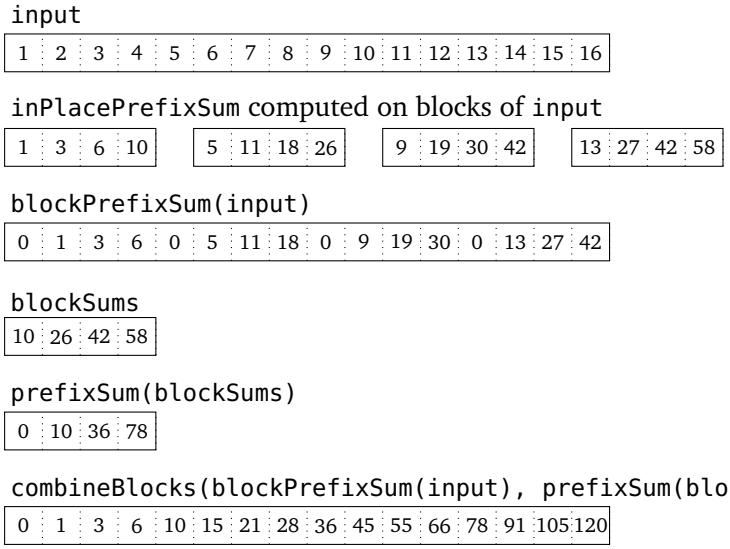


Figure 4.3: Computation of the optimized prefix sum on an array with the numbers from one to sixteen.

Counting Sort The prefix sum, shifted one element over, of the `gridCount` array, computed with the aforementioned optimized prefix sum method, is stored in the grid cell property array `gridOffset`. Using this property, the particle property arrays can be sorted, using counting sort, such that the sorted properties of the particles within a single cell lie consecutively in memory.

The counting sort is performed by computing a new sorting index for each particle i as

```
sortIndex[i] = gridOffset[particleCell[i]] + particleCellOffset[i].
```
(4.1)

Then the respective particle property array is copied to a temporary array and each particle property in the temporary array is copied back to the original array into their respective sorting index location. It should be noted that only the `position`, `velocity`, `deformationGradient`, `particleCell`, `particleCellOffset` and `searchFrom` properties are sorted, as they are the only particle properties that needs to be retained across time steps.

With the property arrays sorted by this method, the property of the first particle in a cell will be located at the index which is equal to the cells `gridOffset`, while the number of particles in the cell will be equal to the cells `gridCount`. Thus, iterating through all properties within a cell c amounts to iterating from index gridOffset[c] to, but not including, `gridOffset[c] + gridCount[c]`.

Complete Particle Reordering The complete particle reordering method boils down to first initializing the property array of `gridCount` and `gridOffset` to zero, followed by inserting all particles into the grid, then computing the prefix sums of the `gridCount` array and lastly copying relevant properties to temporary arrays

and performing the counting sort reordering. This can be seen in Algorithm 4.7 and an example of its use can be seen in Figure 4.4.

Algorithm 4.7: Particle reordering.

```
function reorderParticles()
    set gridCount to 0
    set gridOffset to 0

    runKernel(insertParticles, generalGrid)

    blockPrefixSum(gridCount, gridOffset, threadsPerBlock)

    copy relevant properties to temporary arrays

    runKernel(countingSort, generalGrid)
```

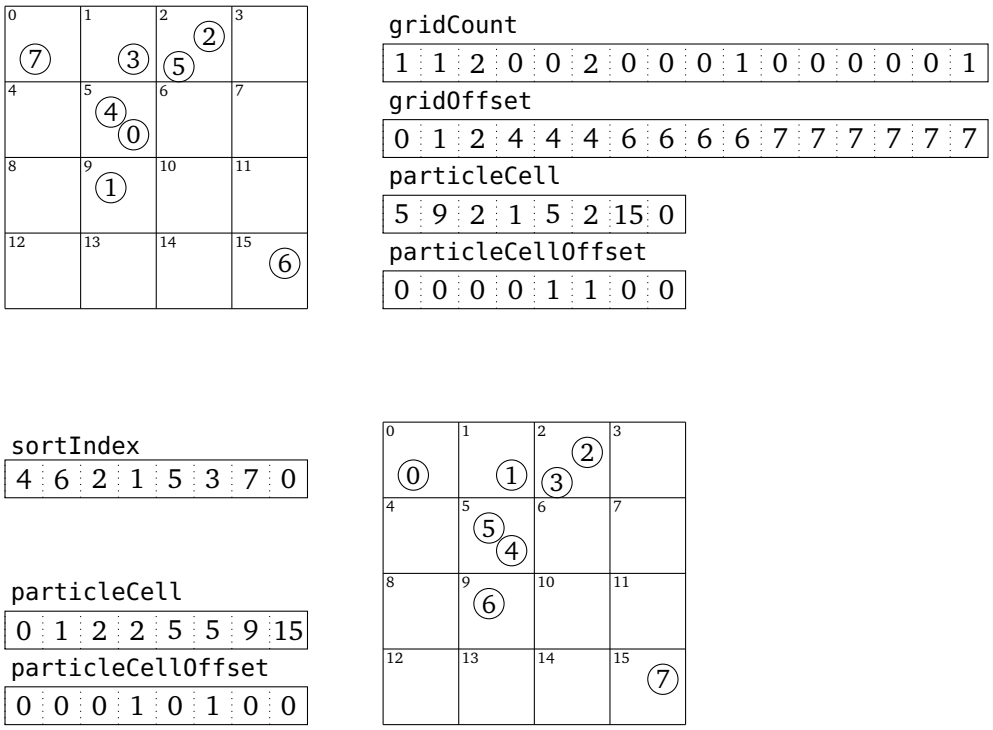


Figure 4.4: Example of particles reordered with counting sort.

It should further be mentioned that, during initialization of the simulator, the boundary particles are reordered using the same method.

Neighbor Lookup As mentioned above, in order to find all neighbors of a particle, one has to look through at most eight cells. Which eight cells one has to look through, depends on which corner of its own cell the particle lies closes to. In order to find these eight cells, the position of the one of these eight cells lying closes

to the cell with index zero is computed for and stored in the particle property `searchFrom`. This value is computes as

$$\texttt{searchFrom[i]} = \left\lfloor (\mathbf{x}_i - \texttt{grid}_{min}) * \Delta\texttt{grid} - \begin{bmatrix} 0.5 \\ 0.5 \\ 0.5 \end{bmatrix} \right\rfloor, \tag{4.2}$$

where $\texttt{grid}_{min}$ and $\Delta\texttt{grid}$ is the smallest value corner of the grid and grid cell side length, respectively. Further more, the index of this cell computed in the same manner as the index of the cell containing the particle, as mentioned above.

The `searchFrom` cell is the cell with the smallest position in all the cardinal directions, compared to the eight cells one has to look through. One can then compute offsets from this cell index to the other seven cells one has to look through.

In order to then find all neighbors of a particle, one goes through all eight offsets, the first being zero, giving a cell index `searchFrom[i] + searchOffset[k]` where k is the current offset. Then one looks through all particles within these cells and checks whether or not they are in fact in the neighborhood of particle i.

Kernel Value Pre-Computation It was noticed quite early on that the neighborhood lookup is, by an order of magnitude with respect to computation time, the greatest bottleneck in the implementation. In order to improve the performance of this are neighbors pre-computed at the beginning of every time step. In addition to this, as the smoothing kernel values are computed by all neighbor dependent kernels, these are also pre-computed.

In order to store these pre-computed values a pre-computed neighbors property type has been constructed. The structure of this type can be seen in Table 4.5.

Table 4.5: Structure of the pre-computed neighbors type.

Field	Type	Memory usage
`count`	`unsigned int`	$4B$
`index`	`unsigned int[N]`	$N \times 4B$
`W`	`float[N]`	$N \times 4B$
`WGrad`	`float3[N]`	$N \times 12B$

As one can see, this type is dependent on some value N, this value is the maximum allowed neighbors of a particle. This implementation has chosen to allow a maximum of 128 snow particle neighbors and a maximum of 32 boundary particle neighbors.

This type stores a count of how many neighbors a particle has, an array of the indices of all neighbors and arrays for the smoothing kernel values and their gradients between a particle and all its neighbors.

As the indices of all neighbors of a particle are stored in this property, iterating through all neighbors of a particle amounts to iterating through the `count` first indices in the `index` array.

Optimized Pre-Computation Due to the workings of GPUs, if one thread within a kernel grid block requires less work than other threads within the same block, it will stall and wait for the other threads to complete. It is thus preferable to limit this behavior.

As described earlier in this section, finding all neighbors of a particle requires one to iterate through eight grid cells. If a thread iterates through all these eight cells and the cells happen to not contain particles, while other threads in the same kernel grid block does find particles in their cells, the thread would have to stall and consequently lessen the performance of the whole system.

In order to mitigate this effect this implementation dedicates one thread to each block needed to be searched through. However, since neighbors are dependent on all eight cells the `count` field of a particles `neighbors` property has to be incremented atomically. The value of this count, before its increment, is further used as the index in which to place neighbor values in their respective arrays.

4.2.2 Singular Value Decomposition

As discussed in Section 4.1.2, Singular Values Decomposition is used both in the integration of deformation gradients and in order to compute the Moore-Penrose pseudoinverse, which is required in the computation of the correction matrices.

While the SVD of a two by two matrix can be computed explicitly, the situation is more complex when dealing with three by three matrices. When computing the SVD of a three by three matrix one has to iteratively find a numerical approximation of the decomposition.

This implementation uses the implementation by Wu [45], which is a CUDA implementation of the method described by McAdams *et al.* [46].

4.2.3 CUDA Grids

Throughout the algorithms shown in this section certain grid values have been mentioned. As described in Section 2.5.2, the performance of a GPU kernel is highly dependent on kernel grid one uses when running kernels on the GPU. Finding optimal configurations of these grids is a difficult task and has not been properly explored during the work on this thesis. However, as matrix operations require a higher degree of register usage from each thread, the kernels performing these operations where limited in their grid choice. Thus these kernels are run with a grid configuration of 256 threads per grid block, which is referred to as the `matrixGrid`. Most of the other kernels where however found to perform better with a grid configuration of 512 threads per block, referred to as the `generalGrid`. The pre-computation kernel was further found to perform best with a grid configuration of 1024 threads per block, referred to as the `precompueGrid`.

Chapter 5

Results

This chapter presents the results that have been generated from the two implementations in this thesis.

The results from the serial implementation first show a selection of general simulation with a focus on differentiating various methods and an inspection of varying time step sizes. It further shows results from simulations with a focus on aspects related to slab avalanches.

The results from the parallel implementation are generated with a focus on inspecting the performance of the implementation.

5.1 Serial Results

This section describes various results from the serial implementation. This includes comparing various methods, comparing various time steps, a closer look at boundary handling and two situations related to slab avalanches. These results are for the most part only visual, as they are generated with the purpose of showing the simulators abilities.

While the following results have varying parameters, some parameters are constant throughout all results. The gravity $\mathbf{g}$ of the simulation is set to earths gravity,

$$\mathbf{g} = \begin{bmatrix} 0 \\ -9.81 \end{bmatrix}. \tag{5.1}$$

The particle spacing Δs is set to

$$\Delta s = 0.00568181818m. \tag{5.2}$$

The tolerance ϵ of both the BiCGSTAB method, Algorithm 2.1, and the implicit pressure solver, Algorithm 3.3, it set to 0.001. Lastly, as the generation of particles is quite uniform, while actual snow is less uniform, all simulations randomly place particles within a square with side length of $\frac{\Delta s}{8}$ around their intended position, unless specified otherwise.

5.1.1 Comparison of Methods

For the serial implementation nine methods were implemented, see Table 4.1. In order to compare these methods, a simulation was configured where two snow balls collide.

Setup The snow balls each have a radius $9.375cm$. They are placed with their centers $30cm$ apart in the horizontal direction and $5cm$ apart in the vertical direction. The leftmost snow ball has an initial velocity of $2\frac{m}{s^2}$ in the right direction and the rightmost snow ball has an initial velocity of $2\frac{m}{s^2}$ in the left direction. Their initial configuration can be seen in Figure 5.1.

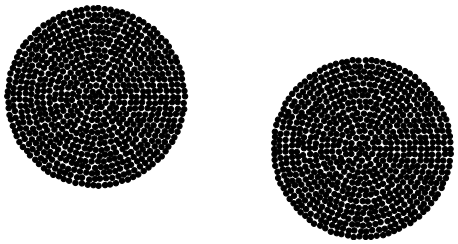


Figure 5.1: Initial configuration of two snow balls colliding.

The snow balls are both configured with the same parameters, inspired by the recommended values by Stomakhin *et al.* [10], which can be seen in Table 5.1.

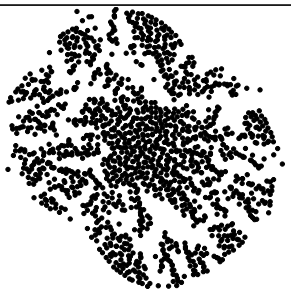
Table 5.1: Parameters for two snow balls colliding.

Parameter	Value
Δ t	$0.0001s$
ρ_0	$400\frac{kg}{m^3}$
E	$140000Pa$
ν	0.2
ξ	10
θ_c	0.025
θ_s	0.0075

Lastly, the simulation is run for a simulation time of $0.1s$, which is slightly after the impact of the snow balls, whereas the position of all particles are captured as the result.

Results The results of the of the two snowballs colliding, using all implemented methods, can be seen in Figure 5.2. The computation time show with every result is the combined computation time of all $\frac{0.1s}{\Delta t} = 1000$ time steps. Where relevant, the solver iterations per time step is shown, where "Min" is the minimum number of iterations used by the respective solver for a single time step, "Max" is the maximum number of iterations used by the solver for a single time step and "Avg"

Implicit stress solved for acceleration

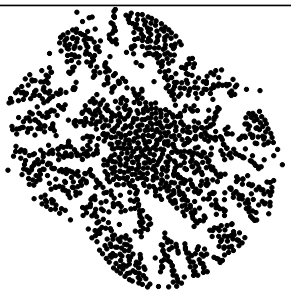


(a) Lamé decoupled
Computation time:
106.61s
$\boldsymbol{a}_\lambda$-solver iterations:
Min: 1, Max: 4, Avg: 2.64
$\boldsymbol{a}_G$-solver iterations:
Min: 0, Max: 2, Avg: 0.86

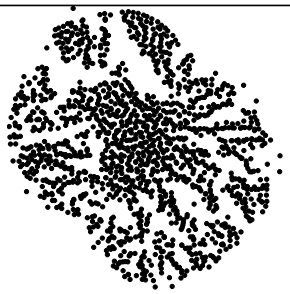
(b) Fully decoupled
Computation time:
108.05s
$\boldsymbol{a}_\lambda$-solver iterations:
Min: 1, Max: 5, Avg: 3.16
$\boldsymbol{a}_G$-solver iterations:
Min: 0, Max: 1, Avg: 0.70

(c) Combined
Computation time:
45.47s
$\boldsymbol{a}_\sigma$-solver iterations:
Min: 0, Max: 2, Avg: 0.72

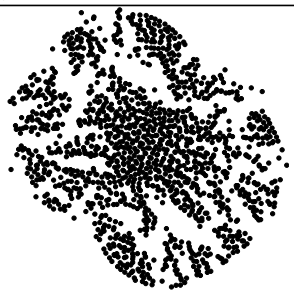
Implicit stress solved for velocity



(d) Lamé decoupled
Computation time:
52.25s
$\boldsymbol{a}_\lambda$-solver iterations:
Min: 1, Max: 4, Avg: 2.64
$\boldsymbol{a}_G$-solver iterations:
Min: 0, Max: 1, Avg: 0.68



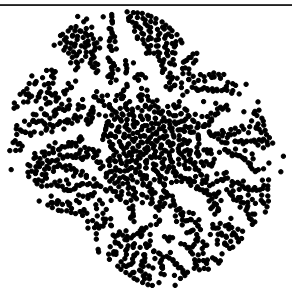
(e) Fully decoupled
Computation time:
56.11s
$\boldsymbol{a}_\lambda$-solver iterations:
Min: 1, Max: 5, Avg: 3.16
$\boldsymbol{a}_G$-solver iterations:
Min: 0, Max: 1, Avg: 0.68



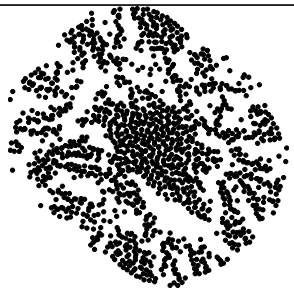
(f) Combined
Computation time:
41.66
$\boldsymbol{a}_\sigma$-solver iterations:
Min: 0, Max: 1, Avg: 0.68

Explicit stress

(g) Lamé decoupled
Computation time:
34.66s
$\boldsymbol{a}_\lambda$-solver iterations:
Min: 1, Max: 4, Avg: 2.65



(h) Fully decoupled
Computation time:
37.51s
$\boldsymbol{a}_\lambda$-solver iterations:
Min: 1, Max: 5, Avg: 3.19



(i) Combined
Computation time:
19.31s

Figure 5.2: Two spheres colliding.

is the combined total number of iterations for the solver divided by the number of time steps. Note that the stress solvers can have an iteration count of zero, which is due to the early stopping condition of the Bi-CGSTAB method, Algorithm 2.1.

The two first rows of Figure 5.2 show that the results from the stress solvers that solve for implicit acceleration is equivalent to the result from the stress solvers that solve for implicit velocity. However, the ones that solve for acceleration requires slightly more solver iterations and is, both due to the increased number of iterations and due to requiring more matrix operations, slower than the ones that solve for velocity.

It should be noted that the Lamé decoupled and the Fully decoupled solvers solving for implicit acceleration show a computation time about twice as large as the ones that solve for implicit velocity. This is for the most part due to issues with garbage collection and just in time compilation, and would not be the case in an performance focused implementation of the methods. Thus the difference in computation time between the combined solvers is a fairer comparison.

The explicit solvers show quite similar results, although they lack some of the stiffness given by the implicit solvers, such that their particles have a tendency to be more spread, thus not producing as clean chunks of snow as the implicit solvers.

Choice of Methods As the solvers that solve for implicit acceleration are slower but produce the same result as the ones that solve for velocity, they will be left out of all the following results. As mentioned in Section 3.2.2, the decoupled solvers are not necessary when simulating snow that does not interact with other materials. The results also show that the combined solvers are the most performant, thus all the following results will only utilize the method of combined stress solved for implicit velocity and the method of combined explicit stress, which will further be referred to a implicit stress and explicit stress.

5.1.2 Comparison of Time Steps

The size of the time step used in an SPH simulation can greatly affect on the simulations correctness and stability. Thus this section compares a couple different time step sizes.

The setup for these results are exactly the same as the setup in Section 5.1.1, with the exception of the time step Δt.

Results The first thing to note in the results, Figure 5.3, is that the explicit method is unstable for this simulation at time steps larger than about 0.0005, while the implicit method remains stable at a time step as large as 0.001.

One can further see that while the explicit method is stable at a time step of 0.0005 it does not show a result that is in line with the expected result at smaller time steps. The implicit solver on the other hand, while still not showing a result close to the expected result at smaller time steps, it is more in line with

Implicit stress

(a) Time step: 0.001
Computation time: 14.25s
$\boldsymbol{a}_\sigma$-solver iterations: Min: 0, Max: 10, Avg: 6.24

(b) Time step: 0.0005
Computation time: 16.00s
$\boldsymbol{a}_\sigma$-solver iterations: Min: 1, Max: 4, Avg: 2.74

(c) Time step: 0.00001
Computation time: 474.11s
$\boldsymbol{a}_\sigma$-solver iterations: Min: 0, Max: 1, Avg: 0.58

Explicit stress

(d) Time step: 0.0005
Computation time: 4.00s

(e) Time step: 0.00001
Computation time: 179.55s

Figure 5.3: Comparison of various time step sizes.

the expected result, and one can assume that, given another simulation setup with lower velocities and smaller forces, it could be viable to use the implicit solver with larger time steps.

Choice of Time Step From the results in Figure 5.3, one can see that a time step of 0.00001 does not contribute much more correctness than a time step of 0.0001 as seen in Figure 5.2. Thus, for the rest of the generated results a time step of $\Delta t = 0.0001$ will be used.

5.1.3 Boundaries

As many properties of snow is related to how it accumulates on the ground, the boundary conditions of the simulation are very important. This section will thus drop a snow ball on a flat boundary ground to see how it interacts with it.

Setup In this simulation a snow ball with a radius of $9.375cm$ is placed such that its center is $18.75cm$ above a flat line of boundary particles. In order to avoid waiting for it to drop from higher, it is initialized with a downwards velocity of $5\frac{m}{s^2}$. This configuration can be seen in Figure 5.4, where the boundary particles can be seen drawn as hollow circles.

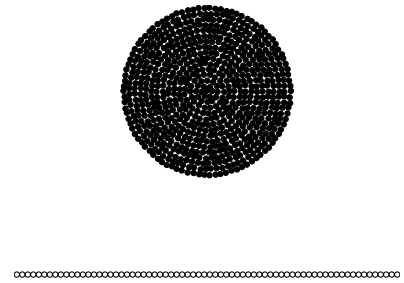
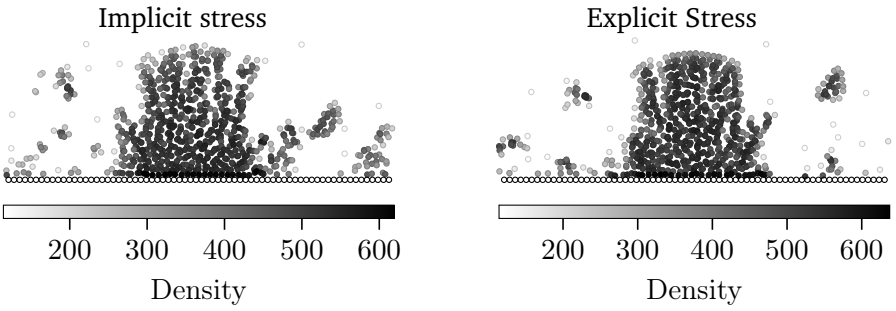


Figure 5.4: Initial configuration of snow ball dropped on ground.

The parameters of the snow particles are set exactly like in Table 5.1. The friction coefficient μ of the boundary particles is set to zero.

The simulation is run for a simulation time of $0.1s$, which is slightly after the snow ball has impacted the ground. Finally, the positions of all particles and their density is captured as the result.



(a) Computation time: 23.83s
$\boldsymbol{a}_\sigma$-solver iterations:
Min: 0, Max: 1, Avg: 0.81

(b) Computation time: 10.97s

Figure 5.5: Snow ball dropped on ground.

Results The results from this simulation can be seen in Figure 5.5, where boundary particles are drawn as hollow circles and the density of snow particles can be seen as a gray scale gradient.

The fact that snow particles lies flat with the boundary line indicates that the volume of the boundary particles is computed correctly.

With perfect boundary conditions, one should see a smooth density gradient through the snow and onto the boundary. In this situation the difference between

the density of the snow particles touching the boundary and the next layer of snow particles is about $50\frac{kg}{m^3}$ to $100\frac{kg}{m^3}$, which is not perfect, but is a quite smooth gradient.

5.1.4 Crack Formation

Although the initial goal of this thesis was to simulate slab avalanches, Section 2.1.3, configuring the simulation for a complete avalanche simulation turned out too strenuous. However, inspired by Gaume *et al.* [6] which uses the material in point method by Stomakhin *et al.* [10], in conjunction with their own implementation of weak layers, to simulate slab avalanches, this section and the following section simulates snow properties that are required for slab avalanches to form.

This section simulates a snow slab lying flat on the ground and shows its ability to form cracks when loosing support from below.

Setup This simulation places a snow slab, with a height of $17.23cm$ and a width of $41.57cm$, flat on a boundary ground. Then a, one particle wide, $15.83cm$ long, line of snow is removed from the bottom right corner of the slab. Furthermore, as the cracking mechanics of the snow is very dependent on the uniformity of the snow, the snow is made less uniform by randomly placing particles within a square with side length of $\frac{3\Delta s}{16}$ around their intended position. This results in the configuration which can be seen in Figure 5.6. One should however note that the figure does not show the left side of the slab as it is only used as support.

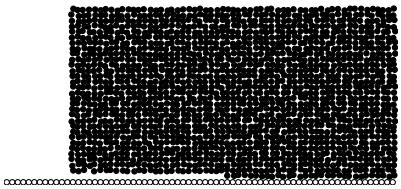


Figure 5.6: Initial configuration for crack formation.

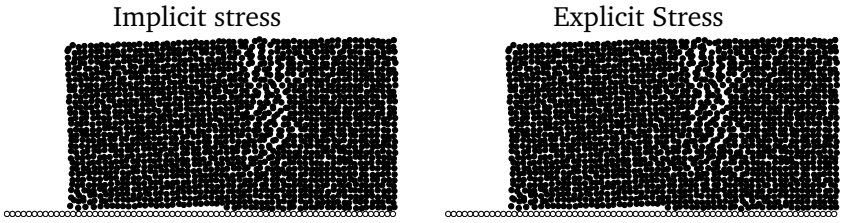
The snow particles in the slab are configured with the parameters shown in Table 5.2.

Table 5.2: Parameters for crack formation.

Parameter	Value
ρ_0	$400\frac{kg}{m^3}$
E	$600000Pa$
ν	0.2
ξ	20
θ_c	0.019
θ_s	0.0055
μ	$10\frac{N}{m^2}$

Inspired by Stomakhin *et al.* [10], the young modulus and the hardening coefficient is increased and the critical stress and stretch coefficients and decreased in relation to their recommended values, in order to produce a more solid piece of snow with a higher tendency to produce cracks. Lastly, the friction coefficient μ is set to a large value in order to avoid sliding which may affect the results.

The simulation is run for a simulation time of $0.1s$ and the position of all particles are captured to give the results.



(a) Computation time: 83.91s
α_σ-solver iterations:
Min: 1, Max: 6, Avg: 1.40

(b) Computation time: 28.84s

Figure 5.7: Crack formation in snow slab.

Results The results from the crack formation simulation can be seen in Figure 5.7. It shows that the section of the slab without support underneath experiences rotational momentum until its corner collides with the ground. This rotation subsequently makes a crack form through the whole slab.

This result indicates that given a simulation of a full scale avalanche, a fracture propagation through a weak layer could lessen the support on the slab making it crack. Being able to form cracks, and consequently breaking off slab blocks is the initial requirement for a slab avalanche to occur.

5.1.5 Slab Sliding on Weak Layer

While the aforementioned crack formation is the snow behavior that triggers slab avalanches in the first place. However, in order for them to travel down mountain sides, the slabs requires the ability to slide on a weak layer, see Section 2.1.3. Thus, inspired by Gaume *et al.* [6], this section simulates a snow slab on top of a weaker snow layer and the slabs ability to slide down a slope. As slab avalanches tends to occur in slopes of 30° to 45°, this simulation places the slab and weak layer on a slope of 37°.

Setup First a line of boundary particles is placed such that it has an angle of 37° from the horizontal axis. Then a weak snow slab with a height of $3.13cm$ and a width of $26.25cm$ is placed right on top of the boundary line with the same angle. Lastly a stronger slab with a height of $12.5cm$ and the same width as the weaker

slab is placed right on top of the weaker slab. This configuration can be seen in Figure 5.8.

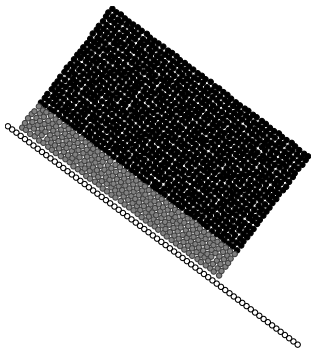


Figure 5.8: Initial configuration of slab sliding on weak layer.

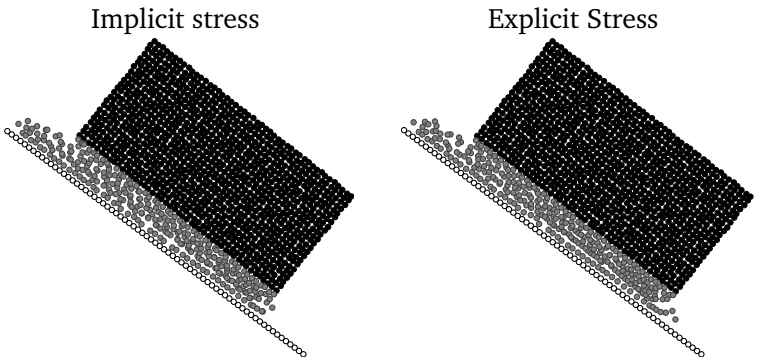
The parameters for strong snow slab is in a similar fashion to the simulation of crack formations. The rest density of the weak layer is set lower by changing the rest density used in the computations of the Lamé parameters, Equation (3.2) and Equation (3.3), and the weak layer is set to have a low young modulus and a higher Poisson's ratio in order to make the weak layer weaker and more granular, respectively. The friction coefficient is set to a high value in order to make sure that the sliding behavior is due to the weak layer and not a slippery boundary. The parameters for the simulation can be seen in Table 5.3.

Table 5.3: Parameters for slab sliding on weak layer.

Weak layer		**Slab**	
Parameter	**Value**	**Parameter**	**Value**
ρ_0	$100\frac{kg}{m^3}$	ρ_0	$400\frac{kg}{m^3}$
E	$10000Pa$	E	$600000Pa$
ν	0.3	ν	0.2
ξ	10	ξ	10
θ_c	0.025	θ_c	0.019
θ_s	0.0075	θ_s	0.0055
Boundary			
Parameter		**Value**	
μ		$10\frac{N}{m^2}$	

The simulation is run for a simulation time of $0.2s$ and the position of all particles are captured as the result.

Result The results of this simulation can be seen in Figure 5.9, in which the weaker layer is unable to hold the weight of the slab and the slab thus slides downhill. The slab as a whole ends up sliding $5.11cm$.



(a) Computation time: 89.30s
$\boldsymbol{a}_\sigma$-solver iterations:
Min: 1, Max: 1, Avg: 1.00

(b) Computation time: 37.43s

Figure 5.9: Slab sliding on weak layer.

While the previous result of crack formations show requirements for avalanches to initiate, this result is a requirement for the avalanche to further propagate. Given a simulation of a full scale avalanche and a mechanism to break off slab blocks, this result show that such a slab block would be able to slide down mountain sides and thus actually enable the full avalanche propagation.

It should further be mentioned that, while the recorded results does not show it, the weak layer in this simulation is in fact stable on its own, meaning it remains solid without the weight of the slab layer.

5.2 Parallel Results

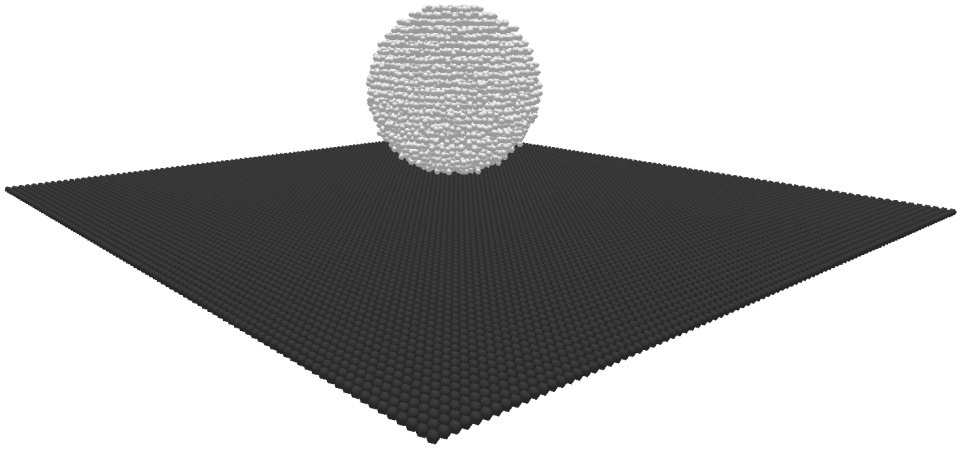
While the results for the serial implementation focus various aspect of the snow simulation and avalanche related snow properties, this section show results for the parallel implementation and focuses on its performance aspects.

5.2.1 Setup

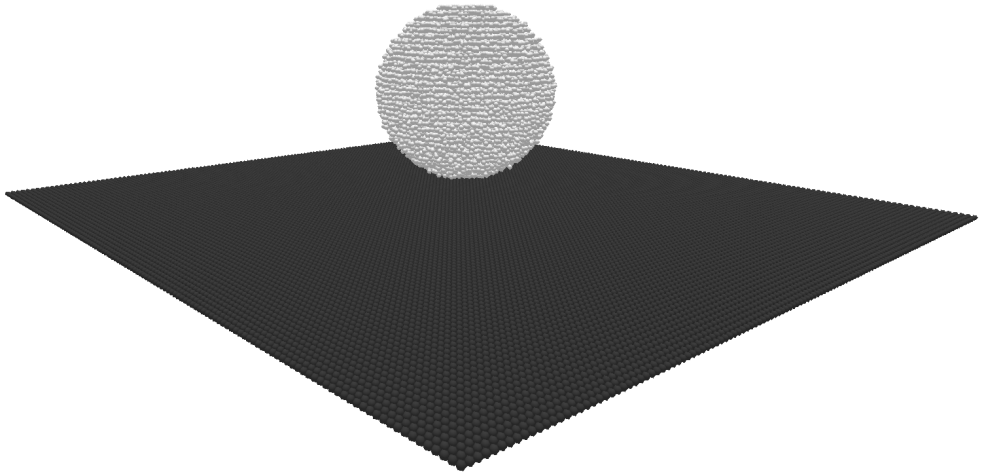
These results are almost identical to the snow ball dropped on the ground in the serial results, Section 5.1.3. A sphere of snow particles with a radius of $9.38cm$ is placed with its center $12.5cm$ above a flat boundary plane. The snow ball is initialized with a downwards velocity of $5\frac{m}{s}$. The initial configuration of the simulations can be seen in Figure 5.10.

Then all parameters of the simulation are set to the same values as in the serial snow ball drop results, with the exception of particle spacing and the friction coefficient. The friction coefficient is set to a higher values as to keep the snow more centered in the simulation. All parameter values can be seen in Table 5.4.

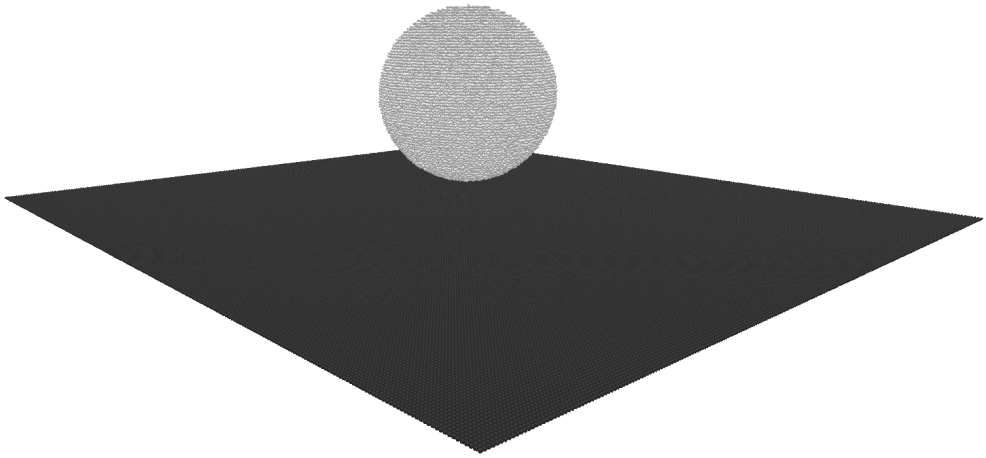
This section has the goal of looking at the performance of the simulation. The performance is entirely dependent on the amount of particles in the simulation



(a) $\Delta s = 7.576mm$



(b) $\Delta s = 5.682mm$



(c) $\Delta s = 2.851mm$

Figure 5.10: Initial configurations for parallel snow ball drop.

Table 5.4: Parameters for parallel snow ball drop.

Parameter	Value
Δ t	$0.0001s$
ρ_0	$400\frac{kg}{m^3}$
E	$140000Pa$
ν	0.2
ξ	10
θ_c	0.025
θ_s	0.0075
μ	4

and the amount of particles is entirely dependent on the particle spacing. Thus, the simulation is run three times with the particle spacings $0.007576m$, $0.005682m$ and $0.002851m$.

The simulations are all run for a simulation time of $0.1s$, which is right after the snow ball has impacted the ground. During each time step the various GPU kernels are timed and processed to give performance results.

Throughout the simulation the particles are all rendered as icosahedrons, which does to some degree affect the total run time of the simulation.

The simulations were performed on an Nvidia RTX 2080 Ti [47].

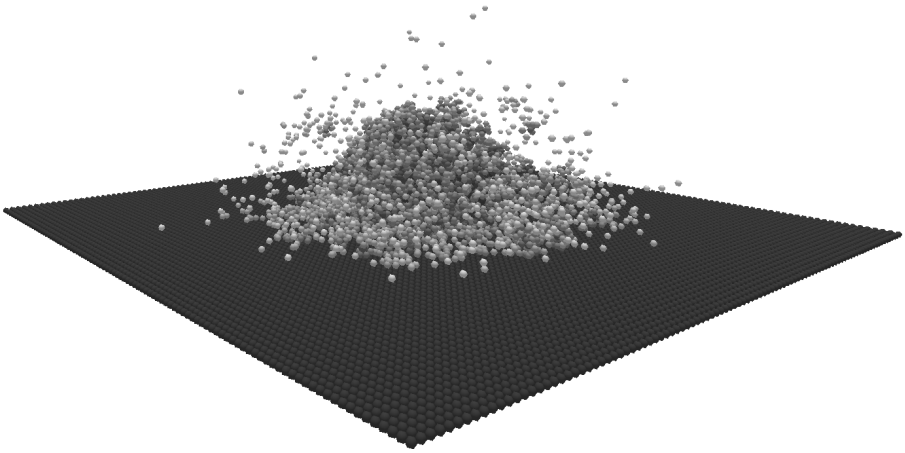
5.2.2 Results

The final results of running the simulations can be seen in Figure 5.11, where boundary particles are colored dark gray and the snow particles are colored in such a way that they are darker the higher density they have. One can see from these results that visualizing three dimensional particle clouds is difficult. They do however show similar results to the ones seen in Section 5.1.3.

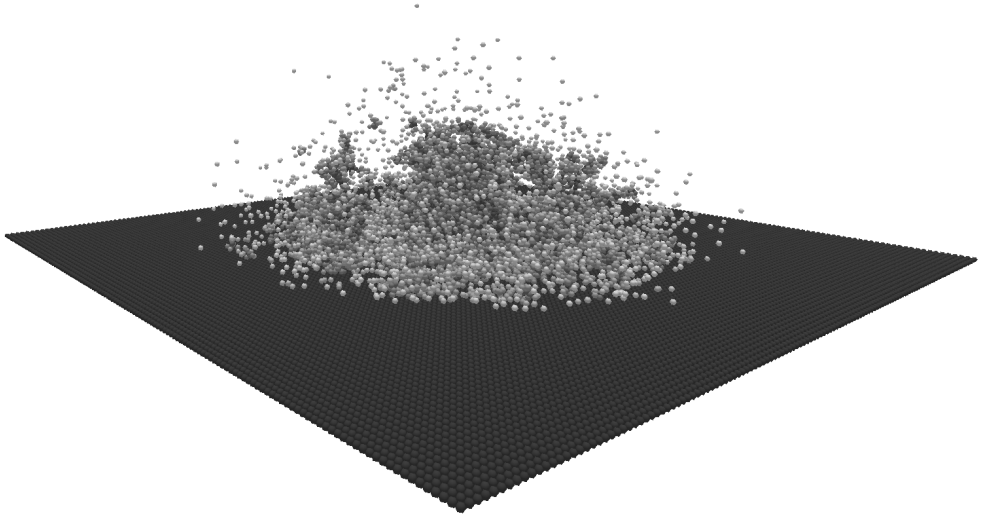
Recorded numbers and timings can furthermore be seen in Table 5.6. The kernel timings in this table are run time per time step, where the maximum, minimum and average kernel run time for a single time step has been recorded.

Memory Usage The memory usage in Table 5.6 is the allocated bytes on the GPU used by the simulation. One can first note that the memory usage of particles is strictly linearly dependent on the number particles. The memory usage of various data structures can be seen in Table 5.5.

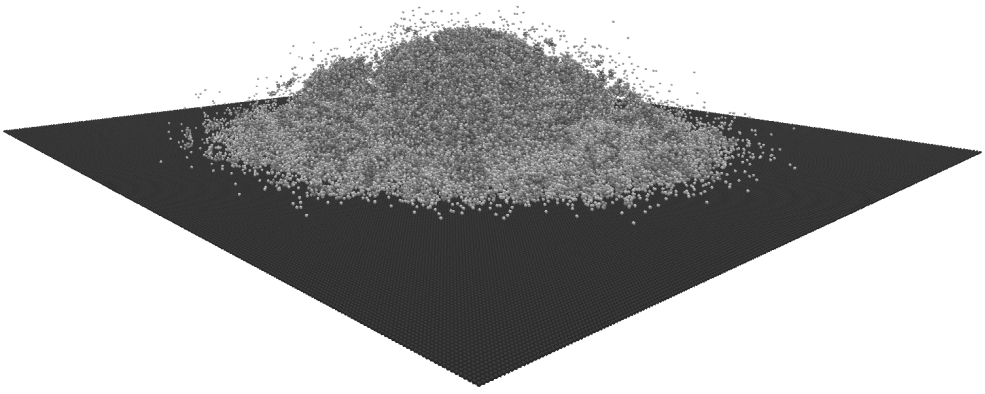
One can see that the number of snow particles and the number of grid cells grows inverse cubically when the particle spacing is made smaller. As the decreasing the particle spacing adds a computational cost similar to making the simulation domain larger, one can see that the memory usage of a simulation is expected to grow cubically with an increased resolution of the simulation, weather the resolution is increased due to a smaller particle spacing or an increased simulation domain.



(a) $\Delta s = 7.576mm$



(b) $\Delta s = 5.682mm$



(c) $\Delta s = 2.851mm$

Figure 5.11: Parallel snow ball drop.

Table 5.5: GPU memory usage of data structures.

Data structure	Memory usage
Snow particle	$3440B$
Boundary particle	$36B$
Grid cell	$8B$
Temporary snow particle	$72B$

While the number of snow particles grow cubically with a increased resolution, the number of boundary particles only grow quadratically, this is due to the fact that one only needs a shell of boundary particles along the boundary edge. Thus boundaries can be added to the simulation at a much lower cost compared to snow.

Table 5.5 shows a temporary snow particle. This temporary particle is used for copying persistent particle properties during the counting sort kernel. As the counting sort is performed each time step, this memory is kept allocated and add an additional memory usage per particle. It should be noted that such an temporary particle is also required when initializing the boundary particles, this can however be deallocated after initialization and is thus not shown here.

During the execution of the prefix sum kernels, one is required to store block sums and their prefix sums. This has an additional memory cost of $8\left\lfloor \frac{n}{b} \right\rfloor + 8$ bytes for each required call to the combine block kernel, where n is the number of block sum elements and b is the block size, see Section 4.2.1. This implementation only allows two calls to combine blocks, and uses a block size of 1024.

Timing It should first be noted that the total run time of the simulations, Table 5.6 is affected by rendering, thus the kernel timings are a better indicator of the performance of the simulation.

It should further be mentioned that the timings “Prefix sum and counting sort”, “Correction matrix and external forces” and “Integration” combine multiple kernels calls into one timed function. Furthermore, during the writing of this section it was noticed that the “Lamé parameters” kernel can and should definitely be a part of the preceding kernel.

The results for the prefix sum and counting sort kernels show that they are highly performant during all runs of the simulation and should not pose a problem for more complex simulations.

The results for the neighborhood pre-computation kernel show that it is the greatest bottle neck of the simulation. This was expected as it is the sole reason for pre-computing kernel values in the first place. By analyzing this kernel with the kernel profiler Nvidia Nsight Compute [48], it was found that the main discrepancy of this kernel is its ability to load particle positions from memory.

For all other kernels, the main performance limiting aspect seems to be matrix operations, as they are required for the computation of correction matrices, stress and the integration of the deformation gradient.

Table 5.6: Results from parallel snow ball drop.

	$\Delta s = 7.576mm$	$\Delta s = 5.682mm$	$\Delta s = 2.851mm$
Numbers			
Snow particles	7901	18656	150238
Boundary particles	9800	17423	69695
Grid cells	24389	59319	474552
GPU memory usage			
Memory usage	28.30MB	66.62MB	533.95MB
Total run time			
Run time	17.54s	28.17s	161.18s
Kernel run times			
Prefix sum and counting sort			
Min	0.09ms	0.10ms	0.33ms
Max	0.54ms	0.65ms	0.99ms
Avg	0.11ms	0.13ms	0.72ms
Neighborhood pre-computation			
Min	3.91ms	8.56ms	54.83ms
Max	7.07ms	14.29ms	92.36ms
Avg	4.81ms	10.14ms	64.61ms
Density, volume and rest density			
Min	0.25ms	0.36ms	4.56ms
Max	0.54ms	0.68ms	7.51ms
Avg	0.30ms	0.40ms	5.31ms
Lamé parameters			
Min	0.01ms	0.01ms	0.04ms
Max	0.01ms	0.03ms	0.05ms
Avg	0.01ms	0.01ms	0.04ms
Correction matrix and external forces			
Min	1.54ms	3.12ms	16.09ms
Max	3.36ms	4.76ms	19.84ms
Avg	2.60ms	3.88ms	17.28ms
Stress			
Min	5.19ms	6.80ms	42.36ms
Max	7.15ms	10.84ms	60.90ms
Avg	6.14ms	7.97ms	47.23ms
Integration			
Min	2.25ms	4.0ms	22.94ms
Max	3.16ms	9.05ms	27.20ms
Avg	2.62ms	4.50ms	23.93ms

Chapter 6

Conclusion and Future Work

This chapter concludes this thesis by first discussing what has been accomplished, both in terms of work and the results it has produces through the implementations that has been developed, before lastly describing certain aspects of snow simulation that was not explored during the work on this theses, but would be beneficial to see future work on.

6.1 Conclusion

This section discusses what has and has not been accomplished by this thesis, both in terms of the knowledge acquired during the research behind the thesis and the results the thesis has generated.

Research This thesis started out with an intention of re-implementing the method by Gissler *et al.* [7] and seen what kind of performance benefits one would get from using graphical processing units to parallelize the method. It however became apparent that implementing such a method requires a great amount of background knowledge. A large portion of the work behind this thesis has thus been allocated to understanding the mathematical basis behind both smoothed particle hydrodynamic and elastoplasticity. All of this research has accumulated into the chapters Chapter 2 and Chapter 3. This will hopefully be a decent starting point and reference for anyone attempting to continue work on snow simulation.

Simulation Results The initial goal of this thesis was to simulate full scale slab avalanches. This was unfortunately not accomplished as it turned out too difficult to construct a global simulation configuration that allows for all aspects inherent in a slab avalanche. This thesis has however shown that a snow simulation using smoothed particle hydro dynamics have the ability to model certain aspects of slab avalanches. The simulation results have shown that it is possible to model a solid slab of snow with the ability to form cracks and fracture into chunks of snow, which is the initial property a snow slab requires to enable the initiation of slab

avalanches. The simulation results have further shown that it is able to model a weaker layer of snow that exhibit a granular flow when compressed by a stronger slab layer above. This is the property required for slab avalanches to travel down mountain sides.

Performance Results The previous snow simulation work at the NTNU HPC-Lab has had the goal of implementing real time simulations, this was never the ambition of this thesis, but it was goal to see how close one could get with the performance benefits of using graphical processing units. The serial results from this thesis show that even with an implicit method, one is required to use quite small time steps in order to achieve desired results. The parallel results have further shown that, due to large amount of matrix operations and a high neighborhood search cost for areas of compact snow, simulating large amounts of snow as an elastoplastic material is very computationally costly, even for graphical processing units. Thus, due to the number of time steps one would have to compute and the computational cost of computing these time steps, achieving a real time simulating of slab avalanches with smoothed particle hydrodynamics seem unfeasible with current methods and hardware.

6.2 Future Work

This section discusses the many aspects the snow simulation that has yet to be explored and tries to point anyone continuing this work in a direction for where they could start their work.

Implicit Method As mentioned in Section 4.2, due to time limitations on writing this thesis, only the combined explicit method was implemented as a parallel method on graphics processing units. It is however a desire to implement implicit methods in the same fashion in order to investigate how well they would behave and in order to compare their performance with respect to explicit methods more properly.

Visualization As the results in Section 5.2 show, visualizing large amounts of particles is a difficult task, and the result tend to not properly show the aspects of the simulation one wants to investigate. With newer graphics processing units supporting real time ray tracing [49], it would be quite interesting to implement such a visualization for a snow simulation. Especially since snow has a high degree of sub surface scattering, which is impractical to model with classical real time visualization methods.

Snowfall One of the reasons Gissler *et al.* [7] mention for using smoothed particle hydrodynamics in their snow simulation method is its ability to model individual snow particles, which is required in order to simulate snow fall. As snow fall is

what builds up and enable the occurrence of avalanches, this would enable a more complete simulation of the whole life cycle of snow using only one method for simulating all aspects of snow. In order to make the snow interact with wind, one would however need to use one of the decoupled solvers described in Chapter 3.

Terrain Previous work on the HPC-Lab snow simulator has implemented terrains which are used as boundaries in other avalanche models. This implementation does however model boundaries more complexly, as it utilizes boundary particles. In order to fully integrate this implementation in the existing solution one has to develop a method for covering the terrain in a sheet of boundary particles.

Memory Localization The results in Section 5.2 show that the biggest bottle neck in the parallel implementation is the computation of kernel values between particles and their neighbors. This is largely due to costly memory access times when accessing particle positions. In order to mitigate this problem one could investigate the use of shared memory in the pre-computation kernel. However in order to better benefit from caching without the need of shared memory, one could employ the method of data oriented design [50] in order to localize the properties that are required for by one kernel.

Matrix Operations The results of Section 5.2 also show that the kernels with matrix operations have a large computational cost. Thus one would greatly benefit from improving the performance of these operations. As modern graphical processing units have hardware accelerated tensor operations [35], it would valuable to see weather or not utilizing such tensor operations could benefit the performance of matrix operations in the simulation.

Tailoring the Simulation for Avalanches While the goal of this thesis was to simulate avalanches, it essentially only simulates the elastoplastic aspects of snow. However, when Gaume *et al.* [6] implement their avalanche simulation, they develop a method specifically for modeling crack formations in weak snow layers. Thus it would be beneficial to look further into their work and investigate whether or not the simulating, as implemented by this thesis, is able to model similar phenomenons, or if one would have to develop a more avalanche specific implementation.

Bibliography

[1] I. Saltvik, A. Elster and H. Nagel, 'Parallel methods for real-time visualization of snow', Jun. 2006, pp. 218–227, ISBN: 978-3-540-75754-2. DOI: 10.1007/978-3-540-75755-9_27.

[2] R. Eidissen, 'Utilizing gpus for real-time visualization of snow', 2009.

[3] Ø. E. Krog, 'Gpu-based real-time snow avalanche simulations', 2010.

[4] I. A. H. Sandvik, 'Adding gpu-accelerated real-time sph-based avalanche simulations to the ntnu hpc-lab snow simulator', 2021.

[5] Ø. L. Boge, 'Avalanche simulations using fracture mechanics on the gpu', 2014.

[6] J. Gaume, T. Gast, J. Teran, A. van Herwijnen and C. Jiang, 'Dynamic anticrack propagation in snow', *Nature Communications*, vol. 9, Aug. 2018. DOI: 10.1038/s41467-018-05181-w.

[7] C. Gissler, A. Henne, S. Band, A. Peer and M. Teschner, 'An implicit compressible sph solver for snow simulation', *ACM Trans. Graph.*, vol. 39, no. 4, Jul. 2020, ISSN: 0730-0301. DOI: 10.1145/3386569.3392431. [Online]. Available: https://doi.org/10.1145/3386569.3392431.

[8] J. Stam, 'Stable fluids', *ACM SIGGRAPH 99*, vol. 1999, Nov. 2001. DOI: 10.1145/311535.311548.

[9] S. Green, *Particle simulation using cuda*, 2010.

[10] A. Stomakhin, C. Schroeder, L. Chai, J. Teran and A. Selle, 'A material point method for snow simulation', *ACM Trans. Graph.*, vol. 32, no. 4, Jul. 2013, ISSN: 0730-0301. DOI: 10.1145/2461912.2461948. [Online]. Available: https://doi.org/10.1145/2461912.2461948.

[11] R. A. Gingold and J. J. Monaghan, 'Smoothed particle hydrodynamics: theory and application to non-spherical stars', *Monthly Notices of the Royal Astronomical Society*, vol. 181, no. 3, pp. 375–389, Dec. 1977, ISSN: 0035-8711. DOI: 10.1093/mnras/181.3.375. eprint: https://academic.oup.com/mnras/article-pdf/181/3/375/3104055/mnras181-0375.pdf. [Online]. Available: https://doi.org/10.1093/mnras/181.3.375.

[12] N. Akinci, M. Ihmsen, G. Akinci, B. Solenthaler and M. Teschner, 'Versatile rigid-fluid coupling for incompressible sph', *ACM Trans. Graph.*, vol. 31, no. 4, Jul. 2012, ISSN: 0730-0301. DOI: 10.1145/2185520.2185558. [Online]. Available: https://doi.org/10.1145/2185520.2185558.

[13] S. Band, C. Gissler, M. Ihmsen, J. Cornelis, A. Peer and M. Teschner, 'Pressure boundaries for implicit incompressible sph', *ACM Trans. Graph.*, vol. 37, no. 2, Feb. 2018, ISSN: 0730-0301. DOI: 10.1145/3180486. [Online]. Available: https://doi.org/10.1145/3180486.

[14] A. Peer, C. Gissler, S. Band and M. Teschner, 'An implicit sph formulation for incompressible linearly elastic solids', *Computer Graphics Forum*, vol. 37, no. 6, pp. 135–148, 2018. DOI: https://doi.org/10.1111/cgf.13317. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/cgf.13317. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1111/cgf.13317.

[15] M. Chunnoo. (2022). Smoothed particle hydrodynamics, [Online]. Available: https://github.com/chunnoo/sph (visited on 20/07/2022).

[16] T. Geldsetzer and B. Jamieson, 'Estimating dry snow density from grain form and hand hardness', Jan. 2000.

[17] N. Calonne, C. Geindreau, F. Flin, S. Morin, B. Lesaffre, S. Rolland du Roscoat and P. Charrier, '3-d image-based numerical computations of snow permeability: Links to specific surface area, density, and microstructural anisotropy', *Cryosphere*, vol. 6, pp. 939–951, Sep. 2012. DOI: 10.5194/tc-6-939-2012.

[18] J. Glen, 'W. s. b. paterson 1994. the physics of glaciers. 3rd edition. oxford, etc., pergamon, 480 pp. isbn 0-08037945 1. hardback. £70; 0-o8o37944 3, flexieover. £25.', *Journal of Glaciology*, vol. 43, no. 145, pp. 594–594, 1997. DOI: 10.3189/S002214300003522X.

[19] K. Müller, H. T. Larsen and G. Sojer, 'Snøomvandling', 2020.

[20] G. A. of the European Avalanche Warning Service, 'Typical avalanche problems', 2017.

[21] N. vassdrag- og energidirektirat. (2022). Snøskredskolen, [Online]. Available: https://varsom.no/snoskred/snoskredskolen/ (visited on 14/07/2022).

[22] H. Norem, 'Veger og snøskred', 2011.

[23] A. Hodges, *Alan Turing: The Enigma*. Vintage, 1992, ISBN: 9780099116417. [Online]. Available: https://books.google.no/books?id=VWvPIWm75XIC.

[24] D. Nuentsa-Wakam and G. Guennebaud, *Eigen*, https://gitlab.com/libeigen/eigen/-/blob/master/Eigen/src/IterativeLinearSolvers/BiCGSTAB.h, 2014.

[25] J. Monaghan, 'Smoothed particle hydrodynamics', *Reports on Progress in Physics*, vol. 68, p. 1703, Jul. 2005. DOI: 10.1088/0034-4885/68/8/R01.

[26] J. J. Monaghan, 'Smoothed particle hydrodynamics', *Annual Review of Astronomy and Astrophysics*, vol. 30, no. 1, pp. 543–574, 1992. DOI: 10.1146/annurev.aa.30.090192.002551. eprint: https://doi.org/10.1146/annurev.aa.30.090192.002551. [Online]. Available: https://doi.org/10.1146/annurev.aa.30.090192.002551.

[27] D. Koschier, J. Bender, B. Solenthaler and M. Teschner, 'Smoothed particle hydrodynamics techniques for the physics based simulation of fluids and solids', *CoRR*, vol. abs/2009.06944, 2020. arXiv: 2009.06944. [Online]. Available: https://arxiv.org/abs/2009.06944.

[28] J. Bonet and T.-S. Lok, 'Variational and momentum preservation aspects of smooth particle hydrodynamic formulations', *Computer Methods in Applied Mechanics and Engineering*, vol. 180, no. 1, pp. 97–115, 1999, ISSN: 0045-7825. DOI: https://doi.org/10.1016/S0045-7825(99)00051-1. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045782599000511.

[29] J. Niiranen, 'Fast and accurate symmetric euler algorithm for electromechanical simulations note: The method became later known as "symplectic euler"', Sep. 1999. DOI: 10.13140/2.1.4663.0080.

[30] M. Shadloo, G. Oger and D. Le Touzé, 'Smoothed particle hydrodynamics method for fluid flows, towards industrial applications: Motivations, current state, and challenges', *Computers & Fluids*, vol. 136, pp. 11–34, 2016, ISSN: 0045-7930. DOI: https://doi.org/10.1016/j.compfluid.2016.05.029. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0045793016301773.

[31] S. Band, C. Gissler, A. Peer and M. Teschner, 'Mls pressure boundaries for divergence-free and viscous sph fluids', *Computers & Graphics*, vol. 76, pp. 37–46, 2018, ISSN: 0097-8493. DOI: https://doi.org/10.1016/j.cag.2018.08.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S009784931830116X.

[32] D. Sulsky, S.-J. Zhou and H. L. Schreyer, 'Application of a particle-in-cell method to solid mechanics', *Computer Physics Communications*, vol. 87, no. 1, pp. 236–252, 1995, Particle Simulation Methods, ISSN: 0010-4655. DOI: https://doi.org/10.1016/0010-4655(94)00170-7. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0010465594001707.

[33] J. Anderson, *Computational Fluid Dynamics*, ser. Computational Fluid Dynamics: The Basics with Applications. McGraw-Hill Education, 1995, ISBN: 9780070016859. [Online]. Available: https://books.google.no/books?id=dJceAQAAIAAJ.

[34] Google. (2017). An in-depth look at google's first tensor processing unit (tpu), [Online]. Available: https://cloud.google.com/blog/products/ai-machine-learning/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu (visited on 14/07/2022).

[35] Nvidia, *Nvidia turing gpu architecture*, 2018.

[36] T. N. Theis and H.-S. P. Wong, 'The end of moore's law: A new beginning for information technology', *Computing in Science & Engineering*, vol. 19, no. 2, pp. 41–50, 2017. DOI: 10.1109/MCSE.2017.29.

[37] M. Flynn, 'Flynn's taxonomy', in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA: Springer US, 2011, pp. 689–697, ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_2. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4%5C_2.

[38] Nvidia. (2022). Cuda toolkit documentation v11.7.0, [Online]. Available: https://docs.nvidia.com/cuda/ (visited on 18/07/2022).

[39] Nvidia, *Nvidia's next generation cuda compute architecture: Fermi*, 2009.

[40] N. Akinci, G. Akinci and M. Teschner, 'Versatile surface tension and adhesion for sph fluids', *ACM Trans. Graph.*, vol. 32, no. 6, Nov. 2013, ISSN: 0730-0301. DOI: 10.1145/2508363.2508395. [Online]. Available: https://doi.org/10.1145/2508363.2508395.

[41] 'Ieee standard for binary floating-point arithmetic', *ANSI/IEEE Std 754-1985*, pp. 1–20, 1985. DOI: 10.1109/IEEESTD.1985.82928.

[42] J. Blinn, 'Consider the lowly 2 x 2 matrix', *IEEE Computer Graphics and Applications*, vol. 16, no. 2, pp. 82–88, 1996. DOI: 10.1109/38.486688.

[43] G. Golub, C. Van Loan, P. Van Loan and C. Van Loan, *Matrix Computations*, ser. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, 1996, pp. 257–258, ISBN: 9780801854149. [Online]. Available: https://books.google.no/books?id=mlOa7wPX6OYC.

[44] R. E. Ladner and M. J. Fischer, 'Parallel prefix computation', *J. ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980, ISSN: 0004-5411. DOI: 10.1145/322217.322232. [Online]. Available: https://doi.org/10.1145/322217.322232.

[45] K. Wu, *3x3_svd_cuda*, https://github.com/kuiwuchn/3x3_SVD_CUDA, 2018.

[46] A. McAdams, A. Selle, R. Tamstorf, J. Teran and E. Sifakis, 'Computing the singular value decomposition of 3x3 matrices with minimal branching and elementary floating point operations', 2011.

[47] Nvidia. (2022). Geforce rtx 2080 ti, [Online]. Available: https://www.nvidia.com/en-me/geforce/graphics-cards/rtx-2080-ti/ (visited on 19/07/2022).

[48] Nvidia. (2022). Nvidia nsight compute, [Online]. Available: https://developer.nvidia.com/nsight-compute (visited on 19/07/2022).

[49] Nvidia, *Nvidia ampere ga102 gpu architecture*, 2021.

[50] N. Llopis. (2009). Data-oriented design (or why you might be shooting yourself in the foot with oop), [Online]. Available: `https://gamesfromwithin.com/data-oriented-design` (visited on 19/07/2022).

Mathias Chunnoo

Simulating Snow as an Elastoplastic Material on the GPU

NTNU
Norwegian University of
Science and Technology