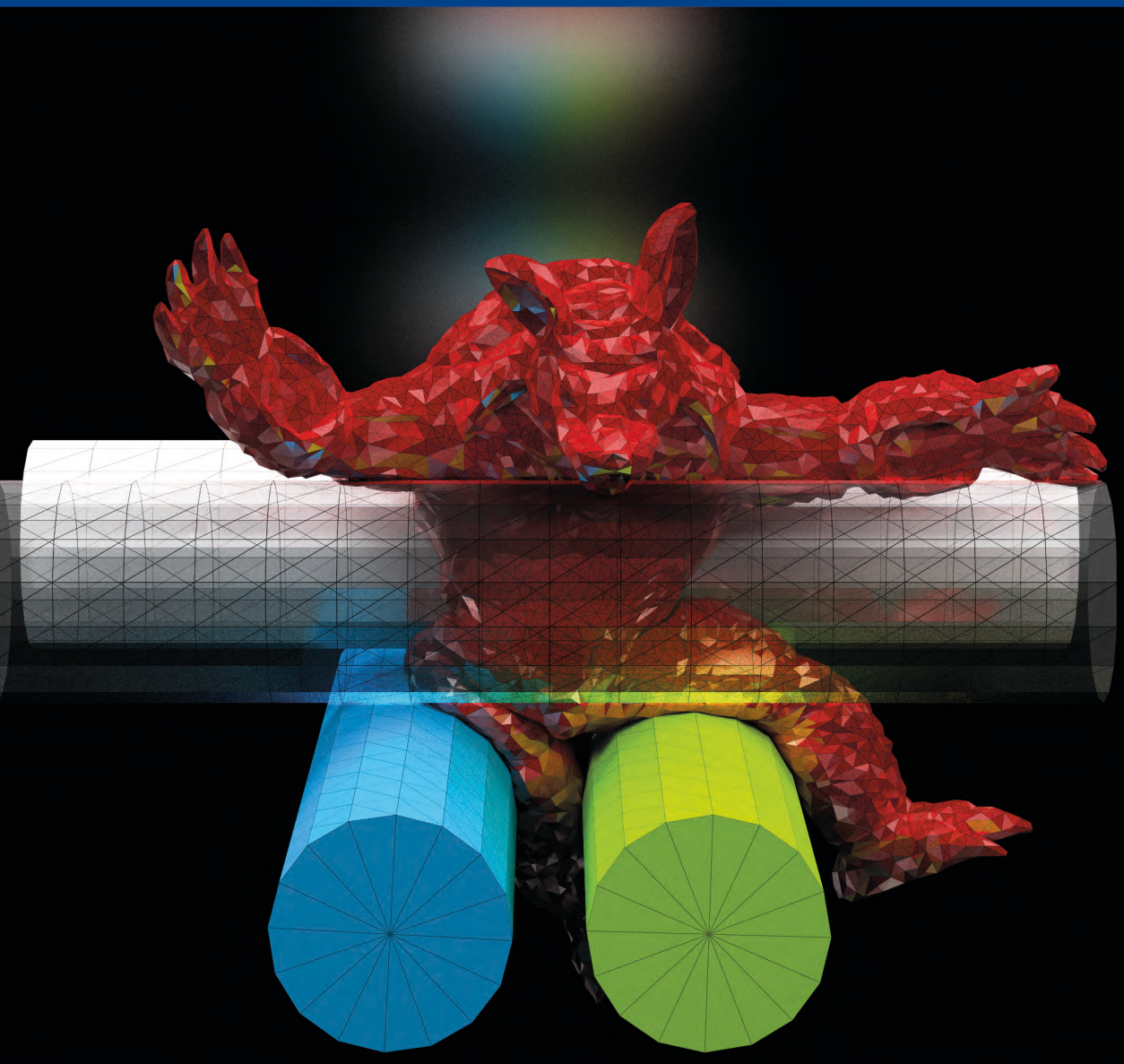
Simulation and Animation of Deformable Solids



Mickeal Verschoor

Simulation and Animation of Deformable Solids

Mickeal Verschoor

Simulation and Animation of Deformable Solids

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof. dr. ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties, in het openbaar te verdedigen op maandag 27 mei 2019 om 16:00 uur

door

Mickeal Verschoor

geboren te Dordrecht

Dit proefschrift is goedgekeurd door de promotoren en de samenstelling van de promotiecommissie is als volgt:

voorzitter:	
promotor:	prof. dr. ir. J.J. van Wijk
copromotor:	dr. A.C. Jalba
leden:	prof. dr. E. Eisemann (Technische Universiteit Delft)
	dr. M.E. Hochstenbach
	dr. M.A. Otaduy (Universidad Rey Juan Carlos)
	prof. dr. I.S. Pop (Universiteit Hasselt)
	prof. dr. A.C. Telea (Rijksuniversiteit Groningen)

Het onderzoek of ontwerp dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

Voor mijn moeder Ineke

Cover design: Brenda Razenberg
Cover image: Rendering of frame 6000 of the experiment shown in Figure 5.1
Plots: Gnuplot
Diagrams: Adobe Illustrator & Sketch (`http://sketch4latex.sourceforge.net`)
Renderings: Bidirectional path tracer, by Mickeal Verschoor
Chapter image: Triangulation of another peak, by Brenda Razenberg
Typesetting: LaTeX
Printed by: Gildeprint, Enschede

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN 978-90-386-4736-4

TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Additional videos and code are available at `http://www.mverschoor.nl`.

An electronic version of this dissertation is available at `http://repository.tue.nl`.

Preface

This dissertation is the result of a long period of learning, researching, programming and debugging, that started before I even decided to follow a Master in computer science. I was only a teenager when I was already attracted to the beauty of computer graphics, but to me this was more like a hobby than something you could study. Many years later, I rediscovered this interest when I was a student at the University of Applied Sciences in Utrecht. There I followed with great interest courses on computer graphics and rendering methods, and I was advised to continue with a Master in this field. Through courses I followed as a Master student at the University of Groningen, I met my future copromotor dr. Andrei C. Jalba. Under his guidance I participated in an internal project on deformable-model simulations. Eventually, this led to a serious interest in doing a PhD. One day I was introduced to my future promotor prof. dr. ir. Jarke J. van Wijk, who is leading the Visualization group at Eindhoven University of Technology, and we talked about being a PhD candidate. I realized that being a PhD candidate is a unique opportunity to deepen your knowledge on a few subjects. Now that this dissertation is done, I can say that being passionate about your work is also important when you are doing a PhD. I can also say that I deeply enjoyed my time as a PhD candidate. A decision I do not regret at all. Along this path I have learned a lot, deepened my knowledge, developed skills and met various people who helped and supported me in reaching this goal. In the following I want to thank everybody who played a direct or indirect role during this journey.

First and foremost, I want to thank my promotor prof. dr. ir. Jarke J. van Wijk for giving me the opportunity to do a PhD in his group. Jack, thank you for all your help and advice. Although my work was not in the group's core subject, visualization, you helped me a lot with your insight, ideas, optimism, and sharp and to the point feedback. Especially on this dissertation. It was a real honor to have you as my promotor.

Second, I want to thank my copromotor dr. Andrei C. Jalba for his daily supervision. Andrei, thank you for asking me to do a PhD and for all your help during my time as a PhD candidate. Especially for helping me with my research, writing, proofreading my work and pushing the boundaries of our field. I enjoyed working with you and I have learned many things from you during my time as a PhD candidate.

I would like to thank prof. dr. Elmar Eisemann, dr. Michiel E. Hochstenbach, dr. Miguel A. Otaduy, prof. dr. Iuliu Sorin Pop and prof. dr. Alexandru C. Telea for accepting the invitation to be part of the PhD committee, participate in the defense ceremony and provide valuable feedback on my dissertation. Additionally, I want to thank Michiel Hochstenbach for his thorough feedback on the draft of this dissertation. Furthermore, I want to thank Miguel A. Otaduy for inviting me to join his research group at Universidad Rey Juan Carlos in Madrid. Although it was not easy to leave everything behind, I am happy that we made the decision to move to Madrid. Miguel, thank you for your additional advice and giving me the possibility to attend SIGGRAPH.

The PhD journey is for every PhD candidate different and in general it is not a straight line. During my time as a PhD candidate it was always good to have fellow PhD candidates around that have different paths with similar difficulties. On this path I have met a great group of people. Thank you Dennie Reniers, Danny Holten, Niels Willems, Yedendra Shrinivasan, Jing Li, Kasper Dinkla, Stef van den Elzen, Roeland Scheepens, Martijn van Dortmont, Paul van der Corput, Bram Cappers, Alberto Corvò, Humberto Garcia Caballero, Dennis Dingen and Dennis Collaris for all the fun and (serious and non-serious) conversations we had during lunches, drinks, diners and other events. Roeland, Kasper and Stef, thank you for sharing your need for coffee with me. I'm happy that I can say that my progress bar is now finally complete. Additionally, I want to thank my other colleagues Michel Westenberg, Huub van de Wetering, Robert van Liere, Romain Bourqui, Mark de Berg, Herman Haverkort, Kevin Buchin, Maike Buchin, Dirk Gerrits, Elisabeth Melby and Meivan Cheng, at the TU/e VIS/ALG group for the nice environment, collaborations during the design based learning courses, and (in)direct support.

After leaving Eindhoven for Madrid, my colleagues at Universidad Rey Juan Carlos helped me with many daily things that are usually easy to arrange, but difficult to do when you are not fluent in Spanish. Thank you Angela Mendoza, Jorge Lopez, David Miraut and Dan Casas for all help regarding social security, tax and healthcare issues, and providing a nice atmosphere in the group. Additionally, I would like to thank colleagues Alvaro G. Perez, Alberto Sánchez, José San Martín, Dirk J. Lehmann, Alejandro Rodríguez, Carlos Aliaga, Elena Garces, Carlos Garre, Eder Miguel, Steven Sinclair, Jesús Pérez, Micah Davis, Joana Garrido, Jessica Illera, and PhD students Juan José Casafranca, Daniel Lobo, José Ángel Canabal, Rosell Torres, Rosa Sánchez, Héctor Barreiro, Alberto Martín, Raquel Vidaurre, Igor Santesteban, Javier Tapia, Kate Kardash, Cristian Romero, Christos Koutras, Zhongyun He for their support, language exchange and discussions on various topics during lunches and other events. Juanjo, also thank you for your advice on non-linear elasticity models.

The decision to do a PhD was not made if I did not had the opportunity to get in touch with the beauty of computer graphics in the first place. Thank you Gibby Koldenhof for your enthusiastic courses on computer graphics and rendering methods, and motivating me to continue with a Master after obtaining my Bachelor of Engineering.

Natuurlijk kan ik mijn ouders niet vergeten. Pa en Ma, bedankt voor jullie onvoorwaardelijke steun en liefde die ik van jullie heb mogen ontvangen voor alle keuzes die ik heb gemaakt. Ma, je was mijn grootste fan van alles wat ik deed. Wat was je zo trots toen ik mijn Master aan de universiteit had gehaald en ik als promovendus begon. Helaas heb je het eindresultaat niet meer mee kunnen maken, maar diep in mijn hart weet ik dat je er al die tijd voor mij was. Pa, dit boekje is hetgeen waar ik de afgelopen jaren aan heb gewerkt. Maar ik denk dat de interesse in computers en informatica al jaren geleden bij ons thuis is gewekt. Weet je nog, die ene keer dat wij samen thuis aan het programmeren waren?

Verder wil ik mijn broer Raymond, zus Nathalie, verdere familie en al mijn vrienden bedanken voor de nodige afleiding en ondersteuning in de afgelopen tijd. De boog kan niet altijd gespannen staan en daarom heb ik veel genoten van de vakanties, weekendjes weg, bezoeken aan concerten en festivals, of door gewoon ergens wat te gaan eten/drinken. De volgende keer in Madrid?

Lieve Brenda, nu ben jij eindelijk aan de beurt in mijn dankwoord. Brenda, ik weet niet hoe ik jou kan bedanken voor alle steun en onvoorwaardelijke liefde die je me de afgelopen jaren hebt gegeven. Het valt niet mee om een vriendin/vrouw te zijn van iemand die aan het promoveren is. Laat staan dat hij er ook nog een full-time baan naast heeft. Ik moet zeggen dat ik je nog heel wat verschuldigd ben voor de tijd dat ik aan het werk was voor mijn onderzoek en dit proefschrift. Tijd die ik niet met jou heb door kunnen brengen. Bedankt dat je met mij het avontuur bent aangegaan om naar Madrid te verhuizen. Bedankt voor het maken van de cover van dit proefschrift en het 'wijzen' op 'de minnetjes' in mijn code ;-) Je bent geweldig!

For everybody I failed to mention, thank you!

Mickeal Verschoor
Madrid, March 2019

Contents

1

Introduction

1.1 Motivation

Since the early 1960's (Sutherland [172]) computers are used to generate all kind of images, ranging from technical *Computer Aided Design* (CAD) images, to photo-realistic images for product presentations, games and movies. In this process, the geometric descriptions of objects in a scene are transformed given the properties of a virtual camera. These transformed geometric descriptions are then projected on the image plane. By using additional properties of the objects' surfaces and lights in the scene, shading effects can be added to the objects' projections to obtain realistic images. This process is known as *Rendering*. An important application of rendering is computer animation for, e.g., games and motion pictures. In these applications the objects' geometric descriptions are dynamic and can be controlled by the player who is playing a game, or the animator who is creating an animation for a motion picture. The animated objects and their motions can vary much, ranging from walking characters and flocking of large groups of animals, to motions of fluids, cloth, hair and snow. An important motion type is induced by physics laws, for example, Newton's laws of motion, friction, gravity, viscosity, elasticity, etc. Animating this kind of phenomena can be challenging. Standard key-frame and interpolation techniques are not able to produce the underlying physical behavior of the animated objects or phenomena in an easy way. Handcrafting the animation is usually very difficult since the number of objects and degrees of freedom can be very large. Therefore, a better approach to animate such objects and phenomena is to use mathematical models of the underlying physics laws in combination with numerical simulations to compute the objects' motions. However, this is a difficult problem to solve, especially if the motion and shape of the objects are (in)directly influenced by motions and deformations of other objects through contact and friction.

1.2 Simulation of Deformable Bodies

The simulation process found in many computer animation applications can be described using Figure 1.1. A *scene* consists of a set of objects, including their current positions. The current velocities are maintained in a separate *velocity state*. Each object can interact with the environment and other objects through contact and friction, or using other external forces and constraints. The scene is rendered and shown to the user, also, the user can interact with the objects and the environment, resulting in additional forces and constraints. The new state of a scene after a time-step is obtained as follows. First, internal forces and dynamics are computed using, e.g., the *Finite Element Method* (FEM) or *Rigid Body Dynamics* (RBD). These forces, additional external forces and constraints are collected in the *Force/Constraint state*. This provides the input for a *Time integrator*, which computes a new *velocity state*. To this end, a system of Partial Differential Equations (PDEs) is derived, which is approximated and numerically solved using a *Solver*. This process is often computationally demanding. Therefore (parts of) the solver can be executed on modern GPUs and CPUs, having highly parallel architectures. Finally, the new positions in the scene are calculated by integrating the updated velocities.

In general, this recipe for simulations needs to be accompanied with additional requirements. Depending on the type of application, particular requirements are more important than others. For simulations we consider the following requirements:

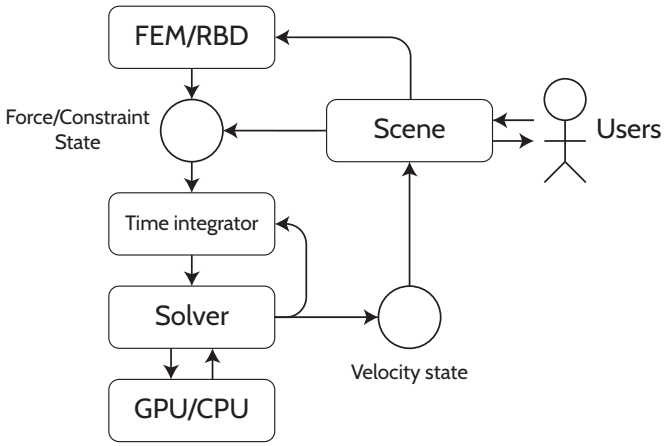


Figure 1.1: The simulation process.

- **Stability:** Stability is related to the feedback loop that is present in simulations. Because Newton's second law of motion is time integrated using a time-integration method, and each time-integration method introduces small truncation errors that depend on the time-step size, truncation errors may accumulate. As long as these errors are not amplified by the time-integration method, the simulation behaves stable. When these errors are amplified, the simulation becomes unstable, the results are unreliable and usually do not look realistic. One approach to increase the stability is to reduce the time-step size in case of explicit time-integration methods, or use implicit time-integration methods that are unconditionally stable. When implicit methods are used, the problem to solve is usually more complex, demanding more computational resources.
- **Accuracy:** Accuracy is related to the computation of certain quantities describing the state of a simulation, for example, the computation of the forces, collisions and the new velocity state. In general, the more accurate approximation of the solution is needed, the more computations are required. If the accuracy of a simulation is set too low, the motions and deformations may not look realistic. When the problem is not solved with a certain accuracy, some residual energy may remain in the system. In case of objects in contact, this residual energy can cause instabilities in the motions of the involved objects and might cause oscillations that negatively affect the realism.
- **Efficiency:** Depending on the required stability and accuracy, a simulation can be more computationally demanding. This affects the running time of the simulation. The running time of the simulation is also affected by the *computational efficiency* of the solver method, which is related to the *complexity* of the methods and operations used by the solver. If the complexity of the simulation increases due to external factors, the computation time should not dramatically increase. If the solver method is not efficient for solving a particular problem, often a reduced form or an approximation of the problem is used, in order to meet the time-constraints.

Contact Accurately simulating contact between deformable (and rigid) objects is a very challenging task that requires accurate and efficient methods. An accurate computation of friction can dramatically increase the realism of the simulation. If friction is not treated accurately, even simple examples may fail and result in non-realistic / non-physical motions. Friction is a complex phenomenon to deal with, because of the tight coupling between the degrees of freedom of the objects in contact. The motion of an object influences the motion of the other objects that are in contact with the former. On top of that, friction forces are tightly coupled with contact forces. A small change in the object motion can result in a small change in the contact-forces, which in turn affects the dissipation of energy through friction. This chain of force- and motion-propagation is very difficult to simulate and is usually approximated, which could result in stability problems. When deformable objects are involved, also their deformation influences contact and friction forces. Due to the tight coupling, such problems are typically non-smooth and non-linear and are therefore hard to solve accurately and efficiently. On top of that, such systems are inherently chaotic, meaning that the final simulation results are very sensitive to changes in initial conditions, rounding errors and the order of computations in case of non-deterministic parallel computations.

1.3 Objective

The main research question addressed by this dissertation is as follows:

"How can we accurately and efficiently simulate rigid and deformable solids that can collide, in a fast and stable way for computer animation applications?"

Simulations used in animations do not necessary need to be fast but accurate, while the same simulation used in interactive applications like *Virtual Reality* (VR) and Gaming require high simulation rates where a smaller accuracy is allowed. All applications require stability and need to be performed efficiently. The computational complexity of existing methods for simulating contact between rigid objects prevents us to directly use them for deformable solids. In this dissertation we develop methods that aim for a faster execution time by exploiting parallelism. We also aim to remove the computational bottlenecks by re-formulating the underlying problem.

In order to answer the research question above we subdivided it in four parts:

- **Parallel acceleration:** *How can we accelerate both numerical methods and simulation methods such that we can solve/simulate large problems in a short amount of time using parallel hardware?* Parallel hardware, such as (Multiple) GPUs, provides us with a tremendous amount of computational resources. Given the hardware, the goal is thus to use as many resources as possible to solve the problem. How to achieve this?
- **Performance analysis:** *How can we provide tools for reasoning about the efficiency of (numerical) methods performed on parallel hardware?* The problems solved using parallel hardware can vary in size and complexity and may or may not be able to use all computational resources. What are the performance limitations and how do they affect the efficiency of the hardware?

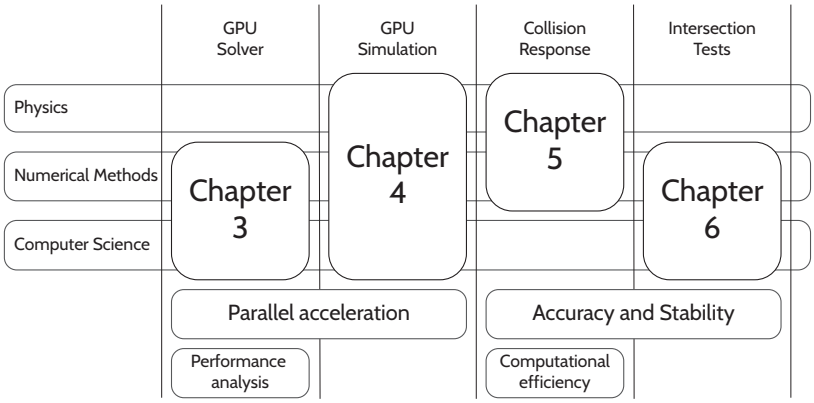


Figure 1.2: Overview of this dissertation.

- **Computational efficiency:** *How can we simulate coupled simulations in an efficient way?* Coupled simulations consider different objects or materials interacting with each other, through a direct or indirect coupling. Due to this coupling, the computational complexity of a simulation can significantly increase for certain problems. How can we make such simulations more efficient?
- **Accuracy and stability:** *How can we increase the accuracy and stability of (coupled) simulations?* In such simulations, the coupling should be performed in an accurate way. Usually coupling is based on physical properties, like contact and friction. How can we treat, e.g., contact and friction in an accurate way? Closely related to this is the stability of a simulation.

Furthermore, we can distinguish three main research areas that are involved in physics-based animation and are also closely related to the research question, namely:

- Physics
- Numerical Mathematics/Analysis
- Computer Science

Each of the main chapters in this dissertation aims to address a few aspects of the research question that are related to these research areas, as shown in Figure 1.2.

1.4 Overview and Contributions

This dissertation is organized as follows: Chapter 2 provides an overview of the research area *Computer Graphics* and briefly introduces *Contact Mechanics* and *GPU Computing*. An overview of methods found in physics-based animation is provided, as well as an overview of methods for performing linear algebra on GPUs. Apart from various simulation models found in physics-based animation, a discussion is provided about various contact solver methods. In the background sections of each chapter, a more detailed discussion among different methods are provided.

Chapters 3 to 6 form the main contribution of this dissertation. Chapter 3 describes methods for storing large sparse-matrices on GPUs and how to efficiently perform the Conjugate Gradient (CG) on (multiple) GPUs. Chapter 4 describes a FEM simulation of deformable objects, solely performed on modern GPUs and uses the machinery developed in Chapter 3. Chapter 5 describes a novel and efficient approach for accurately simulating contact and friction between multiple deformable and rigid solids. Chapter 6 describes the collision handling and intersection tests used in the implementation of Chapter 5.

The key contributions of this dissertation can be summarized as follows:

1. In Chapter 3 we introduce a variation of the Block Compressed Sparse-Row storage scheme that is adapted to better fit the memory access patterns found on modern GPUs. This storage scheme is used in a parallelized version of the Conjugate Gradient method. Additionally, an approach for estimating the performance of the CG and similar methods is introduced that allows one to reason about its performance for a specific unseen problem, compared to the theoretical peak-performance of the GPU device.

2. Chapter 4 presents an implementation of linear elastic deformable bodies through a Finite Element Method (FEM) simulation, running solely on the GPU. Here the machinery presented in Chapter 3 is used for solving the linear systems. The simulation is able to run at interactive simulation rates (> 30 frames per second) on a GTX280 GPU, in a stable way and for relatively complex models ($> 10K$ elements). Since the simulation runs solely on GPU, the method is not slowed down due to transferring data between GPU and CPU memory.

3. Chapter 5 provides a novel approach for accurately computing contact and friction together with the dynamics of the simulated objects. The method is able to solve this problem without computing an intermediate Linear Complementarity Problem (LCP), which can be inefficient for deformable bodies with many degrees of freedom. Coulomb's friction cone is modeled by a force that is always aligned with the sliding velocity, which is more accurate than frequently used pyramid approximations found in many constraint-based methods. Due to the non-linear behavior of the simulated models, linear solvers are not able to correctly compute a perfectly collision-free state at convergence since their shape changes due to contact and friction. The presented method solves this non-linear problem efficiently, thus guaranteeing a perfect collision-free state at convergence.

4. Chapter 6 provides geometric primitives for correctly detecting intersections between vertices, faces and edges of a deformable mesh. With standard vertex-face and edge-edge intersection tests, internal collisions can potentially result in wrongly detected collisions, resulting in non-physical motions or even failure of the underlying method. The chapter also provides details about the collision handling system presented in Chapter 6.

Finally, Chapter 7 concludes the dissertation and Appendices A to D provides additional background information.

1.5 Publications

The following chapters are based on the following publications:

- Chapter 3: "Analysis and Performance Estimation of the Conjugate Gradient Method on Multiple GPUs." M. Verschoor and A.C. Jalba. *Parallel Computing, 38(10-11), 552-575, 2012* [183].
- Chapter 4: "Elastically Deformable Models based on the Finite Element Method Accelerated on Graphics Hardware using CUDA." M. Verschoor and A.C. Jalba. *Journal of WSCG, 20(3), 179-188, 2012* [184].
- Chapter 5: "Efficient and Accurate Collision Response for Elastically Deformable Models." M. Verschoor and A.C. Jalba. *ACM Transactions on Graphics, 38(2), 17:1-17:20, 2019* [185].

Chapter 6 is based on previously unpublished work, but it is part of the implementation of Chapter 5.

2

Background

2.1 Introduction

In the previous chapter a motivation of the research and an overview of this dissertation were given. In this chapter we provide background material and the context of the research. First a brief overview of the main research field *Computer Graphics* is given, which is further narrowed down using *Computer Animation* followed by an overview of *Physics-Based Computer Animation*, *Contact Mechanics* and *GPU Computing*. Related work specific to the methods described in the next chapters are discussed in the related work sections of those chapters. In the appendices additional background material on the methods used in this dissertation is provided. Appendix A provides additional information on the preconditioner introduced in Chapter 5. In Appendix B a brief overview is given on optimization methods used for solving some of the problems described in this dissertation and discusses the methods used for solving optimization problems typically found in physics-based computer animation. Appendix C provides additional material for rigid-body simulations as used in parts of this dissertation. Appendix D gives a brief introduction to *FEM-based Elasticity*.

2.2 Computer Graphics

Computer Graphics is a vast research area which in general concerns techniques used for the creation and manipulation of certain models or data and their transformation into (sequences of) images using computers. These models and data can originate from various sources and other research areas, e.g., mathematics/statistics, physics, meteorology, engineering, while also human interaction leads to new input. Computer graphics has led to a variety of specialized subdisciplines, such as data visualization (providing insight into large data collections) [53, 91, 92, 155, 191], scientific visualization (providing (temporal) insight into physical processes or objects) [60, 83, 93, 190], virtual reality (aiming at immersion of users in virtual worlds) [162] and rendering (aiming at highly realistic images) [51, 100, 144]. In this dissertation we focus on animation: the synthesis of sequences of images to show dynamic scenes. Animation is a discipline that encompasses *Traditional Animation* and modern *Computer Animation*. Traditional animation can be considered as the art of creating animations using physical objects and processes. Computer animation concerns 2D and 3D animation methods for creating animations using computers and can be considered as a subdiscipline of computer graphics. *Physics-based Computer Animation* is a subdiscipline of computer animation and concerns the application of simulation techniques for animating the physical behavior of objects or phenomena. The main focus of this dissertation is on techniques related to simulation methods used to create animations – more specifically, the simulation of deformable and rigid objects and interactions between objects through contact and friction. In the following sections we further describe the field of computer animation, followed by a description of physics-based animation.

2.2.1 Computer Animation

The art of animation can be traced back to the 19th century when the first devices were created for showing small animations of hand-drawn images. With the invention of the *Cinematograph* and the development of photography in the early 20th century, the first short animations were created by showing sequences of hand drawn images or photos. Using techniques like *Cell Animation*, *Key-frame Animation* and *Rotoscoping*, animators were able to create motion pictures with detailed and static background images, while the foreground contained various hand drawn animation loops that were re-used among different animations. Another form of

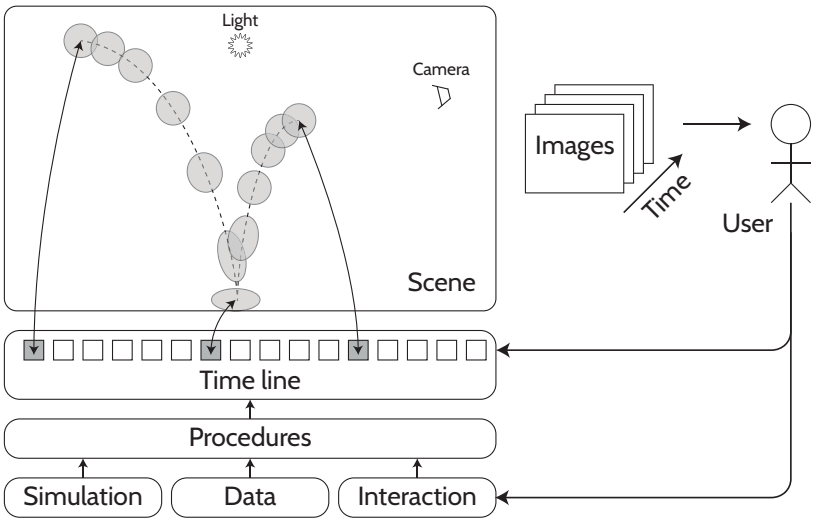


Figure 2.1: A 2D/3D key-frame animation system. The key-frames (dark-gray blocks) on the time line are set directly by the user (animator) and the in-between frames are obtained by interpolating between key-frames. Alternatively, procedures can be used to directly set the key-properties of the objects for each frame. These procedures can have a simulation, data or interaction as input.

animation is *Stop Motion* animation in which the properties of physical objects in a scene are changed between two consecutive frames. By showing the frames at a sufficiently high rate, each of these techniques will create the illusion of smooth moving characters or objects.

With the development of computers and computer graphics, animators were able to use computers for creating animations. In the early days these animations were mainly 2-dimensional or relatively simple 3-dimensional animations. Many concepts known in traditional animation were re-applied in computer animation, for example, key-frame and cell animation. With the adoption of computers, animations were not solely created by animators, but were also created by any form of data, user interactions, mathematical models or procedures, or combinations of these inputs. This has led to a wide variety of applications such as (interactive) data visualization, gaming, VR, education, art, user interfaces, etc. Since the discipline of computer animation is very broad, we further narrow this down to 3D animation methods and animation systems.

2.2.2 Animation Systems

Modern 3D animation systems can be described using Figure 2.1 and consists of a *scene* containing the objects to be animated, a virtual camera used to generate the final images, and lights that describe the virtual light conditions in the scene. Each entity in the scene has its own properties, like position, orientation, scale, intensity, color, etc. Furthermore, animation systems have a *time line*. A time line contains all frames of the animation. The animator can set each of the key properties of the objects in the scene for a specific frame in the animation; such frames become key-frames. The objects' properties between successive key-frames are obtained by interpolating between the key-properties. In this way the animator is able to create animations in which the key-properties change over time.

Animated objects can consist of a single geometric mesh, or they are composed using multiple smaller objects that are linked to each other using a hierarchical structure, usually involving joints. For example, a car object is a composition of a chassis, doors, wheels, etc. The position and orientation of each individual part of this car depends on the position and orientation of its parent object in this structure. For example, when the position of the car changes, also its wheels should move along. On top of that, a local offset can be added to the position and orientation of each wheel relative to the chassis. These additional offsets are properties that can also be specified as key-values on the time-line.

Character animation extends this idea of hierarchical structures. Usually a skeleton is created for a particular character. On top of this skeleton, the skin of the character is placed and connected to the underlying skeleton. When the animator animates the skeleton, the skin of the character follows the motion of the skeleton. Using techniques like *Linear Blend Skinning* [102, 116], the deformation of the skin is defined by a combination of the motions of the nearest skeleton parts. Such techniques give an animator control over a complex skin using a simplified version of the character, its skeleton. On top of this, the skeleton can also be controlled directly by data obtained through *Motion Capture* [195], user interactions or computational models for controlling characters like *Flocking* [150] or *Inverse Kinematics* [6].

The advantages of such methods are: (i) the animator has direct control over the final result of the animation, and (ii) by carefully changing the key-values, usually very artistic results are obtained. The disadvantages are: (i) no physical correctness and (ii) the difficulty to animate scenes with many degrees of freedom. The lack of correct physical models is usually visible as non-physical motions or deformations of the objects or characters, or as interpenetrating surfaces. The latter can be corrected using post-processing methods. When a large number of characters or objects are animated, also the number of degrees of freedom grows rapidly. Creating good and artistic-looking animations now becomes a difficult process. If various characters and objects interact, also the animation method must be aware of the physical constraints that emerge in such complex cases. On top of that, some animations require very complex but realistic motions of daily objects or phenomena. For example, the motions of trees, grass, hair, cloth, fluids, smoke, sand, dust, snow, etc., subject to various external forces like wind, friction, viscosity, and cohesion. Such examples are very difficult to animate in a realistic way using key-frame animation because the interpolation of specific properties between two key-frames is not aware of the underlying physics laws or corresponding constraints.

Many of these difficulties of traditional 3D computer animation are addressed by physics-based computer animation methods, as we will describe in the next section. Many of these methods require physics simulations and often use sophisticated numerical methods for solving the underlying physics problems. Our work can be positioned in the area of physics-based computer animation methods and aims to provide efficient (numerical) methods for performing the underlying simulations.

2.3 Physics-Based Computer Animation

Physics-based computer animation is an active area of research which is concerned with animating complex models, behaviors or phenomena based on physics laws. Within physics-based animation methods, usually some simulation is performed in which objects or phenomena are simulated. In general, a system of Partial Differential Equations (PDEs) is created and solved, which leads to the new state of the degrees of freedom in the simulation, following

the outline in Figure 1.1. Degrees of freedom in this context can refer to simple rigid objects, complex deformable objects, or systems of particles that resemble the flow of fluids or granular materials. Animating such complex systems is often too demanding using traditional key-frame methods. The motion of these systems can be subject to various constraints provided by the animator, actor, or other objects in the simulation or scene. This in turn enables interactions between (parts of) objects and the user, which results in realistic motions of the objects in the environment. However, these motions can be difficult to control precisely due to the chaotic nature of the underlying systems. This can be problematic for animators who demand a certain animation as a result.

Although physics-based animation increases the realism of animations, it requires a thorough understanding of the involved physics, numerical mathematics and hardware to design and implement methods that can simulate the desired phenomena. Furthermore, when collisions or contact occurs between different parts of the simulated objects, energy is dissipated through friction. To attain a high degree of realism, accurately simulating friction is of key importance. Additionally, depending on the complexity of the underlying problems, this kind of simulations can also demand substantial computational power and memory resources. Therefore, many methods are performed on GPUs or even clusters. This also requires the methods to be designed to run in parallel. Therefore, GPU and parallel computing has received a lot of interest from this area.

When applied in interactive applications, one has to face time-constraints for the computations in order to produce animations at interactive frame-rates. A trade-off between speed and realism is often made. For offline animations time is not always a hard constraint and the amount of computing power can be easily scaled up. When time-constraints apply, the desired realism is not always easy to obtain. For example, stacking a few rigid objects on top of each other can be a difficult problem to solve when time-constraints apply and computational resources are limited. In such simulations the underlying numerical methods are not able to produce an accurate solution within the required time, resulting in a non-smooth behavior over time. Furthermore, to produce accurate results, also the collision detection must be precise. If this is not done properly, collisions can be missed and detected in a later stage in the simulation, resulting in spurious motions.

In the following subsections a brief overview of commonly used methods in Physics-based animation is given.

2.3.1 Rigid Body Simulations

One of the most elementary type of simulations for animations are rigid body simulations, see [20]. A rigid body is, as its name suggests, is a body that cannot deform. Thanks to this, the state of a rigid body can be described using one position and one orientation, see Figure 2.2. Rigid bodies are used in many scientific and engineering disciplines, but are nowadays also used in many computer games and virtual reality applications. Game engines [57, 182] and virtual environments are usually equipped with specialized physics engines [22, 45, 128] that are heavily optimized for simulating rigid bodies. These engines also perform collision detection between all objects and take care of the computation of collision responses. Since rigid objects can not deform, collisions between *two* objects can be computed very accurately. Rigid objects also appear as series of connected articulated objects. These connections are often modeled using joints based on constraints [9] or springs. Granular materials can also be considered as a special kind of rigid body simulation. By modeling particles as small spheres, collision

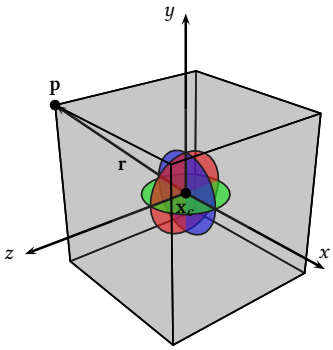


Figure 2.2: Rigid body: the state of a rigid body is expressed by the position of its *Center of Mass* $\mathbf{x}_c$, and three rotations around axes x, y, z. A point $\mathbf{p}$ on its surface is expressed as $\mathbf{p} = \mathbf{x}_c + \mathbf{r}$, with $\mathbf{r}$ a vector from the center of mass to point $\mathbf{p}$. The linear motion of the body affects the center of mass $\mathbf{x}_c$ and the angular motion affects vector $\mathbf{r}$.

detection can be simplified. However, granular materials can also be simulated using hybrid approaches, in which the material is regarded as a fluid. In Appendix C a full description of rigid body dynamics is provided as used in Chapter 5.

2.3.2 Deformable Body Simulations

Deformable bodies are volumetric objects that will deform under a load or application of forces. In the following subsections, a number of methods are mentioned.

Mass-spring systems Mass-spring systems are systems of particles with a particular mass, connected through springs with a certain stiffness, see Figure 2.3. These mass-spring systems can be used to model simple cloth or deformable models. In general, the force generated by a linear spring can be computed using Hooke's law, i.e.,

$$\mathbf{f} = -k\Delta\mathbf{x}, \tag{2.1}$$

with k the stiffness of the spring and Δx the change in length with respect to the initial configuration of the spring. This equation also forms the basis of Linear Elasticity, described in Appendix D.2.

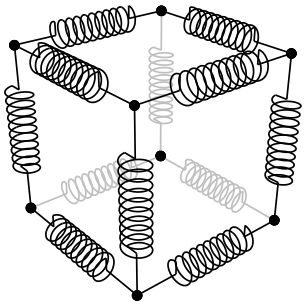
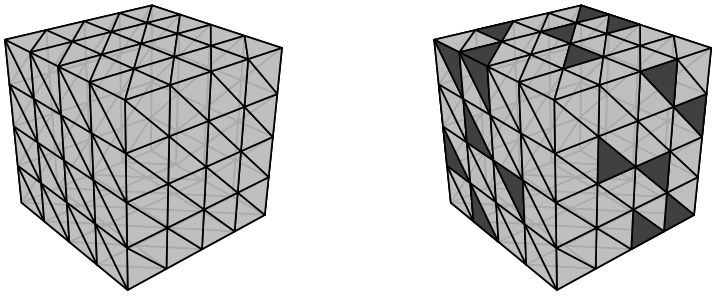


Figure 2.3: A mass-spring system: a number of point-masses are connected through springs. When the length of a spring changes with respect to its rest configuration, a force is exerted on the connected masses. This eventually results in a motion of the corresponding points.



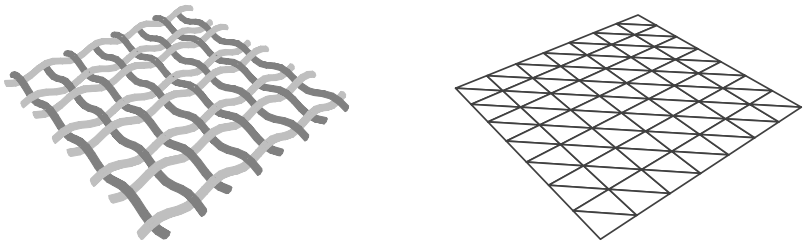
(a) FEM-based elasticity: each tetrahedral element in the object contributes to the global deformation.

(b) Subspace methods: only a few carefully-selected tetrahedral elements (dark) are used to describe the global deformation.

Figure 2.4: FEM and subspace methods.

Finite Element Based Elasticity methods The Finite Element Method (FEM) [143] is a standard method among engineers for performing all kinds of simulations. FEM based methods require a mesh to discretize the computational domain, see Figure 2.4a. In three dimensions, tetrahedral, cubical or hexahedral elements are often used. Per element a local instance of the global problem is approximated using so-called *shape functions*. These shape functions can be linear or a higher order polynomials. Finally, all equations of all individual elements are assembled into a large system of equations. This system is typically solved using the Conjugate Gradient (CG) [88] method. The Finite Element Method allows for elements with different shapes and sizes, and often smaller elements are used in regions where a higher accuracy is needed. The advantage of FEM is that the residual of the approximation solution is minimized at the nodes, resulting in a good global approximation. Furthermore, an implicit time-integration scheme can be easily used, starting from a solution from an earlier time-step. However, this still requires to solve a large linear system. In graphics applications, the topology of the underlying mesh is often kept fixed, while engineering applications perform a re-meshing of the domain. A popular method used in Computer Graphics is co-rotational FEM [124]. This method keeps the structure and stiffness of the model fixed, but takes a correction with respect to rotations of the elements into account. Other popular methods use a Finite Volume approach [95, 96, 168, 176], which models the problem on the boundaries of the elements. Using the deformation gradient tensor and its Singular Value Decomposition (SVD), the pure deformation and the rotation of each element is obtained. The pure deformation is then used to compute the stress in a more straightforward way. In Appendix D a full description of FEM based deformable bodies is provided as used in Chapters 4 and 5. Chapter 4 describes a GPU implementation of the co-rotational method [124], and is accelerated using the method described in Chapter 3.

Subspace methods The number of degrees of freedom in typical FEM simulations can grow rapidly, which increases the computation time. By projecting the deformations on a smaller sub-space, the computational overhead can be reduced. Subspace methods are often based on statistical methods that reduce the set of basis-vectors which are obtained from, e.g., FEM based methods. By projecting a vector from the full space on the subspace, a dimensionality reduction is obtained. For example, using *Principal Component Analysis* (PCA) such a reduction can be obtained by selecting the basis vectors corresponding to the most significant eigenvectors. The



(a) Fiber-based cloth. **(b)** Mesh-based cloth.

Figure 2.5: Fibers and cloth simulations.

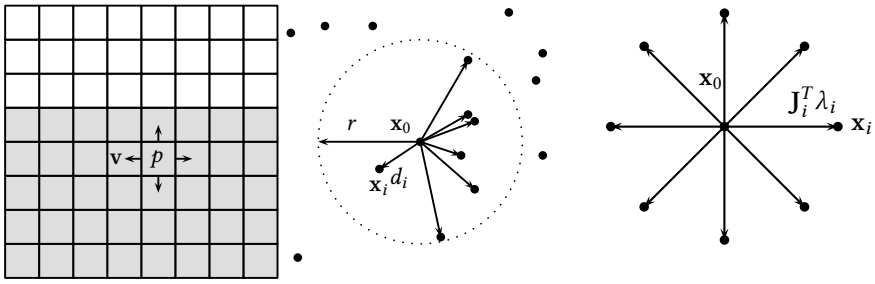
selected basis-vectors then define the so called *reduced subspace*. Other approaches inspect the modal bases of the deformation and select a number of modes with low frequencies, or select a few representative elements to describe the global deformation, see Figure 2.4b. The number of selected basis vectors largely determines the dimensionality reduction and the quality of the animation. Using training data originating from the original models, typical deformations can be learned from examples such that the final set of basis vectors can reproduce similar deformations. Methods that perform this kind of reductions are, e.g., *Cubature* and *Sub-Space Methods* [4, 14, 15, 28, 140, 175, 181]. Applications are typically found where deformations of objects should be obtained fast, e.g., interactive 3D animation applications, virtual reality, and games.

2.3.3 Fibers and Cloth

In addition to volumetric deformable bodies, also non-volumetric deformable objects exist, e.g., fiber and cloth simulations. Simulations of fibers are both found in applications in which hair is simulated [25, 47] or in which cloth is simulated at yarn level [41], see Figure 2.5. In both areas single fibers are simulated through chains of single rods. In hair simulations, each fiber can potentially collide with all other fibers. In the case of yarn based cloth simulations, it is known beforehand where and which (segments of) fibers are in contact, so at each yarn crossing several forces can be modeled. Furthermore, yarns are allowed to slide over each other, resulting in friction. By modeling cloth at yarn level, the behavior of the cloth is mainly defined by the interactions between the yarns and thus deliver very realistic results compared to more traditional mesh based cloth simulations. Similarly, by modeling hair using individual fibers, the resulting appearance is very realistic. Traditional cloth simulations are modeled using a lattice mesh or using a mass-spring approach and define bending and shear forces on the crossings [11, 29].

2.3.4 Fluids

Simulation of fluid is an active research area. Fluid simulations are successfully applied in many feature movies, games and virtual environments. Many methods applied in graphics originate from computational physics in which one seeks for accurate methods for solving the Navier-Stokes equations for modeling, e.g., the flow of oceanic currents, weather patterns and two-phase flows. Within fluid simulation techniques, a distinction between Eulerian and Lagrangian methods can be made. Eulerian methods model the flow of fluids using an Eulerian grid. Lagrangian methods model fluids using particles. Hybrid methods combine Eulerian and Lagrangian approaches.



(a) Grid-based fluids: the velocity $\mathbf{v}$ at the boundaries of each cell affects the pressure p inside the cell.

(b) Smoothed Particle Hydrodynamics: a quantity at $\mathbf{x}_0$ is approximated by the weighted sum of its neighbors. The weights are obtained by a kernel function and distance d_i.

(c) Position-based dynamics: particles are connected through constraints.

Figure 2.6: Various fluid simulation methods.

Grid Based Methods Classical methods for simulating fluids are based on computations in a grid. Here the computational domain is formed by a grid in which each cell has the same shape, see Figure 2.6a. Each cell of the grid represents a number of different quantities, e.g., velocity, density and pressure. By solving the Navier-Stokes equations on the grid, an accurate simulation of the flow of a fluid can be obtained. Therefore grid based methods are frequently used in engineering and scientific applications. Since the size of the grid can be very large, this kind of methods are computationally intensive. Examples of grid-based methods in Computer Graphics are the flow of fluids [55, 64, 65, 166], simulation of smoke [61] or clouds [82]. Furthermore, some of these simulations can be coupled with a surface tracking method to also simulate the surface of the fluid, using, e.g., (particle) level-set methods [16, 54, 56, 137].

Smoothed Particle Hydrodynamics Smoothed Particle Hydrodynamics (SPH) [79, 110] is a Lagrangian approach for simulating various phenomena found in astrophysics [165], fluid [123] and granular material simulations [1]. The method uses large amounts of particles. Each particle carries some physical properties, like a mass or temperature. Using smoothing kernels, a weighted average of these quantities can be computed. Particles closer to the point of interest receive a larger weight than distant particles, see Figure 2.6b. When a derivative of a quantity must be computed, the derivative of the kernel is used in the convolution. SPH enables fast simulations of fluids in games or for offline animations. All computations are performed per particle and are therefore candidates for parallelization. The advantage of the method is that it can change the resolution by changing the size of the kernel. Furthermore, the method is able to simulate objects that do not have a fixed topology. However, since the method works in a small area of the global domain, global constraints are difficult to enforce, for example the incompressibility of fluids and the computation of the pressure inside a fluid. The incompressibility is for instance better handled in Position Based Methods.

Position Based Dynamics Position Based Dynamics [21, 115, 125] can be regarded as an extension of SPH. The method adds additional constraints to the particles which enforce the incompressibility between individual particles. Additionally, the same constraints also allow

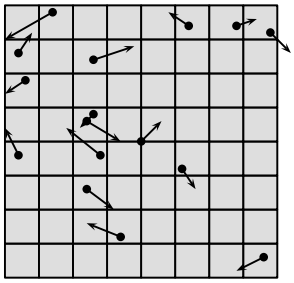


Figure 2.7: Hybrid methods: the Material Point Method (MPM) transfers quantities and velocities from the particles to the grid using APIC. Forces are computed in the grid, followed by an update of the grid velocity. The velocity and other quantities are transferred back to the particles, and their positions are updated.

particles to 'stick' to each other, see Figure 2.6c. In this way, deformable or even rigid bodies can be simulated using particles. By allowing constraints to disable, also fracturing can be modeled. Constraints are solved using similar techniques that are found in contact mechanics, i.e., Lagrangian multipliers are computed that represent a force that repel or attract particles. However, when many particles are constrained, the method requires more time to converge since only a local approximation is computed, while one is seeking for a global one.

Hybrid methods Hybrid methods typically combine Eulerian and Lagrangian type of simulations. The Material Point Method (MPM) [171] is a particle-based method that computes some properties using kernels found in SPH methods and transfer other properties to a grid for which a global problem is solved, see Figure 2.7. The result is then transferred back to the particles. Nowadays these hybrid methods are very popular for simulating various phenomena, like snow [169] or sand [198]. The main issue with these methods is how to transfer a quantity from particles to the grid and back. Transfer methods are for example PIC, APIC and FLIP [80, 98].

2.3.5 Learning and Example Based Methods

Machine Learning is a field within computer science that develops models that allow the computer to 'learn something'. At first glance computer graphics and machine learning seem two distinct fields, but recently the graphics community is adopting machine learning methods, for example, the use of *Convolutional Neural Networks (CNNs).* Since many physics-based simulations can be very computationally intensive, one interesting approach is to make them interactive by replacing the underlying physics model by a model that is learned from examples. Usually the new model is some kind of CNN that is able to reproduce a similar behavior, see [36, 179]. Furthermore, with highly sophisticated simulation models available, it can be difficult to find the optimal set of parameters to simulate a particular material. Using machine learning and computer vision methods one can find these parameters given the examples obtained from recorded images or videos [197].

2.4 Contact Mechanics

The previous sections described various simulation methods found among graphics applications. In this section a brief overview is given about contact mechanics and methods that take contact between objects into account.

When two objects are in contact, an equal (but opposite) contact force is applied on both objects. Depending on the net forces acting on the bodies, this contact force has a normal component in the normal direction of the contact surface, and a tangential component working in the plane of the contact surface. These forces can result in additional motions which in turn results in an additional change in the contact forces, the orientation of the contact surface and possibly a change in the shape of the contact area. This process continues until no change in the contact forces and velocity is found, i.e., when a stable state is found. When such systems are simulated using discrete time-steps, the state of the simulation at the end of the time-step must also be stable, i.e., the obtained forces and velocities should not result in yet additional changes in the forces and velocities. Due to this tight coupling of deformation, motion and contact forces, this problem in non-linear. A stable configuration can be considered as a root of this non-linear problem. This problem is also known as Signorini's problem [158]. Antonio Signorini posed a problem of an elastic body resting on a rigid friction-less body. The problem he described was 'a problem with ambiguous boundary conditions'. For any point of contact, the boundary condition should either impose an equality or inequality. The problem is that it is not a priory known which type of boundary condition should be applied for each point in contact, and if that solution is unique.

From the initial Signorini's problem, the Signorini's conditions were derived, see [33]:

- The contact force exerted by body 1 on body 2 can be positive only if the contact is closed.
- If the contact is open, then the contact force exerted by body 1 on body 2 is zero.
- The contact force can take only nonnegative values.
- The bodies cannot interpenetrate.

These conditions describe the complementarity between the normal contact-force magnitude f_n and some contact distance d as:

$$0 \leq f_n \perp d \geq 0, \tag{2.2}$$

which is known as a complementarity problem. When f_n is a linear function of d, this problem is known as a *Linear Complementarity Problem* (LCP). In Appendix B.1.5 an overview is given of this kind of problems.

2.4.1 Coulomb Friction

The Signorini problem does not take frictional forces into account. In general, apart from the normal force acting on both objects involved in the contact, also a tangential friction force is present. This friction force is the result of imperfections between the surfaces. Rougher surfaces observe a larger friction force compared to smooth surfaces. Apart from these imperfections also the magnitude of the normal force determines the maximum tangential friction force. The larger the normal force is, the larger the maximum friction force in the tangential direction can be. An approximation of this relation is known as Coulomb's friction model,

$$\|\mathbf{f}_{t,max}\| = \mu \|\mathbf{f}_n\|, \tag{2.3}$$

which relates the magnitude of the maximum tangential friction force $\|\mathbf{f}_{t,max}\|$ with the magnitude of the normal force $\|\mathbf{f}_n\|$ times some friction coefficient μ, which depends on, e.g., the materials involved in the contact. The maximum friction force is the maximum force in the

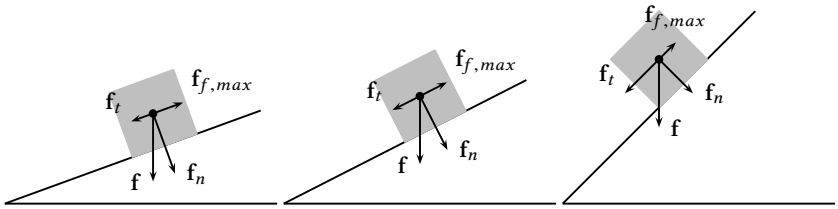


Figure 2.8: An object is placed on a slope; by increasing the slope, the magnitude of the tangential force $\mathbf{f}_t$ eventually becomes larger than the maximum friction force $\mathbf{f}_{f,max}$. Therefore, the type of friction changes from static (a)(b) to kinetic friction (c), for $\mu = 0.5$, and the object starts to slide after (Figure 2.8b).

tangential direction that the contact can resist. If the force applied in the tangential direction has a smaller magnitude than the magnitude of the maximum friction force, the objects involved in the contact will not slide over each other: the maximum friction force is large enough to keep the net tangential force at zero. This case is called *static friction* or the *stick* condition. For larger applied forces in the tangential direction, the friction force is bounded by Coulomb's friction law in Equation (2.3). As a result, the net force in the tangential direction will not be zero, and the objects involved in the contact will slide over each other. This case is called *kinetic friction* or the *slip* condition, see Figure 2.8.

2.4.2 Maximum Dissipation

Coulomb's friction law dictates the boundaries on the tangential friction component. However, no description on its direction is given. In principle, a friction force working between two sliding surfaces should *always* result in a dissipation of energy. If, for example, the friction force works in the same direction as the relative motion of the objects in contact, the friction force will accelerate the motion and adds additional energy to the system, which is not correct. When the friction force is working exactly in the opposite direction of the objects' relative motion, then the friction force will slow down the motion the most, i.e., the maximum amount of energy is dissipated. When this is the case, the *maximum dissipation principle* is obeyed. In real natural cases, this principle is always obeyed. In simulations however, the direction of friction is determined by some computations. Therefore it is possible that the chosen direction of the maximum friction force eventually works in a non-optimal or even a wrong direction such that the applied friction force will introduce too less dissipation, or spurious motions of the involved objects. By enforcing the sliding direction to be opposite of the friction force, maximum dissipation is enforced.

Since the direction of friction can be in principle anywhere in the tangential plane, Equation (2.3) can be considered as a cone that clamps the tangential friction force. The slope of the cone is given by the friction coefficient.

2.4.3 Non-Linear and Non-Smooth Problem

As mentioned in the introduction of this section, contact between two objects is a non-linear problem. Contact forces will result in a change in motion of the involved objects and a change in orientation of the contact surface. By solving this non-linear problem, eventually a stable

configuration is found in which no change in the contact forces is observed. When there is also a friction force working between the surfaces, the maximum dissipation principle should be obeyed when objects are sliding over each other. This means that the tangential friction force direction directly depends on the sliding velocity. A change in the sliding direction then affects the friction force direction, which can lead to changes in the motion of the objects and configuration of the contact surface. Additionally, since friction for a certain contact can change between static and kinetic friction, and between *impact* and *release*. The latter two occur when objects collide or move away from each other. Therefore, this problem is also considered as non-smooth since contact forces are not smooth at the boundaries of these states. Simulating this kinds of problems is therefore challenging. Replacing rigid bodies with deformable does in principle not change the handling of contacts. However, contact will also result in a deformation of the objects that must be taken into account when finding the optimal contact configuration.

2.4.4 Painlevé Paradox

In rigid body simulations, the use of Coulomb's friction model *can* lead to situations in which the state of a single contact is undetermined or inconsistent. It can be shown that for certain simple configurations of a rigid body with some frictional contact, and by imposing additional conditions or constraints on the acceleration of the body, the corresponding *Linear Complementarity Problem* (LCP), see Appendix B.1.5, can have a non-unique or no solution [33, 107]. The method cannot decide in which state the contact is, because the corresponding complementarity conditions cannot be met, or have multiple solutions. This inconsistency is known as the Painlevé paradox, named after the French mathematician Paul Painlevé. This paradox can be demonstrated by drawing a line on a blackboard using a piece of chalk. Depending on the magnitude of the normal force and the angle of contact, either a smooth line is drawn, or a dashed line appears. This is due to the 'hopping' or 'bumpy' motion of the chalk over the blackboard and typically occurs for large friction coefficients μ.

Moreau's time-stepping method [122], allows one to simulate the Painlevé paradox. In this time-stepping method, an implicit method is used to simulate the smooth terms of the problem. For the non-smooth contacts, their impulses and relative velocity are used in a complementarity problem. For deformable models the time of contact is usually larger than one time-step, therefore, this impulse approximation can be simplified by taking the impulse over the discrete time-step. For rigid bodies simulated with reasonably small time-steps, a similar approximation can be used. However, for simulating the Painlevé paradox, these impulses need to be approximated accurately. In case the approximation of the impulses is very rough, the method can be regarded as an implicit method with the complementarity conditions on the velocity level. For the classical Painlevé example, it can be shown that using this approach the resulting LCP will have a unique solution, see, e.g., [107].

2.4.5 Collision Detection

Contact mechanics requires information of features that have collided or are about to collide. Features are for example pairs of vertex-face or edge-edge pairs. Finding these features requires collision detection methods that report all features that have a negative signed distance [23, 24, 71, 121]. Typically, one wants to detect collisions before they occur (a priori). By extrapolating the motion of the objects from the beginning of the time-step to the the end of the time-step, a prediction about a collision is made. Since rigid bodies do not deform, their motion can be predicted accurately as long they are not in contact. Due to this, also a distance

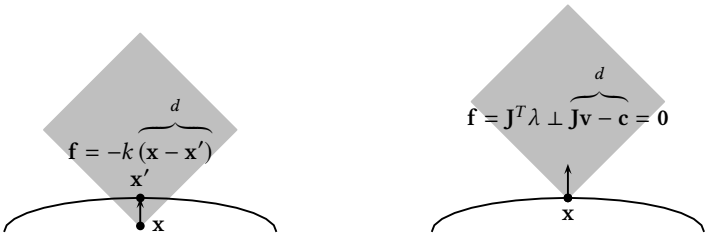
approximation of a feature can be predicted accurately, e.g., using *Controlled Conservative Advancements* [173]. Using a distance function for these features, the location and time of impact can be found by finding the roots of this function. If a valid collision is detected, then the feature can be taken into account. This procedure is known as Continuous Collision Detection (CCD).

In general we can distinguish between two main approaches: sampling a *signed distance field* (SDF), or computing the signed distances between the features. By sampling the signed distance field of an object using a particular vertex, one is able to determine if the vertex is inside the object and thus if an collision had occurred. However, if the distance between the sample at the beginning of the time-step and at the end is large, it is possible that the path in between intersects the object, while the sampled distances are positive. This can lead to missed collisions, especially if the object is thin. In these cases multiple samples along the path could be taken, but this does not guarantee that a collision is found. Furthermore, finding edge-edge features can be difficult. When the object is deformable, the signed distance field needs to be recomputed frequently.

The second approach treats the surface as a piecewice linear approximation of the underlying object. Each linear approximation can be considered as an infinite plane. Given the initial and approximated configuration of this plane, and a certain discrete point on the other object, one can easily determine if both features could potentially collide. By testing the signed distances at the initial and current approximation, a change in the sign of the signed distance could indicate a potential collision. Using a root finding method, a configuration can be found in which the signed distance becomes zero. If the intersection point lies inside the triangle associated with the plane, the collision is valid. The main advantage of such approaches is that no matter how fast the motion is, a collision is always detected. However, the computation of the intersection point requires robust root-finding methods for each type of feature. On top of that, intersections and signed distance computations of these features need to be robust. Finding potential colliding pairs can be accelerated using *Bounding Volumes Hierarchies* (BVHs) [156, 178, 187, 188]. In Section 6.1 intersection tests are discussed for vertex-triangle and edge-edge features for deformable models. The difficulty with deformable models is that between two states, the geometry can change 'significantly'. Face normals can change rapidly for small changes in positions and the geometry can change locally from convex to concave. All this must be taken into account. In these tests, also the possibility of internal collisions must be taken into account.

2.4.6 Contact Solver Methods

Within animation and graphics applications both *Penalty-based* and *Constraint-based* methods are frequently used, see Figure 2.9. Penalty-based methods define a penalty force based on the deviation between the actual and the desired contact configuration. Constraint-based methods constrain the motion of the object or vertices in a particular direction such that they can never interpenetrate. Since the response force does not directly depend on the penetration depth, constraint-based methods are able to compute a response force that exactly prevents penetration. Since the computed response forces may deform the objects, also other collisions can occur or the current contact configuration might not agree with the actual (deformed) geometry. Hence, a few iterations of the contact method must be performed in order to refine the contacts and to search for new collisions. In the following subsection we briefly discuss a number of methods for resolving contact between (deformable) objects.



(a) Penalty-based contact. **(b)** Constraint-based contact.

Figure 2.9: Penalty and constraint-based contact: penalty contact forces require a certain penetration depth d, while constraint-based contacts compute a force $\mathbf{f}$ that resolves penetration depth d.

Penalty-Based Contact Methods As mentioned before, penalty-based methods [30, 87, 121] compute a penalty force between two objects based on the penetration depth of certain features, similar to springs, see Figure 2.9a. Penalty-Based Contact methods have problems with completely resolving the collisions and may introduce oscillations and additional energy to the system. Since the contact force depends on some penalty term, usually the penetration depth, there is always some penetration when objects are in contact and contact forces are computed. Due to this, the response might not be physically correct or the stiffness of the penalty force might be too large, resulting in oscillations. *Continuous Penalty Forces* [174] combines *Continuous Collision Detection* with penalty forces. The method approximates the time of collision using continuous collision detection and computes the impulses needed for resolving the collision. Furthermore, implicit penalty contact [196] treats penalty forces in an implicit way, resulting in stable contact between many objects. Additionally, friction forces are often modeled using anchor forces.

Constraint-Based Contact Methods Constraint-based contact methods prevent penetration of objects by constraining the motion of the degrees of freedom in a particular direction, see Figure 2.9b. A constraint measures the actual penetration depth, or the volume of the intersection. The contact solver solves the obtained *Constrained Optimization Problem* such that each constraint measures no penetration at the end of the time-step. Per constraint a *Lagrange Multiplier* or *Karush-Kuhn-Tucker Multiplier* [106] is computed that represents an impulse that resolves the collision. In addition to this, also friction can be modeled through constraints by coupling the non-penetration and friction multipliers. Although this approach seems straight-forward, in practice it is very difficult to obtain a good coupling between the non-penetration and frictional problem. The non-penetration multipliers influence the frictional multipliers and vice-versa. Furthermore, when the collision detection phase generates the collision points, a number of these constraints may actually be similar, resulting in dependencies between the constraints. This can eventually result in an overdetermined system, which does not have a unique solution.

2.4.7 Constraint Solvers

Constraint solvers are used to solve a problem that is defined in terms of a set of constraints. In general we can make a distinction between methods that solve the reduced problem by first constructing the *Linear Complementarity Problem* (LCP) matrix or *Delassus operator* (see Equation (B.13)), and methods that do not create this reduced problem and solve the *Mixed Linear Complementarity Problem* (MLCP) directly. Depending on the type and size of the problem, either the problem can be efficiently solved through LCPs or MLCPs.

LCP methods LCP methods are a special class of complementarity problems that fit constrained problems with rigid body simulations very well. Within these problems, the original MLCP is converted such that the free variables are separated from the multipliers, resulting in an LCP [43, 44, 59]. During this conversion, a smaller matrix in $\mathbb{R}^{m \times m}$, is explicitly created, which is often called an *LCP* matrix or *Delassus operator*, with m the number of constraints, see Appendix B.1.5 for more details. Once the LCP matrix or Delassus operator is created, it solves the pure contact problem. This can be for example solved using a Projected Gauss-Seidel method that project or clamp the computed multipliers. Furthermore, also sophisticated methods can be used that model friction accurately. Since the construction of this problem requires the inverse of the system matrix, it can be efficiently obtained for simulations in which the system matrix is block-diagonal with small blocks. For more general and larger problems, the inverse must be obtained differently using, e.g., a Cholesky decomposition or by computing the inverse matrix by solving n linear problems. Methods that solve the contact problem for simulations in which the system matrix is block diagonal are, e.g., rigid body problems [9, 37, 180], or hair dynamics [25, 47]. Furthermore, contact with deformable models can be solved using the Gauss-Seidel-Like method [50], which is also applied for volumetric constraints [2]. Once the normal and frictional impulses are obtained, they can be applied directly to the original problem, resulting in a new state of the system.

MLCP methods MLCP methods work in general in a larger dimension compared to LCP problems. MLCP methods work in the $\mathbb{R}^{(n+m)\times(n+m)}$ space, while LCP methods are in $\mathbb{R}^{m \times m}$. In this context n stands for the size of the system matrix and m for the number of constraints. MLCP methods are often used in cases when it is not easy to transform the problem into an LCP. These methods are for instance used when the system matrix is very large and not block-diagonal, such that inverting it or computing the Cholesky decomposition would require significant computational resources. Methods found in Computer Graphics that apply to large deformable models and that work in $\mathbb{R}^{(n+m)\times(n+m)}$ space are for instance *Iterative Constraint Anticipation* [138] and *Staggered Projections* [101]. Iterative Constraint Anticipation uses two nested solvers. The inner solver solves an approximate LCP in which the inverse of the system matrix is replaced by its inverted diagonal. Next, the obtained multipliers are applied to the degrees of freedom in the system matrix by taking the Jacobi decomposition into account. Next, the right-hand side of the approximated LCP is updated and the LCP is solved again. Eventually, an optimal solution is obtained. Staggered Projection separates the computation of the non-penetration and friction multipliers completely. When the non-penetration multipliers are computed by solving a *Quadratic Programming* (QP) problem, the frictional problem is kept fixed. When the friction multipliers are computed, the non-penetration problem is kept fixed. Eventually, the method converges to an optimum. Originally, the method relies on dense solvers.

2.5 GPU Computing

GPU Computing, or *General Purpose GPU* (GPGPU) computing is a field concerned with solving large problems using GPUs. The field itself does not completely fit into Computer Graphics, but also fits Parallel Systems. GPUs are also used to accelerate numerical methods applied in several scientific simulation applications.

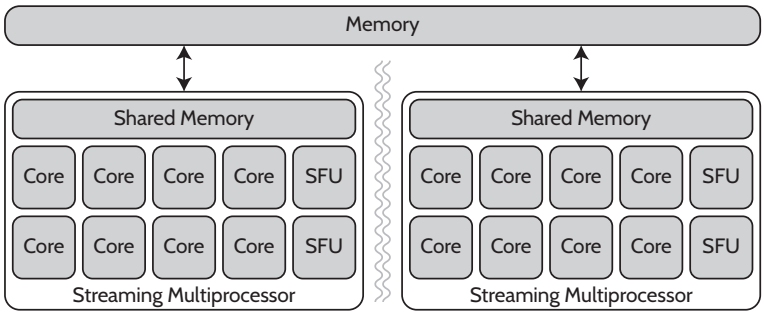


Figure 2.10: Schematic representation of current generation GPUs: A GPU is equipped with a number of *Streaming Multiprocessors*. Each Streaming Multiprocessor contains a number of *Cores* and *Special Function Units* (SFU). Each Core and SFU can access the shared memory of the Streaming Multiprocessor, which in turn can access the main memory. Each type of GPU has a varying number of Multiprocessors, Cores and SFUs, but in general each GPU follows this design.

2.5.1 Graphics Processing Unit

The early Graphics Accelerators were designed for displaying 2D images and *sprites* using palettes with a fixed amount of colors. With the introduction of *Blitter* techniques, fast block-transfers of images to the frame-buffer were made possible. This resulted in faster operations on sprites and faster routines for line and polygon drawing. In the following years the 2D performance, resolutions and amount of colors increased, but still no real 3D acceleration was available. This came available through additional expansion cards that were dedicated to 3D graphics only. After that, GPUs were integrated and able to do both 2D and 3D graphics. With this, also software APIs were designed for standardizing the communication between an application and the GPU. One notable API is *OpenGL* [103], which is still the standard today.

The pipeline of the early GPUs were fixed. Typically the application was sending commands for drawing triangles with several properties, using a particular transformation and projection to the screen. Using clipping techniques the triangles were clipped against the view volume. Invisible triangles were not processed further, while triangles that were intersected by the view volume were subdivided such that they fit the view volume. Finally, the triangles in the view volume were rasterized resulting in *fragments* for each pixel for each triangle. These fragments are then processed further. For example, a depth test can be performed that compares the depth value of a fragment with other fragments on the same location. Finally, the selected fragments are drawn on the screen.

With the introduction of *shaders* one is able to write short programs for manipulating parts of this pipeline [151]. For example, fragment shaders are used to modify the generated fragments. This also enabled the possibility to perform mathematical computations within these shaders. The input of these algorithms were stored in textures, and the output was stored in other textures. By properly managing this 'pipeline' one was able to perform a large amount of numerical operations on the input data. Currently, the fixed pipeline with programmable shaders is replaced by a fully programmable pipeline with programmable shaders. This enables the application of GPUs as massive parallel processors that can be completely programmed. Since these platforms also support single- and double-precision floating point operations, the hardware is very attractive for any kind of computational intensive tasks. APIs used for this kind of tasks are *CUDA* and *OpenCL*. Figure 2.10 gives a schematic overview of the current

generation GPUs. For example, the NVidia GTX 980 GPU has a total of 16 *Streaming Multiprocessor*, each with 128 cores and 96 KB shared memory, which makes a total of 2048 available cores and 1, 536 KB of shared memory. Furthermore, the GTX 980 GPU has a total memory of 4 GB. Within GPU computing one can make a distinction between *Memory Bound* or *Compute Bound* problems, which are further described in the following sections.

2.5.2 Memory-Bound Problems

Memory-bound problems require per floating-point operation a large amount of data. For example, to perform a component-wise multiplication on two n-sized vectors, one needs $2n$ floating-point values to be read from the memory in order to perform one floating-point operation, in this case a multiplication. Furthermore, the result is written back to a certain memory location. In total $3n$ memory operations are needed for performing n floating-point operation. We assume that the data does not fit in caches or local memory. Problems that fit in this category are operations performed on large matrices and vectors, or many finite-difference-based computations on large grids. In order to obtain the best performance for these problems, one needs to attain the best throughput of the data from the main memory to the local processors. Since the total memory throughput on high-end GPUs is about a few hundreds of gigabytes per second, the peak performance is mainly bound by the memory throughput. For example, a GeForce GTX980 has a bandwidth of 336 GB/s, which implies that the component-wise multiplication of two large vectors will have at most a performance of 28 GFLOP/s, while the peak-performance of the device is 2306 GFLOP/s for single-precision operations. Therefore it is very important to optimize the throughput of the data as much as possible. Furthermore, the performance can be increased by combining operations that perform on the same (or same intermediate) values. This strategy is also applied in Chapter 3 in which a few operations in the CG method are performed by the same program.

2.5.3 Compute-Bound Problems

Compute-bound problems are a class of problems for which the number of floating-point operations per floating-point value is bounded by the performance of the cores. Problems that reach this boundary usually have a large number of floating-point operations per transferred floating-point value. Examples of such problems are found in Digital Signal Processing (DSP), filtering and compression methods. Most of these examples require a small amount of data or re-use data between consecutive computations.

2.5.4 Intermediate Problems

In general most problems are neither purely memory- nor compute-bound, but also other aspects determine the final performance of a specific GPU program. For memory-bound problems one needs to achieve the best throughput of the data from the memory to shared memory. The maximum bandwidth is reached when the memory bus is saturated. This can be achieved by issuing a large number of concurrent memory transactions per multiprocessor. While the multiprocessor is waiting for a relative slow memory transaction, it can issue other memory transactions, or perform some computations in the meantime. This pipeline therefore increases the total throughput of the data. This typically occurs for larger problems. If the problem is too small, the relative high latency will dominate the computation time. Therefore, the size of the problem can be a measure of the number of issued memory transactions and the average data throughput. In Chapter 3 this observation is used to determine the performance of memory-bound problems. Also for compute-bound problems, the size of the problem does matter. If the problem is too small, one cannot rely on a straightforward parallelization of the

problem and therefore it is not possible to fully occupy all the processors of the GPU. In many cases the use of GPUs on small problems therefore results in a slowdown compared to a CPU version of the same method.

2.5.5 Matrix Storage Schemes and Krylov Subspace Methods

Many physics or mathematical (simulation) problems boil down to solving some large sparse linear system. Solving these linear systems efficiently on a GPU therefore significantly accelerates the overall computation or simulation method. Methods for solving these linear problems are for example Krylov-subspace methods. These methods iteratively approximate the solution by performing a sparse-matrix vector-multiplication (SpMV) in each iteration. Each iteration moves the current approximation closer to the unknown solution. Depending on the properties of the matrix, this operation can be the most resource demanding in the method. Optimizing this operation as much as possible is therefore of key importance for accelerating Krylov solvers. Since each arithmetic operation in the SpMV operation requires only a few other variables, this problem can be considered as a memory-bound problem. Therefore, it is important to make the memory transactions on the GPU as efficient as possible. Using a straightforward implementation of a sparse-matrix storage scheme and a multiplication operation with a vector, usually results in a faster execution of the simulation compared to a CPU implementation, but it is in general not the most efficient with respect to the available resources on the GPU. Currently, there are a number of different approaches available for storing sparse matrices and performing a multiplication with a vector, e.g., *Compressed Sparse Row* (CSR) [67, 86, 194], *Diagonal Format* (DIA), Row-packed formats (ELLPACK/ITPACK), *Coordinate list* (COO), *Packet format* (PKT), *Hybrid format* (HYB) [18], *Block Compressed Sparse Row* [34, 183], *Jagged Diagonals* (JDS) [38], *Block ELLPACK* (BELLPACK) [40]. An extensive overview is given in [62]. Although some storage scheme is preferred over the other for some kind of matrices, the most important factor is the efficiency of the memory transactions performed by the storage scheme. Since the performance of the sparse-matrix vector multiplication is very important for various Krylov methods, this operation has received a lot of attention, see [18, 40, 76, 120, 192]. These operations are then used in GPU based Conjugate Gradient methods, see [26, 34, 38, 39, 68, 183], or applied on multiple GPUs [68, 78, 183]. Apart from Krylov methods, also Multigrid methods have received interest [49, 69].

In Chapter 3 an alternative GPU storage scheme for the BCSR format is presented. By reordering the individual matrix blocks in memory, large memory chunks can be transferred to the local processors, containing the blocks for multiple rows in the matrix. By using such memory transfers, it is easier to attain the maximum throughput of data and so to maximize the performance of the computations. Since the timings of these memory transactions are very predictable, it is even possible to approximate the expected throughput for any kind of matrix stored using this format.

3

Conjugate Gradient on GPUs

The Conjugate Gradient (CG) method is a widely-used iterative method for solving linear systems described by a (sparse) matrix. The method requires a large amount of Sparse-Matrix Vector (SpMV) multiplications, vector reductions and other vector operations to be performed. We present a number of mappings for the SpMV operation on modern programmable GPUs using the Block Compressed Sparse Row (BCSR) format. Further, we show that reordering matrix blocks substantially improves the performance of the SpMV operation, especially when small blocks are used, so that our method outperforms existing state-of-the-art approaches, in most cases. Finally, a thorough analysis of the performance of both SpMV and CG methods is performed, which allows us to model and estimate the expected maximum performance for a given (unseen) problem.

3.1 Introduction

The *Conjugate Gradient* (CG) method [88] is a widely-used iterative approach for solving linear systems. At each iteration the method performs a *Sparse-Matrix Vector* multiplication (SpMV). For an $M \times M$ *Symmetric Positive Definite* (SPD) matrix, it provably converges in M iterations [88]. For non-symmetric matrices, more general iterative methods have to be used, such as the *BiConjugate Gradient* (BiCG), *BiConjugate Gradient Stabilized* (BICGSTAB) or other related approaches [17, 153, 189]. In practice, the number of iterations required by the CG method is smaller than M, but the speed of convergence depends on the conditioning of the matrix [163]. However, even if the method converges in fewer than M iterations, typically a large number of iterations is still needed. Therefore, improving the speed of the SpMV operation is an important task, and one way of achieving this is through parallelization.

Modern programmable Graphics Processing Units (GPUs) have a highly-parallel architecture that provides a vast amount of computing power. This, together with the large memory bandwidth of these devices, made GPUs an important means for accelerating certain scientific computations [31, 32, 42, 117]. However, mapping a specific algorithm on such a parallel architecture is in general non-trivial.

The CG and the related numerical methods mentioned above are in fact notoriously difficult to parallelize efficiently because of their low arithmetic intensity and high memory-bandwidth demands. Since these methods perform a large number of SpMVs, it is of paramount importance to achieve the best possible performance for such operations. Depending on the underlying problem, different representations can be used for efficiently storing a sparse matrix and multiplying it by a vector. Additionally, PDE discretizations based on structured grids (e.g., typically used with the Finite Difference Method) result in a structured matrix, for which specialized storage representations can be used, whereas discretizations based on unstructured grids (e.g., Finite Element-based on tetrahedral grids) yield unstructured matrices. Throughout this chapter we mainly focus on problems yielding *sparse, unstructured matrices*. General formats for representing such matrices are the *Compressed Sparse Row* (CSR) and its extension – the *Block Compressed Sparse Row* (BCSR), see Section 3.2.

In this chapter, first we propose fast BCSR-based GPU mappings for the SpMV operation using CUDA, see Section 3.3. As suggested elsewhere [18, 34] and confirmed by the results presented here, a block-based layout for SpMV fits very well with the computational model of a GPU. Therefore, we start with a basic mapping which is subsequently transformed and optimized in a step-by-step fashion, so that in the end, we obtain an SpMV method that operates close to the limits of the hardware. Then, in Section 3.4, we propose an efficient GPU mapping of

the CG method, based on our SpMV operation, accelerated on single- and dual-GPU setups. Further, in Section 3.5, we introduce a framework for analyzing the performance of SpMVs and various vector operations performed on GPUs. Since most of these operations are bandwidth limited, we investigate the behavior of the memory throughput with respect to the problem size. Furthermore, we expect the performance of the CG method to be mainly driven by that of the SpMV operation, while the scalability among multiple GPUs should be greatly influenced by the bandwidth difference between the GPU memory and the inter-GPU throughput. Indeed, since the communication bandwidth between multiple GPUs is an order of magnitude smaller compared to the memory bandwidth on the GPUs itself, not all problems scale well across multiple GPUs [72]. Thus, estimating the performance of such parallel systems gives insight into the scalability issue, and enables one to answer questions like 'How many GPUs can be used to solve the problem efficiently?' or 'What performance can be expected for a given problem?'.

Since all operations appearing in the CG method are bandwidth limited (low arithmetic intensity), the behavior of the data throughput is studied and modeled using a mathematical model, see Section 3.5. First, the performance estimates through the proposed model are compared and checked against actual (measured) performance figures, on a number of linear systems. Comparisons are performed for different settings of the SpMV operation, using one or two GPUs. Then, a number of maps are computed (through extrapolation), which allows one to estimate the maximum and average performance of the CG method, given some parameters of the matrix describing an (unseen) problem, see Section 3.6. Such performance estimates can be used to quickly check if the method performs well on the given hardware, thus allowing the user the possibility of considering a different hardware setup. For instance, the user can choose to increase/decrease the number of GPUs used, or he/she can decide to even use the CPU, for better performance. In Section 3.7 we discuss additional aspects influencing the overall performance of the CG method, such as scalability for future devices, matrix reordering schemes and performance figures for double-precision computations.

Please note that throughout this chapter we focus on the CG method, but our model can also be used for the analysis of related Krylov-subspace methods, like the *BiConjugate Gradient* (BiCG) or *BiConjugate Gradient Stabilized* (BICGSTAB) method [17, 153, 189], or other numerical algorithms which perform SpMV and vector operations.

3.1.1 Previous and Related Work

The CG method and the SpMV operation have been implemented on various SIMD (multicore) platforms. In [192] an overview is given on the performance of the CSR-based SpMV operation for a number of modern CPU architectures. A similar comparison is made in [194], where the CG method is implemented on Woodcrest CPUs and NVidia 8800GTX GPUs. The authors report a speedup of about 3 times when using the CSR format on the 8800GTX GPU.

GPU implementations of the CG and multigrid sparse solvers were presented by Bolz et al. [26]. Their methods rely on the programmable graphics pipeline of modern GPUs and were implemented using fragment shaders. Sparse matrices are stored in the CSR format, enhanced by an additional array for storing the main-diagonal elements. The work of [34] is closely related to ours, in that they present CUDA-based GPU mappings of the CG and SpMV operation using the BCSR format. However, since their methods are not optimized, e.g., by using coalesced memory accesses, the peak performance of the underlying hardware was not reached, see Figure 3.9.

Bell and Garland [18] propose several methods for efficient sparse matrix-vector multiplication, which take into account the structure of the input matrices. They implemented efficient multiplication routines for various sparse matrix representations, such as the Diagonal format (DIA), Row-packed format (ELLPACK/ITPACK), Coordinate list (COO), Compressed sparse row (CSR), Packet format (PKT) and a *hybrid format*. Their hybrid layout is most suitable for unstructured matrices and delivers in general the best performance for such matrices. This approach stores a part of the matrix using ELLPACK and the remaining elements using the COO format. It is known that ELLPACK becomes inefficient if the numbers of elements per row varies greatly [18, 40, 120]. Although extensive results were provided, a complete analysis of their methods was not performed. However, they did suggest that the use of blocks may potentially improve the performance even further, but this was left yet to be explored. This extension was first presented by Monakov and Avetisyan [120]. These authors introduced a hybrid SpMV operation as a combination of the BCSR and BCOO format. Splitting the matrix and choosing the representation format is done using dynamic programming, but this approach has large memory requirements. According to their performance evaluation, 4×4 blocks seemed to yield the best efficiency, but no thorough analysis was performed to support this finding.

Cevahir et al. [38] propose an enhanced *Jagged Diagonals* (JDS) format, which reorders the matrix according to the number of non-zeros per row, and stores it similar to the CSR method. In [39] a parallel implementation of the CG method on a GPU cluster is presented. For the SpMV operation, their enhanced JDS format [38] along with the other formats from [18] were considered. For a given problem, the best layout for the SpMV operation is found by first benchmarking the performance of each individual format, and then selecting the one which delivered the fastest run. This is clearly disadvantageous, since first the input sparse matrix has to be off-line converted to a number of different layouts, which are then used to perform the SpMV operation.

In [40], the so-called *BELLPACK* method for SpMV multiplication is introduced. Although this method is similar to ours, there are also some important differences. In BELLPACK, first matrix blocks are identified, created and ordered, similar to [38]. After that, the ELLPACK method is employed on the ordered blocks. However, BELLPACK does not initially use coalesced memory transactions when loading the blocks. To improve on that the blocks are stored interleaved in memory, such that the memory transactions become coalesced, see [40]. As mentioned above, ELLPACK performs poorly when the numbers of elements per row varies greatly. This problem is addressed by sorting block rows, prior to storing them in the ELLPACK format. Since BELLPACK uses one CUDA thread to process one row, the number of registers used varies with the block size, making an analysis of the method more difficult. Within our method the number of registers used does not depend on the actual block size, which makes it easier to analyze its performance.

3.2 Background

3.2.1 The Conjugate Gradient Method

The CG method [88] is used to solve linear systems of the form

$$\mathbf{b} = \mathbf{A}\mathbf{x}, \tag{3.1}$$

with $\mathbf{A}$ a symmetric ($\mathbf{A}^T = \mathbf{A}$) and positive-definite matrix ($\mathbf{y}^T\mathbf{A}\mathbf{y} > 0$, with $\mathbf{y} \neq \mathbf{0}$), and $\mathbf{x}$ the vector of unknowns. In various textbooks (e.g., [17, 35, 145]), it is shown that the CG method minimizes the quadratic function

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}, \tag{3.2}$$

thus solving for $\mathbf{x}$ in Equation (3.1), see Appendix B.2 for more details. Algorithm 2 in Section 3.4 shows a parallel realization of the CG method using a Jacobi preconditioner. Please note that it is straightforward to extend Algorithm 2 to other related methods, such as the BiConjugate gradient or the BiConjugate gradient stabilized methods. Further, the algorithm can easily be modified to accept other preconditioners, such as the Incomplete Cholesky [75], although the computations of such preconditioners are very difficult to parallelize efficiently because of data dependencies.

Due to the iterative nature of the method (see Algorithm 2), a large number of SpMV multiplications have to be performed when solving the linear system. Therefore, it is essential to optimize the SpMV operation as much as possible. Depending on the structure of the sparse matrix, different approaches exist to represent the matrix and to perform SpMVs on various types of hardware. Below, we briefly describe the *Compressed Sparse Row* (CSR) and *Block Compressed Sparse Row* (BCSR) formats; see, e.g., [18] for full details and other Sparse-Matrix storage schemes.

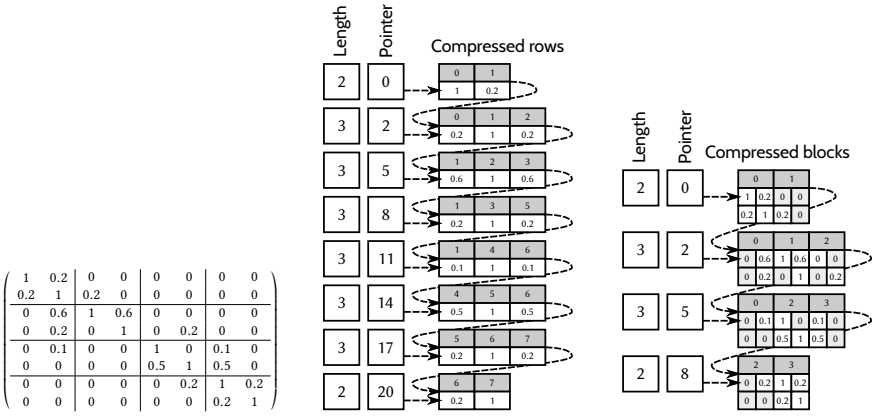
3.2.2 Compressed Sparse Row (CSR)

The CSR format is a well-known, general storage scheme suitable for *unstructured* matrices. Each non-zero element in a row and its column index are stored in two continuous arrays. Because of this, each row needs a *pointer* to the first element in the array of data elements and indices. The number of non-zero elements in a particular row can be determined by computing the difference between the pointer of the current and the next row. Figure 3.1b illustrates the CSR storage scheme.

3.2.3 Block Compressed Sparse Row (BCSR)

BCSR constitutes a generalization of the CSR format. This scheme divides the input matrix of size $M \times M$ in blocks of $P \times Q$ elements, and stores each non-empty block similar to the CSR method, see Figures 3.1a and 3.1c. Each block row contains a number of non-empty blocks, and each block contains a number of non-zero elements. Note that zero elements inside a block are stored explicitly. For each block, the column index is stored, and for each block row, its length and pointer to the first block in the block row are represented, see Figure 3.1c.

Since all values of a block are consecutively stored in memory, the use of blocks reduces cache misses [76], when employed on CPUs, and improves the efficiency of memory transactions on GPUs. Therefore, larger block sizes should in principle lead to better performances, but they



(a) Sparse matrix divided in blocks of size 2×2.

(b) CSR. Rows in (a) are compressed and each non-zero element is stored with its column index (gray cells). For each row, its length and pointer to the first element are stored.

(c) BCSR. 'Block rows' in (a) are compressed, and each *non-empty* block is stored along with its column index (gray cells). For each block row, its length and pointer to the first block are stored.

Figure 3.1: Compressed Sparse Row (CSR) and Block Compressed Sparse Row (BCSR) storage schemes.

may introduce many zero elements. This usually results in wasted computations and memory space. We choose to use square blocks of size $N \times N$, with N a power of two. This fits best the architecture of many SIMD CPUs, as well as the architecture of modern GPUs.

The best block size with respect to performance depends on many aspects. Apart from the block density, also the problem size, hardware architecture and the runtime GPU configuration determine which block size gives the best performance; in Section 3.7.3, we shall further discuss this issue.

3.2.4 CUDA Overview

With the release of NVidia's G80 [129] and NVidia Tesla [109] series GPUs, general purpose computing was truly enabled via the so-called *Compute Unified Device Architecture* (CUDA). In this section we give a short overview of the architecture and its constraints, as exposed through CUDA; detailed information can be found in [132].

A typical modern GPU consists of a large number of *unified shaders* which can be used as either *vertex-, pixel- or geometry-shaders* in graphics applications. Within the context of general-purpose computing, a group of unified shaders is called a *multiprocessor* [129, 131, 132]. On a global scale, the multiprocessors are connected with the *global memory* of the device through a number of memory controllers. On a local scale, each multiprocessor contains a relatively small amount of memory that is shared among the scalar processors of the multiprocessor (*shared memory*).

The communication between the global memory and the individual scalar processors has a relatively high latency. Between a memory request and the moment when data is available, each processor has to wait between 400 and 800 clock cycles. However, each multiprocessor

is able to execute up to 1024 threads, so that this latency can be hidden as follows. When a thread requests data from (non-cached) global memory, the scheduler activates another thread, whereas the thread requesting data is put to sleep. Once data becomes available, the idle thread is activated and allowed to continue its execution. By scheduling a large amount of threads on a small number of scalar processors, most of the memory latency can be hidden.

Within CUDA, thread scheduling is done automatically. Launching a program on the GPU (*kernel*) creates a *grid* containing a number of *thread blocks*, each containing threads that will execute the kernel function. Within each thread block, threads can communicate with each other through shared memory. However, threads belonging to different thread blocks cannot directly communicate, since thread blocks can be executed on different multiprocessors and at different time intervals. Each thread block is divided in smaller *warps* of 32 threads which are executed on the scalar processors. For multiprocessors having eight scalar processors, each instruction is executed four times with different sets of threads. Depending on the shared memory and register requirements of a kernel, a multiprocessor can run concurrently a given number of thread blocks. The ratio between the number of active threads and the maximum number of threads per thread block (1024) is called the *occupancy* of the kernel [134]. For a typical problem solved using CUDA, a large number of thread blocks are created which are distributed over all available multiprocessors. Once a thread block has finished its computations, a new thread block is started. This process is repeated until all thread blocks have performed their task.

Threads that belong to the same warp, execute the same instruction. If a thread within a warp follows a different branch of a conditional statement than the other threads, *thread divergence* occurs, and each thread follows both branches of the statement. If a particular thread does not need the results obtained by following one of the branches, these results are simply discarded. Although thread divergence reduces the overall performance of a kernel, in some cases it is unavoidable.

The architecture has several limitations also with respect to the pattern of global memory accesses. The global memory is divided in a large number of segments with a particular size. A typical memory segment is 128 bytes wide, depending on the version of the architecture. If all threads of a half-warp access data stored in the same segment of the global memory, only one combined memory transaction is initiated. This is known as a *coalesced memory access*. If threads access different segments, separate memory transaction must be initiated, which increases the total latency and should be avoided if possible. With the release of the Fermi architecture [131] global memory accesses are cached, leading to increased data throughput.

Modern GPUs have a large amount of computing power compared to a single CPU. For example, a GTX570 has 15 multiprocessors each containing 32 scalar processors, which makes for a total of 480 scalar processors running at 1464 MHz. The theoretical peak performance of such a device is about 1405 GFLOPS, if only floating-point multiply-add operations are performed. The theoretical memory bandwidth is about 152 GB/s. In order to reach the best performance for a specific problem, some guidelines must be followed, see [132]. First, the memory transactions should be coalesced. Second, thread divergence should be avoided. Finally, the utilization level should be maximized, i.e., for a specific task, as much as possible computational resources should be used. In practice this means that a problem should be solved using a large amount of threads.

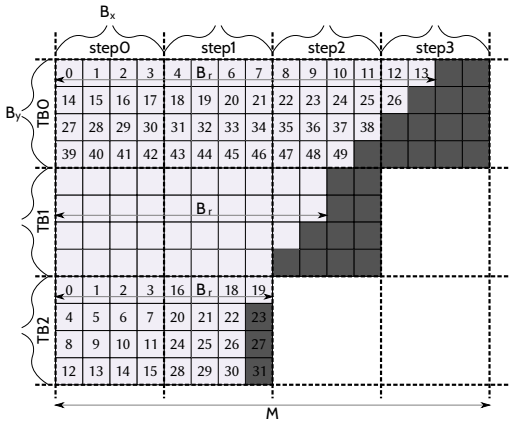


Figure 3.2: Mapping the sparse matrix on thread blocks. Depending on strategy, B_x and B_y are computed as in Table 3.1, given the number of threads T and block dimension N. The dark-gray regions indicate non-existing (empty) blocks used for padding. Each individual cell represents one matrix block with size $N \times N$. B_r indicates the number of blocks in the largest block row assigned to a particular thread block. Each thread block (TBi) performs a number of steps (proportional to $\lceil B_r / B_x \rceil$), to cover all assigned matrix blocks. During each step, each thread block processes collectively $B_x \times B_y$ matrix blocks containing $N \times N$ elements. Block rows were first *sorted* by length as described in Section 3.3.5. For thread block $TB0$, the initial indices of the matrix blocks, stored in the BCSR format, are shown. The matrix blocks assigned to thread block $TB2$ are reordered as described in Section 3.3.4.

3.3 Proposed SpMV Method using CUDA

The SpMV operation can be mapped to a GPU using different strategies, each with its own advantages and disadvantages. In general, mapping a certain problem to a GPU starts with identifying those parts of the algorithm that can run independently from other parts. Within the SpMV operation this is clearly the computation of a single element in the result vector. Throughout this section we shall seek for a GPU mapping of the SpMV operation which gives the best overall performance.

Thread blocks offer the lowest level of parallelism on GPUs, so they are used to compute one or multiple elements of the result vector. Figure 3.2 shows how a sparse matrix, stored using the BCSR format with $N \times N$ blocks, can be mapped to a GPU using CUDA. The actual mapping depends on the number B_y of *block rows* processed by each thread block, and the number B_x of *matrix blocks* processed collectively per block row, see Table 3.1. The total amount of thread blocks executed concurrently on a GPU depends on the number of multiprocessors and the number of active thread blocks, and represents the *occupancy* of a kernel [134]. Table 3.1 defines and describes each parameter appearing in Figure 3.2 for each different strategy. Since the *warp size* is 32, we use square blocks of dimension N, where N is a power of 2. This choice results in easy-to-implement kernels and reduces the amount of inactive threads. For each different mapping, each individual matrix block is processed by threads with consecutive thread indices. For an $N \times N$ matrix block with index i, threads $i \times N \times N$ till $(i+1) \times N \times N - 1$ perform the computations and data lookup. Here i represents the index of the matrix block in the *current step* in *row-major order*, starting with zero, which should not be confused with the index of the block in memory.

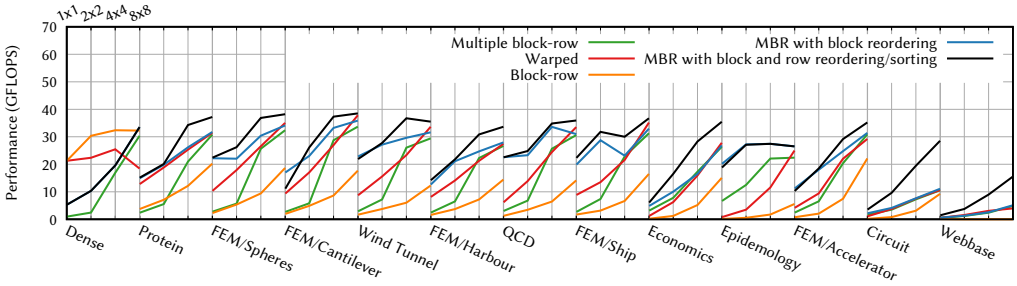


Figure 3.3: Raw performances for the SpMV operation by our mappings, with varying block sizes, performed using a GTX280 GPU.

Parameter	Description	Block row		Warp		Multiple block-row
B_x	Number of collectively processed *matrix blocks* per *block row*.	$\frac{T}{N \times N}$	$>_*$	$\frac{W}{N \times N}$	$>_{**}$	1
B_y	Number of collectively processed *block rows* per *thread block*.	1	$<_*$	$\frac{T}{W}$	$<_{**}$	$\frac{T}{N \times N}$
$B_T = \frac{M}{B_y \times N}$	Total number of *thread blocks* needed to cover the sparse matrix.	$\frac{M}{N}$	$>_*$	$\frac{M \times W}{N \times T}$	$>_{**}$	$\frac{M \times N}{T}$
Steps $= \frac{B_r}{B_x}$	Total number of steps needed to process all *matrix blocks* per *block row*.	$\frac{B_r \times N \times N}{T}$	$<_*$	$\frac{B_r \times N \times N}{W}$	$<_{**}$	B_r

N	Dimension of one *matrix block*, which is a power of 2. Usually 1, 2, 4 or 8.		
T	Number of threads per *thread block*, usually 128 or 256.	**Remarks**	
W	*Warp size* = 32.	$*$	Holds if $T > W$.
M	Matrix dimension.	$**$	Holds if $N < 4$.
B_r	Largest number of *matrix blocks* per *block row* per *thread block*.		

Table 3.1: Parameter definitions for each mapping strategy, see Figure 3.2.

The following subsections describe the three proposed basic strategies presented in Table 3.1: *block row mapping*, *warp mapping* and *multiple block-row mapping (MBR)*. Each strategy maps the computations differently among the available threads of one thread block. On top of the best basic strategy, we apply a few optimizations which further increase the performance, see Sections 3.3.4 to 3.3.6.

To estimate the efficiency of each mapping with varying block sizes, we compute what we call 'raw-performance', defined as the number of GFLOPS achieved, if each matrix block had a *density* of 100%, i.e., each block contained $N \times N$ non-zero elements. Figure 3.3 shows the raw-performance of each mapping for the set of matrices in Table 3.4. We prefer to use raw-performances instead of actual (measured) performances, because they better reflect the differences between each mapping.

3.3.1 Block-Row Mapping

'Block-row mapping' assigns one block row to one thread block. In each *step*, each thread within a thread block loads one matrix element and its corresponding *vector element* from *global memory*. Each vector element is loaded by first computing its index using the *column index* of the matrix block and the *thread index*. Once both matrix- and vector-elements are loaded, they are multiplied and added to a per-thread intermediate value. When all matrix blocks of the current block row are processed, the intermediate values are *reduced* (addition

operation) to a column vector of size N. Finally, after reducing N row vectors, the result is stored in global memory. Note that $B_y = 1$ for this strategy, i.e., one block row is processed by one thread block.

The number of collectively-processed matrix blocks per block row, B_x, is computed using the formulas in Table 3.1. Since each thread block processes exactly one block row, a large number of thread blocks (B_T) are needed to cover the whole matrix. Since in each step a large number of matrix blocks are processed together, the number of *steps* performed by each thread block is small. Thus, computations can be wasted if B_r is not a multiple of B_x. For example, if $N = 1$, then $B_x = T$, which implies that a number T of 1×1 matrix blocks are processed together. In general T equals 128 or 256, which means that B_r must be a multiple of 128 or 256 in order to minimize the number of wasted computations. Clearly, this is impractical in most situations. Note that increasing N gives in general better performances, since the amount of wasted computations is reduced, and the memory transactions for vector elements become more efficient.

3.3.2 Warp Mapping

'Warp mapping' assigns one block row to one warp of threads. The computation of the individual elements is similar to the block-row mapping. The difference between both methods lies in the configuration parameters, see Table 3.1.

Since each block row is mapped to one warp, the number of collectively-processed matrix blocks per block row is smaller than with the block-row mapping strategy. This implies that the number B_y of block rows processed per thread block is larger. Since B_x is smaller, the amount of wasted computation is in general smaller compared to block-row mapping. Furthermore, the number of steps required to process one block row is increased, which results in less additional overhead and thus a higher *memory throughput*. Figure 3.3 clearly shows the performance improvement of this mapping compared to the single block-row case, except for matrix 'Dense'.

3.3.3 Multiple Block-Row Mapping

'Multiple block-row' mapping (*MBR*) is the opposite of the block-row strategy. Instead of processing multiple blocks belonging to one block row, the mapping is transposed such that in each i-th step, each i-th block of the block rows is processed together. For each block row, exactly one block is processed, together with blocks belonging to other block rows. The actual computation of the result vector is similar to the previously described mappings, while the layout is different.

Within the MBR mapping, the number B_x of matrix blocks processed together per block row is exactly one. This implies that B_y should be as large as possible. Because $B_x = 1$, the number of steps required becomes exactly B_r. Furthermore, the number of necessary thread blocks decreases. Since the number of steps is maximized and the number of thread blocks is minimized (while the total amount of work remains constant), the work per thread is maximized. The main advantages of this approach are that the additional overhead is reduced and less space and computations are wasted. However, if the variation of row lengths of block rows (assigned to the same thread block) is large, a large number of threads *can* become idle. Another potential drawback is that matrix elements are loaded in a different order than they are initially stored, which results in un-coalesced memory accesses for specific block sizes. Figure 3.3 shows that (for all matrices) the performance when $N = 1, 2$ is smaller compared to warp-mapping, due

to non-coalesced memory transfers in these cases. However when $N > 2$, the performance is similar to that of the warp-mapping. To overcome these problems when $N = 1, 2$, we use two reordering steps described below.

3.3.4 Block Reordering

For each mapping strategy, each matrix block is processed by *consecutive threads*. Furthermore, blocks are stored at memory locations given by their indices. (The top-most thread block of Figure 3.2 shows these block indices.) Because of this, memory transactions are coalesced (when loading the matrix blocks) if $B_x \times N^2 \geq 16$. For the MBR mapping, coalescing becomes problematic if $N < 4$, since $B_x = 1$. Note that for other strategies this can never happen, because $T \geq W$ in all other cases. Figure 3.3 shows this effect happening when $N < 4$. For example, if $N = 1$ and $B_x = 4$, each step loads data from 16 blocks that are clearly not stored in the same memory segment. The first step of the top-most thread block in Figure 3.2 loads blocks $0 - 3$, $14 - 17$, $27 - 30$ and $39 - 42$ from memory.

To overcome this problem, the order of the matrix blocks in memory must be changed, such that each thread within a half-warp reads from the same memory segment. In Figure 3.2, matrix block with index 14 of thread block $TB0$ is moved to position 4, block 4 to position 16, so that the final configuration is similar to that shown for thread block $TB2$. By reordering matrix blocks such that those blocks processed within the *same step* of the *same thread block* are consecutive in memory, all memory transactions (required for loading matrix blocks) become coalesced. In Figure 3.2, all blocks within $TB2$ are reordered such that all threads read from consecutive memory locations, while all threads assigned to $TB0$ read from non-consecutive memory locations. However, if $N = 4$, this problem is solved, since each half-warp reads exactly one matrix block from memory. Since blocks now are stored at different memory locations, all blocks that are loaded within each step are consecutive in memory, for any size of N. This reordering strategy boosts significantly the performance of the SpMV operation for matrix blocks with $N < 4$, as shown in Figure 3.3.

If block rows within one thread block have different lengths, or if B_r is not a multiple of B_x, empty matrix blocks must be added until each block row assigned to the same thread block has the same amount of matrix blocks. This enables an efficient computation of the memory locations of the blocks, given the thread-id's, step number and a starting offset for the current thread block. Once each thread has computed a memory location for the first step, it is increased by the number threads T, for each following step. Further, since after padding, each block row processed by a thread block has the same length, we do not have to check if the currently processed block exists. This eliminates the possibility that thread divergence occurs at this stage.

If the block rows are sorted by their length, described in Section 3.3.5, the number of additional empty blocks is reduced since block rows with similar lengths are processed together, see Figure 3.2. Furthermore, reordering the matrix blocks does not affect the structure of the matrix: blocks are only stored at different memory locations, but the structure of the matrix remains untouched.

For looking-up vector elements used to multiply with one matrix block, the column index of the matrix block is required, see Figure 3.1c. These indices are stored in the same order as previously described. All threads that are used to load one particular matrix block, also load the column index for that block. Since each thread accesses the same memory location, and because column indices are stored in the same order as for the matrix blocks, this transfer is

always coalesced. Using the column index, block-size and thread index, the index of the vector element is computed and used for looking-up a vector element. This lookup is only coalesced if $N \geq 4$. For smaller block-sizes, threads within a half-warp *can* read data from multiple memory segments. In this case, multiple transactions (one per memory segment) are needed.

3.3.5 Block-Row Sorting

Sorting block rows by their lengths results in an ordering so that block rows with similar lengths are spatially close to each other. Thus, sorting reduces the variation of block-row lengths for block rows assigned to the same thread block, as shown in Figure 3.2. This in turn results in a more balanced computation within one thread block, leading to an improved performance. Furthermore, the amount of empty matrix blocks used for padding (required after reordering the matrix blocks) is reduced. As shown in Figure 3.3, sorting improves the performance in most cases. Additionally, for matrices with a large variation in row-lengths (e.g., 'Circuit' and 'Webbase'), an even larger improvement is obtained, since the amount of added empty blocks is significantly reduced. Note that sorting block-rows does not influence the performance of matrix 'Dense'. This matrix has homogeneous row lengths, so sorting does not change the order of the rows.

Since in general, the order of the block rows is changed, one extra index per block row has to be used, such that the result is stored at the right position in the result vector. This ensures that sorting the block rows does not change the result of an SpMV operation. Since only one extra value is transferred per block row, the added overhead is negligible while the improvement can be significant. Note that sorting block rows improves the performance only if $B_y > 1$. If $B_y = 1$, no sorting is required since only one block row is processed per thread block.

3.3.6 Fine Tuning

The MBR mapping yields in most cases the best performance. One drawback of this mapping is that the amount of thread blocks is relatively low compared to the block-row mapping strategy, while the number of computations per thread is high. For example, using the MBR mapping, matrix 'Dense' with $M = 2000$, 256 threads per thread block and $N = 1$, each thread block maps to 256 rows in the matrix. So, in total 8 thread blocks are used for the multiplication operation. Clearly, this number is too low to reach a good utilization of the GPU. However, by setting $B_x = 2$, twice the amount of thread blocks are created because B_y is halved, leading to better performance. Note that another way of increasing the amount of thread blocks is by increasing N, see Table 3.1. Additionally, for matrices having a high variation in row length, increasing B_x further reduces the number of empty blocks required for block-row padding, which also results in better performance figures, see Section 3.6.1.

3.3.7 Final SpMV Mapping

Throughout this chapter we use the MBR mapping combined with block-row sorting and block reordering. Because the number of thread blocks is the smallest, and the number of steps is the largest (Table 3.1), the amount of work per thread is the largest among our mapping strategies. In general, this results in a higher memory throughput, as shown in Figure 3.3. Finally, in a few cases it is worthwhile to increase B_x (see 'Best' performance in Figure 3.9), however for our performance analysis from Section 3.5 we have used $B_x = 1$.

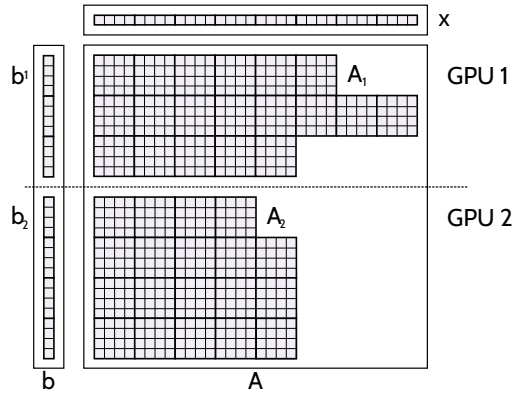


Figure 3.4: Block distribution over two GPUs. Each GPU is assigned a similar amount of matrix blocks: here, each GPU processes 19, 4×4 matrix blocks, distributed over a (different) number of block-rows.

Further, *textures* are used to enable cached memory accesses, and thus to improve the memory throughput when fetching vector values, similar to [18]. The effects of cached memory accesses is significant if $N < 4$. For $N \geq 4$, the improvements are minimal, since the memory transactions are already close to optimal. Finally, the results in Section 3.6 show that this method delivers the best performance in most of the test cases.

3.4 Parallel (Multi-GPU) Conjugate Gradient

Algorithm 2 shows the pseudo-code of our parallel (multi-GPU) Jacobi-preconditioned CG method. In the following subsections the individual steps of this method will be explained further.

3.4.1 Parallel SpMV

Since the computation of one element in the result vector is independent from the computation of other elements in the result vector, the SpMV operation is parallelized by distributing the computation of the elements in the result vector over the available GPUs.

Because the BCSR format represents a sparse matrix by a collection of block rows, the matrix is divided in a certain number of segments of consecutive block rows, where each segment has a similar amount of matrix blocks. Each segment is then mapped to one GPU. Sorting the block rows as described in Section 3.3.5 is preferably done after the segments are created. If sorting is performed prior to segmentation, this results in an unbalanced GPU load, i.e., the block rows assigned to the first GPU will generally be longer than the block rows assigned to the other GPUs. Note that in order to perform an SpMV operation, vector $\mathbf{x}$ must be available to each individual GPU. Accordingly, sub-matrix $\mathbf{A}_i$ of (global) matrix $\mathbf{A}$, stored on GPU i, is multiplied by $\mathbf{x}$, see Figure 3.4. The result of the SpMV operation on GPU i, is $\mathbf{b}_i = \mathbf{A}_i\mathbf{x}$, where $\mathbf{b}_i$ is a vector which size corresponds with the number of rows of sub-matrix $\mathbf{A}_i$, Lines 1 and 15 of Algorithm 2.

Algorithm 1: parallel_reduction($\mathbf{x}_i, n, i$)

Input : Vector $\mathbf{x}_i$, n: number of GPUs, and i: index of current GPU.

Output: $\sum \mathbf{x}_i$

1 $r_i = \sum_{j=1}^{\#\mathbf{x}_i} \mathbf{x}_{i,j}$ /* Parallel reduction */

2 SyncGPUs()

3 **return** $\sum_{j=0}^{n} r_j$ /* Collect and return */

3.4.2 Vector Operations

Standard vector operations, like addition, subtraction, scalar multiplication or elementwise multiplication, can easily be parallelized. Figure 3.4 shows the distribution of vector $\mathbf{b}$ over two GPUs, where each part, $\mathbf{b}_i$, matches the number of rows of the sub-matrix assigned to GPU i, $i = 1, ..., n$, with n the number of GPUs. Other vectors appearing within the CG method are distributed similarly among the available GPUs, see Algorithm 2.

Since standard vector operations need to access vector elements with the same indices, no dependencies exist between the data segment of GPU i and the data of other GPUs, i.e., all required data is stored in the memory of GPU i. This means that it is possible to group such vector operations in one CUDA kernel, and thus perform a larger number of vector operations sequentially, without the need to synchronize the GPUs. For example, in Algorithm 2, all vector operations in Lines 2 to 6 are performed by one CUDA kernel – CG1. This is advantageous, as shown in Section 3.5.2.

3.4.3 Vector Reductions

A parallel vector reduction [81] using multiple GPUs requires more effort. Since each GPU contains only a part of a complete vector to be reduced, a final reduction needs to be performed among these parts to yield the final result, see function `parallel_reduction` (Algorithm 1). Before the final reduction can be computed (Line 3), each GPU must have finished computing its (partial) reduction and must have stored its result in the host memory, Line 1. By synchronizing among the GPUs, one can assure that each GPU has finished its work. Finally, each host thread (initiating computations on one GPU) computes the final reduction result, by collecting and reducing the results of all GPUs, which were previously stored in the host memory.

In order to prevent *race conditions*, a synchronization barrier among GPUs would also be required, after computing the final reduction per host thread. If this synchronization is omitted, a thread can for example perform a subsequent reduction and overwrite the previous reduction result. Preventing overwriting previously stored values, without using a synchronization point, can only be guaranteed if two successive parallel reductions use different storage locations. In Algorithm 2, function `parallel_reduction` uses the latter approach, which explains the need for three temporary vectors $\mathbf{tj}_i$, with $j = 1, 2, 3$. Although additional storage is required, the reduction in the number of synchronization points among GPUs results in improved overall performance.

3.4.4 Parallel CG

The CG method is parallelized by replacing each vector operation, reduction and SpMV operation, by their parallel equivalent, see Algorithm 2. Since the parallel SpMV operation requires a complete vector $\mathbf{v}$, and each GPU stores just a part ($\mathbf{v}_i$) of the complete vector, that vector

has to be reconstructed and updated at each iteration on each GPU. First, each vector part, $\mathbf{v}_i$, is copied to the host memory by host thread i; this is denoted by $\mathbf{v}_{\mathbf{host,i}} \leftarrow \mathbf{v}_i$ in Line 11. Synchronizing among the GPUs (Line 12) ensures that after the synchronization point, each individual part of $\mathbf{v}$ is available in the host memory. Then, each host thread copies the other parts of $\mathbf{v}$ to their assigned GPU, denoted by $\mathbf{v} \leftarrow \mathbf{v}_{\mathbf{host,j}}$ in Line 14. Once $\mathbf{v}$ is reconstructed on a GPU, that GPU can start immediately performing the SpMV (Line 15), followed by an elementwise multiplication in Line 16, which is denoted by '$\cdot*$'.

When vector $\mathbf{v}_i$ is computed on a GPU, the corresponding part of the complete vector $\mathbf{v}$, stored on the same GPU, is also updated. This is denoted by $\mathbf{v} \leftarrow \mathbf{v}_i$ in Line 30, and is combined with other vector operations in order to increase the memory throughput.

3.5 Performance Analysis

In this section we analyze the performance of the CG method described in Section 3.4. Since the operations appearing in the CG method are bandwidth limited, the best performance is reached when the memory throughput is maximized. Thus, here we focus on analyzing the memory throughput of our method. First, the pure memory throughput is obtained for different kernel configurations. Next, the actual performance and scalability of the operations are estimated, which leads to the (*maximum* and *average*) performance estimate of the complete CG method. When multiple GPUs are used, also the memory throughput between the devices plays an important role. All observations are combined into a model, which is then used to estimate the theoretical maximum performance and the average performance of the CG method, given the properties of both the matrix and hardware. Furthermore, this model is also used to determine the scalability of the CG method, given an unseen linear system. The analysis and performance estimations are performed on a machine equipped with an Intel Q6600 quad-core CPU and two NVidia GTX280 GPUs managed by an NVidia nForce 790i SLI chipset.

3.5.1 Memory Throughput Estimation

The CG method consists of two different types of operations: vector operations and the SpMV multiplication. Each kernel implementing these operations requires a suitable run-time configuration, in which the thread block and *grid* dimensions are specified. The thread block dimensions are in general fixed, while the grid dimensions can either be *fixed* or *variable*. To reveal the behavior of each configuration, we have performed a simple benchmark, in which a number of n vectors of a given size m are loaded from (global) memory. The results presented in Figure 3.5 were obtained by repeating this test for different vector sizes m and different numbers n of vectors. For each configuration the same amount of data is transferred.

For *fixed grids*, the minimum number of thread blocks B_T needed to fully occupy the device is given by $B_T = 1024\,P/T$, with 1024 the maximum number of threads per multiprocessor, P the number of multiprocessors and T the number of threads per thread block. In this case, each thread executes a loop, and in each step n vector elements are loaded.

With *variable grids*, $B_T = \lceil m/T \rceil$, so that the number of thread blocks directly depends on the problem size and the number T of threads per block. In this case, each thread loads exactly n vector elements. Therefore, the total number of thread blocks needed to cover the complete computation is much larger than with fixed grids. Since the amount of work per thread

Algorithm 2: Parallel CG on multiple GPUs.

Input : Matrix $\mathbf{A}$ and vector $\mathbf{b}$, $\mathbf{c}$: preconditioner, n: number of GPUs, i: index of GPU i, TOL: tolerance, and n_{iter}: maximum number of iterations.

Output: Vector $\mathbf{x}$, with $\mathbf{Ax} = \mathbf{b}$.

```
1  r_i = A_i x;                                                    /* SpMV */
2  r_i = b_i − r_i;                                                /* CG1 */
3  w_i = c_i · ∗r_i;                                               /* CG1 */
4  v_i = c_i · ∗w_i;                                               /* CG1 */
5  t1_i = w_i · ∗w_i;                                              /* CG1 */
6  t2_i = v_i · ∗v_i;                                              /* CG1 */
7  v ← v_i;                                                        /* CG1 */
8  α = parallel_reduction (t1_i, n, i);
9  r = sqrt (parallel_reduction (t2_i, n, i));
10 for k_i ← 0 to n_iter ∧ r < TOL do
11     v_host,i ← v_i;
12     SyncGPUs();
13     foreach 0 ≤ j < n ∧ j ≠ i do
14         v ← v_host,j;
15     u_i = A_i v;                                                /* SpMV */
16     t1_i = u_i · ∗v_i;                                          /* CG3 */
17     t = α/ parallel_reduction (t1_i, n, i);
18     x_i = x_i + t v_i;                                          /* CG4 */
19     r_i = r_i − t u_i;                                          /* CG4 */
20     w_i = c_i · ∗r_i;                                           /* CG4 */
21     t2_i = w_i · ∗w_i;                                          /* CG4 */
22     β = parallel_reduction (t2_i, n, i);
23     s = β/α, α = β;
24     if β < TOL then
25         t1_i = r_i · ∗r_i;                                      /* CG2 */
26         if parallel_reduction (t1_i, n, i) < TOL then
27             return x;
28     v_i = c_i · ∗w_i + s v_i;                                   /* CG5 */
29     t3_i = v_i · ∗v_i;                                          /* CG5 */
30     v ← v_i;                                                    /* CG5 */
31     r = sqrt (parallel_reduction (t3_i, n, i));
```

block is independent of m, the kernel start-up cost (overhead) per thread block is relatively high compared to the total time used by one thread block, which results in a lower memory throughput.

The variation in throughput with the number of memory transactions can be approximated using a *sigmoid function*, see Figure 3.5. For vector operations, fixed grids deliver in general the best performance, while variable grids also give satisfactory results if each thread loads more than one vector, i.e., $n \geq 2$. Therefore, all vector operations are implemented using a fixed-grid approach.

If a fixed-grid approach is used for the SpMV operation, an extra loop is introduced in the CUDA kernel, which results in a larger amount of registers being used. This negatively affects the *occupancy* of the SpMV kernel, and so its performance. Thus, the SpMV operation (Section 3.3) uses a variable-grid approach. Since the SpMV kernel already transfers a large amount of data in a loop, a high throughput is obtained, thus justifying the choice of a variable-grid.

Figure 3.5 shows the graph of the memory throughput versus the total number of transferred elements. We use the following scaled and shifted sigmoid function to model the memory throughput

$$B(x, \mu, \sigma, \nu) = \nu \left(1 + e^{-\left(\frac{\log_2 x - \mu}{\sigma}\right)}\right)^{-1}, \tag{3.3}$$

with x the total number of transferred elements, and μ, ν and σ, model parameters. Further, the Levenberg-Marquardt algorithm [145] was used to fit the sigmoid curve, leading to the (hardware-specific) parameters given in Figure 3.5. Note that all graphs below, in which Equation (3.3) is used, are plotted using a logarithmic scale.

In order to reach the best memory throughput, each multiprocessor must be fully occupied, i.e., each multiprocessor should have at least the maximum number of threads running. Also, each multiprocessor must initiate as many as possible memory transactions. For example, the NVidia GTX280 GPU can handle up to 1024 threads per multiprocessor and contains 30 multiprocessors, hence a minimum number of 30, 720 threads must be active to fully occupy the GPU. If each thread initiates exactly one memory transaction, each kernel launch introduces a relatively large amount of overhead compared to the memory latency, hence the relatively low throughput in this case, see Figure 3.5. Further, to saturate the memory bus, each thread needs to initiate a large number of memory transactions, such that latencies can be hidden. Equation (3.3) can be used to approximate the memory throughput for any GPU, however in Figure 3.5 we show results for the GTX280 GPU. If a particular kernel is bandwidth limited (similar to those implementing the SpMV and vector operations), one can use this approximation to estimate the total execution time of the kernel, given the problem size. However, the computations performed by a kernel do change slightly the estimation parameters.

3.5.2 Analysis of Vector Operations

A number of vector operations, within the CG algorithm, can be *combined* into a few larger kernels. This allows better hiding of memory latencies, so that the performance is improved. Figure 3.6a shows the measured throughput for each combined vector operation and vector reduction (*Red.*), whereas Table 3.2 shows the number of floating point operations and transferred elements for each kernel. Further, Algorithm 2 shows which operations are executed by which kernel listed in Table 3.2. The data is obtained from both single and dual GPU benchmarks of the CG method, using a large set of test matrices [48]. The numbers of transferred

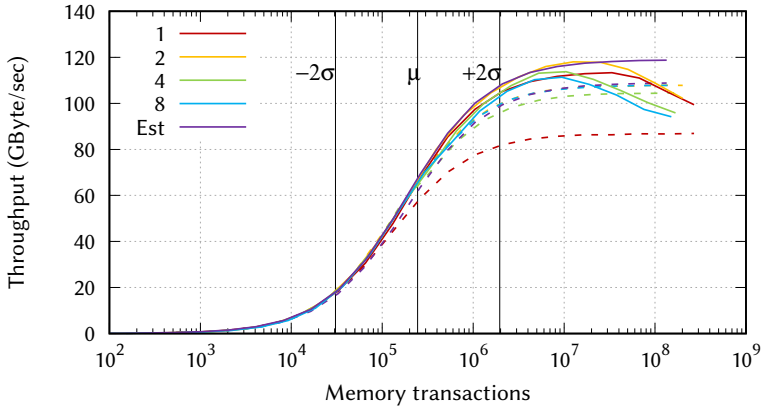


Figure 3.5: Measured and estimated ('Est') memory throughput on a GTX280 GPU for fixed grids (solid), with $\mu = 17.9, \sigma = 1.5, \nu = 119$, and variable grids (dashed), $\mu = 17.9, \sigma = 1.5, \nu = 109$, using an increasing amount $n = 1, 2, 4, 8$ of loaded vectors.

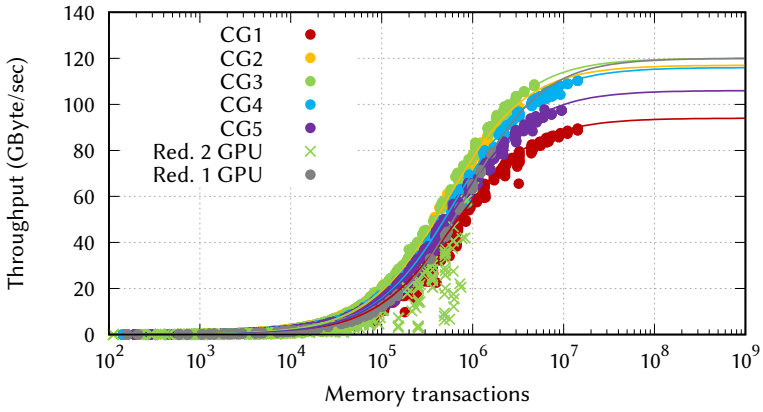
Kernel	transactions m	FLOPS	μ	σ	ν
CG1	$x \times 9$	$x \times 5$	19.2	1.45	94
CG2	$x \times 2$	$x \times 1$	19	1.45	117
CG3	$x \times 3$	$x \times 1$	19	1.45	120
CG4	$x \times 9$	$x \times 6$	19.3	1.45	116
CG5	$x \times 6$	$x \times 4$	19.2	1.45	106
Red.	$x \times 1$	$x \times 1$	19.7	1.4	120

Table 3.2: Number of memory transactions and FLOPS for each kernel used in Algorithm 2. Here x is the vector size, and parameters μ, σ and ν are used to approximate the memory throughput of that specific kernel.

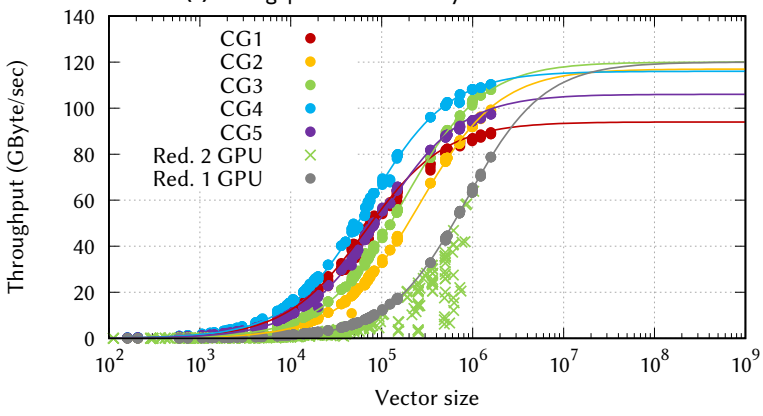
elements and FLOPS in Table 3.2, are obtained by counting the number of vectors loaded or saved by the kernel and by counting the number of floating point operations performed by that kernel. This information is obtained via Algorithm 2. For example, kernel *CG1* load vectors $\mathbf{r}_i, \mathbf{b}_i$ and $\mathbf{c}_i$, and writes vectors $\mathbf{r}_i, \mathbf{w}_i, \mathbf{v}_i, \mathbf{v}, \mathbf{t1}_i$ and $\mathbf{t2}_i$, which makes for a total of $x \times 9$ memory transactions, with x the vector size. Furthermore, five floating point operations can be identified for kernel *CG1*. The other numbers in Algorithm 2 are derived similarly.

These results show trends similar to those in Figure 3.5. Note that these kernels actually perform a number of computations, so that slightly more time is consumed, which affects the maximum throughput. Furthermore, one must be aware that Figure 3.6a shows the performance versus the number of *transferred elements*. Figure 3.6b shows the performance as a function of the *vector size* x, in which large differences are visible between each kernel. A kernel processing multiple vectors of size x, initiates more memory transactions. Such kernels will reach the maximum performance for smaller vector sizes, while kernels processing only one vector, reach the maximum performance for larger vectors. Therefore we have decided to combine as much as possible vector operations, such that the performance of these kernels is increased.

The performance of the vector reduction kernel, Figure 3.6b, is significantly lower compared to other vector operations. Since the reduction kernel performs only $x \times 1$ memory transactions, with x the vector size, this kernel only performs well for large vectors. Furthermore, parameter μ has a slightly higher value, because a complete vector reduction launches at least two kernels.



(a) Throughput versus memory transactions



(b) Throughput versus vector size

Figure 3.6: Measured throughput per combined CG vector operation as a function of the number of memory transactions (a) and as a function of the vector size (b). Lines represent performance estimates using the parameters in Table 3.2; the dots are the actual measurements. The graph contains results from both single and dual GPU setups. In the dual-GPU case, individual results of both GPUs are plotted.

For the dual-GPU cases, also some time is spent on synchronization among the devices explains why the estimations do not agree with the measurements, see *Red. 2 GPU* in Figure 3.6a. Given these observations one can conclude that vector reductions can be problematic for the (parallel) CG method, even though an efficient algorithm [81] was used to implement them.

3.5.3 Analysis of the SpMV Operation

Estimating the performance of the SpMV operation is more difficult because it is influenced by additional aspects of the matrix, like the average density of the matrix blocks. By estimating *raw performances*, the block density is neglected, i.e., we consider that each matrix block contains 100% non-zeros, which artificially increases the number of non-zeros. By doing so, we get more insight in the behavior of the SpMV operation on the used hardware and the different kernel mappings, since the measurements are not affected by the average block density. However, variations between row lengths still influence the distribution of the computations, and thus the efficiency of SpMV. Figure 3.7 shows the *raw memory throughput* versus the number

Block-size	transactions m	FLOPS	μ_{max}	σ_{max}	ν_{max}	μ_{avg}	σ_{avg}	ν_{avg}
1×1	$2\times e$	$2\times e$	19	1.2	85	19.08	1.35	52.12
2×2	$2\times e$	$2\times e$	19.6	1.4	135	19.65	1.15	80.92
4×4	$2\times e$	$2\times e$	19.9	1.4	140	20.39	1.28	117.70
8×8	$2\times e$	$2\times e$	20.11	1.5	145	20.81	1.51	132.39

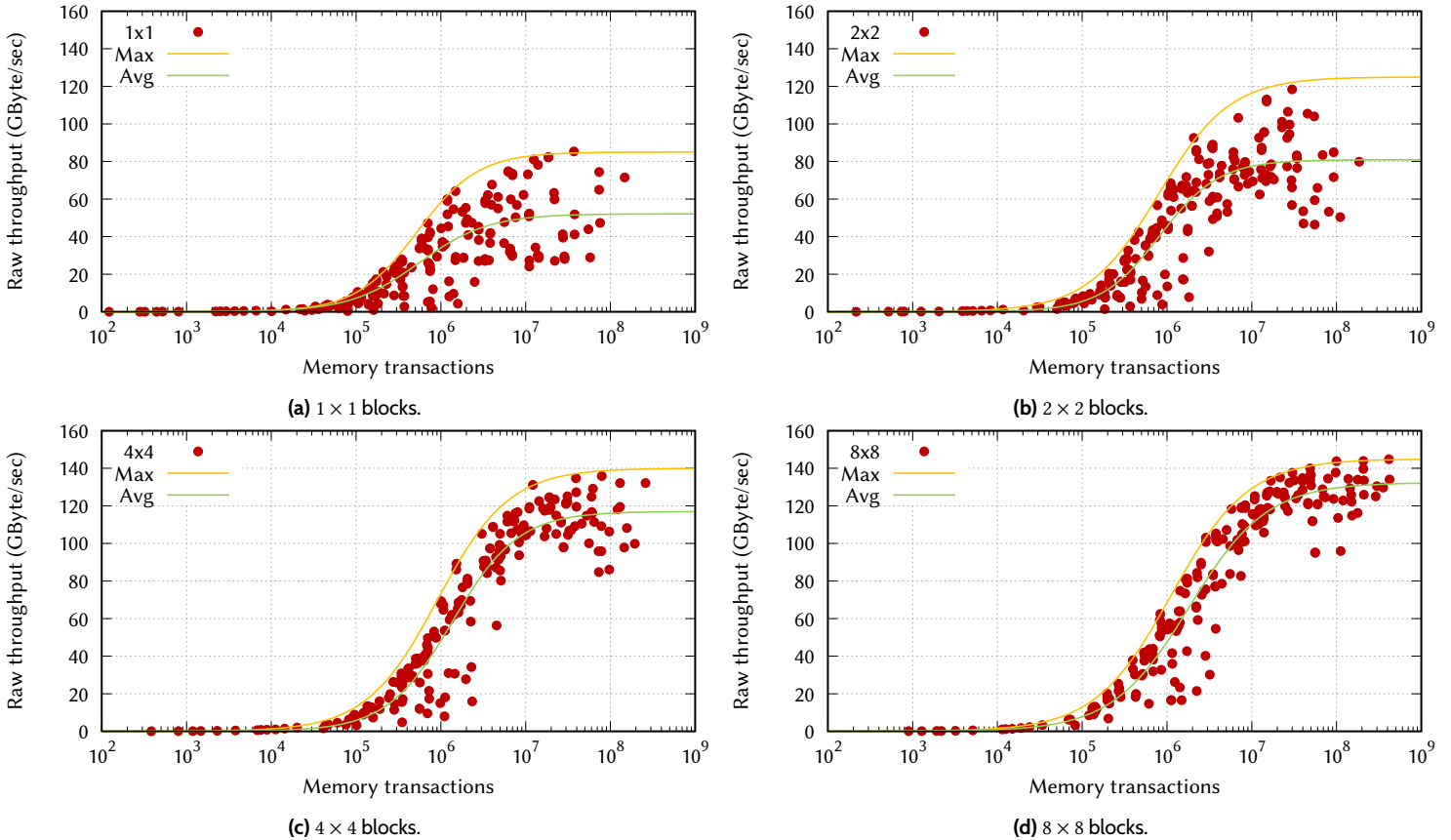
Table 3.3: The number of transactions and FLOPS for our SpMV operation using $N \times N$ blocks, where e is the total number of stored elements. Parameters μ, ν and σ are used to approximate the maximum raw memory throughput (performance) of that specific SpMV operation.

of memory transactions (per block size), which gives an estimate for the upper limit or maximum throughput for a particular block size and the average throughput. Note that Figure 3.7 contains measurements from both a single and dual GPU setup for the set of matrices in [48]. For the dual-GPU results, each measurement of each individual GPU is presented. The *upper limit* and average throughput are estimated using Equation (3.3) and the parameters are given in Table 3.3. Note that this behavior agrees with the observations in Figures 3.5 and 3.6. The 'average' throughput parameters were obtained by fitting the non-linear sigmoidal curve in the measured data using the Levenberg-Marquardt algorithm. The 'maximum' throughput parameters were obtained by adapting the average parameters such that the envelope of the measurements fits the sigmoidal function.

The difference between the measured throughput and its estimated upper limit is mainly caused by the variation of the row lengths. Further, if most rows contain less than 16 blocks, the expected throughput and performance is usually lower than the upper limit. The 4×4 and 8×8 blocks usually deliver the best raw performances, whereas 2×2 and 1×1 blocks also have less efficient vector lookups. Thus, in general, the larger the blocks, the less the extra overhead, yet larger blocks *can* result in a lower block density, which degrades the overall performance. Furthermore, as noted in Section 3.3.7, if $N \geq 4$, the lookup of the vector elements are coalesced by default. This is also clearly visible in the results presented in Figure 3.7. For $N = 1$ and $N = 2$, this effect is visible as the variation of the measurements compared with the average throughput, i.e., the difference between the maximum and the average throughput. The measurements for $N = 4$ and $N = 8$ show that the maximum and average throughput are much more closer to each other compared to the smaller block sizes, as expected.

3.5.4 Scalability

The scalability of the CG method on a single GPU depends on the scalability of each individual operation. Vector operations scale well if the maximum memory throughput is reached. If the total number of memory transactions is larger than 5 million, these operations scale well, see Figure 3.6a. By combining multiple vector operations in a single kernel, the total number of memory transactions is increased for that kernel, which eventually increases the performance, see Figure 3.6b. Unfortunately, vector reductions are problematic, because the amount of memory transactions is low and the operation itself cannot be combined with other vector operations. This results in a low memory throughput and a poor scalability. Furthermore, a vector reduction using multiple GPUs requires some additional synchronization among the devices, which further degrades the performance and scalability of the method.



(a) 1×1 blocks.

(b) 2×2 blocks.

(c) 4×4 blocks.

(d) 8×8 blocks.

Figure 3.7: Measured raw throughput versus total number of memory transactions for the SpMV operation. The graph contains results from both single and dual GPU setups. In the dual-GPU case, individual results of both GPUs are plotted.

The SpMV operation scales also well, if the maximum throughput is reached, i.e., when about 5 million memory transactions are initiated. For each element in the sparse matrix, roughly two values are looked up. One matrix block value, and one corresponding vector value. This implies that sparse matrices with more than 2.5 million elements *can* achieve a good scalability.

When the CG method is executed on multiple GPUs, vectors and matrices are divided in parts (segments), such that each individual GPU processes a part of the vector or matrix, see Section 3.4. This division also means that the total number of memory transactions per GPU is distributed among all available GPUs. Recalling Figure 3.5, a performance drop can be expected if the throughput does not reach the maximum. If the number of memory transactions is larger than 5 million, this performance drop is relatively small. Therefore, the individual operations scale well if the problem is large enough, although the communication between the devices does affect the scalability of the CG method significantly. Furthermore, when the CG method is executed using multiple GPUs, extra synchronizations are needed: one for updating vectors on each GPU, and one synchronization within each vector reduction. However, the time spent on each synchronization is constant.

3.5.5 Inter-Device Communication

When the CG method, or other similar methods, is executed on multiple GPUs, the GPUs need to communicate with each other in order to update their current result vector. Since on our test system, GPUs cannot directly exchange data with each other, each GPU transfers data via the PCI Express bus and the system memory. Because the bandwidth between the GPU and the system memory is limited, and in general, an order of magnitude lower than the bandwidth of the memory bus on the graphics card, this communication negatively and significantly affects the total performance, see Figure 3.8. The effective throughput between the GPUs is also estimated using Equation (3.3). On our test system we found the following parameters, $\mu = 16.7$, $\sigma = 1.4$ and $\nu = 10$ GByte/s, indicating that the maximum throughput is reached when more then one million elements are transferred. Given this approximation, the total time for this operation can be estimated for different data sizes.

First generation GPUs were not able to directly communicate with each other. However, by involving the host memory, GPUs could indirectly exchange data, see above. Current generation GPUs (Fermi) and motherboard chipsets support direct communication between GPUs via the PCIe bus. With the release of CUDA 4.0, a large single address space can be created using *Unified Virtual Addressing* (UVA), which encompasses the memory of each individual GPU and the host memory. Therefore, by interchanging GPU pointers, GPU data can directly be exchanged. This simplifies the communication mechanism and improves the memory throughput, thus increasing the performance of the parallel CG method, see [130–132, 136].

3.5.6 Performance of the CG Method

In order to evaluate the performance of the CG method, as shown in Algorithm 2, and given the properties of a specific problem, the maximum and average performances are estimated. Since all operations appearing in the CG method are bandwidth limited, the memory throughput is used to estimate the total running time. Given the number of transferred elements and the corresponding estimated memory throughput, the time per iteration $T(x, e)$ is obtained as follows,

$$T(x, e) = T_{spmv}(e) + T_{CG2}(x) + T_{CG3}(x) + T_{CG4}(x) + T_{CG5}(x) + 3T_{red}(x), \tag{3.4}$$

where T_{spmv} denotes the time spent for the SpMV operation, T_{CGi} on vector operations of kernel *CGi*, and T_{red} is the vector-reduction timing, e the total number of elements stored in the matrix (including the stored zeros) and x the dimension of the matrix and vectors. Note that kernel *CG1* is executed exactly once and does not contribute to the time per iteration. Each individual timing in Equation (3.4) is given by

$$T(x) = \frac{4m}{B(m, \mu, \sigma, \nu)}, \tag{3.5}$$

with m the amount of transferred elements (which is a function of x or e) and μ, ν and σ the fitting parameters; each timing can be estimated using the parameters and functions in Tables 3.2 and 3.3. Since Algorithm 2 executes approximately 3 vector reductions per iteration, the corresponding timing T_{red} is multiplied by a factor of 3.

The total number of floating point operations per iteration can be computed by

$$F(x, e) = 2e + 15x, \tag{3.6}$$

with x the dimension of the problem and e the total number of elements stored in the matrix, *including* the zeros stored in non-empty blocks. The *raw performance* is then given by

$$P(x, e) = \frac{F(x, e)}{T(x, e)}, \tag{3.7}$$

were T end F are given by Equations (3.5) and (3.6).

Figure 3.8a shows the maximum raw performance of the CG method, executed on one GTX280 GPU. In general, the more elements a matrix has, the better the performance becomes. This can be verified by keeping the vector dimension fixed, while increasing the number of elements. Increasing the dimension of the matrix can have different effects on the total performance. For matrices having approximately 10^5 elements, the performance will increase while increasing the dimension of the matrix and keeping the amount of elements fixed. In this case the total computation time is dominated by the vector operations, which will perform better for larger dimension. Contrary, for matrices having more that 10^7 elements, the performance will decrease while increasing the dimension of the matrix. In this case the total computation time is dominated by the SpMV operation. This operation becomes less efficient when the matrix becomes sparser, hence a drop in the total performance in observed. Please note that increasing the dimension results in more efficient vector operations, while the performance of the SpMV operation decreases since the matrix becomes sparser. If the SpMV operation dominates the total computation time, the total performance will decrease. If the vector operations dominate the total computation time, the performance will increase.

In order to estimate the *real performance*, the raw performance of the SpMV operation is multiplied by the average density of the matrix blocks, i.e.,

$$F(x, e, d_N) = 2ed_N + 15x, \tag{3.8}$$

with d_N the average matrix-block density for a $N \times N$ block. Furthermore, if the matrix contains rows with uneven lengths, a large number of computations are not contributing, due to the added empty blocks (Section 3.3.5). Note that it is difficult to quantify this lost performance, because it requires a lot more information about the input matrix and the used hardware.

3.5.7 Performance of the Parallel CG Method

The parallel performance of the CG method is estimated similarly, but two GPUs run in parallel with approximately 50% of the data. In this case, the communication between the devices and the synchronization time also affects the performance of the parallel CG method. In general, the total time per iteration using n GPUs becomes

$$T_p(x, e) = \max\left(T_1\left(\frac{x}{n}, \frac{e}{n}\right), T_2\left(\frac{x}{n}, \frac{e}{n}\right), \ldots, T_n\left(\frac{x}{n}, \frac{e}{n}\right)\right) + T_{idc}(x\,n) + 4T_{sync}, \tag{3.9}$$

where T_i is the computation time of GPU i, for the sub-matrix (segment) stored on that GPU, T_{idc} is the time spent on communication (Section 3.5.5) and T_{sync} is the average time spent for synchronization ($35\mu s$ on our test system). Each GPU has to copy an x/n-sized vector to the main memory, then $(n-1)$, x/n-sized vectors are copied to n GPUs, which yields a total data transfer of $x\,n$ elements. If GPUs are able to communicate directly using the *Unified Virtual Addressing*, each GPU transfers x/n elements, hence $T_{idc}(x\,n)$ becomes $T_{idc}(x)$.

Figure 3.8b shows the maximum raw performance of the CG method, executed on two GTX280 GPUs. Furthermore, Figure 3.8c illustrates the speedup $S = T/T_p$ of the CG method accordingly, given the parameters of the problem. The thick black line represents the region where the speedup $S \approx 1$. For problems falling below that line, a slowdown can be expected, while for cases above that line, a speedup should normally be obtained. For different systems and GPUs the exact location of this line may vary. This 'map' quickly shows if it is worthwhile to use multiple GPUs for a particular problem.

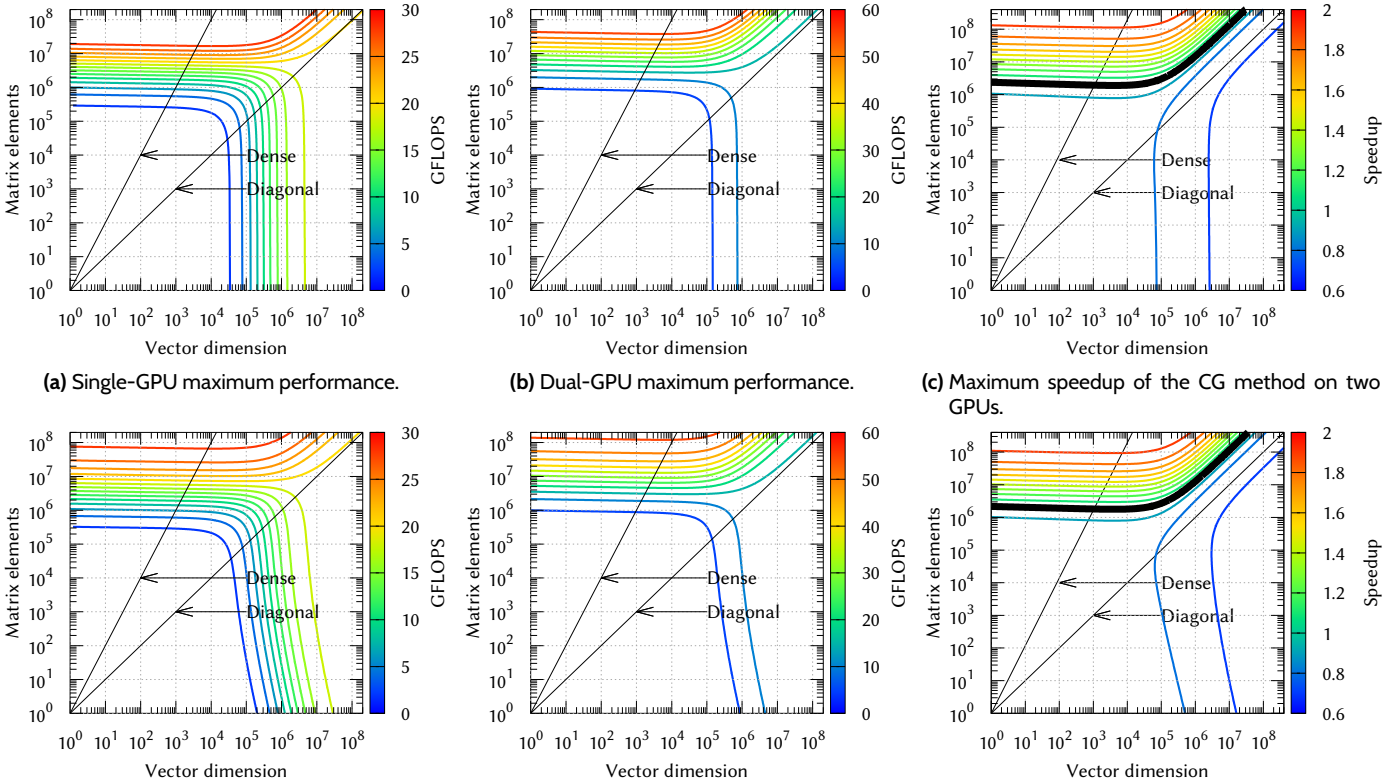
Figures 3.8d to 3.8f show the average raw performance using one and two GPUs and the speedup when two GPUs are used. For these plots the average parameters in Table 3.3 were used. Section 3.6.2 compares the timing results for a large set of test matrices with the average estimated time derived by applying the method described in this section.

The density of the matrix blocks d_N, does not have a significant influence on the speedup, i.e., on the blocks stored on each GPU, the average densities are similar, such that the performances on each GPU are similar. Hence, the speedup will not be affected significantly.

3.6 Results

In this section the results of our SpMV implementation and (parallel) CG method are presented. Both our SpMV and CG methods were benchmarked using different collections of test matrices/problems. The machine used for benchmarking was equipped with an Intel Q6600 quad-core CPU and two NVidia GTX280 GPUs managed by an NVidia nForce 790i SLI chipset; some of our benchmarks were also conducted using an NVidia GTX570 GPU.

First, in Section 3.6.1 the results of our SpMV implementation are compared to those in [18, 34, 135]. Next, the actual and estimated performances of our CG method using one and two GPUs are used to verify the model presented in Section 3.5, see Sections 3.6.2 and 3.6.3. Finally, in Section 3.6.4 we compare the performance of our CG implementation to a similar implementation using the CUSP library [19], on different GPUs and with different precision settings.



(a) Single-GPU maximum performance.

(b) Dual-GPU maximum performance.

(c) Maximum speedup of the CG method on two GPUs.

(d) Single-GPU average performance.

(e) Dual-GPU average performance.

(f) Average speedup of the CG method on two GPUs.

Figure 3.8: Estimated maximum (a)-(b) and average (d)-(e) raw-performances and speedup ((c) and (f)) for the CG method using 4×4 matrix blocks. Black lines depict dense and diagonal matrices, whereas the region between them represents sparse matrices. The black curve in (c) and (f) represents the area in which the speedup is approximately one.

Name	Dim ($\times 10^3$)	nz ($\times 10^6$)	max	avg	d_2	d_4	d_8
Dense	2×2	4	2000	2000	1.00	1.00	1.00
Protein	36×36	4.3	204	119	0.86	0.66	0.48
FEM/Spheres	83×83	6	81	72	0.77	0.55	0.34
FEM/Cantilever	62×62	4	78	64	0.75	0.54	0.35
Wind tunnel	217×217	11	180	53	0.82	0.70	0.51
FEM/Harbour	46×46	2.3	145	50	0.77	0.55	0.37
QCD	49×49	1.9	39	39	0.64	0.47	0.28
FEM/Ship	140×140	7.8	102	55	1.00	0.63	0.34
Economics	206×206	1.2	44	6	0.29	0.10	0.04
Epidemology	525×525	2.1	4	4	0.41	0.20	0.05
FEM/Accelerator	121×121	2.6	81	21	0.64	0.23	0.08
Circuit	170×170	0.95	353	6	0.40	0.15	0.06
Webbase	1000×1000	3.1	4700	3	0.47	0.21	0.09
Rail	1092×4	11279	56181	3	0.48	0.20	0.03

Table 3.4: Properties of the test matrices from [192]. Dim is the dimension of the matrix, nz the total number of non-zeros, max the largest number of non-zeros in a row and avg is the average number of non-zeros per row; d_N is the average block density when the matrix is represented using $N \times N$ blocks. $d_1 = 1$ for all matrices.

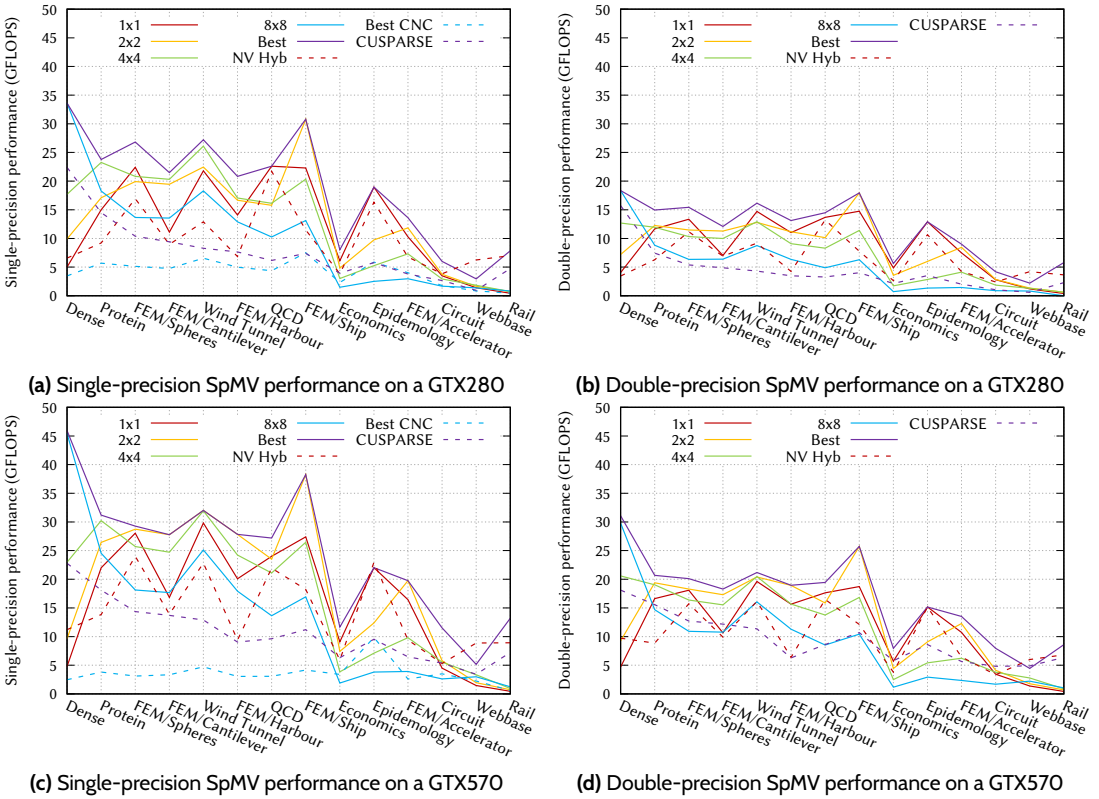
3.6.1 SpMV: Performance Comparison

Figure 3.9 shows the results of our SpMV approach as described in Section 3.3.7 for varying block sizes. 'Best' denotes our best result after increasing parameter B_x as described in Section 3.3.6. 'NV Hyb' denotes the hybrid method of Bell and Garland [18] (implemented in CUSP 0.2, [19]), 'Best CNC' shows the best results obtained by Buatois et al. [34] and finally, 'CUSPARSE' denotes the results of the CUSPARSE library [135] from CUDA 4.0 in which the CSR storage was used. The test set used in this benchmark was introduced in [192] to evaluate the performance of the SpMV operations on various multi-core platforms; some of the properties of the test matrices are given in Table 3.4. Finally, we have performed this benchmark on both a GTX280 and a GTX570 GPUs, in single and double precision.

The results in Figure 3.9 show that our method gives the best performance in most cases, except for matrices that have very few non-zero elements per row. Since such matrices have in general a low average block density (d_N in Table 3.4) for larger blocks, they generally cannot benefit from the BCSR storage format. For example, in the worst case scenario, the usage of 2×2 blocks means that 75% of the computations and memory throughput are useless since blocks contain only one non-zero element. In fact, matrix 'Economics' represents just such an example: it has on average a block density of 29% for 2×2 blocks.

Matrices 'Circuit', 'Webbase' and 'Rail' also exhibit poor performance figures. Table 3.4 shows that the number of non-zeros per row is highly unbalanced. Matrix 'Webbase' has on average three elements per row, yet a few rows contain several thousands elements. In such cases, a lot of threads become idle, while others are busy with processing very large block rows. Combining this with the low number of non-zeros per row, makes that these matrices are difficult to process on GPUs using BCSR. By changing the layout of the mapping as discussed in Section 3.3.6, the performance is improved, see 'Best' performance.

Matrix 'FEM/Ship' shows a large performance boost for 2×2 blocks compared to 1×1 blocks. Table 3.4 shows that $d_2 = 1$, which means that each block contains four non-zero elements. Hence, no computing resources are wasted. This clearly shows the benefit of the BCSR layout compared to the others, if blocks have high densities d_N.



(a) Single-precision SpMV performance on a GTX280

(b) Double-precision SpMV performance on a GTX280

(c) Single-precision SpMV performance on a GTX570

(d) Double-precision SpMV performance on a GTX570

Figure 3.9: Single and double precision performances for the SpMV operation using various storage formats performed on a GTX280 and a GTX570 GPUs. Numbers in the legend indicate block sizes for our method, 'Best' indicates our best results after increasing parameter B_x as in Section 3.3.6, 'NV Hyb' represents the hybrid format of [18, 19], 'Best CNC' – the best result obtained by the method in [34] and 'CUSPARSE' – [132] the CSR implementation from the CUSPARSE library in CUDA 4.0.

Since the set of test matrices in Williams et al. [192] is small, we have also benchmarked the implementations of Buatois et al. [34], Bell and Garland [18] and ours on a substantially larger set of matrices. This large set contains all matrices from Harwell-Boeing, SPARSKIT, the sample collection of The University of Florida [48], Williams et al. [192] and the matrices described in Table 3.5, making for a total of 486 matrices. We have benchmarked each implementation on a GTX570 GPU. We found that in 95% of the cases our method performs better than the hybrid implementation of Bell and Garland. Further, we measured a median speedup of about 8× and a total speedup, with respect to the total computation time (wall-clock time), of about 1.25×. Because of the large difference between the median and total speedups, after careful inspection of the results, we found that our method performed in three cases substantially worse than the method of Bell and Garland. It turned out that these matrices have highly unbalanced row lengths, which made our method to perform poorly, see discussion in Section 3.3. After neglecting these outliers, the speedup becomes 2.5× in favor of our method. The method of Buatois et al. [34] performed in 33% of the cases better than ours and significantly better than the hybrid method of Bell and Garland [18]. This happened for most of the matrices contained in the Harwell-Boeing set. However, in all these cases it turns out that a parallel CPU implementation of Algorithm 2 (using four threads on our quad-core CPU) was still faster than *any* of the GPU methods. Finally, comparing the total computation time for the complete set showed that our method was about 3.7× faster than the best implementation of Buatois et al. Note that by neglecting the three outliers mentioned above, our method performed 6.1× better than their method.

Dense matrix-vector multiplication

The very good results obtained by our method for matrix 'Dense' inspired us to use our representations for sparse matrices to perform matrix-vector multiplications for *dense* matrices. We gradually increased the dimensions of a square and dense matrix from 10 to about 8000, and measured the time taken to perform the multiplication by our methods and function `cublasSgemv` from NVidia's CUBLAS library [133]. Our multiple-block-row mapping using 2×2 blocks yields exactly the same performance as the CUBLAS function, whereas using 1×1 blocks performed poorly compared to CUBLAS. All other combinations, except single block-row with 1×1 blocks, perform better than CUBLAS: the maximum performance is about 15% higher. More importantly, single block-row with 8×8 blocks reaches the peak performance at problem sizes 10 times smaller than when using the CUBLAS function. We chose to use the single block-row mapping since the multiple block-row mapping is not efficient if the dimension is too small.

3.6.2 Performance of our Conjugate Gradient Method

Figure 3.10 shows the performances of our CG method using the SpMV approach from Section 3.3 (performed using different block sizes), and the corresponding performance estimations, as described in Section 3.5. Since the estimations are made for a single GTX280 GPU, we use the same GPU for comparing the results with the estimation. The results in Figure 3.10 are ordered with respect to the dimensions of the matrices. The test set (see Table 3.5 for some properties) represents a subset of the entire University of Florida sparse matrix collection [48], in which all matrices are Symmetric Positive Definite.

The best performances were obtained for matrices 'Bone010', 'af_shell3', 'nd24k', 'nd12k' and 'nd6k' for each block size. As shown in Table 3.5, these matrices have several millions of elements and have relatively high block densities. Also the maximum and average numbers of

Id	Name	Dim	nz	max	avg	d_2	d_4	d_8	Id	Name	Dim	nz	max	avg	d_2	d_4	d_8
1	Trefethen_20	20	158	9	4.93	0.65	0.47	0.27	35	bodyy6	19366	134748	9	6.95	0.49	0.24	0.12
2	mesh1e1	48	306	8	6.37	0.36	0.20	0.17	36	Trefethen_20000b	19999	554435	29	27.72	0.53	0.29	0.15
3	Trefethen_150	150	2040	15	12.75	0.58	0.34	0.21	37	Trefethen_20000	20000	554466	29	27.72	0.53	0.29	0.15
4	Trefethen_200	200	2890	16	13.89	0.57	0.34	0.21	38	smt	25710	3753184	414	145.97	0.74	0.45	0.27
5	bcsstk34	588	21418	47	36.17	0.76	0.52	0.34	39	nd12k	36000	14220946	519	395.02	0.84	0.73	0.59
6	msc00726	726	34518	88	46.89	0.55	0.38	0.26	40	jnlbrng1	40000	199200	5	4.98	0.50	0.25	0.12
7	msc01050	1050	29156	128	27.60	0.60	0.41	0.23	41	bcsstm39	46772	46772	1	0.99	0.50	0.25	0.12
8	plbuckle	1282	30644	44	23.64	0.54	0.36	0.27	42	gridgena	48962	512084	17	10.45	0.75	0.37	0.18
9	msc01440	1440	46270	45	32.13	0.62	0.42	0.24	43	cvxbqp1	50000	349968	9	6.99	0.26	0.10	0.04
10	nasa1824	1824	39208	42	21.49	0.52	0.35	0.23	44	ct20stif	52329	2698463	207	51.56	0.82	0.57	0.36
11	Trefethen_2000	2000	41906	22	20.95	0.55	0.30	0.17	45	nasasrb	54870	2677324	276	48.78	0.90	0.58	0.39
12	nasa2146	2146	72250	36	33.44	0.76	0.60	0.44	46	Dubcova2	65025	1030225	25	15.83	0.37	0.15	0.06
13	Chem97ZtZ	2541	7361	101	2.89	0.49	0.24	0.12	47	qa8fm	66127	1660579	27	25.11	0.51	0.29	0.16
14	nasa2910	2910	174296	175	59.85	0.70	0.53	0.37	48	cfd1	70656	1828364	33	25.87	0.46	0.24	0.13
15	sts4098	4098	72356	784	17.59	0.43	0.22	0.13	49	nd24k	72000	28715634	520	398.82	0.84	0.73	0.58
16	nasa4704	4704	104756	42	22.26	0.53	0.33	0.22	50	finan512	74752	596992	55	7.98	0.40	0.14	0.06
17	crystm01	4875	105339	27	21.58	0.32	0.22	0.14	51	apache1	80800	542184	7	6.71	0.39	0.19	0.09
18	Kuu	7102	340200	98	47.88	0.99	0.71	0.45	52	thermal1	82654	574458	11	6.94	0.35	0.13	0.05
19	Muu	7102	170134	49	23.94	0.50	0.35	0.22	53	2cubes_sphere	101492	1647264	31	16.22	0.28	0.09	0.03
20	bcsstk38	8032	355460	614	44.25	0.75	0.53	0.34	54	cfd2	123440	3087898	30	25.01	0.55	0.30	0.17
21	aft01	8205	125567	21	15.29	0.63	0.35	0.18	55	Dubcova3	146689	3636649	49	24.78	0.42	0.22	0.11
22	nd3k	9000	3279690	515	364.08	0.85	0.74	0.60	56	bmwcra_1	148770	10644002	351	71.53	0.75	0.49	0.28
23	fv1	9604	85264	9	8.86	0.50	0.32	0.16	57	G2_circuit	150102	726674	6	4.84	0.45	0.16	0.06
24	ted_B	10605	144579	49	13.62	0.55	0.40	0.30	58	F1	343791	26837113	435	78.06	0.68	0.38	0.18
25	ted_B_unscaled	10605	144579	49	13.62	0.55	0.40	0.30	59	inline_1	503712	36816342	843	73.09	0.69	0.39	0.21
26	msc10848	10848	1229778	723	113.36	0.80	0.56	0.34	60	af_shell3	504855	17588875	40	34.83	0.84	0.65	0.46
27	cbuckle	13681	676515	600	49.39	0.94	0.77	0.56	61	parabolic_fem	525825	3674625	7	6.98	0.33	0.13	0.05
28	crystm02	13965	322905	27	23.11	0.32	0.22	0.12	62	apache2	715176	4817870	8	6.73	0.39	0.19	0.09
29	Pres_Poisson	14822	715804	50	48.26	0.73	0.47	0.31	63	tmt_sym	726713	5080961	9	6.99	0.50	0.25	0.12
30	Dubcova1	16129	253009	25	15.67	0.37	0.15	0.07	64	bone010	986703	71666325	81	72.63	0.80	0.58	0.37
31	olafu	16146	1015156	89	62.81	0.89	0.69	0.50	65	ecology2	999999	4995991	5	4.99	0.49	0.25	0.12
32	bodyy4	17546	121938	9	6.94	0.49	0.24	0.12	66	thermal2	1228045	8580313	11	6.98	0.34	0.12	0.04
33	nd6k	18000	6897316	514	383.18	0.84	0.73	0.59	67	G3_circuit	1585478	7660826	6	4.83	0.36	0.15	0.06
34	bodyy5	18589	129281	9	6.95	0.49	0.24	0.12									

Table 3.5: Properties of the test matrices used [48] for estimating the performance of our GPU mapping of the CG method. Dim is the dimension of the matrix, nz the total number of non-zeros, max the largest number of non-zeros in a row and avg is the average number of non-zeros per row; d_N is the average block density when the matrix is represented using $N \times N$ blocks. $d_1 = 1$ for all matrices.

Block size	#GPUs	Average rel. error	Variance
1×1	1	0.18	0.108
2×2	1	0.12	0.037
4×4	1	0.07	0.008
8×8	1	0.06	0.004
1×1	2	0.16	0.042
2×2	2	0.14	0.023
4×4	2	0.11	0.012
8×8	2	0.07	0.006

Table 3.6: Average relative-error, and the variance, of the estimated performance with respect to the measured performance.

non-zeros per row are close to each other. This implies that each row has a similar amount of elements, and thus, a more balanced computation. Contrary, matrices 'F1' and 'inline_1', which have also a high number of elements, show a large deviation between the maximum and the average number of non-zeros per row. This implies that computations are unbalanced, which is reflected in the performances of these matrices.

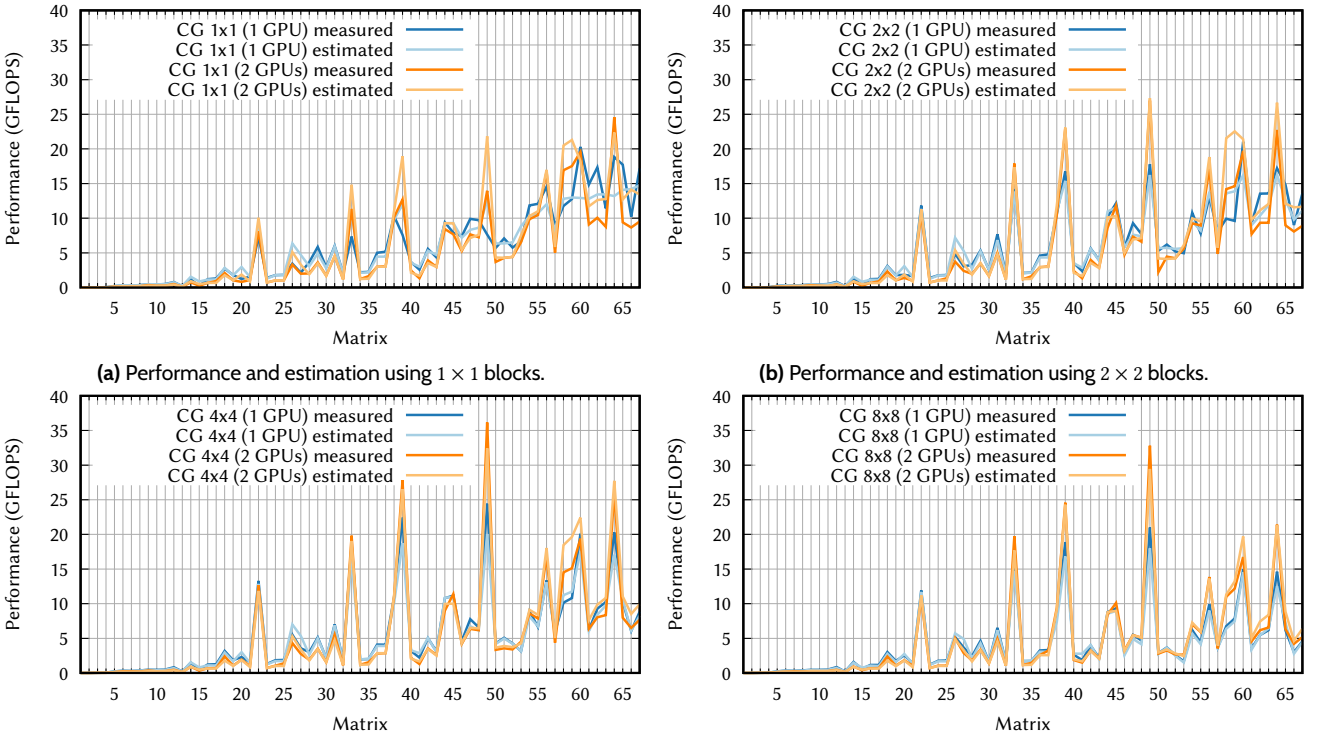
In general, poor performances are obtained if (i) there is a large difference between the maximum and average number of non-zeros per row, (ii) the dimension of the problem is too small or (iii) the number of non-zeros is small. Increasing the block size increases the raw performance in general, but also the average block density *can* decrease, which *can* result in a lower actual performance.

Within Figure 3.10 the dashed lines represent the estimated performance for the corresponding matrices. We have analyzed the average relative error and the corresponding variance of the relative error, as shown in Table 3.6. From this analysis and Figure 3.10 we can conclude that the estimated performance comes closer to the measured performance, when the block size increases. According to Figure 3.7, this behavior is to be expected since the variation between the maximum and average performances for the SpMV operation is larger if $N < 4$. Since the estimated performances are close to the measured ones, our estimation method from Section 3.5 can be used to compute a first indication of the expected maximum and average raw performances, on unseen problems.

3.6.3 Performance of Our Parallel CG Method

Figure 3.10 also shows the performance results and the estimated performances for our CG method accelerated using two GTX280 GPUs. The solid lines show the measured performance, whereas the dashed lines show the estimated performance for the corresponding matrices. Table 3.6 shows the average relative error of the estimation.

The largest speedup and performance is obtained for matrix 'nd24k'. This matrix is relatively dense, i.e., about 400 elements per row on average, with a dimension of 72,000 and a total of 28×10^6 non-zeros. According to Figure 3.8b a maximum raw performance of 42 GFLOPS can be expected. Given the density $d_2 = 84\%$, the maximum expected performance is about 35 GFLOPS, which agrees with the results in Figure 3.10 for two GPUs. The measured speedup is about 1.7, which also agrees with the observations in Figure 3.8c. The dimension of the problem is relatively small, which means that the vector operations do not perform optimal, see Section 3.5. However, most of the computation time is spent on the SpMV operation,



(a) Performance and estimation using 1×1 blocks.

(b) Performance and estimation using 2×2 blocks.

(c) Performance and estimation using 4×4 blocks.

(d) Performance and estimation using 8×8 blocks.

Figure 3.10: Performance and estimations for our CG methods using one and two GPUs. The x-axis represents the indices of the used matrices, as found in Table 3.5. Table 3.6 shows the average relative-error and variance of the estimation compared with the actual results for each configuration.

Method	GTX280					
	Single-precision			Double-precision		
	Time (s)	Speedup	Iterations	Time (s)	Speedup	Iterations
CUSP	3108	1.00 *(1.00)*	333 *(9070)*	2500	1.00 *(1.00)*	264 *(4368)*
Ours	1285	5.73 *(2.41)*	320 *(8734)*	1388	5.20 *(1.80)*	263 *(4329)*
Method	GTX570					
	Single-precision			Double-precision		
	Time (s)	Speedup	Iterations	Time (s)	Speedup	Iterations
CUSP	2031	1.00 *(1.00)*	321 *(8526)*	1675	1.00 *(1.00)*	263 *(4322)*
Ours	895	5.95 *(2.26)*	322 *(8352)*	1058	5.69 *(1.58)*	263 *(4264)*

Table 3.7: Performance comparison for the CG method using our implementation and CUSP [19]. *Time* represents the total time needed to solve the complete set of linear systems, *Speedup* represents the median and average speedup relative to the CUSP-based implementation, *Iterations* denotes the median and average number of iterations used to solve the complete set of linear systems.

which is much larger than the time spent on the vector operations. This results in a relatively large speedup of about 1.7. Note that according to our analysis, reaching a speedup of two is practically impossible, see Section 3.5.

Figure 3.10 shows that in most test-cases the parallel performance is similar to the performance of the single GPU case. For these problems it is not worthwhile to use extra computational resources. However, in a small number of cases, the performance is significantly increased when two GPUs are used. Our model also reports similar performances and speedups in those cases. This means that our approach can be used to determine the number of GPUs that would solve a problem efficiently, given some properties of the corresponding matrix.

3.6.4 Performance Comparison for the CG Method

In this section we compare our GPU mapping of the CG method with a similar one using the CUSP library [19]. For a fair comparison, we have re-implemented Algorithm 2 (with the same preconditioner and an absolute tolerance of 10^{-8}) using CUSP. We performed a large number of benchmarks on both a GTX280 and a GTX570 GPUs using single- and double-precision arithmetic. For each combination we measured the total time to solve a particular linear system. The set of matrices used for benchmarking is a subset of the University of Florida Sparse matrix collection [48], obtained as follows. We selected all matrices that were Symmetric Positive Definite and could fit in the system (and GPU) memory. Furthermore, we selected all matrices that converged using single-precision arithmetic. Finally, we obtained a set of 185 matrices originating from various problems. Table 3.7 shows a summary of our findings.

As can be seen from Table 3.7, our CG implementation performs significantly better than the CUSP-based one. For each possible combination of precision and hardware, our implementation performed in about 98 % of the cases better than the one using CUSP. Note that a comparable percentage was also found in Section 3.6.1, in which we compared our SpMV operation using a larger set of matrices.

Since this test set contains matrices coming from various problems and with very different sizes, the time required to solve a linear system varies a lot. Since a few large matrices dominate the total computation time (wall-clock time), both the median and average estimates are given.

Switching from single to double precision yields similar speedups, but the total computation time changes slightly. For our implementation, the total time increases on both GPUs, while the total time for CUSP decreases. Recalling the results presented in Figure 3.9, one can see that the performance difference between our method and CUSP/Hyb is smaller for double precision than with single precision. However, our double-precision version is not at all optimized. Furthermore, the median and average number of iterations decrease when using double-precision arithmetic. In a large number of cases the amount of iterations do not change, resulting in a higher computation time. Only for the cases in which the number of iterations is reduced significantly, a better computation time is obtained. In section Section 3.7.4 we further discuss the differences between single- and double-precision performances.

3.7 Discussion

The analysis performed in Section 3.5 enables us to estimate the *maximum* and *average* performance of the CG method, accelerated using modern GPUs. One can conclude that these estimates are close to the measured performances (Section 3.6), provided that some conditions are met. First, the number of blocks per block row must not vary greatly. If the variation in row lengths is large, threads may become idle, so that the overall performance drops. Second, if the number of blocks per block row is very low, a larger error between the estimation and the real performance can be expected. If these conditions are met, the average or maximum (raw) performance can properly be estimated by considering the memory throughput. Furthermore, the larger the blocks, the smaller the variance (Figure 3.7, Table 3.6), and the better the estimation becomes.

When using two GPUs, a speedup can be expected for matrices with more than 2.5 million elements, see Figure 3.8. As reported by others [72] and also found by us, a good scalability of the CG method using GPUs can be achieved when the problem size is large enough to fully occupy the GPU. Moreover, the bandwidth between the devices plays an important role, see below.

3.7.1 Scalability for Future Devices

The current trend with GPU development is to increase the number of streaming processors, raster output units and the amount of memory per GPU. Clearly, this increase will lead to a higher total performance. However, in order to reach the peak performance, the problem size should grow accordingly. For example, if the number of streaming processors is doubled, the maximum performance is reached for problems that are twice as large. If multiple GPUs are used, the bandwidth between the GPUs becomes critical. If the available bandwidth *on* GPUs becomes larger, the bandwidth *between* the devices will represent even a larger bottleneck. This makes it even more difficult, especially for bandwidth-limited problems, to achieve a good scalability using multiple GPUs. Note that such changes can be accommodated by our analysis framework, by adjusting the value of parameter v for the inter-device communication time.

Current-generation GPUs support *Unified Virtual Addressing* [131], such that data stored on the GPUs can be accessed by other GPUs via the PCIe bus. This clearly improves the total memory throughput when, e.g., broadcasting the result vector to all GPUs. Furthermore, the communication time in Equation (3.9) will be reduced from xn to x, regardless the number n of GPUs connected over the PCIe bus.

3.7.2 Matrix Reordering

The differences between the raw and actual performance are caused by the density of the matrix blocks. If matrix blocks are completely filled with non-zero elements, no computations are wasted. Therefore it is important to maximize the average density of the matrix blocks. *Reordering* matrices using the (Reversed) Cuthill-McKee [46], Approximate Minimum Degree (AMD) [3] and King [104] matrix reordering schemes, does not usually improve the density of the matrix blocks significantly (results not shown). However, we expect that specific reordering methods, tailored for the BCSR layout, will lead to better performance figures.

3.7.3 Best Block Size

To answer the question about which block-size performs the best, we carefully studied the results presented in Section 3.6. First, we have found that for a large amount of matrices in the Harwell-Boeing set, the dimensions and the number of non-zeros are too small to fully utilize a GPU. The results showed that in 58% of the cases, blocks with $N = 8$ give the best results, even if the average density of the blocks is very low. For the multiple block-row mapping, increasing N automatically increases the amount of thread blocks, see Section 3.3. This in turn results in a higher utilization of the GPU. For the remaining cases we have found that 23% of the cases reported the best performance for $N = 1$, and in 11% of the cases the best performance was obtained for $N = 4$.

Inspecting the results obtained using the test set of Williams et al. [192] using a single GTX570 showed that in 50% of the cases, $N = 2$ yields the best performance. Contrary, the same test on a GTX280 gives in 35% of the cases the best performance when $N = 4$, while in 25% of the cases the best performance was reported when $N = 1$ and $N = 2$.

In order to find which configuration yields the best performance, we selected all test cases (from all benchmarks performed on a GTX570) in which the GPU was faster than the CPU. We found that in 42% of the cases $N = 1$ yields the best performance, followed by 25% of the cases when $N = 2$.

Since the dimension of the problem, the GPU mapping and the sparsity pattern of the matrices influence the performance, it is difficult in general to indicate which block size leads to the best results. However, one can compute the amount of needed thread blocks, given the dimensions of the problem and the block size. If this number is too small to fully utilize a GPU, increasing N and/or B_x will result in a mapping which has a higher utilization and therefore a better performance.

3.7.4 Double Precision

In this chapter we have used both single (32-bit wide) and double (64-bit wide) precision representations within our matrix and vector operations. If the multiple block-row mapping is used in combination with block reordering and block-row sorting, all matrix blocks are loaded in a coalesced fashion, also if double precision is used. In this case two memory transactions are needed. Loading the corresponding vector values also results in coalesced memory transfers if $N \geq 4$, similar to the single-precision case, but with twice the amount of transactions. When $N < 4$, memory transactions are not coalesced anymore and require more transactions for loading the corresponding vector values. In the case of double precision, this does not automatically lead to twice the amount of memory transaction, since the second part of the 64-bit value is stored in the same memory segment as the first 32 bits. This gives a small improve-

ment in these cases. Furthermore, depending on the version of the architecture, *bank-conflicts* [132] can occur. For the GTX280 bank-conflicts happen if threads access 64-bit values in shared memory; for the GTX570 (Fermi) such conflicts do not happen.

We have computed the relative slow-down if one uses double precision. The differences between the block-sizes is especially visible on the GTX280. When $N \geq 4$, the slow-down was approximately 2×, while for $N < 4$ it was between 1.25× and 1.75×. A similar slow-down was measured for the hybrid implementation of [18]. The figures for a GTX570 GPU were slightly different. For any N we have measured a slow-down between 1.4× and 1.6×, which was smaller than on the GTX280. According to the architectural differences between the GPUs, we assume that besides the absence of bank-conflicts, also cached accesses improve the performance when 64-bit data is fetched from the global memory.

Finally, using double precision for solving large linear systems can improve the total computation time, although individual operations become roughly two times slower. On both GTX280 and GTX570 we have found that for the CG benchmark described in Section 3.6.4, about 50% of the cases were faster using double precision. Since the accuracy is higher, the CG method converges faster to the solution, especially for stiff problems. In the remaining cases the CG method converged in the same amount of iterations, resulting in a larger computation time. These finding are reflected in Table 3.7.

3.7.5 Textures

To improve the memory throughput of random memory accesses on the GTX280 and older devices, textures are used frequently. The texture units provide a caching mechanism, which improves the throughput if the memory transactions are not coalesced and highly random.

Current generation GPUs [131, 132] now provide a cache mechanism for global memory. This might imply that the used of textures (for caching) is now deprecated. To test this possibility, the benchmark described in Section 3.6.1 was performed with and without texture cache on the newer GTX570 GPU. We found that the use of textures still improves the performance significantly on the newer hardware. When $N = 1$ and $N = 2$, the improvement is up to $25 - 50\%$, while for larger blocks it is minimal and sometimes slightly negative. Also the implementation of Bell and Garland [18] benefits from the use of textures. Therefore, it is still worthwhile to use textures if memory accesses are highly random.

3.8 Conclusions

In this chapter we have investigated a number of mappings for block-based SpMV operations on GPUs, using CUDA. *Block row mapping* maps one complete block row (a row containing a number of $N \times N$ matrix blocks) to one thread block. This method is straightforward to implement, but not very efficient, since a lot of computational resources are wasted. Within this mapping one thread block processes a large number of matrix blocks. By transposing the block row mapping, the *multiple block-row mapping* is obtained. This mapping assigns multiple block rows to one thread block. One thread block processes a large number of matrix blocks, which belong to different block rows. This has positive implications on the performance, i.e., less thread blocks are needed and the amount of wasted computational resources is decreased. Furthermore, since each thread block processes a larger number of matrix blocks, better memory throughput was obtained and thus a better performance. This is in general only the case if $N \geq 4$. If $N < 4$ the data must be reordered to obtain efficient (coalesced) memory transactions

for loading the matrix blocks. This *block reordering* significantly improves the performance of the SpMV operation for matrices stored using the BCSR layout with blocks of size $N < 4$, if the MBR mapping is used. Sorting the block rows such that block rows with similar lengths are processed by the same thread block, increases the performance significantly.

By mapping the computations differently on the GPU, and by applying row sorting and block reordering, the best performances for the SpMV operation were obtained. Experimental results showed that our SpMV mapping outperforms existing methods in most cases, and performs close to the limits of the hardware. Our optimized SpMV operation was used to accelerate the CG method, given one or multiple GPUs. Together with the optimized vector operations, this makes (in most cases) our CG mapping about five times faster than existing methods.

We have also provided a recipe for estimating the maximum achievable performance and the average performance of a (parallel) CG method, given the properties of the problem. This method can be applied to similar numerical algorithms. Analyzing the memory throughput revealed a clear trend between the number of memory transactions and the performance. This analysis has been done for each kernel performing vector operations, as well as for the SpMV kernel. The resulting trends were modeled by a particular sigmoid function, which was then used to estimate the memory throughput of each individual operation appearing in the CG method. Finally, this led to an approximation of the maximum or average performance of the method. We further extended our performance-estimation framework such that also multiple GPU setups can be modeled. The results showed that our performance estimates were very close to the measured performance, and in general, the estimates became more accurate when larger blocks are used.

In future work, we plan to investigate methods for matrix reordering, suitable for the BCSR format. Existing matrix reordering methods optimize, e.g., the bandwidth of the matrix, which does not necessarily result in increased block densities. Further, our analysis may be improved by taking into account the variations in GPU load, due to block-row padding by empty blocks. Finally, our method performs very well on matrices for which the variation in row lengths is not too large.

4

Deformable Models on GPUs

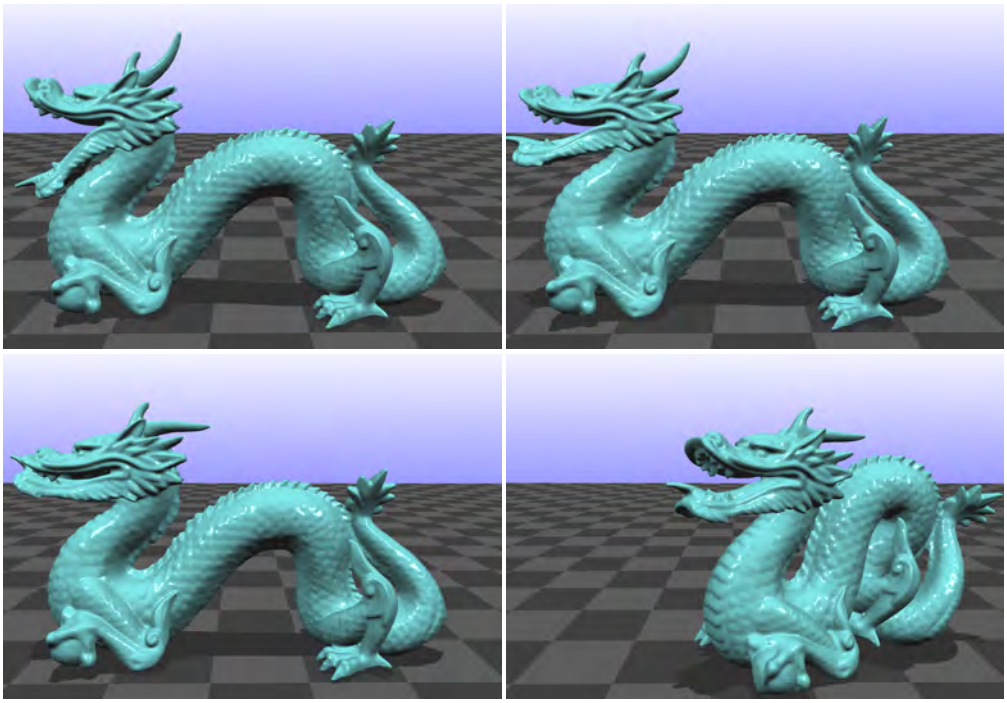


Figure 4.1: Effect of external (stretching) forces on an 'elastic' dragon.

Elastically deformable models have found applications in various areas ranging from mechanical sciences and engineering to computer graphics. The method of Finite Elements has been the tool of choice for solving the underlying PDE, when accuracy and stability of the computations are more important than, e.g., computation time. In this chapter we show that the computations involved can be performed very efficiently on modern programmable GPUs, regarded as massively parallel co-processors through Nvidia's CUDA compute paradigm. The resulting global linear system is solved using a highly optimized Conjugate Gradient method. Since the structure of the global sparse matrix does not change during the simulation, its values are updated at each step using the efficient update method proposed in this chapter. This allows our fully-fledged FEM-based simulator for elastically deformable models to run at interactive rates. Due to the efficient sparse-matrix update and Conjugate Gradient method, we show that its performance is on par with other state-of-the-art methods, based on, e.g., multigrid methods.

4.1 Introduction

Mathematical and physical modeling of elastically deformable models has been investigated for many years, especially within the fields of material and mechanical sciences, and engineering. In recent years, physically-based modeling has also emerged as an important approach to computer animation and graphics modeling. As nowadays graphics applications demand a growing degree of realism, this poses a number of challenges for the underlying real-time modeling and simulation algorithms. Whereas in engineering applications modeling of de-

formable objects should be as accurate as possible compared to their physical counterparts, in graphics applications computational efficiency and stability of the simulation have most often the highest priority.

The Finite Element Method (FEM) constitutes one of the most popular approaches in engineering applications which need to solve Partial Differential Equations (PDEs) at high accuracies on irregular grids [143]. Accordingly, the (elastically) deformable object is viewed as a continuous connected volume, and the laws of continuum mechanics provide the governing PDE, which is solved using FEM. Other popular methods are the Finite Difference Method (FDM) [177], the Finite Volume Method (FVM) [176] and the Boundary Element Method (BEM) [97] (see [70, 126]). FDM is the easiest to implement, but as it needs regular spatial grids, it is difficult to approximate the boundary of an arbitrary object by a regular mesh. FVM [176] relies on a geometrical framework, making it more intuitive than FEM. However, it uses heuristics to define the strain tensor and to calculate the force emerging at each node. BEM performs the computations on the surface of the model, thus achieving substantial speedups as the size of the problem is proportional to the area of the model's boundary as opposed to its volume. However, this approach only works for objects whose interior is made of homogeneous material. Furthermore, topological changes are more difficult to handle than in FEM methods [126].

In this chapter we present a fully-fledged FEM-based simulator for elastically-deformable models, running solely on GPU hardware. We show that the involved computations can be performed efficiently on modern programmable GPUs, regarded as massively parallel coprocessors through Nvidia's CUDA compute paradigm. Our approach relies on the fast GPU Conjugate Gradient (CG) method of [183] to solve the resulting linear system. Since the topology of the deformed mesh does not change during the simulation, the structure of the sparse-matrix describing the linear system is reused throughout the simulation. However, during the simulation, the matrix values have to be updated efficiently. To achieve this, we propose a method that updates the sparse-matrix entries respecting the ordering of the data, as required by the CG method of [183], see Section 4.5.4. Thanks to the optimized CG method and the efficient sparse-matrix update procedure, we obtain results similar to state-of-the-art multigrid methods [49].

The chapter is organized as follows. Sections 4.3 and 4.4 describe the involved discretizations using FEM. Next, Section 4.5 presents the non-trivial parts of our GPU mapping, i.e., computing the local matrices, updating the global sparse matrix and solving the linear system. Finally, in Section 4.6 results are presented and analyzed.

4.2 Previous and Related Work

Bolz *et al.* [26], and Krüger and Westermann [105] were among the first to implement CG solvers on graphics hardware, using GPU programming based on (fragment) shaders. These authors had to deal with important limitations, *e.g.*, the lack of scatter operations, limited floating-point precision and slow texture switching based on pixel buffers, as exposed by the 'rendering-based' GPU-programming paradigm. One of the first GPU implementations of FEM is due to Rumpf and Strzodka [152], in the context of solving linear and anisotropic diffusion equations. Related work on GPU-accelerated FEM simulations also include the papers by Göddeke and collaborators [72–74]. However, the emphasis is on improving the accuracy of *scientific* FEM-based simulations. Prior related work with respect to elastically deformable

models, discretized using FEM, can be found in [84, 96, 124]. They proposed methods which compensate for the rotation of the elements. Liu *et al.* [111] also present a FEM-based GPU implementation. Their results show that the involved CG method dominates the total computation time.

Since FEM often involves a CG solver, considerable research was done on efficiently mapping the CG method and Sparse Matrix-Vector Multiplications (SPMV) on modern GPUs using CUDA, see [18, 34, 183] and the references therein. Other approaches for solving the resulting PDE use multigrid methods, see, e.g., [69]. An efficient GPU implementation of a multigrid method, used for deformable models, was recently presented in [49]. Although multigrid methods typically converge faster than CG methods, implementing them efficiently on a GPU is a much more elaborate process. For example, invoking an iterative solver such as CG, constitutes only one of the steps of a multigrid method, the others being smoothing, interpolation and restriction.

4.3 Elasticity through the Method of Finite Elements

As common in computer graphics applications (see [124] and the references therein), we employ a linearized model based on *linear* elasticity theory [143]. Further, to solve the underlying PDE we use the Method of Finite Elements with *linear tetrahedral elements*, see Appendix D.1 for additional details.

4.3.1 Continuum Elasticity

In continuum elasticity, the deformation of a body, *i.e.*, a continuous connected subset M of $\mathbb{R}^3$, is given by the *displacement* vector field $\mathbf{u}(\mathbf{x}) = [u(\mathbf{x}), v(\mathbf{x}), w(\mathbf{x})]^T$, where $\mathbf{x} = [x, y, z]^T$ is some point of the body at rest. Thus, every point $\mathbf{x}$ of the undeformed body corresponds to a point $\mathbf{x} + \mathbf{u}(\mathbf{x})$ of the deformed one.

The equilibrium equation of the deformation is usually written in terms of the *stress tensor*, $\boldsymbol{\sigma}$. However, since it cannot be measured directly, one uses Cauchy's *linear strain tensor*, $\boldsymbol{\epsilon}$, and some material parameters to approximate the stress inside the body. Similar to Hooke's law for a 1D spring, in 3D one has

$$\boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\epsilon}, \tag{4.1}$$

for each point of the body, where $\mathbf{D} \in \mathbb{R}^{6\times6}$ is the so-called *material stiffness matrix* representing material parameters. The elastic force $\mathbf{f}_e$ acting at a point of the body is given by

$$\mathbf{f}_e = \mathbf{K}\mathbf{u} = \left(\mathbf{P}^T\mathbf{D}\mathbf{P}\right)\mathbf{u}, \tag{4.2}$$

with $\mathbf{K} \in \mathbb{R}^{3\times3}$, $\mathbf{f}_e$ and $\mathbf{u} \in \mathbb{R}^{3\times1}$. $\mathbf{K}$ represents the local *stiffness matrix* and $\mathbf{P} \in \mathbb{R}^{6\times3}$ is a matrix of partial derivative operators, see Appendix D.3 for their derivations.

4.3.2 System Dynamics

Having defined the elastic forces acting in a body, we now derive the equations of motion required to simulate the dynamic behavior of the object. The coordinate vectors $\mathbf{x}$ are now functions of time, *i.e.*, $\mathbf{x}(t)$, such that the equation of motion becomes

$$m\ddot{\mathbf{x}} + c\dot{\mathbf{x}} + \mathbf{f}_e = \mathbf{f}_{ext}, \tag{4.3}$$

where m is the mass of a body particle at position $\mathbf{x}$, c the damping coefficient, $\mathbf{f}_e$ the elastic force and $\mathbf{f}_{ext}$ the vector of external forces, i.e., the gravitational force. We approximate Equation (4.3) using a *semi-implicit* method, *i.e.*,

$$m\frac{\left(\mathbf{v}^{i+1}-\mathbf{v}^{i}\right)}{\Delta t}+c\mathbf{v}^{i+1}+\mathbf{K}\mathbf{u}^{i+i}=\mathbf{f}_{ext}^{i}. \tag{4.4}$$

$$\mathbf{x}^{i+1}=\mathbf{x}^{i}+\Delta t\,\mathbf{v}^{i+1}, \tag{4.5}$$

with $\mathbf{u}^{i+1}=\Delta t\mathbf{v}^{i+1}+\mathbf{x}^{i}-\mathbf{x}^{0}$, which can be rearranged as

$$\left(m+\Delta tc+\Delta t^{2}\mathbf{K}\right)\mathbf{v}^{i+1}=m\mathbf{v}^{i}-\Delta t\left(\mathbf{K}\mathbf{x}^{i}-\mathbf{K}\mathbf{x}^{0}-\mathbf{f}_{ext}^{i}\right). \tag{4.6}$$

4.3.3 Discretization using FEM

Within FEM, the continuous displacement field $\mathbf{u}$ is replaced by a discrete set of displacement vectors $\tilde{\mathbf{u}}$ defined only at the nodes of the elements. Within each element e the displacement field is approximated by

$$\mathbf{u}=\mathbf{N}_e\tilde{\mathbf{u}}, \tag{4.7}$$

where $\mathbf{N}_e\in\mathbb{R}^{3\times12}$ is the matrix containing the element *shape functions* and

$$\tilde{\mathbf{u}}=[u_1,v_1,w_1,\ldots,u_4,v_4,w_4]^T \tag{4.8}$$

the vector of the nodal displacement approximations. Next, Galerkin's method of weighted residuals is applied over the whole volume V, in which the *weighting* functions are equal to the shape functions. Each term in Equation (4.6) is weighted and approximated as in Equation (4.7), which results in

$$\begin{aligned}\int_V \mathbf{N}_e^T\left(m+\Delta tc+\Delta t^2\mathbf{K}\right)\mathbf{N}_e\tilde{\mathbf{v}}^{i+1}\,dV=\\ \int_V m\mathbf{N}_e^T\mathbf{N}_e\tilde{\mathbf{v}}^{i}\,dV-\Delta t\int_V\mathbf{N}_e^T\left(\mathbf{K}\mathbf{N}_e\tilde{\mathbf{x}}^{i}-\mathbf{K}\mathbf{N}_e\tilde{\mathbf{x}}^{0}-\mathbf{N}_e\tilde{\mathbf{f}}_{ext}^{i}\right)dV,\end{aligned} \tag{4.9}$$

with $\mathbf{N}_e^T$ the weighting functions. The equation above is defined for each individual element and generates one matrix consisting of the local mass ($\mathbf{M}_e$), damping ($\mathbf{C}_e$) and element stiffness ($\mathbf{K}_e$) matrices. Additionally, a local force matrix ($\mathbf{f}_e$) is generated, representing the net external force applied to the object. These local matrices are given by

$$\begin{aligned}\mathbf{M}_e&=m_e\int_V\mathbf{N}_e^T\mathbf{N}_e\,dV\\ \mathbf{C}_e&=c\int_V\mathbf{N}_e^T\mathbf{N}_e\,dV\\ \mathbf{K}_e&=\int_V\mathbf{N}_e^T\mathbf{P}^T\mathbf{D}\mathbf{P}\mathbf{N}_e\,dV\\ \mathbf{f}_e&=\int_V\mathbf{N}_e^T\mathbf{N}_e\tilde{\mathbf{f}}_{ext}\,dV,\end{aligned} \tag{4.10}$$

with m_e the mass of element e. See [143] and Appendix D for more details on computing these matrices.

Algorithm 3: Simulation algorithm.

```
1  Compute Q_p;
2  foreach element e do
3      Compute K_e, Equation (4.10);
4  while simulating do
5      foreach element e do
6          Compute volume v_e;
7          Compute R_e, Section 4.3.3;
8          Compute K_re = R_e K_e R_e^{-1} v_e;
9          Compute M_e = m_e Q_p v_e;
10         Compute C_e = c Q_p v_e;
11         Compute f_e0 = R_e K_e x_0 v_e;
12         Compute f_e = Q_p f̃_ext v_e, Equation (4.10);
13         Compute f_te = M_e v^i − Δt (f_e0 − K_re x^i − f_e);
14         Compute K_te = M_e + Δt C_e + Δt^2 K_re;
15     Assemble global K and f using K_te and f_te of elements;
16     Solve K v^{i+1} = f for v^{i+1};
17     Update x^{i+1} = x^i + Δt v^{i+1}, Section 4.3.2;
```

Finally, the global matrix $\mathbf{K} \in \mathbb{R}^{3n \times 3n}$ (with n the number of mesh vertices) is 'assembled' from individual element matrices. This resulting system is then solved using the *Conjugate Gradient* method for the unknown velocity $\mathbf{v}^{i+1}$, which is then used to update the positions of the nodes, see Equation (4.5). Equation (4.5) shows a first order method for updating the positions which can be replaced by higher order methods as described in [96].

Unfortunately, the above equations for simulating elastic deformation only work fine as long as the model does not undergo *large rotations*. This is because linearized elastic forces are used, which are only 'valid' close to the initial configuration. Therefore we use the so-called *Element-based Stiffness Warping* or *Corotational Strain* method [84, 124] to compensate for the rotation of the elements. To extract the rotation part of the deformation, we use the polar decomposition method proposed in [90]. The rotation-free element stiffness matrix $\mathbf{K}_{re}$ then becomes $\mathbf{K}_{re} = \mathbf{R}_e \mathbf{K}_e \mathbf{R}_e^{-1}$, with $\mathbf{R}_e \in \mathbb{R}^{12 \times 12}$ the rotation matrix for element e. Note that this gives rise to an initial elastic force $\mathbf{f}_{e0} = \mathbf{R}_e \mathbf{K}_e \mathbf{x}_0$, which replaces the term $\mathbf{K}\mathbf{N}_e \tilde{\mathbf{x}}^0$ in the right-hand-side of Equation (4.9).

4.4 Overview of the Algorithm

Algorithm 3 gives an overview of the simulation of elastically deformable models as described in Section 4.3. First, a tetrahedralization of the polygonal mesh representing the surface of the object is computed, see Section 4.5.5. Each tetrahedron is considered as an element in FEM. Then, the initial stiffness-matrices of the elements are computed (line 3); these matrices do not change during the simulation and thus are precomputed. Additionally, as the shape functions are constant during the simulation, we can precalculate most matrices from Equation (4.10), using $\mathbf{Q}_p = \int \mathbf{N}_e^T \mathbf{N}_e \, dV$ for a unit volume. This matrix is identical for all elements and is therefore only computed once. After all local matrices have been computed and stored (line 14),

the global matrix is assembled (line 15). The resulting linear system of equations is solved for velocities (line 16), which are then used to advance the position vectors (line 17).

4.5 GPU Mapping using CUDA

In this section we describe our GPU mapping of the simulation on NVida GeForce GPUs using CUDA [132]. First we shall give details about implementing the rotation extraction through polar decomposition. Then, we describe the computation of the local stiffness matrices which are used to assemble the global (sparse) stiffness matrix (matrix $\mathbf{K}$ in Algorithm 3). The resulting system of linear equations is solved using a Jacobi-Preconditioned CG Method.

4.5.1 Rotation Extraction

As mentioned in Section 4.3.3 we have to estimate the rotation of each element in order to calculate displacements properly. Finding the rotational part of the deformation matrix is done using a Polar Decomposition as described in [84, 90, 124]. Although a large number of matrix inversions is involved, this can be done efficiently because small 4×4 matrices are used. Since each matrix contains 16 elements, we chose to map the computations of 16 such matrices to a single CUDA thread-block with 256 threads.

For computing the inverse of a 4×4 matrix we perform a *co-factor expansion*. Each thread within a thread-block computes one co-factor of the assigned matrix. Since the computation of a co-factor requires almost all values of the matrix, memory accesses have to be optimized. In order to prevent for possible bank-conflicts during the computation of the co-factors, each matrix is stored in one memory bank of shared memory. Accordingly, the shared-memory segment (of size 16×16 locations) is regarded as a matrix stored in row-major order, where each column represents a 4×4 local matrix. Therefore, each column (local matrix) maps exactly to one memory-bank. Since a large number of rotation matrices are computed in parallel, a large performance boost is obtained.

4.5.2 Local Stiffness Matrices

Solving a specific problem using FEM starts with describing the problem locally per element. Since a typical problem consists of a large number of elements, the computations involved per element can be easily parallelized. Further, since the matrices used to construct $\mathbf{K}_e$ are in $\mathbb{R}^{12\times12}$, we map the computation of each individual local element stiffness matrix to a thread-block containing 12×12 threads. The inner loop in Algorithm 3 is implemented using one or two CUDA kernels, depending on the architecture version. Instead of creating kernels for each individual matrix operation, we combine a number of them into one larger kernel. Since data from global memory can be reused multiple times, less global memory transactions are required, which improves the overall performance.

4.5.3 Solving the Linear System

Given the local element matrices and load vectors, the global stiffness matrix of the system is assembled. Next, the system has to be solved for the unknown velocity $\mathbf{v}^{i+i}$. The (Jacobi-Preconditioned) CG method performs a large number of sparse matrix-vector multiplications and other vector-vector operations. Therefore, solving a large linear system efficiently, requires a fast and efficient implementation of sparse matrix-vector multiplications, which is highly-dependent on the layout used for storing the sparse matrix. Since three unknown values (components of the velocity vector) are associated to each mesh vertex, a block with 3×3

elements in the global matrix corresponds to each edge of the tetrahedral mesh. Therefore, a *Block-Compressed Sparse Row* (BCSR) format is very well suited for storing the global matrix, and thus improving the speed of the CG method.

Furthermore, since the vertex degree of internal nodes is constant in a regular tetrahedralization (see Section 4.5.5), the variation of the number of elements per row in the global matrix is minimal. Therefore, we use the optimized BCSR format from [183]. This method efficiently stores a large sparse-matrix in BCSR format and reorders the blocks in memory to improve the efficiency of the memory transactions. This fact is very important since the main bottleneck of the CG method is the memory throughput. In [183], through extensive experiments, it is shown that their optimized BCSR layout outperforms other storage formats for efficient matrix-vector multiplication on the GPU.

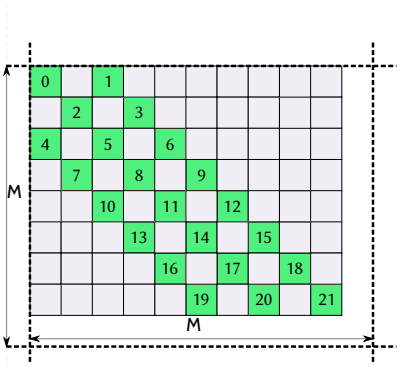
4.5.4 Global Matrix Update

Each local matrix represents a set of equations for each individual tetrahedron. To obtain the global system of equations, each component of each local matrix is added to the corresponding location of the global matrix. The location is obtained using the indices of the vertices for that specific element. Since the structure of the underlying mesh does not change during the simulation, also the structure of the global matrix remains unchanged. Therefore we assemble the global matrix only once and updates its values every time-step. In this section, we propose an extension of [183] which allows us to efficiently update a sparse matrix stored in the BCSR format.

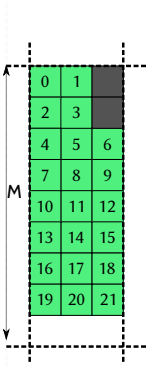
For updating the global matrix, two approaches are possible. Within the first approach (*scatter*), all values of a local matrix are added to their corresponding values in the global matrix. When the local matrices are processed on the GPU, many of them are processed in parallel. Therefore, multiple threads *can* update the same value in the global matrix at the same time, which results in *race conditions*. In order to prevent race conditions from appearing, access to the values of the global matrix would have to be serialized.

The second approach is to *gather* per element in the global matrix, the corresponding values from the local matrices. To do so, the indices of all associated local values are stored per element in the global matrix. Each index represents the position of the local value in an array A, which stores the values of all local matrices. Given these indices per global element value, the local values are looked-up and used to update the corresponding value in the global matrix.

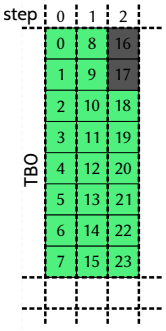
Within the optimized BCSR implementation of [183], the global sparse-matrix is divided in $N \times N$-sized blocks, Figure 4.2a. Next, block rows are compressed and sorted by length, Figure 4.2b. Finally, a number of consecutive block rows are grouped and mapped to a CUDA thread block. Within each group of block rows, the blocks are reordered in memory, such that accessing these blocks is performed as optimal as possible. Accessing the blocks (for, e.g., a multiplication) is done as follows. First, all threads of a thread-block ($TB0$) are used to access the blocks mapped to it in the first step (step 0), see Figure 4.2c. Each thread computes an index pointing to these blocks. Next, blocks $0 - 7$ are loaded from the global memory. Note that these are the same blocks appearing in the first column of Figure 4.2b. For the next step, each thread increases its current index, such that the next set of blocks ($8 - 15$) can be loaded (step 1). Note that all block rows must have the same length, and therefore, empty blocks must be padded (blocks 16 and 17).



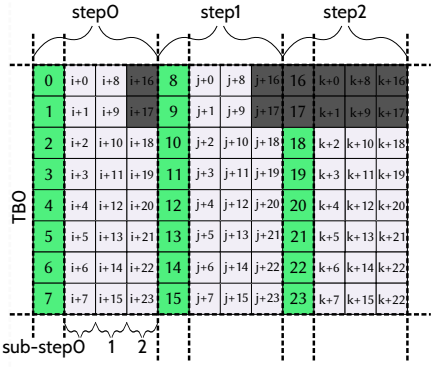
(a) Block layout of a sparse-matrix: Each green block stores $N \times N$ values and its position within the block-row. Numbers represent memory locations. Each gray block contains zero-values and is not explicitly stored. M represents the dimension of the matrix.



(b) BCSR layout: Each block-row is compressed; an additional array with indices to the first block in a block row is necessary (not shown here).



(c) Blocks of consecutive block rows are mapped to a thread block ($TB0$). Blocks mapped to the same thread block are reordered so that blocks processed in the *same step* are continuous in memory. Padding is required (gray blocks).



(d) Updating matrix blocks (green), requires the associated local values. The indices of these values are stored in *index blocks* (gray), in the same order as the matrix blocks. Within each sub-step, a set of continuous index-blocks are loaded and used to fetch the corresponding values from the local matrices. The dark-gray blocks are used for padding and contain -1's. i, j, k represent (starting) offsets in memory.

Figure 4.2: Updating the sparse matrix: the initial sparse matrix is created, stored and processed, (a), (b) and (c). Updating the matrix is done by collecting the corresponding values from the local matrices, (d).

To actually update the data blocks of the global matrix, we use a *gather* approach. Accordingly, $N \times N$-sized *index blocks* are used for each matrix block, see Figure 4.2d. Since the matrix blocks have a specific ordering, the same ordering is used for the index-blocks. For each step, a number of *sub-steps* is performed. Within each sub-step a set of index-blocks is loaded from memory, given a start offset (i, j or k in Figure 4.2d). Then, for each index-block, its $N \times N$ values (indices) are used to fetch the corresponding $N \times N$ data values from local matrices, stored in global memory. Please note that the $N \times N$ data values fetched using one $N \times N$ index-block, do not come, in general, from the same local matrices. To accumulate the local contributions, an array (stored in shared memory) is used. If an index has value -1, no update is performed. For the next sub-step, the indices pointing to the index blocks are increased. Therefore, per step, the number of index blocks for each processed matrix block must be equal, which requires padding with '-1' index blocks. The advantage of this approach is that loading the indices and writing the updated values always result in an optimal throughput. Loading the actual local-element values is in general not optimal.

4.5.5 Tetrahedralization and Rendering

The quality of the tetrahedral mesh is essential for efficiently simulating a deforming elastic object represented by a polygonal mesh. We have experimented with tetrahedralizations in which the surface mesh forms the outer boundary of the tetrahedral mesh. Since the triangles of the surface mesh can have a high variety in size, the generated tetrahedralization also contains tetrahedral elements with a high variation in size and configuration. This can have a negative effect on the quality of the tetrahedralization. Therefore, we chose to create a tetrahedral mesh, using equi-sized elements, which however, may result in a rather rough approximation of the original surface mesh. We tackle this problem by coupling the input polygonal mesh to the (deforming) tetrahedral mesh, as follows.

First, a regular 3D grid of N^3 voxels is created, in which each voxel containing a part of the surface is marked as important; typical values for N are 32, 64 or 128. Next, a regular tetrahedralization of the grid is created using equi-sized tetrahedral elements, and each element containing at least one important vertex of the grid, is stored. Further, the inner volume of the object is tetrahedralized using the same equi-sized tetrahedral elements. Next, in order to reduce the amount of elements, those elements belonging to the inner volume are merged together into fewer larger ones. This reduces the total amount of elements and thus the total computation time. Note however that this approach is most useful with models which have large internal volumes, similar to the bunny in Figure 4.5. Finally, the original surface mesh is coupled with the tetrahedral one similar to [124]: each vertex in the original surface mesh is mapped to exactly one tetrahedron, and its barycentric coordinates in that tetrahedron are stored along with the vertex coordinates.

When the new positions of the tetrahedra are computed, the surface mesh is also updated. To compute new positions of the deformed surface mesh, for each vertex of the input mesh, the positions of the four vertices of the corresponding tetrahedron are looked-up and interpolated using the barycentric coordinates of the original vertex.

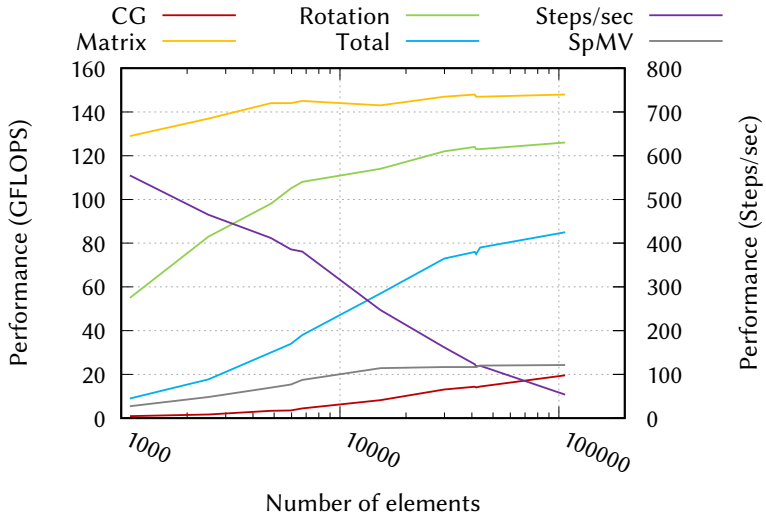


Figure 4.3: Performance results with different numbers of elements. *CG* represents the performance of the CG solver, *Matrix* – the performance for computing the local element matrices, *Rotation* – the performance of the rotation extraction procedure, *SpMV* – the performance of the SpMV operation; *Total* represents the overall performance. *Steps/sec* represents the number of simulation steps per second. The global-matrix update was performed with an effective throughput of 50 GB/sec.

4.6 Results

All experiments have been performed on a machine equipped with an Intel Q6600 quad-core processor and a GeForce GTX 570 with 1.2 Gb of memory. Figure 4.3 shows the performances obtained for computing the local element matrices (*Matrix*), the rotation matrices (*Rotation*), solving the resulting linear system (*CG*), performing a single SpMV (*SpMV*), and the total performance (*Total*) as a function of the number of elements. *Steps/sec* is the corresponding number of simulation steps performed per second. Similarly, Figure 4.4 shows the computation time per simulation time-step. For each model, we have used the following material parameters: Young's modulus of elasticity, $E = 5 \times 10^5 N/m^2$; Poisson's ratio, $\mu = 0.2$; density, $\rho = 1000 KG/m^3$. Furthermore, the time-step of the simulation $\Delta t = 0.001$ and the volume of each initial element $v_e = 1.65 \times 10^{-6} m^3$. Each model used in this chapter is scaled such that each dimension is at most 66 cm and is tetrahedralized as described in Section 4.5.5. With these settings, the CG solver found a solution for each model in 5 to 18 iterations. In order to obtain a generic performance picture, we have fixed the number of iterations to 18, which resulted in the performances from Figure 4.3.

Within Figure 4.3 a number of interesting patterns can be seen. First, the performance for computing the local element matrices reaches its maximum very soon. Since each matrix is mapped to exactly one thread-block, a large amount of thread-blocks is created, resulting in a 'constant' performance. Second, the performance figures for computing the rotation matrices show a larger variation. Since 16 rotation matrices are processed by one thread-block, a significantly smaller amount of thread-blocks is used. Finally, the performance of the CG method seems to be low compared to the other operations. The CG method operates on a global sparse-matrix and performs a large number of sparse-matrix vector multiplications

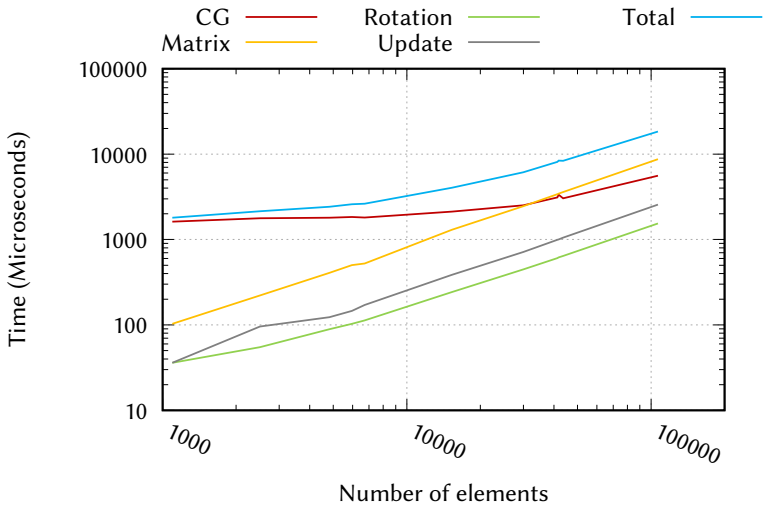


Figure 4.4: Timing results with different numbers of elements, per time-step. *CG* represents the time of the CG solver, *Matrix* – the time for computing the local element matrices, *Rotation* – the time of the rotation extraction procedure; *Total* represents the total elapsed time per time-step.

(SpMVs) and vector-vector operations, for which the performances are mainly bound by the memory throughput. However, the CG performances from Figure 4.3 agree with those from [183], given the dimensions of the problem.

The measured, effective throughput for updating the global matrix was about 50 GB/sec, in all cases with more than $5k$ elements. Since this operation transfers a large amount of data, the memory bus is saturated very soon, resulting in a good throughput. However, since not all transactions can be coalesced, the maximum throughput is not reached. This operation is very similar to an SPMV with 1×1 blocks, but now for a matrix containing d times more elements, with d the degree of internal nodes in the model. This observation shows that the measured throughput is close to the expected one, according to the results in [183].

As expected, the total performance increases with the number of elements. This shows that the computational resources are used efficiently for larger models. The number of elements, for which the maximum performance is reached, depends on the actual GPU mapping of the computations. For example, the CG solver does not reach its maximum performance for $100k$ elements, while the computation of the local element matrices reaches its peak at $5k$ elements. Due to this, one can expect better performances for the CG method when larger models are used. Furthermore, for models having less than $30k$ elements, the total computation is dominated by the time spent by the CG solver. For larger models, more time is spent on computing the local matrices, see Figure 4.4.

The measured overall performance is based on the total time needed per simulation step, which includes all operations performed, except the rendering of the model. Figure 4.3 also shows the number of simulation steps performed per second, given the number of elements; these numbers are based on the total computation time. Accordingly, even for large models, interactive frame rates can be reached. A rough comparison of the obtained performance and frame rate with other state-of-the-art multigrid GPU implementations [49] shows that, even if in theory the CG method converges slower than multigrid, comparable results *can* be obtained for simi-

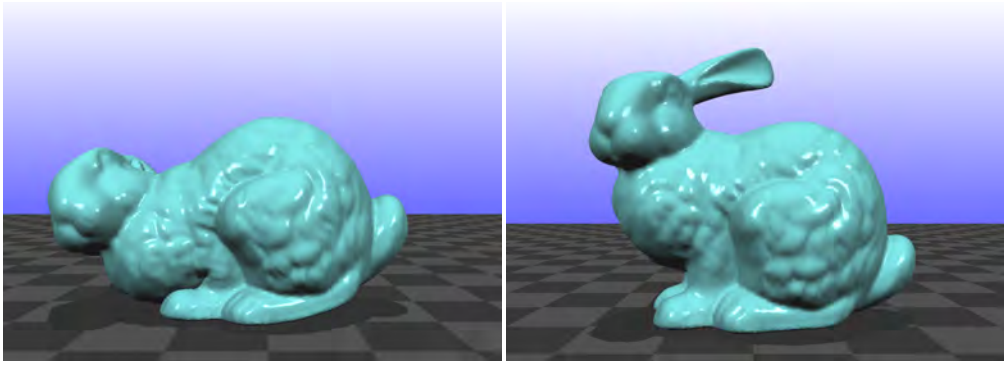


Figure 4.5: Material properties and collision handling. *Left*: flexible material ($E = 5 \times 10^4$). *Right*: stiffer material ($E = 5 \times 10^5$). Simulation rate: 120 frames per second.

lar models. We assume that memory transactions in our method are more efficient, despite of transferring more data. However, more research is required to get a full understanding of the differences between both methods performed on modern GPUs, with respect to performance figures. Finally, Figures 4.1 and 4.5 to 4.9 show example results from our simulations.

4.7 Conclusions

We have presented an efficient method for simulating elastically deformable models for graphics applications, accelerated on modern GPUs using CUDA. Our method relies on a fast Conjugate Gradient solver and an efficient mapping of the SPMV operation on modern GPUs [183]. Since the topology of the underlying grid does not change during the simulation, data structures are reused for higher efficiency. To further improve the performance, we proposed a scheme which allows to efficiently update the sparse matrix, during the simulation.

In future work we will investigate the performance of this method when multiple GPUs are used. Furthermore, we will investigate the performance difference between traditional CG methods and multigrid methods performed on modern GPUs. Also, we plan to enhance the simulation to allow for plastic behavior as well as brittle and fracture of stiff materials.

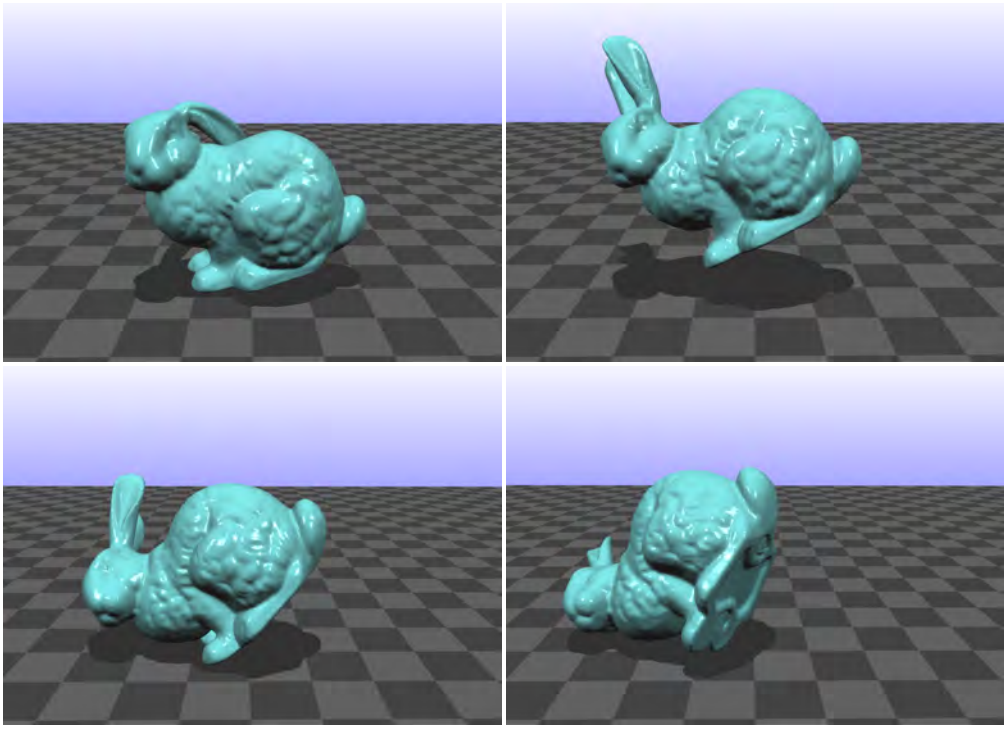


Figure 4.6: Bunny bouncing on the floor. Simulation rate: 120 frames per second.

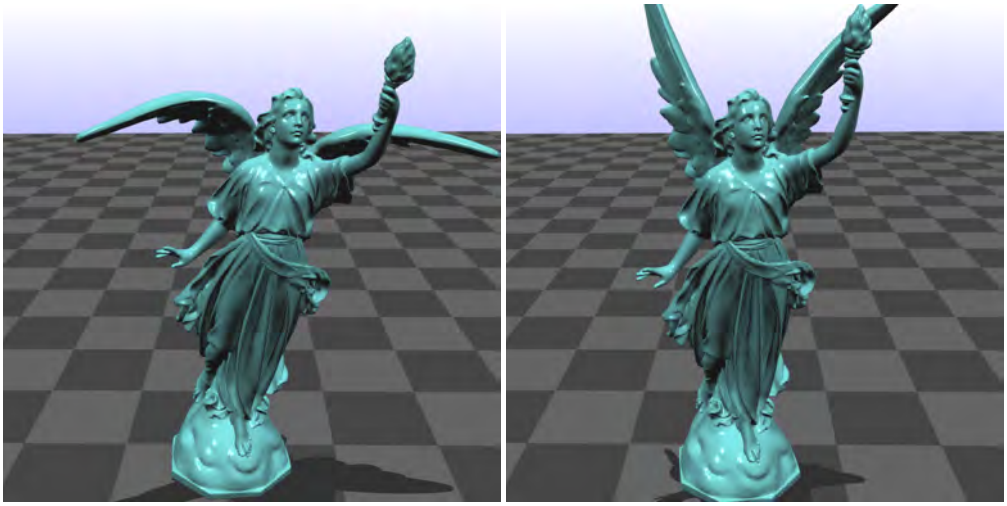


Figure 4.7: Left: applying external forces on the wings. Right: after releasing the external forces. Simulation rate: 116 frames per second.

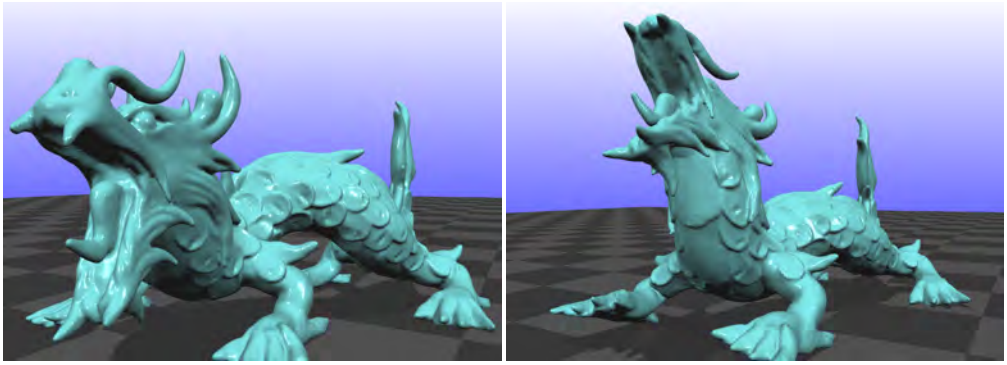


Figure 4.8: Left: stretching and deforming a model using external forces. Right: deformation after releasing external forces. Simulation rate: 118 frames per second.

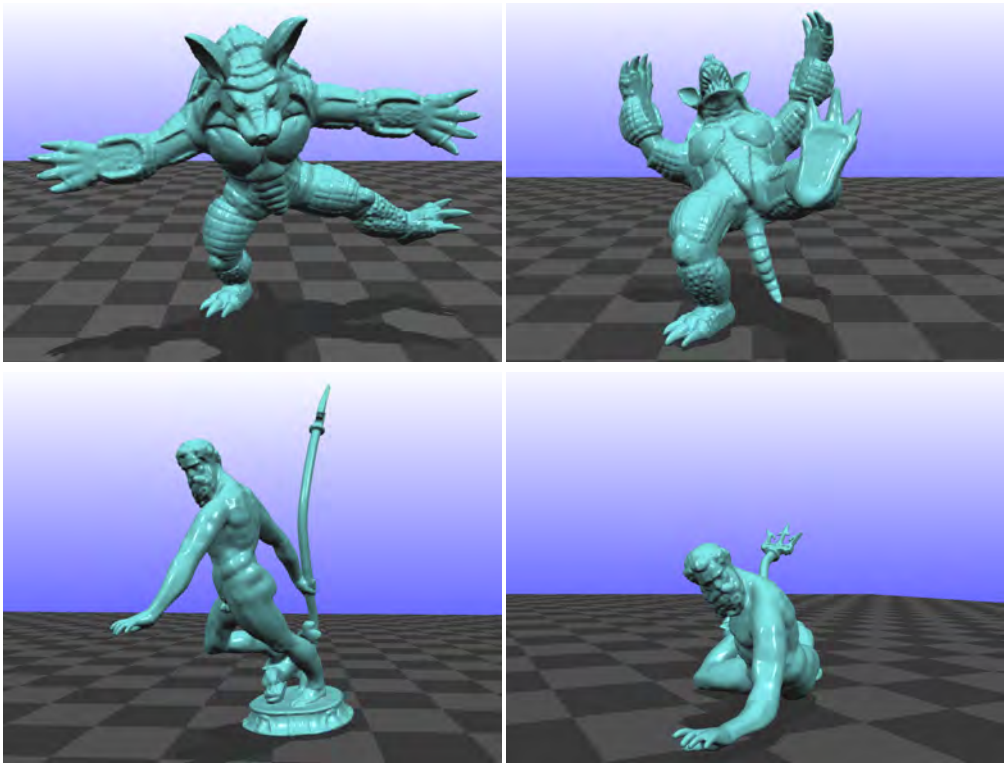


Figure 4.9: Other simulation results. Simulation rate: 160 frames per second.

5

Collision response

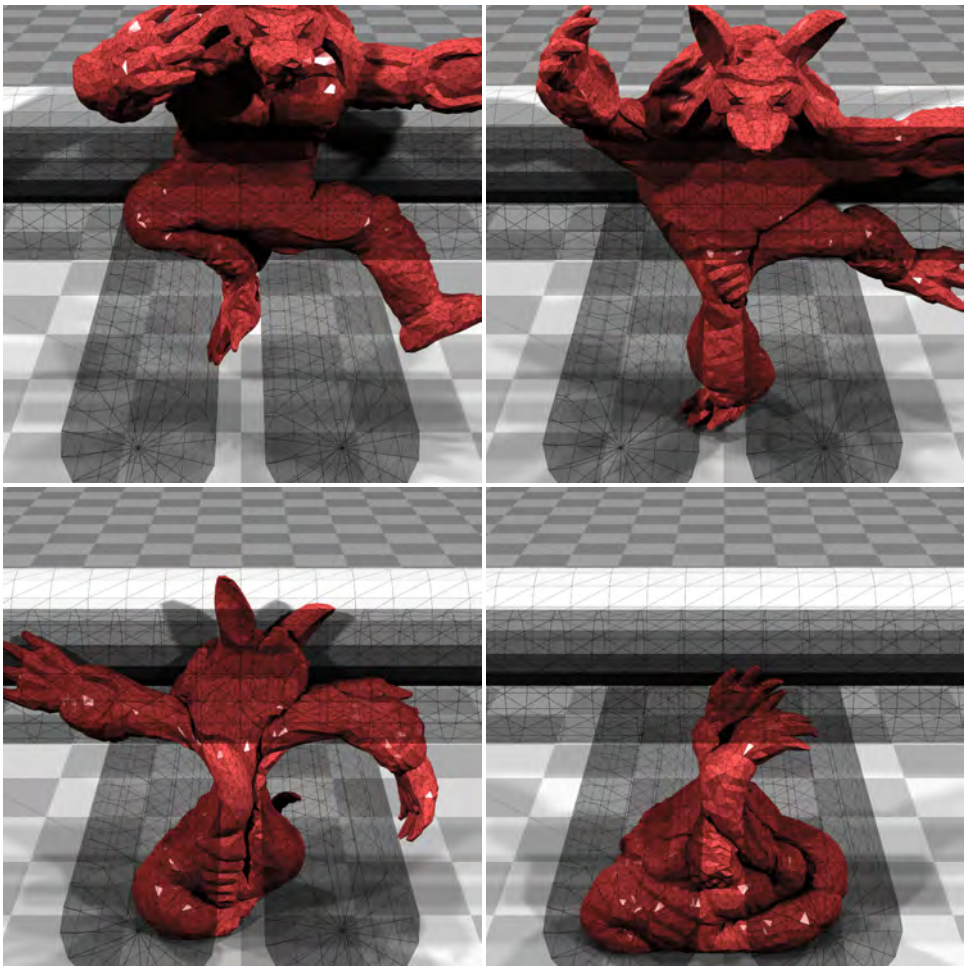


Figure 5.1: A hyper-elastic Armadillo pulled through a set of (transparent) rotating cylinders at time-steps 5000, 7500, 10000 and 12500. The timing results of this experiment for various methods are shown in Figure 5.11.

Simulating (elastically) deformable models that can collide with each other and with the environment remains a challenging task. The resulting contact problems can be elegantly approached using Lagrange multipliers to represent the unknown magnitude of the response forces. Typical methods construct and solve a Linear Complementarity Problem (LCP) to obtain the response forces. This requires the inverse of the generalized mass matrix, which is in general hard to obtain for deformable-body problems. In this chapter we tackle such contact problems by directly solving the Mixed Linear Complementarity Problem (MLCP) and omitting the construction of an LCP matrix. Since a convex quadratic program with linear constraints is equivalent to an MLCP, we propose to use a Conjugate Residual (CR) solver as the backbone of our collision-response system. By dynamically updating the set of active constraints, the MLCP with inequality constraints can be efficiently solved. We also propose a simple yet efficient preconditioner that ensures faster convergence. Finally, our approach is faster than existing methods (at the same accuracy), and it allows accurate treatment of friction.

5.1 Introduction

Physically-based models are nowadays widely used in many computer graphics applications, such as animated movies and video games. Typical animations consist of many (complex) objects that can interact with each other and the environment. The behavior of such objects is based on their internal dynamics, modeled through some material simulation, e.g., rigid-body, cloth or fluid simulation. However, the very motion of each object can also influence the motion of other objects in the scene. Therefore, accurate treatment of collision events dramatically improves the realism of the simulation, delivering rich and visually pleasing animations. In addition to translational and rotational motion, the motion of a deformable object is also affected by shape changes (deformations). Since the deformation is generally unknown, no exact collision time and location can be directly computed. Furthermore, the collision response influences the dynamics of the objects and thus their motion, and ideally, it should also account for accurate friction.

Typical approaches for handling contact problems reformulate them as velocity constraints and use *Lagrange multipliers* to represent the unknown magnitudes of the contact forces. Then, the resulting *Mixed Linear Complementarity Problem* (MLCP) has to be solved, which yields a solution that agrees with both the collision response problem and the model dynamics. The constrained problem is usually tackled by reformulating it as a *Linear Complementarity Problem* (LCP), which typically is solved using (Projected) Gauss-Seidel methods. The construction of the LCP matrix requires the inverse of the generalized mass matrix, which can be directly computed for rigid-body simulations. However, for deformable bodies, this inverse is not directly available and has to be approximated (e.g., see Otaduy et al. [138]). Therefore, one often seeks an approximation or solves a system of coupled problems. Alternatively, directly solving the original MLCP constitutes a viable approach for deformable models. Ramage and Wathen [147] compared the performance of the Conjugate Residual (CR) method for solving coupled indefinite problems, stemming from finite-element discretizations of the Stokes equations, to that of a two-level approach in which two nested Conjugate Gradient (CG) solvers were used. Their results show that the CR method outperforms the coupled CG approach for such equality-constrained problems. Inspired by their results, we shall investigate here how the CR method can be extended for efficiently solving contact problems involving inequality constraints.

For stable simulations of coupled rigid and deformable bodies, it is important that all collisions are resolved completely at the end of each time-step. However, when a collision is resolved, the resulting deformation may lead to new collisions or changes in contact forces elsewhere. This often results in a state in which contacts do not agree with the actual geometry. If such errors are not corrected, these contacts will eventually introduce energy to the system, resulting in oscillations and instabilities. Additionally, the larger the deformation is, the more likely that such mismatches appear. Such mismatches can occur even for rigid bodies. To minimize these errors, one needs to solve non-linear contact problems.

In this chapter we present a method that focuses on solving the contact problem for deformable bodies, without the need to compute the *Delassus operator*. Due to this, contacts can be efficiently linearized, which allows us to solve the non-linear contact problem, resulting in a guaranteed collision-free state at convergence. The method is based on the CR method which simultaneously provides estimates for both deformation and contact forces. By allowing constraints to change status (switch among active or inactive states), inequality (*non-penetration*)

constraints can be handled directly. Furthermore, friction is modeled similarly, such as by switching between static and kinetic friction constraints. By continuously updating the sliding directions of the kinetic friction constraints, Coulomb's friction model is approximated. Additionally, we propose a simple yet effective preconditioner that significantly reduces the number of iterations and computation time. The method allows for large deformations and contact forces, stable-stacking of (almost) rigid objects, and an accurate approximation of Coulomb's friction cone, see Figures 5.1, 5.2 and 5.9. Finally, our approach is faster compared to other methods, with the speedup typically increasing with the complexity of the simulation.

5.1.1 Previous and Related Work

A tremendous amount of work has been done addressing physical simulations for computer animation, including rigid-body, cloth, deformable-object and fluid simulations. Here, we shall only give a very brief overview of highly related methods. For background material on contact mechanics, we refer to the work of Wriggers [193] and references therein.

Rigid Bodies Although penalty methods for collision response [13, 121] are relatively fast and easy to implement, the computed penalty forces have to compete with all other forces acting on the simulated bodies. Therefore, the forces may fail to avoid inter-penetration. Impulse-based methods [7] model the response of a collision event through the application of contact impulses. These methods apply impulses on objects to resolve collisions, but can trigger new ones. This problem can be solved by using shock propagation techniques [77]. Other approaches used in rigid-body simulations often model collision response as a constrained optimization problem [8, 58, 125, 148]. This class of methods also includes approaches based on LCP formulations [43], typically used in rigid-body simulations [9], interactive applications [37, 180] and cloth simulations [30]. LCPs are frequently used in combination with implicit time-integration schemes, such that a collision-free state is guaranteed for the next time-step [167]. It is this last property that makes such approaches very appealing for solving the contact problem for deformable models as well. Anitescu and Hart [5] develop fixed-point iteration algorithms that solve convex sub-problems that are guaranteed, for small friction coefficients, to obtain the unique velocity solution of the non-convex friction LCP. Their method converges at a linear rate. A non-smooth non-linear CG method is presented by Silcowitz-Hansen et al. [160], which combines Projected Gauss-Seidel (PGS) with a Fletcher-Reeves CG method. Bertails-Descoubes et al. [25] present a method for simulating contacts between small rods or fibers. The method models exact Coulomb friction and uses a non-smooth Newton solver, which relies on an easy construction of the Delassus operator. Xu et al. [196] present a method for rigid bodies, which solves a Quadratic-Programming (QP) problem based on contact constraints. A Singular-Value-Decomposition (SVD) solver is used to form the (singular) Schur complement matrix, which is then used in combination with Cholesky decomposition and back-substitution. Friction is modeled using friction anchors. Although a penalty-based method, due to the implicit-integration scheme used, this approach can also be interpreted as a method that solves constraints. Mazhar et al. [118] introduce the so-called Accelerated Projected Gradient Descent approach to accelerate the simulation of large systems of rigid bodies interacting through normal and frictional contact. The method can easily be parallelized, resulting in speedups of one to two orders of magnitude. Silcowitz-Hansen et al. [159] propose a PGS method working on a subspace of the problem. At each iteration, the method first performs an approximation using PGS, followed by a more accurate solve concerning a subset of active constraints having non-clamped multipliers. A detailed overview of numerical methods

for solving LCPs is given by Erleben [59]. Finally, an extensive overview of rigid-body simulations is given by Bender et al. [20]. Other related projected Krylov methods, used in rigid many-body / granular-material simulations, are found in [89, 149].

Deformable models Early work on collision handling for deformable models includes the method of Baraff and Witkin [10], which models the collision response through constraints yielding contact forces. By solving a QP problem, an optimal solution is obtained. Although not directly used for deformable objects, *Constraint Anticipation* [9] is a technique that transforms an MLCP into an LCP. After solving the LCP for the unknown Lagrange multipliers, collision response impulses are applied on the colliding models. When applied to rigid bodies, this technique is very efficient. Otaduy et al. [138] extend this approach for deformable models as *Iterative Constraint Anticipation* (ICA). ICA computes the multipliers and the collision response using two nested (Projected) Gauss-Seidel solvers that approximate both the multipliers and velocities. Raghupathi and Faure [146] use an *Active Set* approach for solving the contact response problem: Given a set of active and inactive constraints, a QP problem is solved using the CG method. On convergence, constraints can change state between active and inactive. The Staggered Projections (SP) method [101] computes an unconstrained velocity that is corrected using two coupled projection steps (each solved using a QP solver). This iterative process continues until the residual is minimized. The method of Allard et al. [2] constructs volume-based constraints that ensure a zero intersection volume of two colliding objects. The resulting MLCP is solved using the Gauss-Seidel-like method of Duriez et al. [50]. The latter approximates exact Coulomb friction and explicitly constructs the Delassus operator (an LCP matrix), for which some approximations were made. Daviet et al. [47] present a scalable and robust solver for capturing Coulomb friction in large assemblies of tightly packed fibers, such as hair. The method can handle a few thousand fibers subject to tens of thousands frictional contacts at a reasonable computational cost. Li et al. [108] present an efficient Gradient Projection method for computing contact responses by decoupling constraints. Unfortunately, their method cannot handle coupled frictional constraints efficiently. Works on collision detection [178], coupling of rigid and deformable bodies [157], deformable bodies [66, 94, 142, 164], and penalty forces [174] are all closely related to the contact problem for deformable models.

Comparison to our method We build on methods that involve velocity-level constraints and compute Lagrange multipliers in combination with implicit time integration. As seen earlier, many methods found in rigid-body applications require the construction of an LCP, Schur complement matrix, or Delassus operator, which relies on the availability of the inverted generalized mass matrix. For rigid-body problems, this inverse can be obtained relatively easily, whereas for Finite Element Method (FEM)-based deformable-body problems, this is generally not immediately possible. Therefore, many of these methods are not efficient for simulating deformable bodies with a large number of degrees of freedom, see Section 5.6.2. Indeed, for deformable-body problems, often coupled solvers are used [101, 138] and/or PGS methods are used [2, 50]. Otaduy et al. [138] iteratively approximate interleaved normal and friction responses, but their method does not satisfy the Maximum Dissipation Principle (MDP). In Kaufman et al. [101] this principle is guaranteed. This method iteratively solves two coupled QP problems and has been demonstrated for rigid and reduced-order flexible multibody systems. The method of Duriez et al. [50] models exact Coulomb friction (satisfying the MDP) and solves the problem using a PGS approach.

All methods just mentioned first solve an unconstrained problem, which is corrected later, or first compute collision impulses, which are applied as external forces to objects. Our approach differs from all these works in that no coupled solvers are used, neither is an LCP, Delassus, or Schur complement matrix constructed or used. We approximate the Coulomb friction model using an additional kinetic friction force, aligned with the sliding velocity, thus satisfying the MDP. The corresponding QP problem (containing the generalized mass matrix of the simulation and its constraints) is solved using the CR method. By changing the state of the constraints, projections are performed that directly apply to the global problem such that friction and normal responses affect the internal (elastic) energy of the objects. In addition, constraints can be added or re-linearized at convergence to guarantee that a global, collision-free state is obtained, satisfying all constraints. To accelerate the convergence of the method, we derive a preconditioner that significantly reduces the amount of iterations. Finally, in this work, we create constraints per contact point, although we could also use constraints based on a volumetric description of a collision, similar to Allard et al. [2].

5.2 Background

5.2.1 Linear and Hyper-Elasticity

The deformation of a body can be described by a mapping $\boldsymbol{\phi}$ from material coordinates $\mathbf{X}$ to world coordinates $\mathbf{x}$ (i.e., $\mathbf{x} = \boldsymbol{\phi}(\mathbf{X})$). The stress $\mathbf{P}$ at a point $\mathbf{X}$ depends solely on the deformation gradient $\mathbf{F} = \frac{\partial \boldsymbol{\phi}}{\partial \mathbf{X}}$ and is obtained through some energy density function $\Psi(\mathbf{F})$ via $\mathbf{P}(\mathbf{F}) = \frac{\partial \Psi}{\partial \mathbf{F}}$. Since the stress is invariant under rotations, $\mathbf{P}(\mathbf{UF}) = \mathbf{U}\mathbf{P}(\mathbf{F})$ holds for any rotation $\mathbf{U}$. Furthermore, if the material is isotropic, $\mathbf{P}(\mathbf{F}\mathbf{V}^T) = \mathbf{P}(\mathbf{F})\mathbf{V}^T$ holds for any rotation $\mathbf{V}$. By diagonalizing $\mathbf{F}$ using SVD, $\mathbf{F}$ becomes $\mathbf{U}\hat{\mathbf{F}}\mathbf{V}^T$, so the stress can be obtained through

$$\mathbf{P}(\mathbf{F}) = \mathbf{U}\mathbf{P}(\hat{\mathbf{F}})\mathbf{V}^T, \tag{5.1}$$

with $\hat{\mathbf{F}}$ a diagonal matrix containing the singular values of the deformation gradient. Given the (isotropic) constitutional model used in Ψ, a wide range of materials can be simulated. If Ψ is a quadratic function, a linear elastic model is obtained, for which the unrotated force gradient is constant; this coincides with co-rotational FEM [124]. Once the stress $\mathbf{P}$ is obtained, the nodal force $\mathbf{f}_i$ is obtained by

$$\mathbf{f}_i = -\mathbf{P}(\mathbf{F})\mathbf{b}_i, \tag{5.2}$$

with $\mathbf{b}_i$ the area-weighted normals of the faces connected to node i for a particular volume and stress $\mathbf{P}$, see [95]. By assembling the nodal forces for all volumes, net force $\mathbf{f}$ is obtained. To obtain the force gradient, $\frac{\partial \mathbf{f}}{\partial \mathbf{x}} = \sum_i \frac{\partial \mathbf{f}}{\partial \mathbf{F}_i} \frac{\partial \mathbf{F}_i}{\partial \mathbf{x}}$, the gradients of the SVD in Equation (5.1) with respect to $\mathbf{F}$ are required, see [161] for its derivation. When dealing with hyper-elastic materials, the computed force responses can become very large (due to the non-linear nature of the energy model) when the compression becomes large. To robustly simulate such materials, we use the method of Stomakhin et al. [168], which linearly extrapolates the energy density function after a certain amount of compression is reached, see Appendix D.4 for additional details.

5.2.2 Dynamics and Numerical Integration

Given Newton's second law of motion $\mathbf{M}\mathbf{a} = \mathbf{f}$, a first-order Taylor expansion of the net force $\mathbf{f}$ is performed, yielding

$$\mathbf{M}\mathbf{a} = \mathbf{f} + \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \frac{\partial \mathbf{x}}{\partial t} + \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{v}} \frac{\partial \mathbf{v}}{\partial t} = \mathbf{f} + \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \mathbf{v} + \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{v}} \mathbf{a}. \tag{5.3}$$

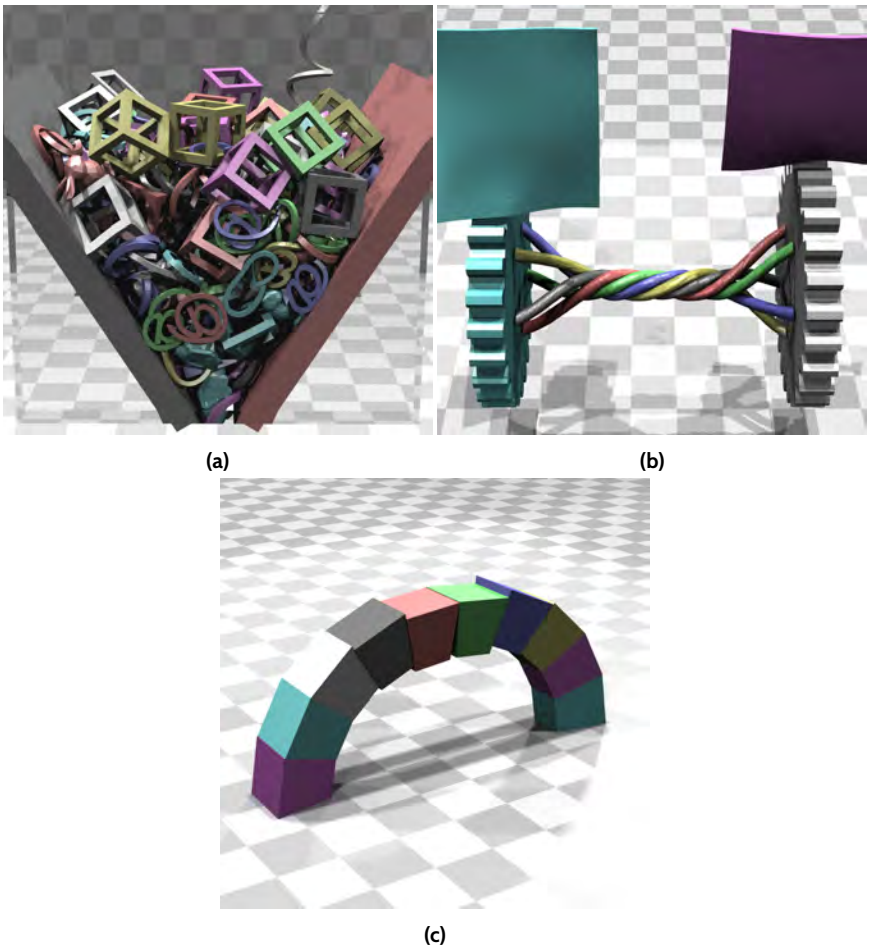


Figure 5.2: Example simulations by our method: (a) difficult case with many deformable objects in a moving wedge; (b) large deformations and elastic forces appear when the shown elastic strings are tangled; by releasing external forces, the strings are untangled; (c) stable stacking of rigid objects, illustrating accurate friction treatment.

Using a first-order forward difference approximation of the acceleration $\mathbf{a}$, the following *semi-implicit* system is obtained

$$\left(\mathbf{M} + \Delta t \mathbf{C}_d + \Delta t^2 \mathbf{K}\right) \mathbf{v}^{i+1} = \left(\mathbf{M} + \Delta t \mathbf{C}_d\right) \mathbf{v}^i + \Delta t \mathbf{f}, \tag{5.4}$$

with $\mathbf{M}$ the mass matrix of the system, $\mathbf{v}$ the velocity, damping matrix $\mathbf{C}_d = -\frac{\partial \mathbf{f}}{\partial \mathbf{v}}$, and stiffness matrix $\mathbf{K} = -\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$, see [11]. Positions $\mathbf{x}$ are updated using

$$\mathbf{x}^{i+1} = \mathbf{x}^i + \Delta t \mathbf{v}^{i+1}. \tag{5.5}$$

We discretize Equation (5.4) in space using FEM. Accordingly, for each element (tetrahedron), a per-element equivalent of Equation (5.4) is generated, given mapping $\mathbf{x} = \boldsymbol{\phi}(\mathbf{X})$. By assembling all these local instances into a global one, the global version of Equation (5.4) is obtained, with now $\mathbf{M}, \mathbf{C}_d$ and $\mathbf{K}$ the *global* mass-, damping-, and stiffness matrix, and $\mathbf{x}, \mathbf{v}, \mathbf{f}$ now vectors representing positions, velocities and all internal (Equation (5.2)) and external forces (e.g.,

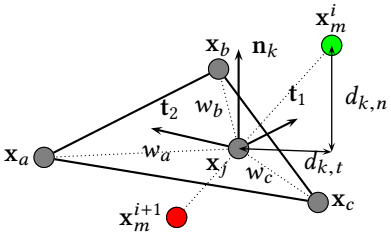


Figure 5.3: Vertex-face contact pair. Vertex $\mathbf{x}_m^i$ (green) at iteration i penetrates a face at $i+1$ (red). $\mathbf{x}_j$ is the collision point on the face, w_a, w_b, w_c its barycentric weights, $\mathbf{n}_k$ the contact normal and $\mathbf{t}_{1,2}$ the tangent vectors; $d_{k,n}$ represents the normal distance at the beginning of the time-step and $d_{k,t}$ the tangential distance.

gravitational forces) of all nodes in the discretized model. This system is stored using a large yet sparse matrix in the form $\mathbf{A}\mathbf{v}^{i+1} = \mathbf{b}$, with $\mathbf{A} \in \mathbb{R}^{3N_v \times 3N_v}$ the generalized mass matrix $(\mathbf{M} + \Delta t\mathbf{C}_d + \Delta t^2\mathbf{K})$, $\mathbf{b}$ the right-hand side, and N_v the total number of vertices. Such systems are typically solved using the CG method for the unknown velocities $\mathbf{v}^{i+1}$. Given the new velocities, the positions are updated using Equation (5.5).

5.2.3 Non-Penetration Constraints

Two deformable bodies have collided if the signed distance between any two points on their surfaces Γ_1 and Γ_2 is negative. Such collision events are detected by considering *vertex-face* and *edge-edge* contact pairs. For the vertex-face case, let $\mathbf{x}_m^i \in \Gamma_1$ be a (discrete) vertex on the surface of the first object, which penetrates Γ_2 in the next time-step, resulting in a collision point $\mathbf{x}_j$ on a triangular face of Γ_2, see Figure 5.3. The (signed) distance $d_{k,n}$ between $\mathbf{x}_j^i$ and $\mathbf{x}_m^i$ is computed using

$$C_k(\mathbf{x}^i) = (\mathbf{x}_m^i - \mathbf{x}_j^i) \cdot \mathbf{n}_k^i = d_{k,n}, \tag{5.6}$$

with $\mathbf{n}_k^i$ the *contact normal*, k the constraint identifier, and $\mathbf{x}^i$ a vector containing all vertex positions at time-step i. To avoid penetration, $C_k(\mathbf{x}^i) \geq 0$ must hold for any $\mathbf{x}_j$ and $\mathbf{x}_m$ pair. Clearly, this should also hold *after* the semi-implicit update (i.e., $C_k(\mathbf{x}^i + \Delta t\mathbf{v}^{i+1}) \geq 0$). Since Equation (5.4) solves for $\mathbf{v}^{i+1}$, Equation (5.6) is transformed into a velocity constraint using a first-order approximation–that is,

$$\begin{pmatrix} \Delta t \frac{\partial C_k}{\partial \mathbf{x}_j} & \Delta t \frac{\partial C_k}{\partial \mathbf{x}_m} \end{pmatrix} \cdot \begin{pmatrix} \mathbf{v}_j^{i+1} \\ \mathbf{v}_m^{i+1} \end{pmatrix} \geq -d_{k,n} = c_{k,n}, \tag{5.7}$$

with $\mathbf{v}_j^{i+1}$ and $\mathbf{v}_m^{i+1}$ the new vertex velocities, and $c_{k,n}$ the constraint constant. After collecting all resulting non-penetration constraints and assembling all instances of Equation (5.7), one obtains

$$\mathbf{J}\mathbf{v}^{i+1} \geq \mathbf{c}_n, \tag{5.8}$$

with $\mathbf{J} \in \mathbb{R}^{N_c \times 3N_v}$ the Jacobian matrix containing the derivatives of *all* positional constraints, N_c the number of non-penetration constraints, N_v the number of vertices, $\mathbf{v}$ a vector representing all vertex velocities, and $\mathbf{c}$ the right-hand side of Equation (5.7), for all constraints. Since $\mathbf{x}_j = w_a\mathbf{x}_a + w_b\mathbf{x}_b + w_c\mathbf{x}_c$ and we consider vertex-face and edge-edge pairs, $\mathbf{J}$ contains per contact point a sparse row-vector $\mathbf{j}_k \in \mathbb{R}^{1 \times 3N_v}$, *e.g.*, $\mathbf{j}_k = \begin{pmatrix} -w_a\Delta t\mathbf{n}_k^T, & -w_b\Delta t\mathbf{n}_k^T, & -w_c\Delta t\mathbf{n}_k^T, & \Delta t\mathbf{n}_k^T \end{pmatrix}$ for a vertex-face constraint C_k, with $\mathbf{n}_k$ the contact normal, see Figure 5.3. At this point we consider $\mathbf{n}_k$ to be constant within one time-step. In Section 5.3.3, an extension is proposed that takes the change of the constraints into account. Furthermore, self-collisions in which

$\Gamma_1 = \Gamma_2$, and degenerate collisions (vertex-vertex/vertex-edge) are treated similarly. However, the latter will produce duplicate contacts that need special care, see Section 5.3.4. Similarly, constraints can be derived for contact involving rigid bodies, see Appendix A.3.1.

5.2.4 Constrained Velocity as a Complementarity Problem

We model collision response using Lagrange multipliers so that normal contact forces are given by $\mathbf{f}_n = \mathbf{J}^T\boldsymbol{\lambda}$, with $\boldsymbol{\lambda}$ the vector of Lagrange multipliers representing the unknown magnitudes of all contact forces. According to Signorini's contact model (see also [50]), a complementarity condition is further imposed–to prevent penetration, the contact forces should push the bodies apart, and hence $\boldsymbol{\lambda} \geq \mathbf{0}$. Furthermore, if the non-penetration constraint is not violated ($C_k(\mathbf{x}) > 0$ for some constraint), its corresponding multiplier should be $\lambda_k = 0$. Hence, $(\mathbf{J}\mathbf{v} - \mathbf{c}_n)^T\boldsymbol{\lambda} = (\mathbf{v}^T\mathbf{J}^T - \mathbf{c}_n^T)\boldsymbol{\lambda} = \mathbf{v}^T\mathbf{f}_n - \mathbf{c}_n^T\boldsymbol{\lambda} = 0$, which gives the complementarity condition.

Contact forces are simply included in the motion equations of the model (see Section 5.2.2), resulting in

$$\mathbf{0} \leq \begin{pmatrix} \mathbf{A} & \mathbf{J}^T \\ \mathbf{J} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{v} \\ \boldsymbol{\lambda} \end{pmatrix} - \begin{pmatrix} \mathbf{b} \\ \mathbf{c}_n \end{pmatrix} \perp \begin{pmatrix} \mathbf{1} \\ \boldsymbol{\lambda} \end{pmatrix} \geq \mathbf{0}, \tag{5.9}$$

(see [43]), which is an MLCP for which the Karush-Kuhn-Tucker (KKT) optimality conditions [106] apply.

5.3 Collision Response through the Conjugate Residual Method

MLCPs can be solved by transforming them into LCPs [9], provided that matrix $\mathbf{A}$ can be easily inverted. They can also be solved by any algorithm for strictly convex QP problems with linear constraints. Given the MLCP from Equation (5.9), the corresponding QP problem is

$$\begin{aligned} \min f(\mathbf{v}) &= \frac{1}{2}\mathbf{v}^T\mathbf{A}\mathbf{v} - \mathbf{v}^T\mathbf{b} \\ \text{subject to } \mathbf{J}\mathbf{v} &\geq \mathbf{c}_n. \end{aligned} \tag{5.10}$$

A sufficient condition to guarantee strict convexity is for matrix $\mathbf{A}$ to be positive definite. In this case, $f(\mathbf{v})$ is a strictly convex function, and if the constraints are linear, it has a unique global minimizer [27]. Since according to Equation (5.4), $\mathbf{A}$ is given by the linear combination of the (positive-definite) mass and (positive semi-definite) stiffness and damping matrices, $\mathbf{A}$ is positive definite. If the material simulation does not produce a positive semi-definite matrix, this property must be enforced by, for example, by removing the negative eigenvalues from the element stiffness matrices.

The quadratic function in Equation (5.10), but subject to $\mathbf{J}\mathbf{v} = \mathbf{c}_n$, is equivalent to

$$\underbrace{\begin{pmatrix} \mathbf{A} & \mathbf{J}^T \\ \mathbf{J} & \mathbf{0} \end{pmatrix}}_{\mathbf{B}} \underbrace{\begin{pmatrix} \mathbf{v} \\ \boldsymbol{\lambda} \end{pmatrix}}_{\mathbf{y}} = \underbrace{\begin{pmatrix} \mathbf{b} \\ \mathbf{c}_n \end{pmatrix}}_{\mathbf{d}}, \tag{5.11}$$

or $\mathbf{B}\mathbf{y} = \mathbf{d}$ in short. Since matrix $\mathbf{B} \in \mathbb{R}^{(3N_v+N_c)\times(3N_v+N_c)}$ is not positive-definite, the system in Equation (5.11) cannot be solved by the CG method. The CR method [113] is a Krylov subspace method, similar to the (more popular) CG method, and minimizes $\|\mathbf{B}\mathbf{y} - \mathbf{d}\|^2$ and thus

solves Equation (5.11). In the following sections, we describe how the CR method is used to solve the contact problem, including friction. Additionally, we describe how constraints are re-linearized in order to obtain a collision-free state. For details on deriving and preconditioning the CR method we refer to Appendices B.2.3 and B.2.4.

5.3.1 Conjugate Residual Method with Non-Penetration Constraints

Luenberger [114] proposed a method for solving constrained nonlinear programming problems by treating inequality constraints as equality ones that can regularly change states. The method uses a *descent step* to improve the current approximation. When the problem is a QP with linear constraints, the CR method can be used to perform these descent steps. In the following, we describe how the method in [114] is applied in case of contact and friction and how it is used in combination with the CR method in Algorithm 4.

For now, let j denote the loop index inside the CR method. Furthermore, recall that each constraint C_k is represented by a sparse row-vector $\mathbf{j}_k$ in Jacobian $\mathbf{J}$, its corresponding Lagrange multiplier λ_k, and some constant $c_{k,n}$, see Section 5.2.3. The complementarity condition in Equation (5.9) states that if $\lambda_k > 0$ is true, then $\mathbf{j}_k\mathbf{v} = c_{k,n}$ also holds for some constraint C_k, with $\mathbf{v}$ the global velocity. Conversely, if $\lambda_k = 0$ is true, then $\mathbf{j}_k\mathbf{v} \geq c_{k,n}$ and $\left(\mathbf{j}_k\mathbf{v} - c_{k,n}\right)^T \lambda_k = \mathbf{0}$ also hold. Therefore, constraint C_k does not contribute to the computation of $\mathbf{v}$ and can be *deactivated* in Line 13 or Line 23. Due to this mechanism, summarized in Algorithms 5 and 6, the CR method can solve the problem in Equation (5.10). In other words, each constraint is *active* or *inactive* throughout the solver iterations, and its state is regularly updated. The method converges when $\|\mathbf{r}_{j+1}\| < \epsilon_1$ and $\|\mathbf{U}\mathbf{r}_{j+1}\|_\infty < \epsilon_2$, with ϵ_1 and ϵ_2 a relative and absolute tolerance, respectively, and $\mathbf{U}$ a matrix that selects the entries from $\mathbf{r}$ that involve active constraints. This ensures a maximum constraint error less than ϵ_2. The second convergence criterion is used to detect local minima, see Section 5.4.3. Furthermore, when the CR method converges, new constraints can be added and inactive constraints are removed. However, the main difference with active set methods is that constraints' state changes are allowed before the solver converges. Therefore, the problem is to decide *when* to activate or deactivate constraints *prior* to convergence. Note that when a constraint is inactive, multiplications $\mathbf{B}\mathbf{p}_j$ and $\mathbf{B}\mathbf{y}_{j+1}$ are performed such that the corresponding entries in $\mathbf{J}$ are neglected. This also applies for the corresponding terms of the preconditioner $\mathbf{C}$, see Section 5.3.4 for details about preconditioning. Furthermore, when the residual is recomputed for inactive constraints, the values in $\mathbf{d}$ corresponding to inactive constraints are also neglected.

Constraint activation The activation of a constraint C_k depends on $c_{k,n}$, $\mathbf{j}_k$ and $\mathbf{v}$ at solver iteration j. Distance $d_{k,n}$ is computed using Equation (5.6) and stored in $c_{k,n}$ when the constraint is added to the system (when a collision is detected), or when the constraint is re-linearized. At the same moment, $\mathbf{j}_k$ is computed and included in $\mathbf{J}$, see Section 5.5 for more details. To determine when a constraint should be activated (assuming it is inactive), the penetration depth is evaluated regularly. If

$$\mathbf{j}_k\mathbf{v} - c_{k,n} \leq 0 \tag{5.12}$$

holds, the constraint is activated and the corresponding Lagrange multiplier will be computed, which would eventually resolve the collision, see Line 8 of Algorithm 5.

Constraint deactivation Since our solver allows for activation and deactivation of constraints at any time, a criterion is needed to decide when to deactivate them. It may seem logical to deactivate a constraint C_k if $\mathbf{j}_k\mathbf{v} - c_{k,n} \geq 0$. However, since $\mathbf{v}$ does not reflect the final velocity, this criterion cannot *solely* be used. When both $\mathbf{j}_k\mathbf{v}$ and λ_k are used instead, the

Algorithm 4: Pseudo-code of our collision handling system.

```
1  Discretize computational domain;
2  Initialize BVH and additional data structures;
3  while Simulating do
4      Update matrix A using FEM, see Equation (5.4);
5      Find all potential collision candidates, Section 5.5.1;
6      Linearize and check all active constraints;
7      r_0 = d − By_0; p_0 = C^{-1} r_0, j = 0;
8      while Not converged do
9          α = (r_j^T C^{-1} B C^{-1} r_j) / (p_j^T B C^{-1} B p_j);
10         y_{j+1} = y_j + α p_j;
11         r_{j+1} = r_j − α B p_j;
12         if α > 0 then
13             EvaluateConstraints(‖r_{j+1}‖, j, ε_1, ε_2);
14         if No constraints changed then
15             β = −(r_{j+1}^T C^{-1} B C^{-1} r_{j+1}) / (r_j^T C^{-1} B C^{-1} r_j);
16             p_{j+1} = C^{-1} r_{j+1} − β p_j;
17             B p_{j+1} = B C^{-1} r_{j+1} − β B p_j;
18         else
19             r_{j+1} = d − B y_{j+1};
20             p_{j+1} = C^{-1} r_{j+1};
21             B p_{j+1} = B C^{-1} r_{j+1};
22         if (‖r_{j+1}‖ < ε_1 ∧ ‖U r_{j+1}‖_∞ < ε_2) ∨ ‖B C^{-1} r_{j+1}‖ < ε_1 then
23             EvaluateConstraints(‖r_{j+1}‖, 0, ε_1, ε_2);
24             if No constraint states have changed then
25                 Check all candidates for new collisions, Section 5.5.2;
26                 if No new constraints are added then
27                     Re-linearize constraints Section 5.3.3 and Equation (5.21);
28                     if No constraints are updated then
29                         Advance simulation;
30                         break;
31             Recompute residual, see Lines 19 to 21;
32         j = j + 1;
```

Algorithm 5: Pseudo-code for constraint evaluations.

```
1  Function EvaluateConstraints(r, i, ε1, ε2):
2      interval = max(1, ⌊log²(r/ε1)/3⌋);
3      if i mod interval == 0 ∨ i == 0 then
4          foreach contact k do
5              if Non-penetration constraint active then
6                  if j_k v − c_{k,n} ≥ 0 and λ_k ≤ 0 then
7                      Deactivate constraint, λ_k = 0;
8              else if j_k v − c_{k,n} ≤ 0 then
9                  Activate constraint, set friction to kinetic friction;
10             if Non-penetration constraint active then
11                 if Friction constraint in static friction mode then
12                     if ‖γ_k‖ > μλ_k and (j_{k,t} v − c_{k,t})^T γ_k > 0 then
13                         Switch to kinetic friction;
14                         δ̄_k = γ̄_k, λ'_k = ‖γ_k‖/μ, γ_k = 0;
15                 else if (j_{k,t} v − c_{k,t})^T δ̄_k ≤ 0 then
16                     Switch to static friction;
17                     γ_k = γ'_k, γ'_k = 0;
18                 UpdateKineticFriction();
19             if Z^{-1} S_d [Δλ_k, Δγ_k^T + Δγ'^T_k]^T > ε2 then
20                 Use updated constraint k and report change to solver;
21             else
22                 Discard changes for constraint k;
23         d = [(b − J_t γ')^T, c_n^T, c_t^T]^T /*Update RHS*/;
```

criterion can be formulated as follows: if

$$\mathbf{j}_k\mathbf{v} - c_{k,n} \geq 0 \text{ and } \lambda_k \leq 0 \tag{5.13}$$

hold, then there is no collision and the constraint is attracting the objects–that is, it tries to enforce the equality $\mathbf{j}_k\mathbf{v} = c_{k,n}$. Therefore, the constraint is deactivated, see Line 6 of Algorithm 5. This is thus the only state that allows deactivation of a non-penetration constraint. For all other combinations, the constraint is kept active since there is still a collision ($\mathbf{j}_k\mathbf{v} - c_{k,n} < 0$) or objects are repelled ($\lambda_k > 0$).

Unlike [114], our CR-based solver for contact problems updates *only* the active set at dynamic intervals determined by the residual norm, see Line 2 of Algorithm 5. When the residual is small, constraints are evaluated more frequently. Conversely, for a larger residual norm, constraints are evaluated less frequently, which reduces the number of residual evaluations (each requiring a multiplication **By**). This makes our method more efficient than the CR solver in [114]. Additionally, our method supports a left preconditioner (matrix $\mathbf{C}^{-1}$ in Algorithm 4).

5.3.2 Incorporating Friction Constraints

Friction is also treated as a constraint on the velocity. According to Coulomb's friction model, friction can be described by a cone that bounds the friction force $\mathbf{f}_f$ in all directions, given the magnitude of the normal force $\mathbf{f}_n$ and a friction coefficient μ, see Figure 5.4. When $\|\mathbf{f}_f\| < \mu\|\mathbf{f}_n\|$, the friction force is not bounded and the sliding velocity is zero. When $\|\mathbf{f}_f\| = \mu\|\mathbf{f}_n\|$, the friction force is bounded and is not large enough to keep the sliding velocity at zero. Equation (5.9) can therefore be extended with additional complementarity conditions, resulting in the following problem:

$$\mathbf{0} \leq \begin{pmatrix} \mathbf{A} & \mathbf{J}^T & \mathbf{J}_t^T & \mathbf{0} \\ \mathbf{J} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{J}_t & \mathbf{0} & \mathbf{0} & -\mathbf{e}^T \\ \mathbf{0} & \boldsymbol{\mu} & -\mathbf{e} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{v} \\ \boldsymbol{\lambda} \\ \boldsymbol{\gamma} \\ \boldsymbol{\beta} \end{pmatrix} - \begin{pmatrix} \mathbf{b} \\ \mathbf{c}_n \\ \mathbf{c}_t \\ \mathbf{0} \end{pmatrix} \perp \begin{pmatrix} \mathbf{1} \\ \boldsymbol{\lambda} \\ \boldsymbol{\gamma} \\ \boldsymbol{\beta} \end{pmatrix} \geq \mathbf{0} \tag{5.14}$$

with $\mathbf{J}_t^T\boldsymbol{\gamma} = \mathbf{f}_f$ the applied friction force, $\mathbf{J}_t \in \mathbb{R}^{N_p N_c \times 3N_v}$ a matrix containing tangent vectors for all friction constraints, $\boldsymbol{\gamma}$ a vector of unknown multipliers representing the magnitude of these vectors, and $\mathbf{c}_t$ containing all tangential distances for all constraints. Per constraint, the tangential distances $\mathbf{c}_{k,t}$ (see Figure 5.3) are computed similar to Equation (5.6), albeit the normal vector is replaced by a tangent one. Please note that $\mathbf{c}_{k,t}$ represents the distance from configuration $\mathbf{x}_m^i$ to the first moment of impact at $\mathbf{x}_j$. For persistent contacts, $\mathbf{c}_{k,t} = \mathbf{0}$ since $\mathbf{x}_m^i$ is already in contact. Matrix $\mathbf{J}_t$ usually contains, per contact point, a number N_p of scaled instances of tangent vectors, each corresponding to one facet of the discretized friction cone. Furthermore, $\mathbf{e}$ and $\boldsymbol{\mu}$ are matrices used to couple all multipliers in $\boldsymbol{\lambda}, \boldsymbol{\gamma}$, and $\boldsymbol{\beta}$. The latter representing the unknown sliding velocities. If β_k is positive for some friction constraint C_k, a positive sliding velocity exists and the friction force is bounded, resulting in dissipation of energy. If β_k is zero, the friction force is not bounded and no sliding occurs. Unfortunately, Equation (5.14) is not symmetric and non-convex, and is therefore not equivalent to a QP problem. In the next section, a modification is proposed that restores this relation.

Modified Friction Model

Given the model described in Equation (5.14), we now proceed by adapting this model. Instead of using a discretized friction cone, a continuous and bounded kinetic friction force is modeled. This approach is summarized in Algorithms 5 and 6, and discussed here. Per constraint C_k, the friction force is decomposed using *only* two perpendicular tangent vectors $\mathbf{t}_{k,1}$ and $\mathbf{t}_{k,2}$, which appear in $\mathbf{j}_{k,t} \in \mathbb{R}^{2\times 3N_v}$ as part of $\mathbf{J}_t$, see Figure 5.4. Due to this, $\boldsymbol{\gamma}$ contains per constraint C_k, two Lagrange multipliers ($\gamma_{k,1}$, $\gamma_{k,2}^T = \boldsymbol{\gamma}_k$, which can be both *positive and negative.* For clarity of exposure, we assume for now that all friction constraints in the system are either in the static or kinetic state. To also keep the presentation brief, we may not make a clear distinction between model and method.

Static friction For static friction, the sliding velocity is zero ($\boldsymbol{\beta} = \mathbf{0}$), so Equation (5.14) can be written as

$$\mathbf{0} \leq \begin{pmatrix} \mathbf{A} & \mathbf{J}^T & \mathbf{J}_t^T \\ \mathbf{J} & \mathbf{0} & \mathbf{0} \\ \mathbf{J}_t & \mathbf{0} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{v} \\ \boldsymbol{\lambda} \\ \boldsymbol{\gamma} \end{pmatrix} - \begin{pmatrix} \mathbf{b} \\ \mathbf{c}_n \\ \mathbf{c}_t \end{pmatrix} \perp \begin{pmatrix} \mathbf{1} \\ \boldsymbol{\lambda} \\ |\boldsymbol{\gamma}| \end{pmatrix} \geq \mathbf{0}, \tag{5.15}$$

with $|\boldsymbol{\gamma}|$ the componentwise absolute values of $\boldsymbol{\gamma}$, and the additional condition that $\mathbf{0} \leq \boldsymbol{e\gamma} \leq \boldsymbol{\mu\lambda}$, which for a constraint C_k means that $0 \leq \|\boldsymbol{\gamma}_k\| \leq \mu\lambda_k$. The second inequality should hold for the static case, because the friction force is not bounded by the friction cone, and the solver

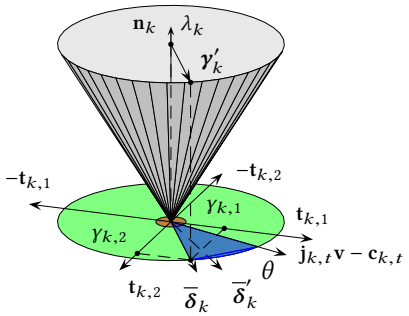


Figure 5.4: A representation of the friction cone in which sliding velocity $\mathbf{j}_{k,t}\mathbf{v} - \mathbf{c}_{k,t}$ is misaligned by angle θ (blue segment) with the friction vector $\overline{\delta}_k$. To address this, we compute the new friction direction $\overline{\delta}'_k$ that is more aligned with the sliding velocity, see Algorithm 6. If the magnitude of the sliding velocity is below the solver tolerance (red area), no update of $\overline{\delta}_k$ is performed.

computes a friction force that satisfies $\mathbf{j}_{k,t}\mathbf{v} = \mathbf{c}_{k,t}$. Conversely, if both

$$\|\boldsymbol{\gamma}_k\| > \mu\lambda_k \text{ and } (\mathbf{j}_{k,t}\mathbf{v} - \mathbf{c}_{k,t})^T\boldsymbol{\gamma}_k > 0 \tag{5.16}$$

hold, the applied friction force is too large and acts in the direction of the sliding velocity; thus, the conditions for static friction are violated and the constraint is switched to kinetic friction, see Line 12 of Algorithm 5.

Kinetic friction For kinetic friction, the sliding velocity is positive ($\boldsymbol{\beta} > \mathbf{0}$), which makes Equation (5.14) non-symmetric. Such a system can be solved as described in [20]. Alternatively, this system can be symmetrized by removing slack vector $\boldsymbol{\beta}$ from the equation. This has the advantage that solutions always exist and can be found by Lemke's algorithm [167]. Unfortunately, dropping $\boldsymbol{\beta}$ results in a violation of the MDP, as $\boldsymbol{\beta}$ selects a discrete vector in $\mathbf{J}_t$ that ensures this principle. In our method, the MDP is enforced differently.

Our approach for handling kinetic friction could be interpreted as a *Bounded Linear Complementarity Problem (BLCP)* [99] reformulation of Equation (5.14), where the alignment of the discretized friction cone depends on the state of the system. The relation in the last row of Equation (5.14) is enforced separately. This means that the kinetic friction force will never be over-estimated, as it could happen in the BLCP model.

For kinetic friction, $\boldsymbol{\beta} > \mathbf{0}$, thus $\boldsymbol{\mu\lambda} = \mathbf{e}\boldsymbol{\gamma}$, which for a particular constraint C_k means that $\mu\lambda_k = \|\boldsymbol{\gamma}_k\|$. The sliding velocity is given by $\mathbf{j}_{k,t}\mathbf{v} - \mathbf{c}_{k,t} = \mathbf{e}^T\boldsymbol{\beta} > \mathbf{0}$. The basic idea is now to estimate both the magnitude and direction of $\boldsymbol{\gamma}_k$.

Let $\boldsymbol{\gamma}'_k = (\mu\lambda'_k\delta_{k,1},\ \mu\lambda'_k\delta_{k,2})^T$ be such an estimate, with λ'_k an approximation of λ_k, and $\overline{\boldsymbol{\delta}}_k = \overline{\delta_{k,1}\mathbf{t}_{k,1} + \delta_{k,2}\mathbf{t}_{k,2}}$ a unit vector, representing the direction of the kinetic friction force in the tangent plane, *e.g.*, at $\mathbf{x}_j$, see Figure 5.4. Next, we eliminate from Equation (5.14) all rows and columns corresponding to $\boldsymbol{\beta}$ and move $\mathbf{J}_t^T\boldsymbol{\gamma}'$ to the right-hand side so that our approximation can be expressed in matrix form as

$$\mathbf{0} \le \begin{pmatrix} \mathbf{A} & \mathbf{J}^T \\ \mathbf{J} & \mathbf{0} \end{pmatrix}\begin{pmatrix} \mathbf{v} \\ \boldsymbol{\lambda} \end{pmatrix} - \begin{pmatrix} \mathbf{b} - \mathbf{J}_t^T\boldsymbol{\gamma}' \\ \mathbf{c}_n \end{pmatrix} \perp \begin{pmatrix} \mathbf{1} \\ \boldsymbol{\lambda} \end{pmatrix} \ge \mathbf{0}, \tag{5.17}$$

supplemented per constraint by the condition $(\mathbf{j}_{k,t}\mathbf{v} - \mathbf{c}_{k,t})^T\overline{\boldsymbol{\delta}}_k > 0$. This condition is violated when the sliding velocity changes direction due to a too large friction force, i.e., when

$$(\mathbf{j}_{k,t}\mathbf{v} - \mathbf{c}_{k,t})^T\overline{\boldsymbol{\delta}}_k \leq 0. \tag{5.18}$$

By then, switching the constraint to static friction, the solver will estimate $\boldsymbol{\gamma}_k$, which eventually satisfies $\mathbf{j}_{k,t}\mathbf{v} - \mathbf{c}_{k,t} = \mathbf{0}$, see Line 15 of Algorithm 5.

Kinetic friction update Earlier, a correct approximation of $\boldsymbol{\gamma}'_k$ was assumed to be available; here, we show how this vector is updated such that the friction force and the sliding velocity are aligned, and that the magnitude of the friction force is bounded. Given the preceding conditions, a violation occurs if

$$\mu\lambda_k \neq \|\boldsymbol{\gamma}'_k\| = \| \left(\mu\lambda'_k\delta_{k,1},\ \mu\lambda'_k\delta_{k,2}\right)^T \|. \tag{5.19}$$

If this happens, *Simple Moving Average* (SMA) $\lambda_{k,a}$ is updated by adding the current λ_k to its history of n values ($n = 2$ in our implementation; see Line 4 of Algorithm 6). Next, λ'_k is set to the SMA of $\lambda_{k,a}$ at Line 5. To impose the MDP, the kinetic friction direction $\overline{\boldsymbol{\delta}}_k$ and the sliding velocity have to be kept aligned. When the angle θ (see Figure 5.4) between $\overline{\boldsymbol{\delta}}_k$ and the current sliding velocity is larger than a given threshold ($\epsilon_\theta = 1.2$ degrees in our implementation), the difference $\Delta\boldsymbol{\delta}_k$ between both vectors is computed (Line 7). Next, $\overline{\boldsymbol{\delta}}_k$ is updated using $\overline{\boldsymbol{\delta}}_k + \alpha_\theta\overline{\Delta\boldsymbol{\delta}}_k$ and then normalized, with α_θ a small step size (0.01 in our implementation; see Line 8 of Algorithm 6). Thus, $\overline{\boldsymbol{\delta}}_k$ is gradually updated to realign it with the current sliding velocity. Finally, $\boldsymbol{\gamma}'_k$ is approximated, see Line 9 of Algorithm 6. Using this strategy, eventually all kinetic friction vectors approach their final configuration and satisfying the conditions specified in Equation (5.17). By using this strategy, a stabilization mechanism is introduced which suppresses undesired oscillations while solving the contact problem. Finally, in a real simulation the system contains both static and kinetic friction constraints, so the problem can be described in matrix form as

$$\mathbf{0} \leq \begin{pmatrix} \mathbf{A} & \mathbf{J}^T & \mathbf{J}_t^T \\ \mathbf{J} & \mathbf{0} & \mathbf{0} \\ \mathbf{J}_t & \mathbf{0} & \mathbf{0} \end{pmatrix}\begin{pmatrix} \mathbf{v} \\ \boldsymbol{\lambda} \\ \boldsymbol{\gamma} \end{pmatrix} - \begin{pmatrix} \mathbf{b} - \mathbf{J}_t^T\boldsymbol{\gamma}' \\ \mathbf{c}_n \\ \mathbf{c}_t \end{pmatrix} \perp \begin{pmatrix} \mathbf{1} \\ \boldsymbol{\lambda} \\ \boldsymbol{\gamma} \end{pmatrix} \geq \mathbf{0}, \tag{5.20}$$

where, for static constraints, entries in $\boldsymbol{\gamma}'$ are set to zero, whereas for kinetic constraints, all entries in $\boldsymbol{\gamma}$, $\mathbf{c}_t$ and those of matrix $\mathbf{B}$ corresponding to $\mathbf{J}_t, \mathbf{J}_t^T$ are similarly set to zero.

5.3.3 The Non-Linear Contact Problem

The problem solved in Equation (5.20) assumes that $\mathbf{J}$ and $\mathbf{J}_t$ are constant during the time-step, which is generally not true. Hence, $C(\mathbf{x}^{i+1}) \perp \boldsymbol{\lambda}$ may not hold at the end of the time-step. To minimize such errors, the complementarity conditions must also hold after updating the geometry. To solve this non-linear problem, $C(\mathbf{x})$ is re-linearized using an interpolated approximation of $\mathbf{x}$ based on its previous and current approximations, followed by a solve of Equation (5.20). Instead of interpolating $C(\mathbf{x})$ globally, one can re-linearize each individual $C_k(\mathbf{x})$, i.e.,

$$C_k^{l+1'}(\mathbf{x}_{i+1}) = wC_k^{l+1}(\mathbf{x}_{i+1}) + (1-w)C_k^l(\mathbf{x}_i), \tag{5.21}$$

with $0 < w < 1$ a weight such that the change in the contact normal is less than, say, 15 degrees, and C_k^l indicates the last, C_k^{l+1} the current, and $C_k^{l+1'}$ the new interpolated configuration of the constraining geometry. The re-linearization of the constraints results in an updated $\mathbf{J}$,

Algorithm 6: Pseudo-code for kinetic friction update.

Function `UpdateKineticFriction()`:
  if *Constraint k in kinetic friction mode* **then**
    if $\|\gamma'_k\| \neq \mu\lambda_k$ **then**
      Compute Simple Moving Average $\lambda_{k,a}$;
      $\lambda'_k = \lambda_{k,a}$;
    if $0 < \overline{\boldsymbol{\delta}}_k^T(\overline{\mathbf{j}_{k,t}\mathbf{v} - \mathbf{c}_{k,t}}) < \cos(\epsilon_\theta)$ /*If angle $\theta > \epsilon_\theta$*/ **then**
      $\Delta\boldsymbol{\delta}_k = (\overline{\mathbf{j}_{k,t}\mathbf{v} - \mathbf{c}_{k,t}}) - \overline{\boldsymbol{\delta}}_k$ /*Compute difference*/;
      $\overline{\boldsymbol{\delta}}_k = \overline{\overline{\boldsymbol{\delta}}_k + \alpha_\theta\Delta\overline{\boldsymbol{\delta}}_k}$ /*Decreases θ*/;
    $\boldsymbol{\gamma}'_k = \mu\lambda'_k\overline{\boldsymbol{\delta}}_k$ /*Update friction force*/;

$\mathbf{J}_t$ and $\mathbf{c}$ (Line 27 of Algorithm 4). Then, the method recomputes the preconditioner and the residual vector, and continues until it converges again for the linearized problem. This procedure continues until, for each active constraint, $0 < C_k(\mathbf{x}) < \epsilon_2$ holds. When $w = 0$, this procedure is exactly Newton's method in which the constraints are re-linearized followed by a solve of the updated Equation (5.20), see Section 6.7.2 for more details. Since Line 25 of Algorithm 4 updates the set of potential collisions, a collision-free state is eventually obtained. Additionally, when a contact point slides off a face or edge, the corresponding constraint must be removed from the system. The main advantage of Algorithm 4 is that updating $\mathbf{J}$, $\mathbf{J}_t$ and $\mathbf{c}$ does not require an update of the LCP matrix or Delassus operator. This allows us to efficiently solve the non-linear contact problem for large systems.

5.3.4 Preconditioning

In this section, we derive a preconditioner that is easy to compute, yet it significantly reduces the amount of iterations compared to a simple diagonal preconditioner. A common approach for improving the convergence rate of Krylov-subspace methods is to use a preconditioner matrix $\mathbf{C}^{-1}$. Many methods exist for approximating preconditioner matrices, such as LU and Cholesky factorizations [75]. Unfortunately, these methods assume that the matrix is positive definite, and therefore these preconditioners are not useful for our indefinite problem from Equation (5.20).

Algorithm 4 shows the pseudo-code of our preconditioned CR solver with inequality constraints. Matrix $\mathbf{B}$ in the solver represents the matrix from Equation (5.20) and $\mathbf{C}^{-1}$ the preconditioner matrix such that $\mathbf{C}^{-1} \approx \mathbf{B}^{-1}$. The blockwise inverse of $\mathbf{B}$ is

$$\mathbf{B}^{-1} = \begin{pmatrix} \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{L}^T\mathbf{S}^{-1}\mathbf{L}\mathbf{A}^{-1} & (\mathbf{S}^{-1}\mathbf{L}\mathbf{A}^{-1})^T \\ \mathbf{S}^{-1}\mathbf{L}\mathbf{A}^{-1} & -\mathbf{S}^{-1} \end{pmatrix}, \tag{5.22}$$

with $\mathbf{L}^T = \begin{pmatrix} \mathbf{J}^T & \mathbf{J}_t^T \end{pmatrix} \in \mathbb{R}^{3N_v \times 3N_c}$ the combined constraint matrix and $\mathbf{S} = \mathbf{L}\mathbf{A}^{-1}\mathbf{L}^T$. Next, we approximate matrices $\mathbf{A}$ and $\mathbf{S}$ by $\mathbf{A}_d = \text{diag}(\mathbf{A})$ and $\mathbf{S}_d = \text{diag}(\mathbf{Z}\mathbf{L}\mathbf{A}_d^{-1}\mathbf{L}^T)$, respectively, with $\text{diag}(\cdot)$ extracting the diagonal part of its matrix argument and $\mathbf{Z}$ a diagonal scaling matrix, see the following. Thus, the computations of $\mathbf{A}_d^{-1}$ and $\mathbf{S}_d^{-1}$ become trivial. Additionally, $\mathbf{S}_d^{-1}\mathbf{L}\mathbf{A}_d^{-1}$ is easy to evaluate and yields the same sparsity structure as that of $\mathbf{L}$. Based on this, we construct the following preconditioners:

Full: $\mathbf{C}^{-1}$ is obtained from Equation (5.22) by replacing $\mathbf{A}$ and $\mathbf{S}$ by $\mathbf{A}_d$ and $\mathbf{S}_d$, respectively;

Diag: $\mathbf{C}^{-1} = \begin{pmatrix} \mathbf{A}_d^{-1} & 0 \\ 0 & \mathbf{S}_d^{-1} \end{pmatrix}$.

When the preconditioner is applied, only those parts corresponding to active non-penetration and static friction constraints are considered. To construct the *Full* preconditioner, we first compute the quantities $\mathbf{S}_d^{-1}, \mathbf{S}_d^{-1}\mathbf{L}\mathbf{A}_d^{-1}, \mathbf{A}_d^{-1}\mathbf{L}^T$ for each constraint and use them when evaluating the blocks of the preconditioner, see Equation (5.22). Furthermore, when constraints are added, removed, or re-linearized, these precomputed quantities are also updated.

If a positive-definite preconditioner is used, like our Diag preconditioner, then the norm of the preconditioned residual will decrease monotonically when the states of the constraints are not updated. As we will discuss in Section 5.4.2, updating constraint states does not always result in a decrease of $\|\mathbf{r}\|$ (see Algorithm 4), but the approximation will always be closer to the new optimum. When an indefinite preconditioner is used, the evolution of $\|\mathbf{r}\|$ is more chaotic. More importantly, α can also be negative, which indicates that the current optimum for the preconditioned problem lies in the opposite direction of $\mathbf{p}$. Such an update does improve the preconditioned problem ($\|\mathbf{r}^T\mathbf{C}^{-1}\mathbf{r}\|$), although generally not the unpreconditioned problem ($\|\mathbf{r}^T\mathbf{r}\|$). Since the constraint updates are based on the unpreconditioned values $\mathbf{x}$ and $\boldsymbol{\lambda}$, an update is only performed if a step towards the unpreconditioned optimum is taken, i.e., when $\alpha > 0$.

Preconditioner Scaling

The Full preconditioner approximated by Equation (5.22) significantly improves the convergence rate of the method. Unfortunately, this preconditioner will have large deviations in the upper-left block when $\mathbf{B}$ becomes singular or ill conditioned due to (nearly) duplicate constraints in $\mathbf{L}$. When $\mathbf{L}$ is full rank, both inverse and pseudo-inverse of $\mathbf{B}$ are equal. By adding duplicate rows to $\mathbf{L}$, $\mathbf{B}$ and so $\mathbf{S}$ become singular. In our approximation, $\mathbf{S}_d$ does not become singular. Computing then the approximated block inverse for our preconditioner using $\mathbf{S}_d$ yields a change in the upper-left block of the preconditioner compared to the non-singular case. If this error becomes too large, the CR method converges significantly slower. In contrast, the pseudo-inverse of $\mathbf{B}$ does not introduce such a change, see Appendix A for more details on this behavior.

To keep the upper-left block invariant for duplicates in $\mathbf{L}$, it is sufficient to see that duplicate rows and columns *can* be added together. As a result, the diagonal values in $\mathbf{S}_d$ *would* have been scaled up by the number of duplicates for their corresponding constraint. Since we do not add duplicates together, the same scaling must be introduced to $\mathbf{S}_d$ using $\mathbf{Z}$, which contains, per constraint in $\mathbf{L}$, the number of similar or duplicate instances. When computing the upper-left block, duplicate constraints are added implicitly through the multiplication $\mathbf{L}^T\mathbf{S}_d^{-1}\mathbf{L}$. Since this multiplication also involves $\mathbf{S}_d^{-1}$ and thus $\mathbf{Z}^{-1}$, the difference introduced by the duplicates is compensated by $\mathbf{Z}^{-1}$.

Similar or identical constraints occur, for example, when an edge intersects other edges close to one of their shared vertices. The barycentric coordinate for the shared vertex is relatively large for all constraints. The other involved vertices have a small contribution, which makes the constraints almost identical to each other. For constraints involving deformable bodies, it is sufficient to compute, for each pair of constraints a and b, a weight based on the differences between their barycentric coordinates, i.e., $(1-|w_{a,i}-w_{b,i}|)$, with i a vertex id. The similarity of a pair is the product of these factors for each vertex that a and b have in common. Eventually, the total scaling factor for one constraint is the sum of all these products. For contacts between

rigid bodies (Figure 5.2c), many constraints *can* act on the same degrees of freedom of the rigid objects. Here the amount of duplicates, and thus the error in the preconditioner, can be potentially large. In this case, it is sufficient to scale all constraints working on the same two bodies by the total number of constraints between those bodies. This is in some way similar to the method of Tonge et al. [180]. Appendix A provides more details on computing this scaling matrix.

5.3.5 Optimizations

The convergence rate of the method described in Algorithm 4 is improved by allowing constraints to update their status at dynamic intervals. In other words, the closer the method is to convergence, the smaller the intervals are between successive constraint evaluations, see Line 2 of Algorithm 5. Eventually, constraints are evaluated every iteration when the residual is small. If we *always* allow constraints to update their status every iteration, in the worst case, the solver would need to recompute the residual vector as in Line 19 every iteration, instead of using the recurrence from Line 15. The performance would then be comparable to that of a *Gradient Descent* method, which can converge slowly due to *zigzagging*. Conversely, when constraints are *only* allowed to update their status at convergence, one has an *Active Set* method. In this case, the number of iterations between successive constraint evaluations *can be* relatively large. Due to this, the solver *can* enter a loop in which constraints continuously change their status. This *constraint cycling* drastically affects solver convergence and should be avoided.

By evaluating constraints at dynamic intervals, the solver responds fast enough to situations in which constraints *should* change states, although also slow enough for the solver to exploit the conjugacy of the residual vectors. A positive side effect of this strategy is that the number of residual re-evaluations (see Line 19) is significantly reduced. We have run a large number of experiments while tuning the relation between the computed interval and the residual norm. We have found that this strategy works very well for efficiently solving contact-response problems. This approach gave good results in all test-cases, but are not optimal for all cases.

Within our method, the possibility still exists that constraints change their status a large number of times, decreasing the convergence rate. To prevent this, one approach is to keep (at some point) constraints in a fixed state, as mentioned in Raghupathi and Faure [146]. Unfortunately this results in a violation of the conditions specified in Sections 5.3.1 and 5.3.2, which could introduce additional energy to the system. However, even in such cases, the current approximation is able to move toward the optimum, meaning that the method is still able to converge, see [114].

When a constraint is about to change (see Line 19 of Algorithm 5), the change is postponed if the change in distance is less than ϵ_2. In this case, the effect of performing the update is negligible. This strategy also eliminates undesired oscillations of constraint states (constraint cycling) that have no effect on the final result. This usually occurs when the solution for a particular constraint is located close to a discontinuity in the derivatives, i.e., when a vertex is about to slide or when two entities are barely touching.

Each time constraints change states, vectors $\mathbf{r}, \mathbf{p}$ and $\mathbf{Bp}$ must be recomputed to restore the conjugacy of the residual vector. Since the computation of the residual requires a full multiplication with matrix $\mathbf{B}$, this is not very efficient. However, since each residual vector can be decomposed as $\mathbf{r} = \mathbf{r}_u + \mathbf{r}_c$, with $\mathbf{r}_u = \mathbf{b} - \mathbf{Av}$ the unconstrained residual and $\mathbf{r}_c$ the remaining constraint residual, a multiplication with matrix $\mathbf{A}$ can therefore be completely omitted since

$\mathbf{r}_u$ is not affected by a state change. Indeed, only $\mathbf{r}_c$ needs to be recomputed, which significantly improves performance. Please note that this decomposition requires that other intermediate vectors and computations are decomposed similarly.

5.4 Convergence

In this section, we discuss the convergence behavior of our method, given the criteria described in Section 5.3 for switching constraints. First, we show that a minimal residual, in combination with all constraints satisfied, implies that a global solution is found. Next, we show that successive updates of constraints lead to a state in which all constraints are satisfied. Finally, we show that when all constraints are satisfied but the residual does not reach the minimum, its gradient will reach the minimum instead.

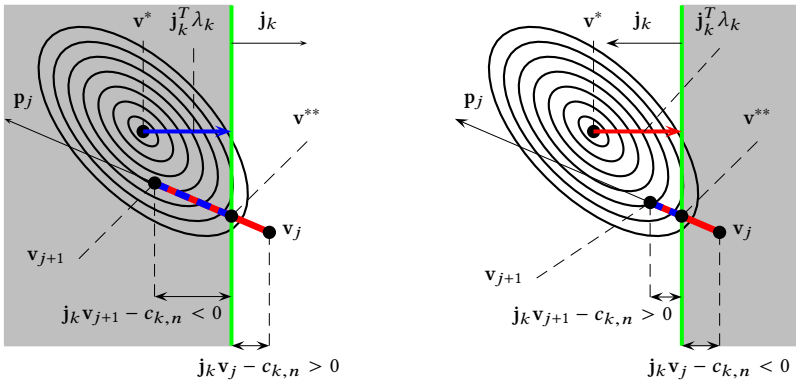
5.4.1 Global Convergence

The Delassus operator ($\mathbf{L}\mathbf{A}^{-1}\mathbf{L}^T$), where $\mathbf{L}^T = \begin{pmatrix} \mathbf{J}^T & \mathbf{J}_t^T \end{pmatrix}$, describes, for *each pair of constraints* that share the *same* object in the inverted generalized mass matrix $\mathbf{A}^{-1}$, their influence on each other. The *inverted* Delassus operator can therefore be seen as the "solution" to the contact problem and describes the influence of *each* pair of constraints onto each other, no matter how many objects and constraints are present in the chain between the two constraints. Hence, this solution is a global one. In our method, the Delassus operator is not used. However, the CR method minimizes $\|\mathbf{B}\mathbf{y} - \mathbf{d}\|$, with $\mathbf{y} = \mathbf{B}^{-1}\mathbf{d}$ its minimum. Given the blockwise inverse of $\mathbf{B}$, see Equation (5.22), minimizing $\|\mathbf{B}\mathbf{y} - \mathbf{d}\|$ also implicitly applies the Delassus operator. This is equivalent to solving both $\boldsymbol{\lambda}$ and $\boldsymbol{\gamma}$ through the Delassus operator, and applying these forces to the bodies to obtain the unknown velocity $\mathbf{v}$.

Within our method, the states of the constraints are actively switched when they do not agree with the currently computed $\mathbf{v}$, $\boldsymbol{\gamma}$ and $\boldsymbol{\lambda}$. When some constraints change states, the method *must* reset vectors $\mathbf{r}$, $\mathbf{p}$ and so $\mathbf{B}\mathbf{p}$, i.e., a state change influences the residual energy of the system. This reset allows the method to search in a direction that minimizes the energy according to the new state. Since all constraints and the deformation problem in sub-matrix $\mathbf{A}$ contribute to the residual, a sufficiently small residual should imply that the problem is solved. However, the solution can only be the global minimizer if all constraints are in their final state at convergence. If not, then constraint states are changed, the residual will be influenced, and thus the current solution cannot be a global minimizer for the problem. The method then continues until all constraints are satisfied.

Assuming that the constraint states are known and only active constraints are considered, a global minimum exists if the constraints are convex and $\mathbf{A}$ and the second derivatives of the constraints (which vanish for linear constraints) are symmetric and positive definite, see [114]. At this global minimum all multipliers of all active constraints are strictly positive (or not clamped in case of friction constraints) and their distances are zero. When also inactive constraints are considered, their multipliers are zero and their distances are strictly positive. Hence, the complementarity conditions are satisfied, and the complementarity error vanishes, see Section 5.6.3. Let us define such a residual that, once minimized, solves Equation (5.20)– that is,

$$\mathbf{r}_g = \begin{pmatrix} \mathbf{A}\mathbf{v} - \mathbf{b} + \mathbf{J}_A^T(\boldsymbol{\lambda})^+ + \mathbf{J}_{t,S}^T(\boldsymbol{\gamma})^{\mu\lambda} + \mathbf{J}_{t,K}^T(\boldsymbol{\gamma}')^{\mu\lambda} \\ (\mathbf{J}_A\mathbf{v} - \mathbf{c}_n) + (\mathbf{J}_I\mathbf{v} - \mathbf{c}_n)^- \\ \overline{\boldsymbol{\gamma}}^T(\mathbf{J}_{t,S}\mathbf{v} - \mathbf{c}_t) + \left(\overline{\boldsymbol{\delta}}^T(\mathbf{J}_{t,K}\mathbf{v} - \mathbf{c}_t)\right)^- \end{pmatrix}, \tag{5.23}$$



(a) Constraint activation **(b)** Constraint deactivation

Figure 5.5: Activation and deactivation of a constraint.

with $\mathbf{J}_A$ and $\mathbf{J}_I$ corresponding to all active and inactive non-penetration constraints; similarly, $\mathbf{J}_{t,S}$ and $\mathbf{J}_{t,K}$ correspond to all static and kinetic friction constraints, respectively. Furthermore, the multipliers and distances are clamped by operators $(\cdot)^-, (\cdot)^+, (\cdot)^{\mu\lambda}$ between $(-\infty, 0), (0, \infty)$ and $(-\mu\lambda,\ \mu\lambda)$, respectively. At the global minimum, $\|\mathbf{r}_g\| = 0 \Rightarrow \|\mathbf{r}\| = 0$. Conversely, $\|\mathbf{r}\| = 0 \Rightarrow \|\mathbf{r}_g\| = 0$ if and only if no constraints are violated. Unfortunately, the final constraint states for which this relation holds are not known *a priori*.

5.4.2 Constraint Updates

In the following, we reason about the implications of updating constraint states, as illustrated by Figure 5.5. We consider only non-penetration and kinetic friction constraints, as static friction can be treated similar to non-penetration constraints.

Activation of non-penetration constraints Figure 5.5a shows the activation of an inactive constraint C_k at iteration j. Its distance $\mathbf{j}_k\mathbf{v}_j - c_{k,n}$ is positive, thus not contributing to $\mathbf{r}_g$ nor to $\mathbf{r}$. By minimizing $\|\mathbf{r}\|$ in the direction of $\mathbf{p}_j$, a new approximation $\mathbf{v}_{j+1}$ is obtained, which is closer to the current minimum $\mathbf{v}^*$ [113]. If distance $\mathbf{j}_k\mathbf{v}_{j+1} - c_{k,n}$ becomes negative, the constraint is activated. Due to this active constraint, the *new* minimum $\mathbf{v}^{**}$ along $\mathbf{p}_j$ now lies on the (green) plane defined by the constraint, which intersects the path between $\mathbf{v}_j$ and $\mathbf{v}_{j+1}$. Since $\mathbf{v}_{j+1}$ is now closer to the new optimum $\mathbf{v}^{**}$ along $\mathbf{p}_j$ than $\mathbf{v}_j$ was to the previous optimum, this constraint switch results in an improvement of the approximation. However, the distance between $\mathbf{v}_{j+1}$ and $\mathbf{v}^{**}$ appears as an additional residual term via $\mathbf{j}_k\mathbf{v}_{j+1} - c_{k,n} = \mathbf{j}_k(\mathbf{v}_{j+1} - \mathbf{v}^{**})$, through the first term in the second row of Equation (5.23). Depending on the used preconditioner, $\|\mathbf{r}\|$ might or might not decrease. However, since approximation $\mathbf{v}_{j+1}$ will now be closer to the *new* optimum, this step including the constraint activation is an improvement.

Deactivation of non-penetration constraints Figure 5.5b shows the deactivation of a constraint C_k. At iteration j, distance $\mathbf{j}_k\mathbf{v}_j - c_{k,n}$ is negative. Next, a new step in direction $\mathbf{p}_j$ is performed, with $\mathbf{v}_{j+1}$ the new approximation. When both $\lambda_k < 0$ and $\mathbf{j}_k\mathbf{v}_{j+1} - c_{k,n} > 0$ hold, the constraint must be deactivated. The negative multiplier in combination with a positive distance indicates that the minimum lies on the positive side of the constraint and not on its plane, and hence the constraint is working in the wrong direction and can be deactivated. Since $\mathbf{j}_k\mathbf{v} - c_{k,n}$ is zero somewhere between $\mathbf{v}_j$ and $\mathbf{v}_{j+1}$, and the new minimum $\mathbf{v}^*$ lies on the

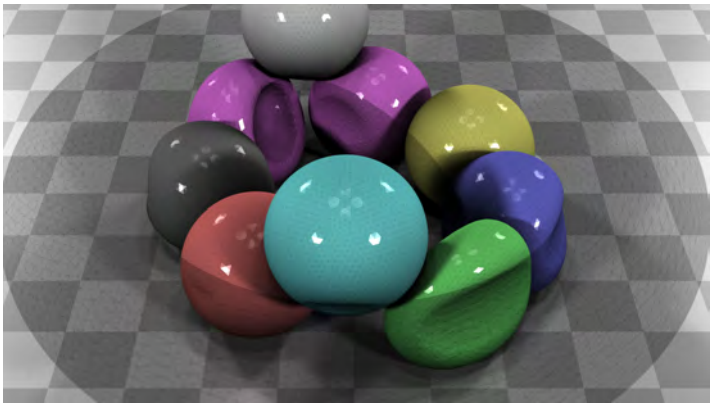


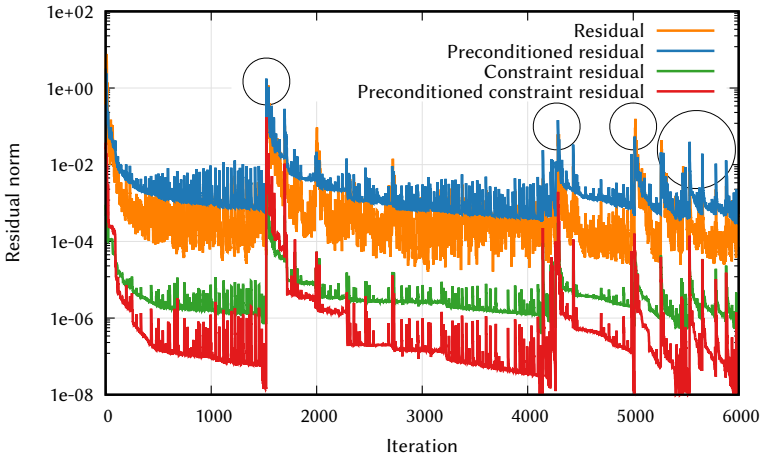
Figure 5.6: Test case in which nine deformable spheres roll between a (transparent) rigid torus and a rotating spherical object. Figure 5.7 depicts the residual norms at a certain simulation iteration.

positive side of the constraint, $\mathbf{v}_{j+1}$ is closer to $\mathbf{v}^*$ than $\mathbf{v}_j$ was. Hence, the state is improved. Since the positive distance $\mathbf{j}_k\mathbf{v}_{j+1} - c_{k,n}$ and negative multiplier are clamped in Equation (5.23), $\|\mathbf{r}_g\|$ decreases. The actual $\|\mathbf{r}\|$ again might or might not decrease since it is based on active constraints and possible negative multipliers.

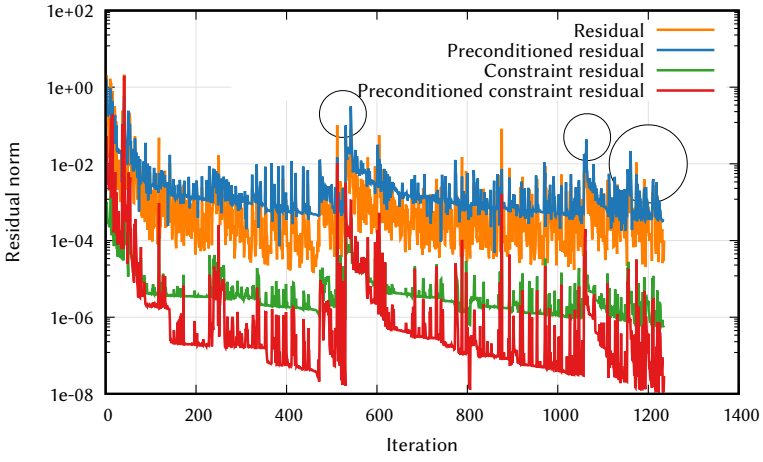
Delayed evaluation When constraints are evaluated and switched at fixed intervals, n optimization steps are performed that move the current approximation closer to the current optimum. This can also be considered as a single optimization step. If constraints do change their state after n steps, a new optimum is obtained and the method proceeds in that direction. This only demonstrates that the method eventually converges to an optimum. The choice of n largely affects the performance of the method. In general, the larger n is, the larger is the possibility that the method *overshoots*, which results in a smaller overall improvement per n optimization steps. Therefore, we keep n large for large residuals, and decrease n with the residual norm, see Line 2 of Algorithm 5.

Kinetic friction constraint update The update of a kinetic friction constraint changes the magnitude λ'_k and direction $\overline{\boldsymbol{\delta}_k}$ of the kinetic friction force. Since the kinetic friction force is placed on the right-hand side of Equation (5.20), any update of $\overline{\boldsymbol{\delta}_k}$ and λ'_k, results in a change of $\|\mathbf{r}\|$. When directions $\overline{\mathbf{j}_{k,t}\mathbf{v} - \mathbf{c}_{k,t}}$ and $\overline{\boldsymbol{\delta}_k}$ are misaligned, $\overline{\boldsymbol{\delta}_k}$ is updated as described in Section 5.3.2. Since each updated $\overline{\boldsymbol{\delta}_k}$ is somewhere between the previous $\overline{\boldsymbol{\delta}_k}$ and $\overline{\mathbf{j}_{k,t}\mathbf{v} - \mathbf{c}_{k,t}}$ due to *moving average* $\boldsymbol{\delta}_{k,a}$, the angle between these vectors becomes smaller, see Figure 5.4. Due to this, also the change in $\mathbf{j}_{k,t}^T\overline{\boldsymbol{\delta}_k}\mu\lambda'$ (the kinetic friction force; see the right-hand side of Equation (5.20)) becomes smaller. Therefore, successive increases of $\|\mathbf{r}\|$ due to these updates (through the last term in the first row of Equation (5.23)) become smaller and eventually vanish when $\overline{\boldsymbol{\delta}_k}$ and $\overline{\mathbf{j}_{k,t}\mathbf{v} - \mathbf{c}_{k,t}}$ are parallel.

Convergence plots Figure 5.7 depicts the convergence behavior of our method for both preconditioners for the simulation time-step shown in Figure 5.6. Figure 5.7a shows the residual norms obtained with the Diag preconditioner. The residual shows a similar behavior compared to the *preconditioned residual* and *(preconditioned) constraint residuals*. Although the residual does not provide much information about the convergence, the preconditioned residual $\|\mathbf{C}^{-1/2}\mathbf{r}\|$ shows a decreasing behavior on the lower bound. A state change of a constraint



(a) Diag preconditioner



(b) Full preconditioner

Figure 5.7: Plots of several residual norms for preconditioners Diag and Full of the simulation time-step shown in Figure 5.6. The non-linear updates of the constraints are indicated by circles. The preconditioned residual norm is computed as $\|\mathbf{C}^{-1/2}\mathbf{r}\|$, the constraint residual norm as $\|\mathbf{U}\mathbf{r}\|_\infty$, and the preconditioned constraint residual norm as $\|\mathbf{U}\mathbf{C}^{-1/2}\mathbf{r}\|_\infty$. Due to the infinity norm, the shown (preconditioned) constraint residuals represent an upper bound of the actual constraint residual.

results in an increase of the residuals. After continuing for a few iterations, the preconditioned residual is (in most cases) smaller compared to its norm before the state update. Additionally, the (preconditioned) constraint residual shows an overall decreasing trend on the upper bound of the error, indicating that, in general, the update of the constraint states eventually results in a better approximation. This behavior also explains the effect of the optimization discussed in Section 5.3.5 in which the constraints are evaluated at a certain rate depending on the residual norm. This allows the method proceed towards the new optimum for a few iterations until constraints are re-evaluated. The large peaks in the plots correspond to the non-linear updates of the constraints. The intervals between successive non-linear updates become smaller as the approximation approaches the final solution, see Section 5.3.3. Figure 5.7b shows the

(preconditioned) residuals obtained with the Full preconditioner, which shows less correspondence with the (preconditioned) constraint residuals. However, the constraint residuals show the same behavior compared to the Diag preconditioned case, albeit the Full preconditioned case results in a much faster convergence. The constraint tolerance for both experiments were $\epsilon_2 = 10^{-6}$ and a relative tolerance for the residual of $\epsilon_1 = 10^{-5}$.

5.4.3 Local Minima

Whenever the problem is over-constrained, no unique solution exists. Hence, the problem has local minima. At such a local minimum, the gradient vanishes. Since the CR method minimizes $\|\mathbf{C}^{-1/2}\mathbf{r}\|^2 = |\mathbf{r}^T\mathbf{C}^{-1}\mathbf{r}|$, the gradient of the minimized function ($2\mathbf{B}\mathbf{C}^{-1}\mathbf{r}$) becomes zero at a local minimum. When this happens, α becomes zero in Line 9 of Algorithm 4, so the approximation cannot be further improved. Next, the computation of β will result in a division by zero, causing the method to break down. To prevent this, the norm of the gradient function (second test in Line 22 of Algorithm 4) is also inspected, as proposed by Hayami [85]. If this norm drops below ϵ_1, the method has computed a least-square approximation for the current local minimum. This allows the method to perform the checks in Lines 23 to 25 and to update the problem.

5.5 System Overview

Non-penetration and friction constraints act on subsets of vertices of the colliding models. To constrain the movement globally, all *vertex-face* and *edge-edge* collision pairs should be detected, collected, and assembled in (global) matrices $\mathbf{J}$ and $\mathbf{J}_t$. Since the solution of Equation (5.20) is collision free, a *standard* collision detection system would not find any collisions in the beginning of each simulation time-step. This suggests that all collisions would have to be detected inside the internal loop of the CR method. However, doing so would be prohibitively expensive in terms of computations.

Algorithm 4 gives the overall pseudo-code of our collision handling system, which is further described in the following sections and in detail in Chapter 6. Within our approach, we search once for all possible *candidate* collision pairs at the beginning of each simulation time-step, see Line 5 of Algorithm 4. Then, all existing constraints are updated. When the solver converges, a collision check is performed, see Line 25. If no new collisions are found, then all active constraints are checked if their configuration and state agree with the actual geometry and velocity; if necessary, constraints are re-linearized or removed from the system, see Line 27. Finally, if the system is not updated, the state of the simulation is guaranteed to be collision free, and the error made by each constraint is guaranteed to be less than the desired tolerance.

5.5.1 Collision Detection

When the simulation is initialized, a *Bounding Volume Hierarchy* (BVH) is constructed, containing axis-aligned bounding volumes enclosing all surface faces (triangles) of the simulated bodies (which coincides with the simulation meshes; see Line 2 of Algorithm 4); our hierarchical data structure is a binary tree. The BVH is used to quickly find potential face-face collisions by testing for intersection of their bounding volumes. Since deformable objects can have self-collisions, all faces of an object must also be tested for collisions with all other faces of the same object. Since bounding volumes of adjacent faces will intersect, one approach to discard false positives is to terminate the tree traversal when both faces belong to the same

(self-collision free) surface patch, see [188]. When there is a possibility that both faces can collide at the end of the time-step, both faces must be included in further collision checks in order to guarantee a collision-free state at the end of the time-step.

To detect nearby faces of a given face, we extend each bounding box by $3\Delta t\mathbf{v}$ in the directions of its vertex velocities. For each face represented in the BVH, a set is created that stores all nearby and potentially colliding faces. All faces whose *extended bounding volumes* intersect the one of the current face are selected by traversing the tree. These face-face pairs are then used to create so-called candidate lists. The candidate lists contain, for each vertex/edge, a list of all nearby faces/edges (the candidates) that can potentially collide; duplicate edge-edge pairs are discarded.

5.5.2 Raw Collision Detection

At convergence, all candidate pairs (vertex-face and edge-edge) are checked for collisions given the current velocity, see (Line 25). First, each collision pair is updated (time integrated) using the current velocity. If the pair has intersected, a root-finding method, such as the Brent-Dekker method [145], is used to find the collision point, contact normal, and barycentric coordinates, given the current and initial geometry state (at the beginning of the time-step). Then, the corresponding non-penetration constraint is initialized, activated, and stored in matrix $\mathbf{J}$, see Section 5.2.3. Likewise, the corresponding friction constraints are initialized and stored in $\mathbf{J}_t$. Since this collision check is performed when the solver converges, new constraints are added as long as the state is not collision free. This is similar to the Constraint Manifold Refinement (CMR) method of Otaduy et al. [138], to ensure that no collisions/constraints are missed. Please note that the collision check relies on the *candidate* pairs, so that (multiple) tree traversals are avoided. In some exceptional cases, a vertex can move out of its extended bounding volume such that collisions might be missed. If such a case is detected at convergence (Line 25), using a simple vertex-in-box check, we need to search locally for additional missed collisions. This search is done by first computing the extended bounding box and rechecking the BVH for collisions with the updated box. If any new box-box intersections are found, the corresponding candidate lists are updated, and a collision check is performed on the updated candidate lists. The overhead of this additional check is relatively small – up to 5 % of the total collision detection time.

5.6 Results

In this section we validate our approach by providing quantitative and qualitative results, and comparing these to those of existing methods. An overview of example results is shown in Figure 5.9.

5.6.1 Varying Parameters

The plot in Figure 5.8 shows the average number of solver iterations during the first second of the experiment shown in Figure 5.9e for different values of Δt. In general, a larger Δt requires more iterations per simulation step, but fewer simulation steps are needed for advancing the simulation to a given timestamp. However, in all cases, the total amount of solver iterations is comparable. We observed that simulations with smaller time-steps reveal subtle motions of objects, but require more iterations; however, such details are not visible when larger time-steps are used.

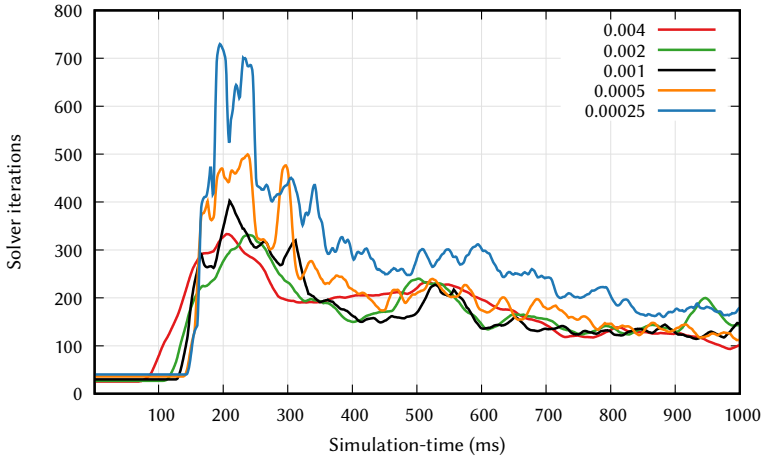


Figure 5.8: Number of solver iterations per simulation step for different values of $\Delta t(s)$ and the experiment shown in Figure 5.9e, with $E = 5 \times 10^6 Pa$.

We have also changed the stiffness parameter E, while keeping Δt fixed at 0.001 seconds. A smaller stiffness parameter generally results in faster convergence for unconstrained problems. When the objects are less stiff, we observed that the method requires more Newton steps due to larger deformations, and this often requires more iterations. For instance, the setup shown in Figure 5.9e is simulated using varying stiffness parameters starting from $5 \times 10^6 Pa$ to $3.125 \times 10^5 Pa$ by dividing E by two for every simulation. Only for the case $E = 3.125 \times 10^5 Pa$ was the total amount of iterations about 1.5 times larger than for $E = 5 \times 10^6 Pa$. In all other cases, these numbers were similar. Please note that it is difficult to quantify these results because a different stiffness parameter results in a different behavior of the material (and simulation). In case of a collision, the time that objects have contact is larger for flexible objects, which influences the behavior of the method.

5.6.2 Comparative Results

In this section, we compare various versions of our method against ICA [138] and SP [101]. For this comparison we have simulated three setups, with varying difficulty degrees, see Figures 5.9b, 5.9e and 5.9h. All methods used double-precision arithmetic and converged to the same absolute error ($\epsilon_1 = 5 \times 10^{-5}$). Additionally, our method uses a threshold for updating vectors $\overline{\boldsymbol{\delta}}_k$ of $\epsilon_\theta = 1.2$ degrees and used a constraint tolerance / safety distance $\epsilon_2 = 5 \times 10^{-6}$. All objects had the following material properties: $\nu = 0.2$ (Poisson's ratio), $E = 5 \times 10^5 Pa$ (Young's modulus), and $\rho = 1000 kg/m^3$ (density). Furthermore, a gravitational acceleration of $g = 9.81 m/s^2$ was used. The FEM machinery is based on co-rotated finite elements [124] and no additional damping was used. Our test machine was equipped with a 4 GHz AMD FX-8350 CPU (using one core) and 16 GB of RAM memory. All methods use a warm start (reuse values from the previous simulation iteration), the same collision detection system, the same optimized routines for vector-vector and matrix-vector operations, and the same tolerances/precision in an attempt to make a fair comparison.

The first experiment from Figure 5.9h contains 120 randomly placed elastic rings, simulated with a time-step $\Delta t = 0.001 s$. The total number of contact points grew to over 1500. The second experiment (Figure 5.9e) contains 140 objects, which are aligned in a pyramid. For Δt,

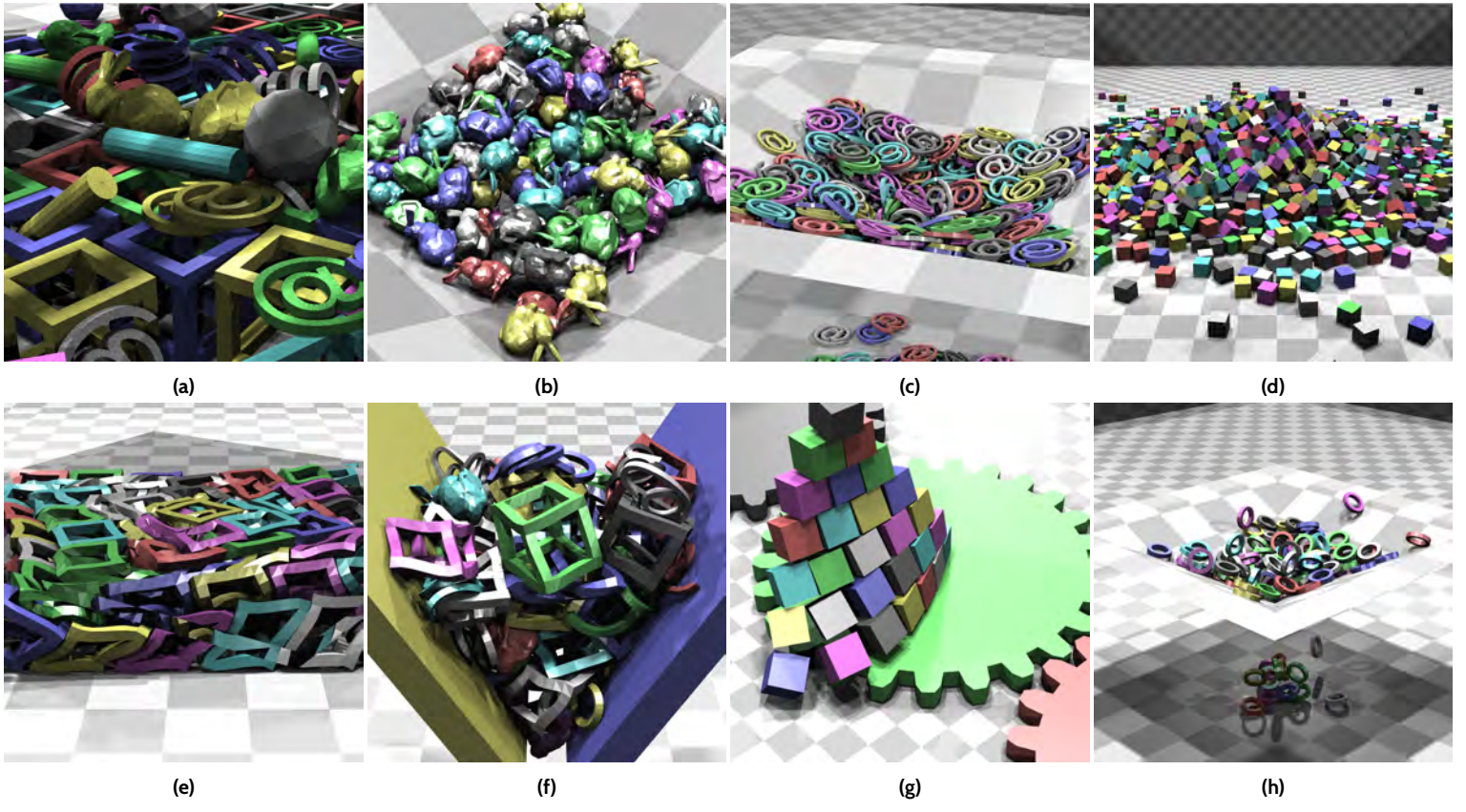


Figure 5.9: Simulations of multiple elastic objects. The collision response is efficiently computed through our method that can handle complex situations, such as: tens of thousands of constraints, large impact forces and complex interactions, see text.

we used 0.0005. The experiment shown in Figure 5.9b uses complex objects in which many internal elements exist. For this experiment, we randomly placed 120 bunnies with a Young's modulus E of $5 \times 10^6 Pa$ and simulated with $\Delta t = 0.001s$. The friction coefficient was set to 0.5 for all simulations. Table 5.1 shows, for each method, the average (and median) time spent on collision detection, contact resolution, solving the whole contact problem, and the average number of iterations performed by all methods. Furthermore, additional details about the simulations are provided. The application of the Full preconditioner required about 15% of the total time, whereas its construction time is negligible – less than 0.1%.

Hyper-Elastic Materials

In addition to linear FEM, we also made a comparison using highly deformable, hyper-elastic models. The stiffness of hyper-elastic materials changes with the amount of compression. For a neo-Hookean material, the stress will approach infinity when the volume of the object approaches zero. Due to this, numerical methods have to deal with large contact forces. Unfortunately, directly using these materials *can* cause problems, as small changes in deformation *could* result in excessively large forces. To deal with this, the underlying energy density function is linearly extrapolated, as described in [168]. Furthermore, since more deformation is involved, all methods need to re-linearize constraints more often. For ICA, this implies that the LCP matrix also needs to be re-computed. On average, constraints are re-linearized 5 times per time-step, with peaks approaching 10 times. In our test case, see Figure 5.1, a neo-Hookean energy density function is used and extrapolated when the volume of an element drops below 70% of its initial volume. Furthermore, the Lamé parameters were set using Young's modulus of elasticity $E = 5 \times 10^5 Pa$ and Poisson ratio $\nu = 0.2$. An Armadillo consisting of roughly $200K$ degrees of freedom is pulled through a set of rotating cylinders, resulting in large compressions, deformation, and contact forces. The total number of iterations and total computation time are presented in Figure 5.11. For all methods, the total number of iterations and computation time increase with the amount of compression. This test also faces the underlying collision detection with numerous challenging cases, which are all resolved correctly.

Comparison to ICA

We compared our approach to the elegant and flexible ICA method of Otaduy et al. [138]. We chose this method for comparison because it constructs and solves complete LCPs (instead of an approximation [50]) and takes the full generalized mass matrix into account when computing collision responses. Both of these aspects are essential when dealing with FEM-based deformable bodies. Furthermore, the computation of non-penetration and friction multipliers are combined, making the method very efficient. Please note that we needed to compute the unconstrained velocity with a slightly lower tolerance in order to guarantee that the final approximation is accurate enough.

In Figure 5.10 and Table 5.1, a comparison is shown between our method and ICA. We compare the average numbers of iterations and average time per time-step. ICA performs, per iteration, up to two Sparse Matrix-Vector (SpMVs) multiplications: one for performing a (block) Gauss-Seidel step in the outer loop and one (block) Jacobi step for setting up the right-hand side of the LCPs. Additionally, the iterations spent on computing the unconstrained velocity are considered. We did not explicitly count the number of iterations used for solving the LCPs, as these matrices generally are very sparse, and the LCPs converge quite quickly.

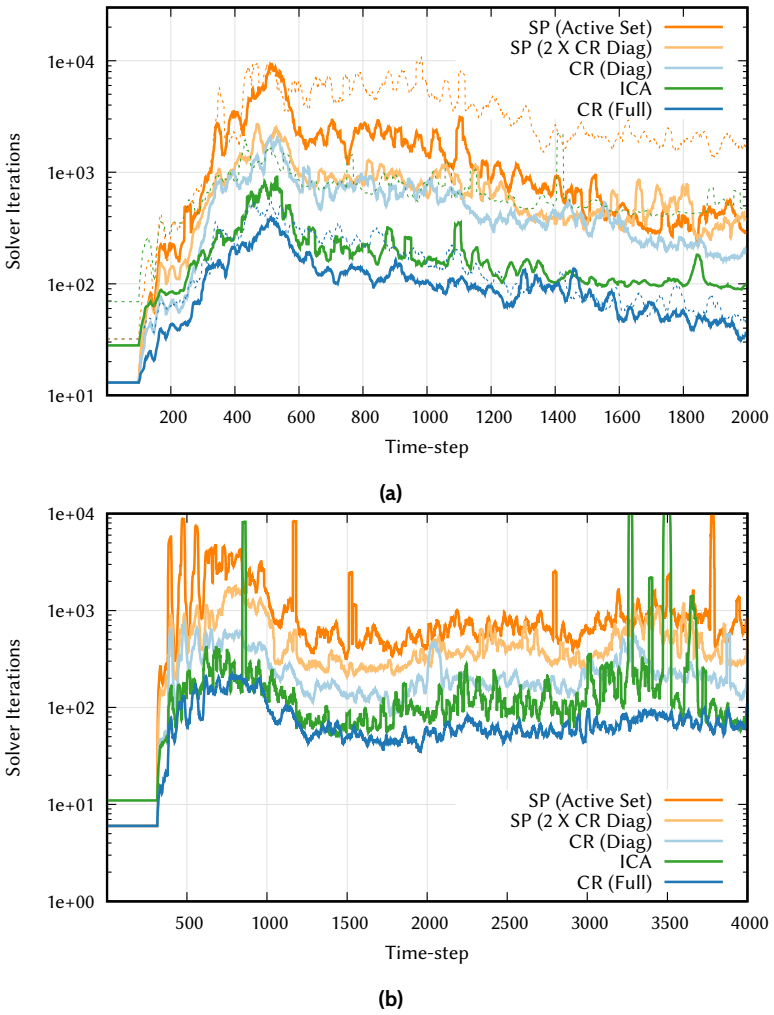


Figure 5.10: Number of solver iterations per time-step for different methods, performing the simulation shown in Figure 5.9h (left) and Figure 5.9e (right). CR (Full) represents our complete method including the Full preconditioner. CR (Diag) represents our method with the diagonal preconditioner, in which the friction cone is replaced by a friction pyramid. SP (Active Set) is an implementation of the SP method, in which the QP problems are solved using two CR-based Active Set methods. SP (2xCR) represents an implementation of the SP method in which the QP problems are solved by a modified version of our diagonally preconditioned CR method. All methods converged to the same absolute error $\epsilon_1 = 5 \times 10^{-5}$, and used the same constraint tolerance $\epsilon_2 = 10^{-6}$ and double-precision arithmetic. The dashed plots represent test case "Rings*", in which the stiffness was increased.

For both methods, we have observed that computing the multipliers converged quickly compared to the global problem. Within ICA, a small LCP is solved per iteration, which converges very rapidly. In our approach, the convergence of the multipliers is slower because each iteration improves the multipliers by just a bit, whereas ICA actually solves an LCP. Nevertheless, the convergence rate of the contact problem is mostly bounded (especially for deformable bodies) by the convergence of the velocity (the deformation part) rather than that of the multipliers. For more complex cases, like the Pyramid setup, a significant amount of time is spent

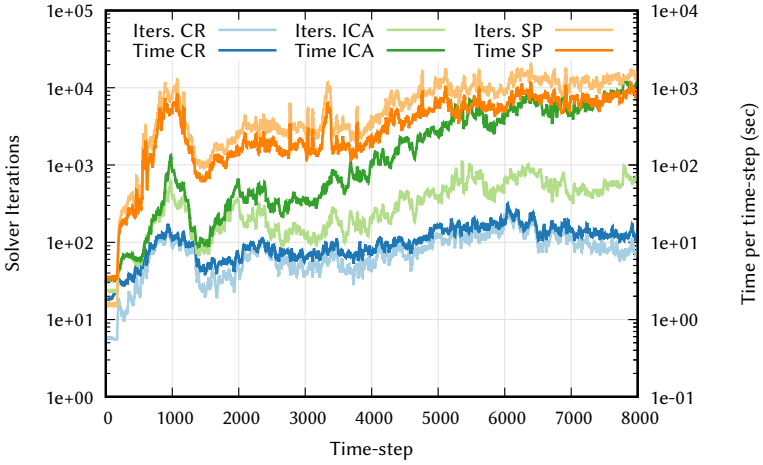


Figure 5.11: Total run time and solver iterations for a hyper-elastic object, see Figure 5.1 and case 'Armadillo' in Table 5.1. The total running-time includes all tasks performed during the process, including collision detection.

on the LCPs. In the hyper-elastic test case (Figures 5.1 and 5.11), ICA tends to perform slower over time. First, per time-step more re-computations of the LCPs are required due to the non-linear motion and deformation. Second, due to the computation of the unconstrained velocity (which releases all stress), more collisions are detected, and more constraints than needed are generated. This results in a much larger overhead spent on constructing and solving the LCPs. Due to the increased stiffness of the problem, the PGS and GS methods converge slower as well. Finally, since the test case in Figure 5.1 also involves rigid bodies, the constructed LCP contains dense regions, which increases the time per PGS iteration. In Section 5.7 we shall further discuss the differences between both methods.

Comparison to Staggered Projections

We also compared our approach to SP [101]. The SP method solves the contact problem by first computing the unconstrained velocity, which is then corrected by separately computing impulses of the non-penetration and friction constraints. This correction procedure first solves a QP problem for non-penetration constraints, provided that some estimate of the friction impulses is available, followed by a QP solve of the friction problem using non-penetration impulse estimates. This process continues until an optimum is found that minimizes both the friction and non-penetration sub-problems. Since a FEM-based discretization results in a large yet sparse matrix $\mathbf{A}$, we have used a diagonally preconditioned MINRES/CR based Active Set approach for solving the QP problems rather than a dense solver, as in the original work [101]. Each individual solve uses a "warm start". The relative error of the friction impulses is computed by solving a linear problem using the CG method. Since the SP method does not guarantee a collision-free state, we have replaced the CR solver within our framework from Algorithm 4 with the SP method such that at convergence, additional constraints can be added and corrected.

In Figure 5.10 and Table 5.1 a comparison between our approach and the SP method is shown. Since SP solves two QP problems per iteration, the total number of QP solver iterations per simulation step is counted. We have also used a modified version of our CR approach as a QP

Method	Resolution (s)	Collision (s)	Time (s)	Iterations	Constraints	Objects	Elements	Faces	Setup	$\Delta t(s)$	$E(Pa)$
SP (Active Set)	5.6 (3.2)	0.68 (0.72)	6.2 (3.9)	1546 (780)	2385 (2919)	120	12960	17280	Rings	0.001	5×10^5
SP (2xCR Diag)	5.0 (4.4)	0.68 (0.72)	5.7 (5.2)	814 (728)	2340 (2886)	120	12960	17280	Rings	0.001	5×10^5
CR (Diag)	2.8 (2.3)	1.1 (1.0)	3.9 (3.5)	451 (340)	2435 (2976)	120	12960	17280	Rings	0.001	5×10^5
ICA	0.8 (0.7)	0.85 (0.85)	1.6 (1.5)	180 (134)	2846 (3180)	120	12960	17280	Rings	0.001	5×10^5
CR (Full)	0.52 (0.44)	0.94 (0.96)	1.46 (1.43)	97 (76)	2280 (2466)	120	12960	17280	Rings	0.001	5×10^5
SP (Active Set)	20.8 (11.75)	2.9 (2.9)	23.7 (14.9)	1088 (604)	8647 (9651)	140	60480	73920	Pyramid	0.0005	5×10^5
SP (2xCR Diag)	15.6 (11.5)	3.7 (3.8)	19.3 (15.4)	463 (332)	8901 (9939)	140	60480	73920	Pyramid	0.0005	5×10^5
CR (Diag)	7.6 (6.1)	4.4 (4.3)	12.0 (10.8)	229 (179)	8871 (9877)	140	60480	73920	Pyramid	0.0005	5×10^5
ICA	14.2 (3.8)	3.77 (3.61)	17.95 (7.77)	558 (75)	8734 (9765)	140	60480	73920	Pyramid	0.0005	5×10^5
CR (Full)	2.3 (1.9)	3.77 (3.70)	6.0 (5.6)	69 (59)	8565 (9546)	140	60480	73920	Pyramid	0.0005	5×10^5
SP (Active Set)	20.2 (15.65)	0.9 (1.06)	21.7 (16.7)	3373 (2423)	2290 (2892)	120	12960	17280	Rings*	0.001	5×10^6
ICA	2.1 (1.7)	0.78 (0.75)	2.7 (2.5)	655 (550)	2439 (3217)	120	12960	17280	Rings*	0.001	5×10^6
CR (Full)	0.8 (0.6)	0.84 (0.80)	1.6 (1.40)	154 (103)	1898 (2232)	120	12960	17280	Rings*	0.001	5×10^6
ICA	174 (160)	4.95 (4.92)	179 (165)	7206 (6261)	876 (1023)	120	152400	88560	Bunnies	0.001	5×10^6
CR (Full)	31.3 (17.8)	4.89 (4.84)	36.2 (22.7)	900 (534)	700 (793)	120	152400	88560	Bunnies	0.001	5×10^6
SP (Active Set)	395 (310)	6.8 (4.5)	404 (317)	6839 (5393)	4335 (4101)	1	385739	20978	Armadillo	0.0005	non-linear
ICA	255 (95)	4.15 (3.81)	259 (99)	344 (286)	4170 (3705)	1	385739	20978	Armadillo	0.0005	non-linear
CR (Full)	6.55 (5.44)	4.14 (3.6)	10.71 (9.65)	81 (71)	4350 (3633)	1	385739	20978	Armadillo	0.0005	non-linear
CR (Full)	19.38 (18.88)	3.54 (3.32)	22.9 (22.4)	480 (463)	4044 (4405)	9	115659	46080	Bearing	0.001	non-linear

Table 5.1: Comparison of our method (CR (Full)) against various instances of Staggered Projections (with Active Set solvers or a modified version of our approach) and ICA. The numbers represent mean (and median) values over a fixed stretch of time-steps. The number of active constraints equals three times the number of contacts.

solver for SP. In this solver, the kinetic friction treatment, described in Section 5.3.2, is replaced by one in which a friction pyramid is used. This is because our approximation of the friction cone is not suitable for use in the SP method.

The main reason for the difference in the number of solver iterations is that SP solves two QP problems, which both influence each other. In our approach, both non-penetration and friction impulses are approximated, and constraints are allowed to update their state more frequently. Due to this, our approach switches the state of a constraint when the current approximation starts to deviate from the (unknown) global minimum. Since SP keeps a part of the constraints fixed, while updating the other set, it can, for instance, omit changing the state of a non-penetration constraint while solving for the friction impulses. This *can* result in a sequence of approximations that is deviating from the (unknown) global minimum for a longer stretch of iterations. In some exceptional cases we have observed a slow convergence due to cycling between non-penetration and friction constraints. The total speedup of our method compared to the Active Set-based SP method is roughly between 7 and 12 times. Replacing the Active Set solvers by modified versions of our method, an improvement in the iteration count of about 2 times was obtained when a diagonal preconditioner was used. Additionally, we have also applied a modified version of our Full preconditioner (in which either the active non-penetration or static friction constraints were present), but this approach was not successful.

To run the experiment from Figure 5.9e using SP, we needed to reduce Δt to $0.0005s$, because for larger time-steps, the method did not converge. Even with the smaller Δt, solving both QP sub-problems converged properly, but applying each outcome to the other sub-problem resulted in an oscillating and diverging behavior. This problem appeared, for example, when the first two layers of objects collided with a large impact; the effects are visible in Figure 5.10b, in which larger iteration counts (spikes) can be seen. A similar observation was done for the hyper-elastic test, Figures 5.1 and 5.11. Here also the method takes a large number of iterations for solving the individual QPs, and then it requires many steps for finding a global optimum. A tighter coupling between the friction and non-penetration constraints seems to be necessary in these kind of situations. Using our method, we were able to increase Δt even further and even for stiffer models, see Section 5.6.1. When the stiffness of the models is increased (setup "Rings*"), SP seems to converge much slower compared to both ICA and CR. When stiffer and more complex objects are used (case "Bunnies"), SP did not properly converge, whereas the QP problems did converge. Therefore these results are not present in Table 5.1.

Comparison to Reduced-Form Methods

Reduced-form methods transform the complete contact problem into a *reduced* one and solve dynamics and contact separately. These forms require the Delassus operator $\mathbf{W} = \mathbf{L}\mathbf{A}^{-1}\mathbf{L}^T$, which is often used in methods in which the number of degrees of freedom per object is relatively small, such as granular materials, rigid bodies, reduced deformable bodies, or fiber simulations. In those cases, the generalized mass matrix $\mathbf{A}$ can be inverted efficiently. For deformable bodies with a large amount of degrees of freedom, the computation of $\mathbf{A}^{-1}$ is more demanding and may bring extra inaccuracies. As suggested in Bertails-Descoubes et al. [25], $\mathbf{A}^{-1}$ can be found through a Cholesky decomposition followed by forward and backward substitutions. Since $\mathbf{A}^{-1}$ contains large and dense blocks, *and* not all columns of $\mathbf{A}^{-1}$ are required for constructing $\mathbf{W}$, it is preferable to compute $\mathbf{W}$ through m forward and backward substitutions involving $\mathbf{A}$, $\mathbf{L}^T$ and $\mathbf{L}$, with m the number of constraints, see [47]. This omits the explicit form of $\mathbf{A}^{-1}$, which can have a very large memory footprint.

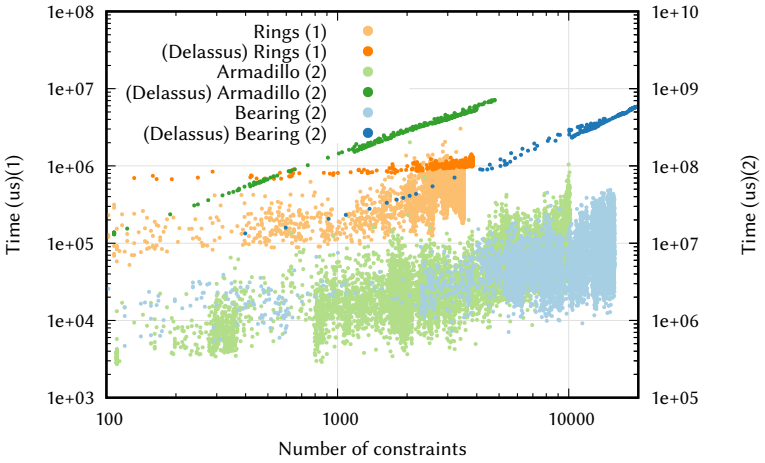


Figure 5.12: Timing comparison between our method, compared to the time required for constructing the Delassus operator $\mathbf{L}\mathbf{A}^{-1}\mathbf{L}^T$ using the optimized approach of Daviet et al. [47], for the Rings, Armadillo and Bearing test cases, respectively, in Table 5.1, see also Figures 5.1, 5.6 and 5.9h. The numbers in the legend refer to the left (1) or right (2) y-axis.

The method in Tonge et al. [180] does not explicitly compute $\mathbf{W}$ but computes, per contact block, each individual $\mathbf{L}\mathbf{A}^{-1}\mathbf{L}^T$ for each constraint and object associated with the contact block and accumulates this in a small matrix. The computational complexity is similar to the method in Daviet et al. [47], in which only the non-zero blocks of $\mathbf{W}$ are computed. Their running-time analysis for constructing $\mathbf{W}$ assumes that the degrees of freedom per object are bounded, and that the computation time is dominated by the squared number of constraints. Hence, by exploiting the block-diagonal structure of $\mathbf{A}$, $\mathbf{W}$ can be assembled efficiently.

The main problem with deformable bodies is to compute $\mathbf{W}$ in reasonable time. Unfortunately, all mentioned methods assume that $\mathbf{A}^{-1}$ is easy to compute, which is not the case for deformable bodies with a large amount of degrees of freedom. Figure 5.12 shows the time consumed for computing $\mathbf{W}$ through a Cholesky decomposition (by also taking the block structure of $\mathbf{A}$ into account [47]) and our method for the same number of constraints. This clearly shows that the computation time of $\mathbf{W}$ grows with the number of constraints, where its slope is determined by the complexity of computing $\mathbf{A}^{-1}$ (which mainly depends on its structure and/or conditioning). Once all blocks of the Delassus operator are computed, the methods can start solving the problem, typically using a (P)GS method in which $\mathbf{W}$ is used as the iteration matrix [47, 180] (both methods explicitly require the diagonal blocks of $\mathbf{W}$), or by using a Newton method in which at each iteration the search direction and step size is found by solving linear systems involving $\mathbf{W}$ [25]. Alternatively, in the latter method, the Delassus operator can be applied implicitly using an operator which solves a linear system involving $\mathbf{A}$. However, we experienced convergence issues when solving a linear problem involving this operator for the examples shown in Figures 5.1 and 5.9h. This is likely caused by the bad conditioning of $\mathbf{W}$ due duplicate constraints or multiple constraints between the same degrees of freedom, which typically occur between rigid and/or deformable objects, or in cases of large compression. In fact, they mention that their method cannot solve this kind of problems, including stacking of rigid bodies as shown in Figure 5.2c for our method. Similar issues are mentioned in Daviet et al. [47] when $\mathbf{L}$ becomes rank deficient. Please note that Figure 5.12 only considers the time

required for constructing the Delassus operator. On top of that, the actual running time of the method must be added. Furthermore, the plot does not take into account additional updates of $\mathbf{W}$ due to re-linearized constraint Jacobians, which boils down to re-computing $\mathbf{W}$ a few times per time-step.

Comparative Timings

Comparing only the iteration count, our approach requires between 10 and 60 times fewer iterations compared to SP. Similarly, our method requires between 2 and 8 times fewer iterations than ICA. When the actual computation time is taken into account, the differences are small for relatively simple models. When the complexity of the models increases (e.g., due to arbitrary shaped tetrahedral elements and many internal vertices), the timing differences become larger. This is due to the increased density and conditioning of matrix $\mathbf{A}$. The conditioning affects the convergence rate, whereas the density affects the time spent per iteration. Furthermore, when a model has many more elements than surface faces (due to internal vertices), the ratio collision detection / collision response, shifts such that collision detection uses a smaller portion of the total time. Experiment "Bunnies" and "Armadillo" in Table 5.1 depicts such cases. The Armadillo test case shows large differences in time and iteration count between the methods, see also Figure 5.11. The differences are mainly caused by the large compression of the material, which results in large contact forces and more linearization steps of the constraints due to the larger deformations. SP has difficulties in finding an accurate global minimum, whereas ICA spends more time on setting up and solving the LCPs.

Figure 5.9 shows additional example results by our method. The first snapshot shows various elastic objects, of different sizes and resolutions, interacting with each other. In total, 201 objects were simulated, containing about $150K$ tetrahedral elements; the maximum number of constraints peaked at $2K$. The average speedup compared to ICA was 4.35×, whereas the time spent on collision detection was negligible. Figure 5.9d shows an example of $2K$ elastic cubes that form a pile. The simulation contains about $88K$ tetrahedral elements, and the number of constraints peaked at $10K$. Figures 5.9e and 5.9g shows examples in which external forces and motions are used along with the simulations. For these examples a Young modulus of $E = 5 \times 10^5 Pa$ and a Poisson ratio of 0.4 were used. Additionally, Figure 5.9e depicts Pyramid test case, in which a pyramid is compressed by a large and rotating glass plate, resulting in large contact forces between the objects.

Memory Usage

The experiment shown in Figure 5.9h required about 380 MB of RAM for the whole simulation. Sparse matrix $\mathbf{A}$ had dimension 25920×25920 with an average density of 17 elements per row. The simulation shown in Figure 5.9e peaked to about 1.8 GB of memory with $\mathbf{A}$ of dimension 109200×109200 and an average density of 17 elements per row. For Bunnies test case, about 2.2 GB RAM was allocated, with $\mathbf{A}$ of dimension 147600×147600 and an average density of 27 elements per row. Most of the allocated memory is used for collision detection related structures and the FEM machinery. We have also compared our CR-based method with a MINRES based implementation. Both versions require similar amounts of memory; this is because both methods store the same matrix and allocate a similar amount of vectors. Therefore, no significant difference exist with respect to memory consumption.

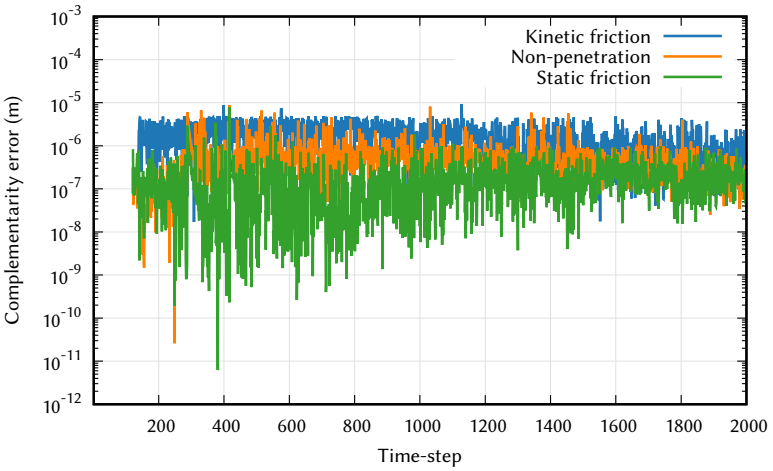


Figure 5.13: Plot of the Fischer-Burmeister function for our method, corresponding to the experiment shown in Figure 5.9h. The solver tolerance was set to $\epsilon_1 = 5 \times 10^{-5}$, and constraints were allowed to change if the corresponding state change was larger than $\epsilon_2 = 5 \times 10^{-6}$.

5.6.3 Error Measurements

The Fischer-Burmeister function,

$$f(x, y) := x + y - \sqrt{x^2 + y^2}, \tag{5.24}$$

can be used to measure the complementarity error made when solving complementarity problems, i.e., $\|(\mathbf{J}\mathbf{v} - \mathbf{c})^T \boldsymbol{\lambda}\|$. Figure 5.13 shows the maximum error measured per constraint for our method for the experiment shown in Figure 5.9h. For each individual constraint, x is set to the corresponding multiplier and y is set to the corresponding distance value, *e.g.*, $\mathbf{j}_k\mathbf{v} - c_k$. For kinetic friction constraints, x is set to the difference between $\mu\lambda_k$ and $\|\boldsymbol{\gamma}_k\|$, which should evaluate to zero if the constraint is satisfied. Furthermore, since x and y contain two different quantities, x is scaled by the corresponding value on the diagonal of matrix $\mathbf{S}_d$, see Section 5.3.4, such that both x and y represent a distance error. As shown in Figure 5.13, both static friction and non-penetration constraints measure in general a complementarity error that is below the specified solver tolerance $\epsilon_1 = 5 \times 10^{-5}$. Furthermore, the plot shows the maximum error among all constraints for a given time-step. Kinetic friction constraints yield, in general, an error smaller than $\epsilon_2 = 5 \times 10^{-6}$.

5.7 Discussion

Solvers for indefinite problems The CR method is very similar to the MINRES method [139], and in fact, for positive-definite matrices, both methods perform identically [63]. Even for indefinite problems, both methods perform similar. The main difference between CR and MINRES is the computation of the updated search vector. CR uses a two-term recurrence, similar to the CG method, whereas MINRES uses a three-term recurrence. We have implemented a MINRES version of the method described in Algorithm 4 using many combinations of preconditioners (None, Diag and Full) and friction treatment (cone and pyramid). For all tested cases (mentioned in Table 5.1), the MINRES version performed most time-steps very similar to our CR version. However, sometimes it had serious convergence issues leading to

failure. Especially when multiple bodies were colliding, convergence issues were observed, which were not observed within our CR-based method. Thus, when constraints are updated prior to convergence (as in our method), MINRES is not recommended. Conversely, we successfully applied MINRES as an Active Set QP solver in the SP method. The results were very similar to the CR-based method in which the constraints were kept fixed. Additionally, both CR and MINRES can break down in similar situations, in which a least-square approximation is computed. To alleviate this, an extra stopping condition is typically used, see Section 5.4.3. Within CR, vector $\mathbf{B}\mathbf{C}^{-1}\mathbf{r}$ is already available, whereas MINRES should explicitly compute it, which makes the method less efficient.

Other popular methods for solving indefinite problems are *generalized* versions of CR and MINRES, such as GCR [52] and GMRES [154], which also apply to non-symmetric problems. Both methods store one additional vector per iteration, which is used in all subsequent iterations. This results in growing storage and computational requirements. Since these methods rely on a history of search vectors, one could ask how GCR or GMRES perform when the methods are restarted after every constraint update. We leave this for future work.

Differences with ICA The speed difference between our method and ICA is mainly due to the different solvers used: Gauss-Seidel in ICA versus preconditioned CR in our method. Using the full preconditioner from Section 5.3.4, the CR solver converges significantly faster compared, for example, to a simple diagonal preconditioner, and this is therefore crucial for efficiently computing the collision response. Within ICA, per iteration, one Gauss-Seidel approximation is computed for refining the velocity correction and one Jacobi approximation for estimating the right-hand side of the LCP. We have observed that if the unconstrained problem is solved relatively fast, the differences between our method and ICA are smaller. Conversely, if the unconstrained problem is more difficult to solve (especially for stiffer and/or more complex models), our method performs significantly better than ICA, see the Bunnies test case in Table 5.1. However, we expect smaller differences in cases where $\mathbf{A}$ has a low density and low condition number. Apart from the underlying numerical methods, other differences also exist. For example, at the beginning of each time-step, ICA computes an unconstrained velocity that is used as input for the collision detection phase. Then, the collision detection stage results in a large number of collisions, which are subsequently used for computing the velocity correction. The problem with performing collision detection using the unconstrained velocity is that all elastic energy is released at once. The collisions found afterwards do not necessarily match the collision state at the end of the previous time-step, and hence the multipliers computed in the previous step will not result in a nearly resolved collision state. Due to this, ICA requires more steps to find the proper set of constraints and needs more iterations to converge. This becomes clearly visible in Figure 5.11. The difference in total computation time can become larger than 10 times. To make ICA more efficient in these situations, we suggest to follow our approach in which the constrained velocity is used for the collision test. The main advantage of this is that valid active constraints are reused, so there is no sudden release of the internal elastic energy, resulting in fewer new collisions. For the test case described in Section 5.6.2, the cylinders were made of rigid objects. As a result, the computed LCPs will have large and dense regions, which affects the construction time of the LCP matrix and running time of the PGS method. This additional overhead is visible as the time per time-step increases more than the number of iteration. This can be largely factored out using a modified PGS method, see [119] for more details. However, the total convergence behavior remains similar. This overhead is not observed while running the same test case using our method.

Differences with Staggered Projections SP uses a loose coupling of the non-penetration and friction constraints, by sequentially solving the non-penetration and friction sub-problems. The advantage of this approach is that kinetic friction is modeled through a BLCP for which the bounds are computed by the non-penetration sub-problem. The disadvantage of this coupling is that the sub-problems respond much later to state changes in the other sub-problem. This can require more iterations to obtain convergence, see Figures 5.10 and 5.11. If we compare our method to SP with Active Set solvers and a hybrid version using our modified solver, we can also see the effects of updating the constraint states while the method has not converged yet, see the results of Rings in Table 5.1. Within the Active Set version, the solvers converge quickly, although not necessarily to the optimal solution. By changing states, eventually the method converges. The hybrid version allows changing of non-penetration or friction constraints, depending on the current sub-problem. This results in a faster convergence, but still the non-penetration and friction constraints have the same loose coupling, which still can lead to a slow converging sequence of approximations. Within our approach, this coupling is much tighter, as constraints are allowed to change at much smaller intervals. Hence the method converges faster, but this mechanism introduces some additional overhead.

Discretization methods We have used a (first-order) semi-implicit time integration scheme that solves an implicit system for the unknown vertex velocities. The new velocities are then used explicitly to compute new vertex positions. However, our method is not bound to this scheme, and instead it can use other/higher-order (implicit) integration schemes, such as the Newmark-Beta method. When higher-order methods are used, the constraint matrices $\mathbf{J}$ and $\mathbf{J}_t$ are extended and the evaluation of the constraints involves additional quantities, but the basic principle remains the same. Furthermore, we have used the FEM for the spatial discretization (represented by matrix $\mathbf{A}$), but this can be replaced by other discretization schemes.

Towards larger time-steps Figure 5.8 shows results of our method for the same simulation, executed with different values of Δt. In general, the method converges faster for smaller Δt, but more time-steps are required to simulate a certain time span. Therefore, the total number of solver iterations remains approximately similar while simulating the same time span with different Δt. For larger time-steps, more Newton steps are required, resulting in more re-linearizations of the constraints, and the set of potential candidates increases; both affect the runtime. When the time-step becomes larger than $0.004s$, the constraint tolerance ϵ_2 should also be scaled with Δt. We have tested the method with time-steps of $0.02s$. In this case, the problem to solve per time-step is more difficult, but agrees with the findings in Figure 5.8. To apply this method in real-time applications, such as haptics [112], either the resolution of the meshes should be decreased, and/or parallelization should be used. With lower-resolution meshes, the number of potential collisions decreases as well.

Limitations We are bound to use a customized collision detection system, whereas ICA and SP, for example, can use any of the highly optimized systems available. Since our method is very accurate, it is less forgiving when conflicting constraints are created due to wrongly detected collisions. This puts additional requirements on the collision detection method such that no incorrect or conflicting constraints are created. Furthermore, the friction treatment in our approach may not allow for an easy replacement of the proposed CR-based solver, for example, by a PATH solver.

5.8 Future Work

In this chapter we have mainly focused on collision response handling for deformable models. However, we plan to further investigate the performance of our approach for coupled deformable and rigid-body simulations. As shown in Section 5.6.2, the LCP matrices lose their sparsity in such cases, resulting in larger computation times for the LCPs. Such a problem is not observed in our method. In Section 5.6.2, non-linear materials were used by linearizing the underlying energy density functions of the materials at the beginning of each time-step. One step further would be to extend our method such that also the complete non-linear problem is solved by using our approach in Newton-based solvers, which are usually built on top of linear solvers. Finally, all computations of our method can be easily parallelized, and we are in the process of porting the entire method on the GPU. Based on the results reported in [183] and our initial experiments, we expect a speedup of at least one order of magnitude.

6

Intersection tests

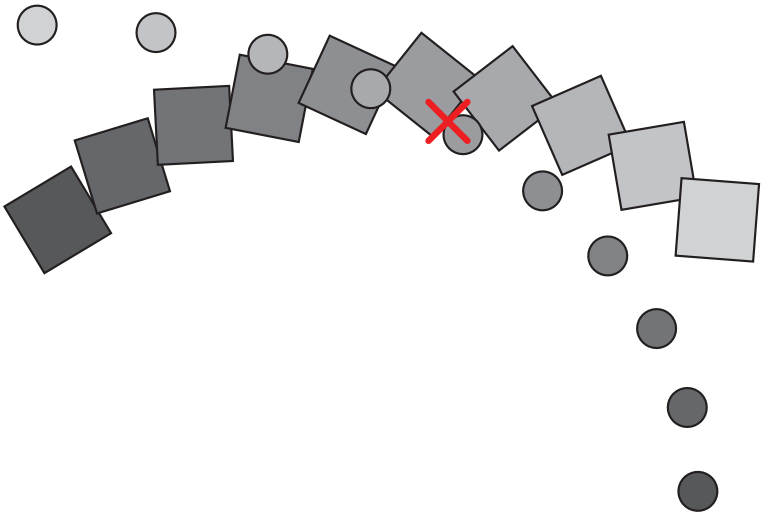


Figure 6.1: Continuous Collision Detection (CCD) will find the time and configuration of objects and individual features of the geometry, at their impact time. Each object in motion describes a path over time. By defining a distance function between both objects, a root-finder will find a configuration of both objects at the same time such that they are touching each other, i.e., the distance function is exactly zero. In this figure, both objects start their path at the dark-gray configuration and both advance through time to the light-gray instances. Both objects collide if their paths intersect at the same time (visualized by the gray-scale). The red cross indicates the location where both objects are colliding.

6.1 Introduction

In this chapter the collision handling system, briefly introduced in Chapter 5, will be described further in detail. In general, it is a *Continuous Collision Detection* (CCD) method that finds the time and configurations of two colliding features, i.e., *vertex-face* or *edge-edge* pairs, at impact. To find the time and configurations at impact, typically a *root-finding* method is used. At the begin of each time-step, the *initial configurations* of each feature is known. After computing a new approximate velocity, the geometry is updated and all *current configurations* of all features are obtained. Now each *pair* contains for both features the begin and current configuration, which describes a path in time for both features contained in the pair. When along these paths the signed distance between both features changes from positive to negative, the paths of both features cross and there must be a configuration in between where the signed distance is exactly zero. Using a root-finding method, the time and configuration of the two features at this impact time can be found, see Figure 6.1. By using the configurations at impact for initializing a constraint between both features, a collision response is computed which separates them and prevents interpenetration.

The main problem with this approach is that the root-finding method requires a *consistent* signed distance function for all possible configurations between the initial and current configuration of a pair. (With consistent meaning that the sign should only change when the state of the features in the pair change from colliding to non-colliding, and vice versa.) Along this path, the configuration of each feature can change significantly. Also the shape of the geometry may locally change between convex and concave. Assuming that the sign of the signed distance is computed correctly, only the computation of the distance remains. This computation is in

general trivial for non-degenerate geometry, but requires special attention when the geometry becomes locally degenerate. Another problem with collision detection of deformable objects is that the geometry deforms and can invert. When inversions occur, inside and outside are swapped, resulting in wrongly-detected collision. This in turn results in wrongly-initialized constraints. Eventually this leads to a situation in which the solver is not able to compute an approximation of the solution that satisfies all constraints.

In order to correctly solve the contact problem, it is therefore important that the signed distance is computed in a robust way, the geometry cannot (locally) invert, and that degeneracies are handled correctly. To do this, Section 6.2 introduces a number geometric primitives that are able to test for intersections, knowing that the configurations of these primitives may change drastically between the initial and current configuration. With this, a number of degenerate cases can be identified which need to be excluded from the intersection tests. Additionally, one must take the non-linear motion and deformation of the geometry into account. When the geometry deforms, the used constraints in the constrained optimization problem are likely not agreeing with the corresponding geometry. This should also be taken into account in order to guarantee a perfect collision-free state after solving the contact problem.

Discrete v.s. Continuous Collision Detection Collision detection methods can be categorized into two types, *Discrete Collision Detection* (*a posteriori*) and *Continuous Collision Detection* (*a priori*). Figure 6.1 shows a continuous collision detection method in which the trajectories of both objects cross at a certain time during one simulation time-step. Such an event is detected *a priori* and a response is computed which prevents the colliding objects to pass through each other. To do so, in our method, a prediction of these trajectories is made using *approximated* or *intermediate velocities* of the involved objects' features. These velocities are called *approximated* or *intermediate* because they do not reflect the final velocity, and they are likely to change due to the computed collision responses. Each time velocities are updated, also the trajectories may change. The main advantage of continuous collision detection is that all collisions are detected, no matter how large the time-step or velocity is. This is sometimes considered as *bullet-proof*. However, this approach also introduces some additional overhead due to testing for crossing trajectories.

The other type is discrete collision detection. When a new velocity is computed, the simulation advances to the next time-step by updating the positions of all objects and features. Instead of checking the trajectories of the objects or features for crossings, the objects or features are tested for intersections instead. If an intersection is detected, a response (and a new state) is computed that separates the intersecting objects. Due to this, collisions are detected *a posteriori*, i.e., after they have occurred. The main advantage of discrete collision detection is that is computationally cheaper than its continuous counterpart. However, before a collision is detected and resolved, the objects or entities were already intersecting. Depending on the chosen time-step size, these intersections could be visible. Additionally, the case shown in Figure 6.1 would not result in a detection of a collision event because the initial configurations have no intersection. Also in the next time-step (the light-most gray instances), no intersection occurs. Hence, the objects may pass through each other when the time-step is chosen too large.

Overview The presented collision detection method consists of three stages. First a broad-phase collision detection is performed which populates the so-called *Candidate Lists*. Each candidate list stores for each vertex or edge primitive, a list of nearby face or edge primitives. These candidate lists are then used to perform a fine collision/intersection test on each *candidate pair*, which is formed by the feature associated with the candidate list, and nearby features

Algorithm 7: Pseudo-code for collision detection

```
Initialize BVH and additional data-structures, Section 6.3;
i = 1;
while simulating do
    Update geometry using x^i;
    Broad-phase collision check, Section 6.4;
    Remove inactive constraints;
    Compute system dynamics, Appendices C and D;
    while Non-linear solution not correct do
        while Linear solution not correct do
            Compute new approximation v, Chapter 5;
            Update geometry using x^{i+1} = x^i + Δt v;
            Detect degenerate cases, add constraints, Section 6.6;
            Perform collision detection, add constraints, Sections 6.2 and 6.5;
        Check for missed candidate collisions, Section 6.5.3;
        Update constraints, Section 6.7.2;
        if No constraints updated or candidates added then
            Handle slid off contacts, Section 6.7.3;
    i = i + 1;
```


that are stored in the list. A candidate pair stores for both associated features their begin and current configuration. Each time the solver converges, a new approximation of the velocity is obtained, which is used to update the geometry, and the current states of features stored in each pair. Next, a fine collision check is performed on the updated pairs. This collision check is typically performed a few times per time-step. Finally, when a candidate pair detects an intersection of both features, a constraint is initialized and added to the solver. The candidate pair keeps track of the constraint and is responsible for keeping the constraint in sync with the geometry. The method described in this chapter can be summarized as in Algorithm 7.

This chapter is organized as follows: First, the geometric primitives used for intersection tests are described. For each type, a description of the intersection test is given. Next, the collision detection framework is described, starting with the broad-phase collision detection, followed by the finer collision tests using the intersection tests described earlier. Special attention is also given to handling degenerate cases. Finally, an approach for dealing with non-linear motion is given.

6.2 Primitives

Detecting collisions between parts of the geometry can be done by computing the signed distance between features in a vertex-face or edge-edge pair. However, the signed distance alone does not provide enough information to determine whether the associated geometry is colliding or not. It can be shown that in a number of cases, the signed distance is negative, but the associated geometry is not colliding or a collision can not be determined, see Figure 6.2. The main problem occurs when the geometry can significantly deform such that it (locally) inverts. When the inversion is resolved in a later time-step, there will be a false detection of

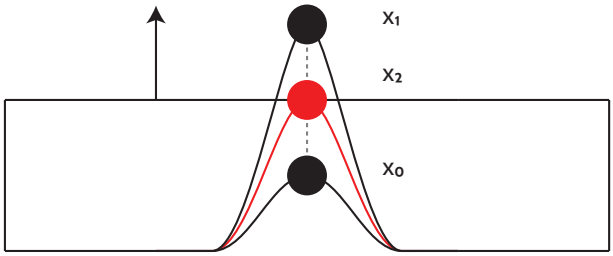


Figure 6.2: Example of a wrongly-detected collision. A vertex $\mathbf{x}_0$ internally crosses the surface from behind. This collision is usually not detected and an inversion is created ($\mathbf{x}_1$). When the inversion is being resolved by, e.g., the elasticity of the material, the same vertex now crosses the same surface from the front. This event *is* usually detected as a collision and a response is computed that prevents the vertex crossing the surface. As a result, the geometry got stuck in the red inverted geometry ($\mathbf{x}_2$).

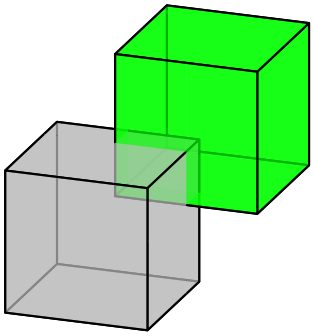
a collision. However, the sign changes from positive to negative, so it can be considered as a collision, but this collision will eventually prevent the geometry to recover from the inversion, see Figure 6.2. To prevent this, no inversions of the geometry are allowed. To do this, internal collisions must be detected and treated as normal collisions. Furthermore, false collisions must be identified correctly such that they can be ignored.

A better criterion for detecting a collision is when the volumes associated with features in a pair have an *intersected volume*. The properties of this intersected volume provides information about the type of intersection and thus how objects are colliding. The problem here is that vertices and edges do not necessarily define a volume. Faces or triangles define an infinitely-large volume that can be used directly. So, both vertices and edges need to be enriched such that they represent volumes. These enriched vertices and edges we call *geometric primitives*. These primitives must be able distinguish between front-face or back-face intersections, i.e., normal and internal collisions, and should correctly deal with temporally-inverted geometries.

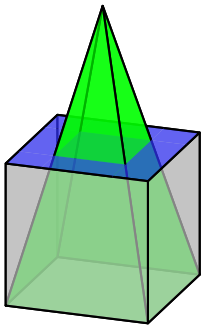
6.2.1 Intersection Definition

As mentioned before, a better approach for detecting collisions is to detect the geometric intersection between two surfaces using their underlying volume. When two closed volumes are intersecting, then the intersected volume is bounded by parts of the surfaces of both volumes, see Figure 6.3a. Each point inside this intersected volume also lies inside both volumes. The vertices and faces of the objects that are involved in the intersection, are also part of the boundary of the intersected volume. Apart from normal intersections, also internal intersections can occur in which a part of the surface penetrates another part of the same surface internally, see Figure 6.3b. Here the intersected volume lies 'inside' the cube. Since the vertex causing the internal collision now lies outside that volume, it is not part of the intersected volume. Therefore, this intersection is not detected using the previously-described conditions. Since this vertex lies outside the volume of the object, the geometry must be negated in order to detect such intersections using the same definitions.

Internal intersections To do so, an additional internal geometry is defined with the surface normals pointing inwards. This negated surface can only intersect with itself, not with internal surfaces of other objects and not with all other outer surfaces of all objects. Furthermore, this additional internal geometry is only needed for deformable objects. Since rigid bodies do not deform, internal collisions cannot occur. When an internal intersection occurs, the outer geometry has the same but inverted configuration, but the geometry of this intersected volume



(a) Intersecting volumes. The part of the volume that is shared by both volumes (the intersection) is bounded by the surfaces of both volumes.



(b) Internal intersection. The green sub-surface intersects the blue sub-surface from behind. The intersected volume is not bounded by the vertex causing the internal intersection since it lies outside the volume of the object.

Figure 6.3: Definition of an intersection. Left: two volumes have an intersected volume that is bounded by both surfaces of both volumes. Right: an internal intersection in which the intersected volume is not bounded by all features involved.

is not bounded by both involved features, thus the internal intersection is detected, while the same intersection for the outer surface is discarded. Figure 6.3b has an intersected volume that is bounded by the blue face, but is not bounded by the vertex causing the intersection. Therefore, this intersection is not valid.

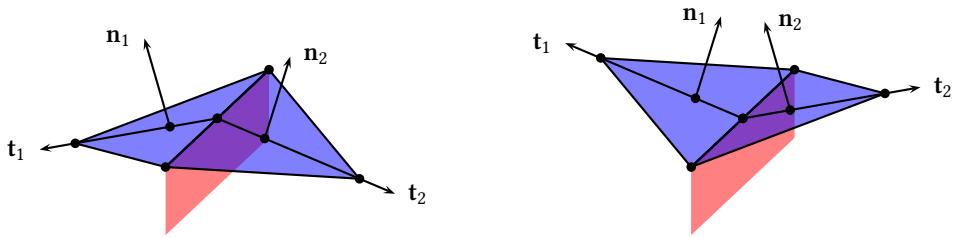
To detect intersections properly, a few primitives are developed that can detect intersections that have a bounded intersected volume. This allows us to distinguish between normal and internal intersections and collisions and treat them as such. In the following subsections each primitive is described and the definition of a valid intersection is given.

6.2.2 Face

A face, or triangle, defines a local boundary between the internal volume of an object and its exterior. Since faces have a strict definition of inside and outside, no special treatment is required for faces. Per face, a normal vector is stored.

6.2.3 Edge

An edge is a region where two faces meet. Since an edge does not represent a surface, it is undefined how an edge can distinguish between the interior and exterior of a surface. Since two faces meet at an edge, information about both faces can be encoded into an edge. To do this, two additional tangent vectors $\mathbf{t}_1$ and $\mathbf{t}_2$ are stored which are perpendicular to the edge and point to one of the two non-referenced vertices of the adjacent faces. Furthermore, two normal vectors $\mathbf{n}_1$ and $\mathbf{n}_2$ are stored which are the face normals of the two adjacent faces, see Figure 6.4. Given this information, an internal plane is defined that lies inside the volume and in between both adjacent faces. This plane is used in the actual intersection tests.



(a) Convex edge. **(b)** Concave edge.

Figure 6.4: Edge primitive. Left: a convex edge. Right: a concave edge. Each edge is enriched using the two adjacent face normals $\mathbf{n}_1$ and $\mathbf{n}_2$, and the two adjacent tangent vectors $\mathbf{t}_1$ and $\mathbf{t}_2$ which point from the edge to the adjacent vertices. Given the additional information, an infinite half-plane (red) can be defined that lies inside the object and lies exactly between the two faces.

Edge convexity Assuming a non-degenerate edge, each edge can be either convex or concave. When the two adjacent faces form a convex region, i.e, when

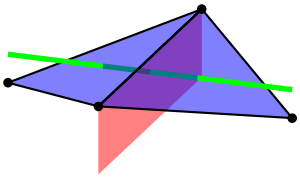
$$(\mathbf{t}_1 \cdot \mathbf{n}_2 < 0) \wedge (\mathbf{t}_2 \cdot \mathbf{n}_1 < 0) \tag{6.1}$$

holds, the edge is said to be *convex* (Figure 6.4a). In all other cases the edge is *concave* (Figure 6.4b).

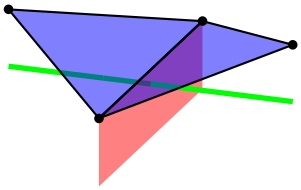
Edge-edge intersections Two edges can only intersect if both edges are convex. If one of them is concave, then the intersected volume is not bounded and thus infinitely large, see Figure 6.5. Furthermore, in such cases some vertices of the edges will have an intersection with the adjacent faces of the other edge in the test. When both edges are convex, then the actual intersection test can be performed. Two edges do intersect if each edge intersects both adjacent faces of the other edge. In this case a closed intersected volume is created. However, this test might become unreliable when both adjacent faces of an edge are almost co-planar. To deal with this, an additional intersection test is performed with the interior *half-plane* of the other edge. This half-plane lies in the interior region of the edge and lies exactly in between both adjacent faces, the red planes in Figure 6.4. When *both* edges intersect the interior half-plane of the other edge in the test, and at least one adjacent face, then both edges are intersecting. A strict closed volume is only obtained when both adjacent faces of both edges are intersected, but in practice it is sufficient to weaken this such that an intersection is valid when one of the adjacent faces *and* the internal face are intersected by the other edge. Please note that this test does not take the actual size of the edge into account, nor the size of the half-plane. Only infinitely large planes and edges are tested.

In order to deal with numerical errors in the computation of the intersection, additional tests must be performed in case no intersection is detected while both edges are convex. By placing small cylinders around the edges with a radius equal to the desired tolerance, additional intersection tests are performed with the edges and the cylinders. If both edge-cylinder intersection tests detect an intersection, then both edges are intersecting, since both edges are convex.

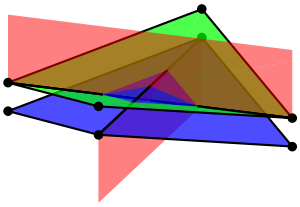
Edge-edge signed distance Given the definition of two intersecting edges, their signed distance can be computed. When two edges intersect, their distance must be negative. When two edges are not intersecting, we leave their sign undefined. If we simply give a positive sign to non-intersecting edges, then there might be a sudden change in the sign when the signed distance is evaluated in the root-finder. It might be possible that along the paths of two edges, their configurations change slightly and violate the definition of an intersection, thus



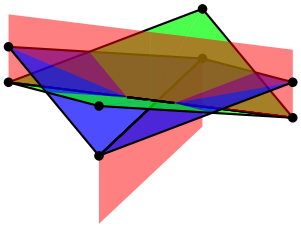
(a) A line (green) intersects a convex edge by intersecting both adjacent faces and the internal half-plane.



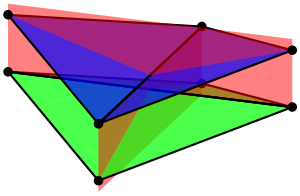
(b) A line (green) intersects a concave edge by only intersecting the internal half-plane.



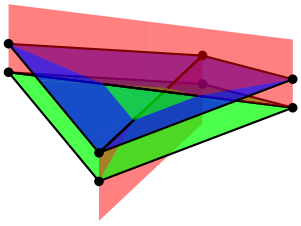
(c) Both convex edges intersect the other edge. The intersected volume is clearly bounded by the local geometry.



(d) A convex (green) and a concave (blue) edge do *not* intersect their internal half-plane. There exists however two infinitely-large intersected volumes that are not bounded by the local geometry. If both edges intersect the half-plane of the other edge, the intersected volume still remains unbounded.



(e) Both concave edges do *not* intersect the other concave edge, but do intersect the half-plane (red) of the other edge. There exists however an infinitely large intersected volume that is not bounded by the local geometry.



(f) Both concave edges do *not* intersect the other concave edge, they do *not* intersect the half-plane (red) of the other edge. There exists an infinitely large intersected volume that is not bounded by the local geometry. The volume that is visible in the center coincides with the outside region of both edges. If also a negated internal geometry is defined, this case would result in an intersection in the negated internal geometry.

Figure 6.5: Convex (a) and concave (b) edge intersections with an infinite line (green). Figures (c) - (f) show various test cases in which two edges are tested. Only (c) is defined as an intersection of two edges.

a sudden change from a negative to a positive signed distance is detected, which disturbs the root-finding process. The root-finder will probably find the configuration between the intersecting and non-intersecting states, but the corresponding edge-edge distance is far from zero. However, we are only interested in the location where this distance *is* zero. To work around this, intersecting edges must have a negative signed distance. When the root-finder searches for an intersection of the paths, the signed distances of each in-between configuration is computed using a *reference vector* and a vector between both projections of each edge on the other edge. When both edges cross each other, the dot product between the reference vector and

the vector between the projection, changes sign. By computing the sign of the signed distance as such, the signed distance function has a smooth behavior and a root-finder is now able to correctly find the location and time of impact of both edges. This reference vector can only be computed when edges are intersecting. When edges are not intersecting, but have intersected previously, this reference vector is realigned at each begin of a time-step.

Edge collapse The geometry can invert between its initial and current configuration. When this happens at a local scale, edges can collapse. An edge is collapsed when there is a transition between the initial and current configuration in which the adjacent faces switch side relative to the interior half-plane between the two adjacent faces. This also implies that a vertex of an adjacent face collides with the other adjacent face. Whenever an edge is *collapsed*, it can not be used in edge-edge intersection tests because the volume associated with the geometry is inverted. Furthermore, such an edge has been changed from convex to concave or vice versa. Hence, the definition of an intersection does not hold in these situations, so it will lead to wrongly-detected intersections. To recover these edges, additional vertex-face constraints are placed between both adjacent faces, if not already present due to other detected collisions. When these collisions are resolved, the corresponding edge is in a non-collapsed state, so it can be used in following edge-edge intersection tests.

6.2.4 Vertex

A vertex represents a single point in the geometry which can be efficiently used together with a plane to determine the signed distance. As long as the involved surfaces cannot invert, like for rigid bodies, or the features do not completely move through objects in one time-step, this approach is sufficient in order to test for collisions. When the model can potentially invert, or features can completely move through objects in one time-step, it is possible that a vertex crosses a face from behind. When this is not correctly handled, the same vertex will cross the same or another face from the front in a later time-step. This is then detected as an undesired collision event. Instead of using the signed distance of a vertex-face pair, a geometric intersection test must be defined, similar to the edge-edge case. To do so, each vertex is enriched with information about all edges and faces connected to the vertex and per edge, one edge normal is stored. The enriched vertex now describes a volume and so an intersection with a plane can be defined. Figure 6.6 shows such an enriched vertex, with $\mathbf{p}$ the actual vertex with its adjacent faces, $\mathbf{n}_i$ their face-normals, e_i the edges between the faces and $\mathbf{n}_{ei}$ the edge normals.

Vertex-face intersection Like the edge-edge case, vertex $\mathbf{p}$ intersects a plane when the intersected volume is bounded by the plane and all faces connected to vertex $\mathbf{p}$. Since the vertex has multiple faces connected to it, it is not possible to make a distinction between a convex or concave state of the vertex. Some parts of the surrounding faces can be convex, while other parts can be concave. To test if the vertex is potentially intersected by the plane, first the signed distance of vertex $\mathbf{p}$ and the plane is tested. If the signed distance is positive, then no intersection is possible. If the signed distance is negative, then all edge vectors $\mathbf{e}_i$ are inspected. If all edge vectors $\mathbf{e}_i$ have a positive dot product with plane normal $\mathbf{n}$, then all faces connected to $\mathbf{p}$ are intersected by the face. If also the volume bounded by the faces associated with vertex $\mathbf{p}$ is positive, then the face and vertex are intersecting. This is similar to the convex condition used for edges. In Figure 6.7 we show all eight possible combinations and the definition of an intersection when all faces are intersected by the plane.

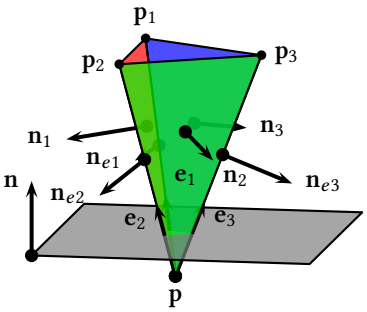


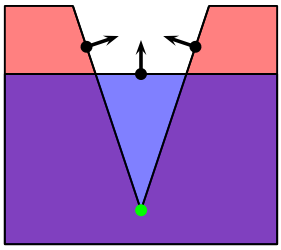
Figure 6.6: A vertex $\mathbf{p}$ with its associated volume spanned by connected vertices $\mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$. The surface of this volume is described by 3 faces with face normals $\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3$. Between the connected vertices and vertex $\mathbf{p}$, edge vectors $\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3$ are found. On these edges, edge normals $\mathbf{n}_{e1}, \mathbf{n}_{e2}, \mathbf{n}_{e3}$ are defined, which are the average of their adjacent face normals. The volume of the vertex is intersected by the gray plane with normal $\mathbf{n}$, which corresponds with Figure 6.7b. In practice the volume is spanned by more vertices and can have arbitrary shapes, which can potentially self-intersect.

It is straight-forward to see in Figure 6.7 when a vertex is intersected by a face. If the intersected volume lies in the interior of both volumes, then the intersection is valid. However, it is possible that the volume represented by vertex $\mathbf{p}$ does not have a nice convex shape. In practice it is possible that this volume is very flat or consists of multiple layers of faces. In this case it is difficult to measure the sign of the volume properly. Alternatively, this problem can be seen as a visibility problem.

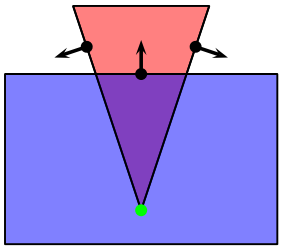
If a volume defined by a set of faces is intersected by a plane, and if the surface of the volume is facing the surface of the plane, then the intersection is valid. To test for this, the face of the volume closest to the intersecting face must be found. The closest face can be found by selecting edge vector $\mathbf{e}_i$ that has the smallest dot product with plane normal $\mathbf{n}$. If this dot product is negative, then the intersected volume is not bounded, so the intersection is not valid. Due to this, the dot product for a valid intersection is positive, so the edge with the smallest dot product must be the closest to the intersecting plane. Assuming that the volume is free from self intersections, no other face of the volume is located between the selected edge and the intersecting plane. Next, if the dot product of the edge normal $\mathbf{n}_{ei}$ and plane normal $\mathbf{n}$ is negative, then the intersecting plane and the edge $\mathbf{e}_i$ are facing each other. Hence, if the closest edge $\mathbf{e}_i$ is visible from the plane and all edges are intersected by the plane, then the intersection is valid.

This test only gives satisfactory results if the edges are not collapsed or when there are no self-intersections of the volume. If the closest edge $\mathbf{e}_i$ had been collapsed, then its corresponding edge normal was negated. When this normal is tested for visibility with the plane, the result is invalid. Therefore, vertices that have a volume that contains collapsed edges or has self-intersections, are excluded from the intersection tests until the volume is recovered properly.

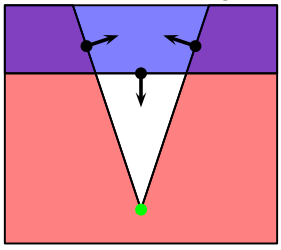
Vertex collapse The intersection of a vertex with a plane can only be valid when the volume associated with the vertex is free from self-intersections. If there are self-intersections in this volume, then there is no guarantee that the intersection test is performed correctly. When self-intersections are detected, the intersection test is not performed until the volume is recovered. To detect a self-intersection of the volume, an intersection-test must be performed between the faces and vertices associated with the volume. If a vertex crosses an opposite or neighboring face, then there is a self-intersection. Although this test can be performed explicitly, since all



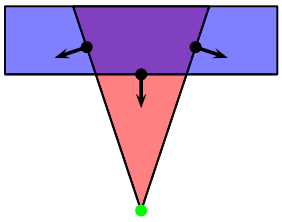
(a) No intersection: Negative signed distance, all faces intersected, no facing normals.



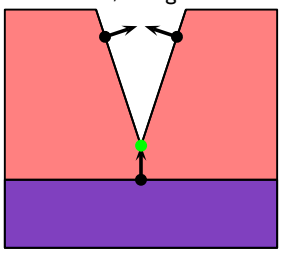
(b) Intersection: Negative signed distance, all faces intersected, facing normals.



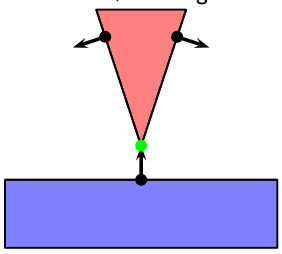
(c) No intersection: Positive signed distance, all faces intersected, facing normals.



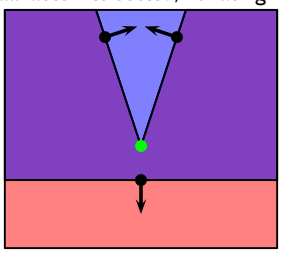
(d) No intersection: Positive signed distance, all faces intersected, no facing normals.



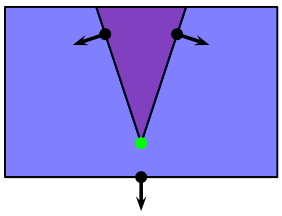
(e) No intersection: Positive signed distance, not all faces intersected, no facing normals.



(f) No intersection: Positive signed distance, not all faces intersected, facing normals.



(g) No intersection: Negative signed distance, not all faces intersected, facing normals.



(h) No intersection: Negative signed distance, not all faces intersected, no facing normals.

Figure 6.7: All eight possible vertex-face cases shown using simplified 2D cross-sections. The vertex (green) lies on the surface of a volume (red). The face represents another volume (blue). All vectors represent the local surface normals. Fig. b shows the only valid cases for an intersection between a vertex and face. Here the intersected volume is bounded by both involved features. Fig. c is a valid back-face collision, which is obtained by negating all normals in case b.

required information is already available, each vertex-face and edge-edge pair in the volume also exist as a pair in the candidate lists. By querying these pairs for sign-changes (vertex-face) or edge collapses, this test can be performed efficiently. Once all self-intersections of the volume are resolved, the vertex-face pair is automatically included in following intersection tests.

6.3 Initialization

In order to perform all intersection tests, all data structures need to be initialized. First an axis aligned hierarchical bounding volume needs to be created. This is done using a bottom-up approach in which the leaf nodes of a binary tree are created, see [187, 188]. Each leaf-node stores one triangle from the surface of a model, one axis aligned bounding volume containing the triangle and one extended bounding volume containing both the triangle and its displaced instance. Next, sub-patches are created by combining two adjacent sub-patches stored in nodes at the same level of the tree. The newly created sub-patch is then stored in a node at one level higher in the tree, and has the two individual nodes as its children. The selection of which two adjacent sub-patches need to be merged, is determined using a greedy algorithm. By measuring the average area and circumference of all possible combinations of sub-patches, the combination is chosen that maximizes the area / circumference ratio. This forces the process to create patches that have a coherent shape, which is favorable when testing for intersections. This process is continued until all sub-patches are merged at all levels, such that they are eventually represented by one node, the root of the object. Next, for each sub-patch in each node of the tree a bounding box is computed given the size of the sub-patch stored at that node. Eventually, each bounding box is a sub-volume of the bounding box of its parent node. The bounding box that is computed at the root, is a representation of the size of the stored geometry of the whole object. After each sub-tree for each individual object is created, all sub-trees are merged in a similar way such that a tree is created containing all triangles in the scene. Figure 6.8 shows a visualization of patches with their bounding volumes stored at different levels in the tree.

Next, for each face, edge and vertex in the scene, empty *candidate lists* are created. Each (empty) candidate list stores the initial, and current configuration of the feature associated with the list. Here initial configuration means the configuration at the beginning of the time-step and the current configuration is the configuration obtained after a computing an approximate velocity. Eventually the current configuration becomes the initial configuration for the next time-step.

6.4 Broad-Phase Collision Detection

Given the initialized bounding volume hierarchy, first an additional set of bounding volumes is initialized. Given the current velocities of each vertex in the triangular mesh, an approximation of the displacement can be made. By multiplying the displacement of a particular vertex by some constant and adding a small offset, a larger bounding box is obtained, the *extended bounding box*, which also contains the approximated displaced feature. Next, all extended bounding volumes at each level of the tree are updated using the extended bounding boxes of their children.

Figure 6.8: Visualizations of different levels of the BVH. Each sub-patch has one (random) color and one bounding box assigned. The levels of the visualized BVH are, 0, 4, 8 and 12.

For each face in the scene, a broad-phase collision check is performed using the extended BVH. Given the extended bounding volume of the associated face, all leaf nodes in the extended BVH are selected by traversing the tree while their extended bounding boxes intersect. Once this tree traversal reaches leaf-nodes for which their extended bounding boxes intersect with the extended bounding box of the queried face, the face associated with that node is added to the list of candidates for the queried face.

Please note that for a particular face, all its neighboring faces are found using this test. This is in general considered as a waste of computational resources since the chance that a triangle intersects its neighboring faces, is in general very small. When the underlying object is rigid, this is not even possible. However, for highly deformable objects, this is actually possible and happens frequently. Therefore, neighboring faces for which the bounding boxes do intersect are included in the candidate lists.

Once all face-face candidate pairs are found, unique vertex-face and edge-edge candidate pairs are extracted. For each pair, a new candidate is created (or updated) and stored in the corresponding candidate lists. Each candidate stores the initial (at the beginning of the time-step), current (end of the time-step) and intermediate configuration of the face, vertex and edges at

the moment of impact (if a collision is detected). Once a new approximate velocity is computed, the current configuration of each vertex, face and edges are updated. After the candidates are updated, they are tested for possible intersections. When a new time-step starts, the initial configurations are set given the last current configuration. The new current configuration is updated later after a new approximate velocity is obtained.

6.5 Intersection Tests

Once all candidate lists are populated with their candidates, the actual intersection tests are performed for each pair described by a candidate. First, at the begin of the time-step, all currently known primitives are updated given the new state of the geometry. By assuming that the previous time-step resulted in a collision-free state, the initial state of the current time-step is *also* free from collisions and collapsed edges. Next, the new state of the system is obtained by solving the constrained dynamics problem. Once a new state is obtained, the current configurations of the primitives are updated. Given the new current configurations of all candidate pairs, intersection tests are performed by finding roots in their signed distance function.

6.5.1 Root Finding

Given the initial configuration at the beginning of the time-step and the current configuration given the new state of the geometry, a finer check is performed for each pair by finding roots in the signed distance function. Using the root-finding method described in Section 6.7.1, the time and configurations at impact can be found, if exists. If the root is found, additional checks are performed on the obtained configurations from the root-finder. First, their distance should be zero. Second, the computed intersection point must lie on the triangle or edges. If this point lies outside the geometry defined by the primitives, then the pair does not intersect and is not considered until the next intersection test. If the intersection point is on both primitives, then a constraint is initialized using the configuration at the obtained root. Last, also the obtained configuration must be a 'valid' intersection. However, since the root finder has found a configuration for which the distances are zero, there will be no formal intersection. However, for both edges and triangles it is important that features are not degenerate, have not too small edges, and the area of the triangle must not be too small.

6.5.2 Additional Intersection Tests

After solving the constrained problem, a new configuration for the current set of pairs is obtained. It is possible that new intersections happen due to deformations of surfaces. By performing a new collision check, these collisions will be found and new constraints are added to the system. This process is continued until no pairs are found that have a real intersection.

6.5.3 Missing Collision Test

At the beginning of each time-step, a broad-phase collision check is performed by traversing the BVH. This step adds new candidates to the candidate lists if the corresponding geometry is likely to collide. This test is performed using axis-aligned bounding volume tests in which the extended bounding volumes of two features are tested against each other. When at this point the bounding volumes of the features of a pair are not intersecting, the pair is not added to the candidate list and no finer collision checks are performed during the current time-step. However, since this test is performed using an approximated velocity at the beginning of the time-step, it is possible that the final velocity is significantly different than the initial approximated velocity. Therefore it is possible that at the end of the time-step, collisions

are missed since the corresponding candidate pairs were not present in the candidate lists. If these candidates are added in the following time-step, then the geometry *can* be intersecting at the beginning of the next time-step. Due to this, no new collision can be detected for that candidate, resulting in an undesired interpenetration that cannot be resolved.

To ensure that at the end of the time-step all candidate lists do contain all potential candidates, we could perform an expensive second broad-phase collision check. Alternatively, each current configuration of each face in the geometry can be tested against the extended bounding volume used to populate the candidate lists. If a face had been moved out of its extended bounding volume, then there could be a potential collision missed. By identifying all vertices, edges and faces that have moved out of their extended bounding boxes, and updating their bounding volumes in the BVH, all missed potential collisions can be found. Given their updated extended bounding volumes, the updated BVH is tested against these identified faces. If new potential collisions are found, the corresponding candidate lists are updated using the new candidates. Eventually, finer collision checks are performed. Contrary, if no new candidates are found, no collisions are missed and the method can continue. This test is performed every time constraints are updated. If all features remain in their extended bounding volume, no additional steps are required.

6.5.4 Treatment for Internal Collisions

In general, each surface of a deformable object can have self-intersections. In principle, these kind of intersections are no different than any other intersection between two different deformable objects. However, if the underlying tetrahedral mesh can potentially invert, then another kind of self-intersection is possible. In these situations it is possible that the surface intersects itself from the inside. If these internal collisions are not resolved, then the mesh could invert. However, thanks to the primitives described earlier, no collision is detected when the inversion is being resolved. Such a collision is detected when only signed distances are considered which could potentially result in the creation of conflicting constraints and could thus result in an unsolvable system.

In order to deal with these *internal collisions*, intersections of the internal geometry must be detected separately from the outer geometry. To do so, additional candidate lists are created for each vertex and edge that are part of the internal geometry. Next, when a face-face pair in the BVH is likely to intersect and both faces belong to the same internal surface, also their back-face versions are used to create the vertex-face and edge-edge candidates for the internal geometry and are stored in the associated candidate lists. By using this additional internal geometry, the intersection test is performed exactly as done for the outer surface. Since the candidate pairs in the inner and outer surface are *mutually exclusive* with respect to intersections, it is not possible that for a particular pair both the inner and outer surface detects an intersection at the same location. However, to guarantee this, the geometry must be free from degenerate cases. If degenerate cases are found, they must be actively resolved first. If this is not guaranteed, then the surface can become in a state in which intersections cannot be detected properly. Furthermore, since the outer and inner surfaces are separated, outer surfaces cannot intersect with inner surfaces. This removes the possibility of the detection of false collisions.

6.6 Degeneracies

While testing the geometry for intersections, a number of degenerate cases can be encountered. Edges can become parallel to other edges of faces, or the distances between edges and vertices in a triangle can become too small, resulting in too small edges or too small areas of triangles. Here we describe how to handle these situations.

Parallel edges When edges become parallel, the edge is also parallel to the half-plane between the two tangent faces of the edge (Figure 6.4). In this case, no intersection is detected, so the pair is automatically neglected in intersection tests. This is in general not a problem, since the endpoints of the edges *could* have an intersection with the adjacent faces of the other edge. So possible intersection are resolved by the vertex-face primitives.

Edge-Vertex In order to guarantee a mesh with no zero-sized edges and triangles, additional constraints are used to constrain the distance between each edge of a triangle and its opposite vertex. If this distance becomes too small, the area of the triangle also becomes very small and / or edges become too short. In these cases the orientation of the triangle may change significantly while the change in position of a vertex is small. This makes it difficult to compute a good collision response.

To detect that a vertex moves too close to an opposite edge, a plane is used with a normal that coincides with the vector perpendicular to the initial edge and pointing to the initial opposite vertex, see Figure 6.9. If the distance measured with respect to this plane drops below the desired tolerance, a root-finding procedure is started in order to find the intersection of the path of the vertex and the cylinder around the edge. Within this procedure the actual signed-distance function is used which *could* have more than one root. If multiple roots are found, the closest to the initial configuration is selected since it represents a crossing from a positive to a negative signed distance. Given the obtained configuration at the root, a friction-less constraint is initialized which is treated like any other constraint, including possible updates due to a changed geometry. As long as the degeneracy is not resolved properly, the face, edges and vertices involved are not used in other intersection tests.

Exact edge-edge intersections Two edges have an exact edge-edge intersection if their signed distance is exactly zero (or smaller than a certain epsilon threshold). The signed distance of the intersecting pair should be negative, but if their distance is exactly zero, the correction using the reference vector cannot be performed since either way the result is always zero. If the reference vector can not be determined properly, we cannot compute the sign of the signed distance. To correctly determine the direction of the reference vector, copies of both edges are *displaced* in the opposite direction of the average face normals and tangent vectors, i.e., *assuming that the edge is convex*, this average vector is obtained by:

$$\mathbf{z} = \mathbf{n}_1 + \mathbf{n}_2 - \mathbf{t}_1 - \mathbf{t}_2. \tag{6.2}$$

If the edge is flat, the normal vectors cancel out, if the adjacent faces are co-planar, the tangent vectors cancel out. Hence, it is always possible to compute a displacement direction. Given vector $\mathbf{z}$, the vertices of the edge are displaced in the negative direction of $\mathbf{z}$. This needs to be done for both edges for both computed directions of $\mathbf{z}$. After this separation is performed, the signed distance can be computed again, which should be positive. If the computed signed distance is negative, the reference vector is negated. The obtained reference vector is then used to re-compute the initial and current distance.

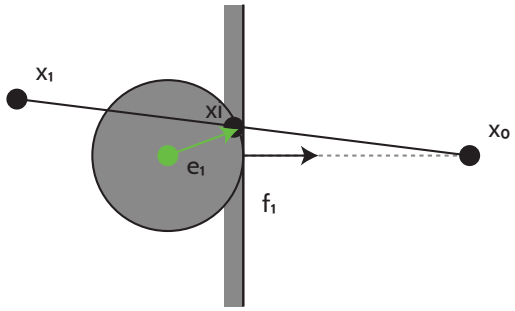


Figure 6.9: Intersection of vertex $\mathbf{x}$ with an edge $\mathbf{e}$. This image show the cross-section of a triangle with an edge $\mathbf{e}$ and its opposite vertex $\mathbf{x}$. If the path between the initial configuration $\mathbf{x}_0$ and $\mathbf{x}_1$ crosses plane f_1, an intersection test is performed between the cylinder around the edge and the path of the vertex. If an intersection is detected, a constraint is added which prevents the vertex to move closer in the direction of the edge. This approach guarantees a minimum distance between edges and vertices, and so between vertices in the same triangle.

Complete edge degeneracy resolution To correctly perform the intersection tests and the creation of constraints, it is very important to perform the intersection tests only when the associated geometry is not degenerate. To do so, the resolution of the degeneracies is prioritized as follows:

1 Enforce a minimum distance between all edges and vertices within one triangle.

2 Enforce constraints on collapsed edges.

3 Resolve self intersections in volumes associated with vertices.

4 Resolve regular intersections.

For case 1 and 2, additional constraints are used that enforce a minimum distance or resolve the collapsed state. Case 3 is resolved by enforcing case 2 and regular collisions between the involved features, case 4. Once all these cases are resolved, the associated features are included in the regular intersection tests. While resolving the degeneracies it is important to first resolve case 1, then 2 followed by 3. For example, if the volume associated with a vertex consists of collapsed edges and the minimum distance is not respected, the minimum distances need to be enforced. Once the minimum distances are enforced, collapsed edges must be resolved. After that, we can add additional constraints, due to regular collision detection, to resolve self intersections for the volume associated with the edge. If this ordering is not respected, then it is possible that a state is obtained for which no solution exists.

6.7 Non-Linear Motion

The method described so far assumes a 'nearly' linear motion of the primitives. In practice, the motions of these primitives are determined by *the motion of the objects*, *the contact responses due to collisions* and *deformations due to contact.* All these aspects contribute to a non-linear motion of the primitives which affects the detection of collisions, the computed responses of the collisions and the state of the objects.

To correctly handle non-linear motions, first a collision between two features with a non-linear motion must be properly detected. Second, due to deformations, the initially computed locations of contact points between colliding primitives will change due to deformations, friction and other collisions. Also the associated contact normal changes accordingly. These changes must be taken into account. Finally, a contact point associated with a collision between two features can slide to neighboring parts of the surface. If these cases are not correctly handled, collisions can be missed, resulting in interpenetrations or large gaps between two surfaces in 'contact'. In the following subsections we describe each aspect in detail.

6.7.1 Non-Linear Collision Detection

Assuming a simulation with discrete time-steps, a collision occurs when the signed distance between two primitives changes from positive to negative. If such a change happens somewhere in a time-step, a time and configuration exists in which the signed distance must be zero. Depending on the configurations of the involved features at this intermediate time, the features are about to collide.

This approach works well when small time-steps are used or when the motion is nearly linear. For larger time-steps, the motion of the involved primitives may be non-linear. For example, vertices at the surface of a deformable body can have a linear motion, but the motion of triangles and edges can be non-linear. A triangle can deform and rotate due to linear vertex velocities. If another vertex crosses this rotating face in the same time-step, the signed distance for this pair can be positive in the beginning of the time-step, and again positive at the end of the time-step. Thus, detecting collisions based solely on the signed distances at the beginning and end of the time-step may result in missed collisions. In order to detected this kind of collisions, a more sophisticated root-finding method must be used.

Root-finding Given the distance function of the example above, there are multiple possible scenarios. There is no zero (or root) between the beginning and end of the time-step, or the function has a multiple of two zeros. In such cases, the derivatives of the distance function in the beginning and end of the time-step have opposite signs, as shown in Figure 6.10. In the first part of the time-step the vertex moves towards the face, in the second part, the vertex moves away from the face, while the face rotates in between. Using this additional information, one can easily determine if there is a potential root in this time-step.

Assuming that the distance function behaves smoothly during the time-step, a minimum of the distance function occurs when its derivative is zero. Since the derivatives in the beginning and end of the time-step have different signs, the derivative must be zero somewhere in between, which coincides with the minimum of the distance function. By finding the root of the derivative distance function, also this minimum is found. If at this minimum the signed distance has a different sign compared to that in the beginning of the time-step, two intervals exist with each one root. In this case, we are interested in the first root, since the derivative in the first segment is negative, indicating that the distance decreases. Using a bounded root-finding

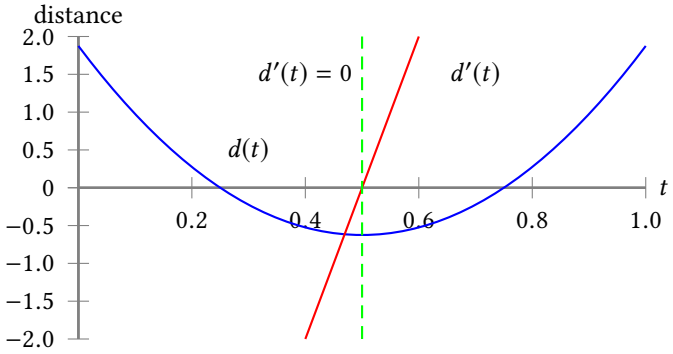


Figure 6.10: Distance function $d(t)$ (blue) evaluated at t in one time-step. The distance function is positive at the beginning ($t = 0$) and at the end ($t = 1$) of the time-step. Its derivative $d'(t)$ (red) is negative at $t = 0$ and positive at $t = 1$, which implies a minimum or maximum of the distance function within the interval of the time-step. This minimum can be found using a root-finder and is located at $t = 0.5$ (green). At this minimum, the distance function is negative. This implies that a root exists between $t = 0$ and $t = 0.5$. Another root exists between $t = 0.5$ and $t = 1$. Since we are interested in the root in the first interval, performing a root-finding method between $t = 0$ and $t = 0.5$ will give the time and configurations at impact.

method like the Brent-Dekker method [145], in combination with a distance function that has different signs in the beginning and end of the interval, a root is guaranteed to be found. Contrary, if the minimum of the distance function has the same sign as the distance function in the beginning of the time-step, then there is likely no intersection. The vertex moves close to the face, but does not intersect. However, at this stage, this cannot be guaranteed since also the derivative can have multiple roots within the interval.

Many roots, many extrema Depending on the time-step size and the type of simulations involved, the distance function, and its derivative, can have multiple roots and extrema within the interval described by the time-step. For example, when candidate pairs contain features associated with (articulated) rigid bodies, their motion can be very non-linear and the distance function may contain multiple roots and/or extrema. According to the *Intermediate Value Theorem*, a continuous function that has different signs at the start and end of such interval, must have at least one root somewhere in this interval. When the Brent-Dekker method is used, the function values in the beginning and end of the interval of interest must be different in sign. If satisfied, then a root within this interval is found. However, it remains unknown if the function contains other roots within the same interval. To find other potential roots, one could sub-divide the interval in smaller sub-intervals and perform a root-finding on those sub-intervals as described in the previous paragraph and Figure 6.10. However, there is no guarantee that an arbitrary root is a good candidate for sub-dividing the interval.

A better approach would be using a root-finding method that does not require different signs of the function value at the boundaries of the interval. Usually such methods are also not bracketed, meaning that they can converge to any root, which could be outside the interval of interest. For example, Muller's method [145] is such a method and does not require a good approximation of the initial guess. Please note that this method requires complex arithmetic. Once a root has been found, a root deflation technique is used to remove the root from the original function, i.e.,

$$f(x)_1 = f(x)_0/(x - x_0), \tag{6.3}$$

with $f(x)_0$ the original function, x_0 the first root found and $f(x)_1$ the deflated function. Now the next root, if exists, can be found in the same way as described above. This deflation process continues until no new (real) roots are found. However, care must be taken. Due to the addition of approximated roots containing rounding errors, also the roots of the deflated function are affected by these errors. One way to reduce this error is to *polish* the obtained roots by performing a root-find method on the original function where the starting point is set to the root. Usually a few iterations are sufficient. Once all roots are obtained, the root is selected for which a real geometric collision happens. If there are multiple valid configurations, the first one is used.

Edge-edge roots When searching for a root of an edge-edge pair distance function, one must be careful. If at both the beginning and end of the time-step the edge-edge pair has no intersection, we cannot determine the sign of the signed distance correctly. This can only be done if the pair is intersecting. When a multiple of two roots are found, the configuration of the edge-edge pair needs to be inspected in between the found roots. If one of them is intersecting, then the reference vector can be computed and used to correct the other distances. If no edges are intersecting at the selected points, then there is most likely no intersection. When the signed distance has only one root in the interval, either the beginning or current configuration of the edge-pair is intersecting, for which we know that the signed distance must be negative.

6.7.2 Constraint Updates

Once the additional intersection tests do not find any new intersections, the state of the current pairs that have a constraint defined should be further inspected. Given the description of the constrained problem, see Equation (5.11), it is immediately clear that at convergence $\mathbf{J}_l\mathbf{v}_{l+1} \geq \mathbf{c}_l$ hold, with $\mathbf{J}_l$ and $\mathbf{c}_l$ obtained from the geometry in the beginning of the time-step. However, for active constraints,

$$\mathbf{J}_{l+1}\mathbf{v}_{l+1} - \mathbf{c}_{l+1} \neq \mathbf{J}_l\mathbf{v}_{l+1} - \mathbf{c}_l = C_l(\mathbf{v}_{l+1}) = \mathbf{0}, \tag{6.4}$$

implies that it is possible that the constraint configuration does not agree with the current state of the geometry. This is visible as too large gaps or interpenetrations between objects. Additionally, a contact point could have slid off an edge or face (the constraint is active where it should not be).

In order to determine that the current state of the geometry is correct, all signed distances of the geometric pairs having a constraint enabled are inspected. If each signed distance is positive and within the desired tolerance, the current configurations of all constraints are correct with respect to the geometry, except that contact points could have slid off a face or edge, which is discussed later. If at least one of the constraints does not agree with the geometry, then all constraints are updated given the current configuration of the geometry. To justify the update of constraints, let us recall Newton's method, i.e.,

$$x_{l+1} = x_l - \frac{f(x_l)}{f'(x_l)}, \tag{6.5}$$

applied to the problem described in Equation (5.11), Appendix B.1.2 and Appendix B.1.5. Considering only equality constraints, this yields the following procedure:

$$\underbrace{\begin{pmatrix}\mathbf{v}\\ \boldsymbol{\lambda}\end{pmatrix}_{l+1}}_{\mathbf{y}_{l+1}} = \underbrace{\begin{pmatrix}\mathbf{v}\\ \boldsymbol{\lambda}\end{pmatrix}_{l}}_{\mathbf{y}_l} - \underbrace{\begin{pmatrix}\mathbf{A} & \mathbf{J}^T\\ \mathbf{J} & \mathbf{0}\end{pmatrix}^{-1}}_{f'(\mathbf{y}_l)^{-1}} \underbrace{\left(\begin{pmatrix}\mathbf{A} & \mathbf{J}^T\\ \mathbf{J} & \mathbf{0}\end{pmatrix}\begin{pmatrix}\mathbf{v}\\ \boldsymbol{\lambda}\end{pmatrix}_l - \begin{pmatrix}\mathbf{b}\\ \mathbf{c}\end{pmatrix}\right)}_{f(\mathbf{y}_l)}, \tag{6.6}$$

in which $f'(\mathbf{y}_l)^{-1}f(\mathbf{y}_l)$ is solved using some iterative method, see Chapter 5, and l indicating the Newton iteration number. When performing Newton's method, $f(\mathbf{y}_l)$ and $f'(\mathbf{y}_l)$ are linearized at $\mathbf{y}_l$, meaning that before using $f'(\mathbf{y}_l)^{-1}$ and $f(\mathbf{y}_l)$, all constraints $C(\mathbf{y}_l)$ are linearized given the state of the geometry at $\mathbf{y}_l$, which is then stored in the system by updating $\mathbf{J}$ and $\mathbf{c}$. Since in general Equation (6.4) applies for the updated constraints, $f(\mathbf{y}_l) \neq \mathbf{0}$ and a new step towards the solution of the non-linear problem is performed. As long Equation (6.4) holds, a new Newton step is performed. When all constraints agree with the geometry, $f(\mathbf{y}) = \mathbf{0}$, implying that a non-linear solution is found.

It is known that Newton's method converges slowly or diverges if the current update on $\mathbf{y}$ does not improve $f(\mathbf{y})$. Since only constraints $C(\mathbf{y})$ are updated, the update of these constraints should be done in a 'smooth' way. In our case, we choose to update each constraint using a linear combination of the configurations used for initializing or updating the constraint, and the current configuration of the geometry, i.e.,

$$C'_{l+1}(\mathbf{y}_{l+1}) = wC_{l+1}(\mathbf{y}_{l+1}) + (1-w)C_l(\mathbf{y}_l), \tag{6.7}$$

with $0 < w < 1$ a weight such that the change in the contact normal between C'_{l+1} and C_l is less than, say, 15 degrees, $C_{l+1}(\mathbf{y}_{l+1})$ the current configuration of the edge-edge or vertex-face pair given the current positions $\mathbf{y}_{l+1}$ obtained by linearizing the constraint after the last Newton step. $C_l(\mathbf{y}_l)$ represents the previous configuration of the constraint in the previous Newton step, and consequently $C'_{l+1}(\mathbf{y}_{l+1})$ is the new *interpolated* configuration of the pairs, which are used to update the current set of constraints.

To put this in the context of the method described in Chapter 5, $f(\mathbf{y})$ equals the negative residual vector. Each iteration of Algorithm 4 improves $\mathbf{y}$. When the residual drops below a certain tolerance $f(\mathbf{y}) \approx \mathbf{0}$. At this point, constraints are updated as described before. By recomputing the residual vector, the solver now improves $\mathbf{y}$ given the new configuration of the constraints, based on the previous solution of $\mathbf{y}$. Since the solver already performs $\mathbf{y}_{l+1} = \mathbf{y}_l - f'(\mathbf{y}_l)^{-1}f(\mathbf{y}_l)$ in one of the search directions, there is no need to explicitly perform the Newton procedure as shown in Equation (6.6), i.e., the update of $\mathbf{y}_{l+1}$ based on $\mathbf{y}_l$ is already performed implicitly. Alternatively, Equation (6.6) can be rearranged as

$$\begin{pmatrix} \mathbf{v} \\ \boldsymbol{\lambda} \end{pmatrix}_{l+1} = \underbrace{\begin{pmatrix} \mathbf{A} & \mathbf{J}^T \\ \mathbf{J} & \mathbf{0} \end{pmatrix}^{-1}}_{f'(\mathbf{y}_l)^{-1}} \begin{pmatrix} \mathbf{b} \\ \mathbf{c} \end{pmatrix}, \tag{6.8}$$

which implies that after solving the problem, constraints need to be re-linearized. Performing a few 'Newton iterations' by updating constraints and solving the newly obtained constrained optimization problem, the constraints will eventually match the state of the geometry. Furthermore, the distances between primitives with active constraints are guaranteed to be positive and less than the desired tolerance. If this update step is not performed, it is possible that after solving the constrained problem, the geometry still has interpenetration or too large gaps, as shown by Equation (6.4). Once a collision-free state is obtained, all constraints associated with contact-points that have slid off a face or vertex, need to be inspected further.

Please note that when contact points are sliding over the surface, one must take the already slid distance into account when updating the constraint. When a constraint is updated, the distance between the initial and current configuration is updated, and stored in $\mathbf{c}$. However,

this allows the solver to freely move the contact point to that location and start the friction computation from there. This basically releases the already computed friction forces, which is not desired. To keep friction consistent after a constraint update, the initial tangential-distances with respect to the initial configuration are used, but need to be aligned with the updated geometry. This prevents a sudden release of (kinetic) friction forces.

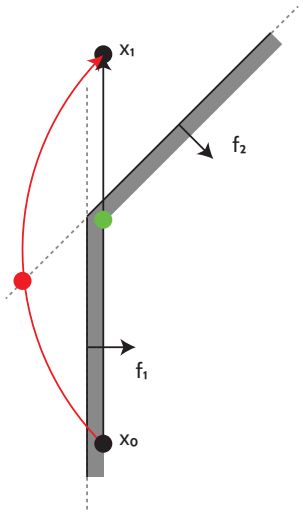
6.7.3 Sliding Contacts

Another type of constraint update is required when two surfaces are sliding over each other. In those cases it is possible that a contact-point slides off a face or edge to a neighboring face or edge. In these cases the constraint should be transferred to its neighboring face or edge. In most cases it is sufficient to just disable such constraint, but this strategy does not always result in a collision-free state, as we will describe here.

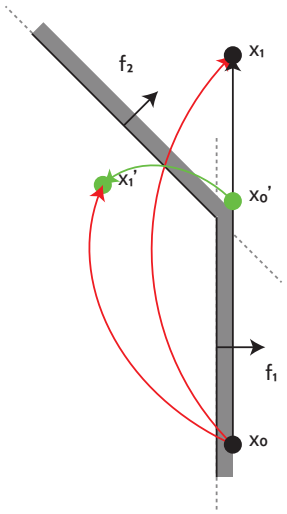
When a vertex slides off a face we have to distinguish two cases. The vertex crosses a *convex* or *concave* edge, see Figure 6.11. When a convex edge is crossed, the vertex in the beginning of the time-step lies behind the neighboring face. Once it crosses the edge, its distance becomes positive. When the original constraint is disabled, the vertex can potentially cross the neighboring face again, its signed distance then changes from positive to negative. In such a case, the root-finder will not find a root along that path because both the start and current configuration of the path lie behind the face. To handle those situations, we have to update the beginning configuration of the vertex such that it lies outside the first face (but on its plane), but also above the neighboring face. When the old constraint is disabled, the vertex now can cross the neighboring face. Since the start configuration now has a positive signed distance and the path clearly intersects the face, a collision is eventually detected and a constraint is enabled, see Figure 6.11b.

When the vertex crosses a concave edge, the vertex already crosses its neighboring face. Although it seems counter intuitive, such collisions are not always detected properly. When the vertex slides over the face, its signed distance will be positive with respect to the face. When it crosses a concave edge, the neighboring face detects a sign change from positive to negative for the vertex. Using a root-finder, the actual configuration at impact can be found. However, if the signed distance function is non-linear, then the root-finder might give unexpected results. For example, if the faces and vertex are in motion, the motion of the vertex relative to the faces will not be a straight line. The curved path of the vertex can intersect the neighboring face at a different location than expected. For example, the path of the vertex between the start and current configuration can intersect the neighboring face just outside the face. When such an intersection is found, it is discarded, because the intersection with the current face is reported, see Figure 6.11a. This in turn results in a non-detected collision with the neighboring face. To handle this case robustly, it is sufficient to enable the constraint between the neighboring face and the vertex once the vertex crosses the edge, without explicitly checking for a collision.

The same procedure applies to edge-edge cases. When two connected neighboring edges form a convex region the same procedure is followed as in the vertex-face case. The start configuration of the slid off edge is updated such that a collision with the neighboring edge can be properly detected. For concave regions, it is sufficient to enable a constraint for the neighboring edge-edge pair without detecting a collision explicitly.



(a) Concave transition: The signed distance of vertex $\mathbf{x}_0$ with face f_2 is initially *positive* and *negative* when the vertex has slid off face f_1 ($\mathbf{x}_1$). The intersection at the intermediate position (green) will not be detected in all cases. It is possible that the non-linear path results in a collision elsewhere (red). Therefore, a new constraint between the vertex and face f_2 is explicitly enabled.



(b) Convex transition: The signed distance of vertex $\mathbf{x}_0$ with face f_2 is initially *negative* and *positive* when the vertex slides off face f_1 ($\mathbf{x}_1$). After updating the initial configuration at the crossed edge (green $\mathbf{x}'_0$), the initial signed distance becomes *positive*. Later, after disabling the constraint between face f_1 and the vertex, the vertex now can move towards the second face and an intersection is detected (green) ($\mathbf{x}'_1$). The red paths show possible trajectories of the vertex without updating the initial configuration. None would be detected as a potential collision with face $f2$.

Figure 6.11: Sliding contacts. A vertex slides off a face. Depending on the convexity of the crossed edge, either a constraint is explicitly enabled (a), or a possible collision is enforced by updating the initial position (b).

The procedure for sliding contacts can be summarized as follows:

1 Assure that all constraints are resolved correctly.

2 Update all beginning configurations of each pair using the current configurations, only if the current configurations are not intersecting and have a positive signed distance.

3 Disable the original constraint if the contact crosses a convex region.

4 Enable a constraint for the neighboring pair once the contact crosses a concave region (if not already activated)

6.8 Conclusion

The method described in this chapter allows us to correctly detect collisions involving deformable models. The difficulty with deformable models is that the geometry can (locally) invert, the collision response changes the contact configuration and the geometry can change significantly within one time-step. If an accurate collision response for deformable models is required, these aspects should be taken into account.

Given these observations, finding all collisions in a correct way is a challenging task. Using a standard vertex-face collision detection based on signed distances usually gives reasonable results, but cannot guarantee a collision-free state because edge-edge contacts are missing. In order to include proper edge-edge contacts, their signed distances must be computed in a robust way. Thanks to our approach we are able to correctly compute signed distances and initialize constraints. Challenging cases, like inversions or objects that are flat, are correctly detected and handled. Additionally, the non-linear problem is solved by updating constraints a few times per time-step, which allows us to obtain a truly collision-free state at convergence. This guarantees a correct detection of collisions in later simulation steps. As demonstrated, this coincides with Newton's method for solving systems of non-linear equations.

6.8.1 Limitations

Although we have designed this approach to handle extreme deformations, there are a few drawbacks.

- When the method tests for intersections, not all intersections are immediately found. Due to the strict criteria of intersections, some cases are detected after collisions of neighboring geometry are resolved first. For example, an intersection of two concave edges is not detected, but is resolved by neighboring vertex-face constraints. As a result, a few iterations of collision detection on the candidate pairs, followed by a solve, is required before all collisions are found. A less strict definition of intersections would help here, but will cause problems in other situations.
- If large time-steps are used, also the displacement $\Delta t\mathbf{v}$ increases. Due to this, also the extended bounding boxes, described in Section 6.4, can become too large. Each feature in the mesh has such an extended box which is used in the actual broad-phase collision detection step. As a result, one face could have a large number of potential collisions with other faces, while only a few matter. This results in the creation of many candidate pairs in the candidate lists, which could result in a large memory consumption. This can be reduced by decreasing the extension of the extended bounding boxes, i.e., reducing $3\Delta t\mathbf{v}$ to something smaller. However, reducing it may result in missed potential collisions, see Section 6.5.3.

7

Conclusions

7.1 Conclusions

In the previous chapters we presented the individual parts of deformable and rigid-body simulations, partially performed on GPUs. This research contains a study on how to store sparse matrices on GPUs while maximizing the efficiency of Sparse-Matrix Vector Multiplications (SpMVs) and methods using these operations. That is, we analyzed their performance and presented a model for estimating it given some properties of the problem. Next we have used this machinery to perform a FEM simulation of deformable bodies in real-time using GPUs. We have introduced a novel method for simulating deformable and rigid bodies coupled through contact and friction in an accurate way. Finally we developed some tools for performing reliable collision detection between (deformable) objects, involving large deformations.

In the following paragraphs the conclusions of each individual chapter are given, followed by a general discussion on other possible applications of the obtained results. Next we reflect upon our work followed by a general discussion on future research directions.

Conjugate Gradient method on GPUs In Chapter 3 we have investigated a number of mappings for block-based SpMV operations on GPUs, using CUDA. *Block row mapping* maps one complete block row (a row containing a number of $N \times N$ matrix blocks) to one thread block. This method is straightforward to implement, but not very efficient, since a lot of computational resources are wasted. Within this mapping one thread block processes a large number of matrix blocks. By transposing the block row mapping, the *multiple block-row mapping* is obtained. This mapping assigns multiple block rows to one thread block, so that it processes a large number of matrix blocks, which belong to different block rows. This has positive implications on the performance, i.e., less thread blocks are needed and the amount of wasted computational resources is decreased. Furthermore, since each thread block processes a larger number of matrix blocks, better memory throughput was obtained and thus a better performance. This is in general only the case if $N \geq 4$. If $N < 4$ the data must be reordered to obtain efficient (coalesced) memory transactions for loading the matrix blocks. This *block reordering* significantly improves the performance of the SpMV operation for matrices stored using the BCSR layout with blocks of size $N < 4$, if the MBR mapping is used. Sorting the block rows such that block rows with similar lengths are processed by the same thread block, significantly increases performance.

By mapping the computations differently on the GPU, and by applying row sorting and block reordering, the best performances for the SpMV operation were obtained. Experimental results showed that our SpMV mapping outperforms existing methods in most cases, and performs close to the limits of the hardware. Our optimized SpMV operation was used to accelerate the CG method, given one or multiple GPUs. Together with the optimized vector operations, this makes (in most cases) our CG mapping about five times faster than existing methods.

We have also provided a recipe for estimating the maximum achievable performance and the average performance of a (parallel) CG method, given the properties of the problem. This method can be applied to similar numerical algorithms. Analyzing the memory throughput revealed a clear trend between the number of memory transactions and the performance. This analysis has been done for each kernel performing vector operations, as well as for the SpMV kernel. The resulting trends were modeled by a particular Sigmoid function, which was then used to estimate the memory throughput of each individual operation appearing in the CG method. Finally, this led to an approximation of the maximum or average performance of the

method. We further extended our performance-estimation framework such that also multiple GPU setups can be modeled. The results showed that our performance estimates were very close to the measured performance, and in general, the estimates became more accurate when larger blocks were used.

Chapter 3 mainly focused on performing the Conjugate Gradient method on GPUs since it is a widely used method in simulations of, e.g., elastically deformable models and simulations based on FEM. However, the same machinery can be applied to implement many related Krylov solvers on GPUs, like the Conjugate Residual method used in Chapter 5, as well as other numerical methods that heavily rely on a fast Sparse-Matrix Vector Multiplication. Furthermore, the analysis method can be applied to different memory-bound problems, can be used to identify bottlenecks in methods and to reason about the maximum reachable speed or data throughput for certain algorithms.

Deformable models on GPUs In Chapter 4 we have presented an efficient method for simulating elastically deformable models for graphics applications, accelerated on modern GPUs using CUDA. Our method relies on a fast Conjugate Gradient solver and an efficient mapping of the SPMV operation on modern GPUs presented in Chapter 3. Since the topology of the underlying grid does not change during the simulation, data structures are reused for higher efficiency. To further improve performance, we proposed a scheme which allows to efficiently update the sparse matrix, during the simulation.

The method performs all necessary computations on the GPU, which completely eliminates the need for transferring data from/to CPU memory. This strategy therefore maximizes the performance of such simulations when GPUs are involved.

Although we mainly focused on the simulation of elastically deformable models on GPUs, this research can be applied to many types of FEM-based simulations. In general the strategy for many FEM simulations is very similar. They all compute some local quantities per element, these are then assembled into a large system, which are then solved using a particular numerical method. As shown in the results, the computation of the local quantities is very fast because in general no additional information from neighboring elements is required. This information is propagated by assembling and solving the large system. We have demonstrated how linear elements can be applied, but also other types of elements with higher order interpolation or test functions can be used.

Efficient and Accurate collision handling In Chapter 5 a novel method is presented for simulating coupled deformable- and rigid-body simulations through contact and friction. Contact and friction are modeled using KKT multipliers (Appendix B.1.3), and additional friction forces. Where friction is usually linearized using a discretization of Coulomb's friction cone, usually an n-faceted pyramid, we model kinetic friction using an additional friction force that is aligned with the sliding velocity. This property ensures a maximum dissipation of energy and accuracy of the computed friction. Instead of solving the KKT problem by converting it to a Linear Complementarity Problem (LCP) and solving this using a Projected Gauss-Seidel method, we solve the underlying Mixed Linear Complementarity Problem (MLCP) directly through a Conjugate Residual (CR) method. The problem with the LCP approach is that it requires the inverse of the stiffness matrix of the simulated bodies. For rigid bodies this inverse can be computed instantly, for deformable bodies the computation of the inverse is computationally intensive.

We have compared a number of methods that are capable of solving the contact problem when deformable objects are involved. The compared methods usually (partially) decouple the computation of the dynamics of the simulated objects, contact response and friction response. Some methods perform a splitting of the stiffness matrix, resulting in a more computationally efficient construction of the approximated LCP. Other methods rely on a Cholesky factorization of the stiffness matrix in order to compute the inverse. Among the compared methods we found that the level of coupling between non-penetration and friction constraints significantly influences the efficiency of the methods. The tighter this coupling, the more efficient the method becomes. The method described in Chapter 5 enforces this tight coupling by updating constraint states and friction forces while searching for an optimum. As demonstrated, the CR method is capable of solving such problems, but in an unpreconditioned setting, the method becomes very inefficient due to a very slow convergence. To overcome this, a preconditioner was introduced that significantly improved the performance of the CR method applied to constrained optimization problems.

Since the problem is solved in its MLCP form, a relinearization of the constraints does not require a re-computation of an intermediate matrix, e.g., the LCP matrix or Delassus operator. Thanks to this, the non-linear contact problem can be solved efficiently, which improves accuracy and stability.

The method presented in Chapter 5 is not limited to contact mechanics alone, but can be applied to other problems involving (in)equality constraints and in which the system matrix is Symmetric and Positive Definite. Non-linear problems can also be solved efficiently with this method since it does not require to recompute intermediate matrices after the re-linearization of the problem. Similar problems can be for instance found in economics, physics or electrical engineering.

Collision detection for deformable models The method described in Chapter 6 allows us to correctly detect collisions involving deformable models, as used in Chapter 5. The difficulty with deformable models is that the geometry can (locally) invert, the collision response changes the contact configuration and the geometry can change significantly within one time-step. If an accurate collision response for deformable models is required, these aspects should be all taken into account.

Given these observations, finding all collisions in a correct way is a challenging task. Using a standard vertex-face collision detection based on signed distances usually gives reasonable results, but cannot guarantee a collision-free state because edge-edge contacts are missing. In order to include proper edge-edge contacts, their signed distances must be computed in a robust way. Thanks to our approach we are able to correctly compute signed distances and initialize constraints. Challenging cases, like inversions or objects that are flat, are correctly detected and handled. Additionally, the non-linear problem is solved by updating constraints a few times per time-step, which allows us to obtain a truly collision-free state at convergence. This guarantees a correct detection of collisions in later simulation steps. As demonstrated, this coincides with Newton's method for solving systems of non-linear equations.

7.2 Reflection

In this section we reflect upon our work to see how our work contributed to answering the research question. In Chapter 1 we divided the research question in four parts, see Figure 1.2. Here we discuss how the presented methods answers our research question.

- **Parallel acceleration:** *How can we accelerate both numerical methods and simulation methods such that we can solve/simulate large problems in a short amount of time using parallel hardware?*
  Parallel hardware and especially modern GPUs have a quite different architecture compared to traditional CPU architectures. Typically, a GPU contains many 'simple' processors, able to perform operations in a *Single Instruction Multiple Data* approach. One instruction is typically executed by many processors and each processor works on a specific part of the data. In order to maximize the performance for certain numerical methods, the processors need a steady stream of data from memory. The key in optimizing the performance on GPUs is by streaming the data to the processors in a particular pattern, such that each processor immediately can start computing, instead of first collecting and/or reordering its data. This can be achieved by mapping the computations on the available processors in a specific way, which implicitly reorders the data in memory such that the maximum data throughput-rate can be approached. In Chapter 3 we showed the relation between the performance and memory throughput, and used this principle to maximize the performance of linear algebra operations performed on sparse matrices and vectors, leading to a GPU implementation of the Conjugate Gradient method. Additionally, the same methodology was used in Chapter 4 – performing a FEM simulation solely on a GPU.

- **Performance analysis:** *How can we provide tools for reasoning about the efficiency of (numerical) methods performed on parallel hardware?*
  Given the answers from the previous question, one can ask the question: what is the maximum performance one could obtain for certain problems and how efficient is this? Given the observations from the previous question, a model was made that modeled the maximum memory throughput for a certain problem size. Since the memory throughput and the performance are closely related in memory-bound problems, we were able to estimate this theoretical peak-performance given the problem-size. This model was extended to also model the communication between GPUs, which visualizes *Amdahl's law*. When a certain problem is solved using multiple GPUs, each GPU needs to communicate its results to the other GPUs. Additionally, since each sub-problem was smaller than its original problem, each GPU also performed less efficient. This demonstrated that achieving a good scalability of numerical methods performed on GPUs is not straightforward.

- **Computational efficiency:** *How can we simulate coupled simulations in an efficient way?*
  Coupled simulations couple the motion of one object to the motion of another object. Usually such a coupling is modeled through *penalty forces*, which are not 100% collision free. The use of *Lagrangian / KKT Multipliers* results in an accurate coupling, which is able to resolve interpenetrations. When solving such constrained problems, the intermediate *Linear Complementarity Problem* is usually too computationally-intensive to obtain. Instead of solving the problem through this LCP, the underlying MLCP is efficiently solved, as described in Chapter 5. Additionally, by actively switching the state of inequality constraints, the method converges faster.

- **Accuracy and stability:** *How can we increase the accuracy and stability of (coupled) simulations?*
  Simulations of rigid and deformable bodies can become unstable. For example, stacking rigid objects is still a difficult problem. The source of this problem originates from contact and friction constraints that are not resolved properly. This could potentially add new energy to the system, resulting in oscillations and non-physical motions. Additionally, when object are colliding, these collisions must be detected properly. Collisions may trigger or invalidate other collisions. In Chapter 6 we have developed a method for correctly detecting collisions when deformable objects are involved. Furthermore, thanks to the efficiency of the method described in Chapter 5, we are able to solve the actual non-linear problem, which guarantees a collision free state at the end of a time-step is obtained and consequently the system reaches a true equilibrium. Another important aspect of our approach is the treatment of kinetic friction, which is modeled as a continuous force that is always aligned with the motion of the objects in contact. All these aspects contribute to both accuracy and stability.

7.2.1 Reusability and Further Integration

The methods described in each chapter can be reused in other applications. Chapter 3 delivers a stand-alone Conjugate Gradient solver that can be executed on multiple GPUs. Although we have focused on the CG method, it is straightforward to use the building blocks for implementing other related Numerical Methods. The most important part here is the storage scheme of sparse matrices and the SpMV operation, and the parallelization among multiple GPUs. The corresponding method for performance analysis could be applied to similar memory-bound problems, since the analysis method tries to model a theoretical memory throughput. Therefore this can also be used for identifying bottlenecks in computations on GPUs. Chapter 4 provides a stand-alone GPU based simulation of FEM based deformable models and uses the building blocks described in Chapter 3. Since all operations on the individual elements in the tetrahedral mesh are performed on a GPU, these operations can also be applied to other problems. Chapter 5 provides a numerical method for solving optimization problems involving (in)equality constraints and can be used to solve various problems. The method mainly focused on solving contact and friction in an accurate way. However, when less accurate collision responses are sufficient, then a less accurate version of this method could be used in interactive settings. For example, by replacing the collision detection method by a simpler and less accurate method, one could apply constraint-based contact in real-time. When combined with the building blocks from Chapter 3 and the FEM simulation of Chapter 4, a real-time simulation of deformable and rigid-bodies in contact is possible. Chapter 6 describes a method for detecting collisions involving deformable objects. This method is combined with the numerical method described in Chapter 5 and could be combined with other methods requiring collision detection. In fact, this method is used in combination with the methods from Chapter 5 to perform a comparison that demonstrates its reusability and integration in existing methods.

Further integration The current chapters can be further integrated by performing the complete simulation pipeline on GPUs. Chapters 3 and 4 only have this pipeline partially implemented on GPUs. By integrating Chapters 3 and 4, and a GPU version of Chapter 6 into Chapter 5, a fast and accurate simulation of deformable and rigid bodies in contact would be possible, which could have many applications, ranging from games, VR to haptics applications, see, e.g., [112, 186].

Other applications The machinery presented in this dissertation can also be applied to other subdisciplines within computer graphics. For example, the rendering method *Radiosity* [51] could benefit from a fast GPU method for solving large linear systems. Furthermore, many problems found in computer graphics use optimization methods for finding the optimal solution of a certain problem. In many cases a large linear system is obtained, possibly in combination with Lagrange multipliers. Such problems are typically found in the fabrication and design of particular objects or systems that have to fulfill a number of constraints on the shape or other physical properties. Many of these methods could benefit from our work.

7.3 Looking Forward

After concluding and reflecting on our work, we have to look forward and consider interesting research directions. In the previous section we have described how we can combine the different methods described in this dissertation. In this section we like to describe a few directions for future research in general.

Accelerating Numerical Methods We have implemented methods for *accelerating* Numerical Methods using GPUs and identified the bottlenecks in a few methods and operations. In many methods, Sparse-Matrix Vector Multiplications form a certain bottleneck on GPUs due to the memory latencies and difficulties in order to saturate the memory bus. If the memory bus was saturated, then the maximum performance obtained was significantly less than the peak-performance of such devices. For example, the Nvidia GeForce GTX280 GPU used in those experiments had a theoretical peak-performance of 933 GFlops and a memory throughput of 141 GB/s. For the SpMV operation we were able to obtain a maximum performance less than 34 GFlops. The current generation Nvidia GeForce RTX280 GPU has a theoretical peak-performance of 8920 GFlops and a memory throughput of 448 GB/s. Although the peak-performance is an order of magnitude larger, the memory throughput only increased slightly more than three times. For memory-bound problems this means that modern GPUs are even less efficient compared to older GPUs, by only looking at these numbers. Furthermore, a similar trend is observed in the development of CPUs. The clock-frequencies used in CPUs are reaching a limit. Making CPUs faster by increasing the clock frequency alone is therefore very difficult. Instead, modern CPUs are currently small parallel computers having multiple smaller processors used for numerical operations. Additionally, such CPUs in combination with GPUs are used in large grids in which, ideally, each processor works on solving the same large problem. This raises basically two questions: *How can we use the full potential of current generation GPUs and CPUs for accelerating Numerical Methods?* and *Can we design/use other non memory-bound numerical methods for solving systems?*

Computational efficiency and accuracy We have investigated efficient methods for solving contact problems by eliminating the need for computing and solving the intermediate problems. We have demonstrated this for deformable bodies based on FEM. Although this approach is significantly faster than existing methods, we have not tested this with other types of simulations. Especially simulations with a large system matrix could potentially benefit from such methods. Furthermore, knowing that we can solve non-linearly-coupled systems in an efficient way, this potentially opens the possibility for simulating other complex non-linearly-coupled phenomena, involving large systems. Furthermore, it is interesting to solve the non-linearity also for the underlying object simulations. In the work presented in this dissertation, these simulations were linearized only once per time-step. Within physics based animation methods found in Computer Graphics, particle based methods in combination with

constraints (Position Based Dynamics) are receiving much attention, as well as hybrid methods like the Material Point Method (MPM) that is used to simulate various phenomena. These methods could potentially benefit from our methods. Especially when MPM is coupled with other kinds of simulations, they could potentially benefit from our research. The accuracy of this coupling or transfer between both simulations is of key importance to obtain accurate enough final results. Apart from using existing methods found in physics based animation, investigating new simulation methods that maximize the efficiency on parallel architectures in general is an interesting direction of research.

Furthermore, one major improvement of the performance in the constrained optimization method was obtained through the presented preconditioner. Although the performance increase was significant, we believe that clever preconditioning schemes may improve the performance of such methods even further.

A

Preconditioner scaling

A.1 Introduction

The Full preconditioner described in Chapter 5 works well, in general, but can introduce large errors in some cases. This is not directly caused by the numerical properties of the individual systems, but merely by the numerical properties of the corresponding Schur complement matrix or LCP matrix. When an MLCP is converted to an LCP, this Schur complement matrix is computed and appears in the solution of the problem, see Equation (B.13). It is known that Schur complement matrix will have a bad conditioning when multiple (more or less) identical rows appear in $\mathbf{J}$. When this happens, numerical methods will have difficulties in solving the corresponding linear system for the unknown multipliers. This happens when a particular collision creates a few nearly identical constraints, e.g., when a vertex intersects a face close to one of its vertices. Such an event will likely create intersections of the edges connected to the involved vertices. In this case, a vertex-face constraint is created and added to the system, and so will constraints for neighboring intersecting edges. All constraints created by this event are very similar and almost identical. When all constraints are added to $\mathbf{J}$, the Schur complement matrix $\mathbf{J}\mathbf{A}^{-1}\mathbf{J}^T$ will have a bad conditioning or will even become singular in case $\mathbf{J}$ becomes rank deficient. Such a system can be inverted using a Singular Value Decomposition, as shown in [196], but can introduce large errors in subsequent computations. The singularity can be described by a few contact forces for which an infinite number of solutions exist, but only the sum of all contact forces is applied via $\mathbf{J}^T\boldsymbol{\lambda}$. So, each solution $\boldsymbol{\lambda}$ of Equation (B.13) is valid. Fortunately, the method in Chapter 5 does not compute or solve such a Schur complement matrix, and will properly converge even if $\mathbf{J}$ contains duplicates, since it actually solves a Least-Squares problem.

When two deformable bodies collide, the involved constraints act on the velocities of a few vertices of both models. If one collision results in duplicate constraints, the amount of duplicate constraints is relatively low. This is because a collision only generates (close to) duplicate constraints if a vertex collides with a face close to one of the vertices of the face. When both objects are rigid bodies, the amount of duplicate constraints can become much larger. A constraint on a rigid body directly acts on its linear and angular velocities, i.e., all constraints acting on the same rigid body share the same entries in $\mathbf{A}$. Collisions that are close to each other, create very similar constraints for rigid bodies, despite the fact that different vertices in the model are involved. The duplicity in this case is not bounded, but scales with the amount of constraints created between the two rigid objects. The finer the meshes used for both models, the more constraints are created that act on the same entries in $\mathbf{A}$.

A.2 Rank Deficiency in Preconditioner

The method described in Chapter 5, does, in principle, not have problems when $\mathbf{J}$ contains duplicates. However, the used preconditioner relies on an approximation of the block-inverse of the system matrix $\mathbf{B}$, see Equation (5.11). This block-inverse contains the inverted Schur complement matrix, see Equation (5.22). Due to this, the preconditioned CR method will fail if the preconditioner uses a singular matrix. Fortunately, a diagonal approximation of this Schur complement matrix is used which is not singular, see Equation (A.1). Due to this, we are only left with a scaling error in the preconditioner, as we will show in this section.

To illustrate the problem, we use a small example that shows the error in the preconditioner when $\mathbf{J}$ is rank deficient. In our example we set $\mathbf{A}$ to the identity matrix, $\mathbf{J} = (1, 1)$ and compute the preconditioner matrix as

$$\mathbf{B}^{-1} \approx \begin{pmatrix} \mathbf{A}_d^{-1} - \mathbf{A}_d^{-1}\mathbf{J}^T\mathbf{S}_d^{-1}\mathbf{J}\mathbf{A}_d^{-1} & (\mathbf{S}_d^{-1}\mathbf{J}\mathbf{A}_d^{-1})^T \\ \mathbf{S}_d^{-1}\mathbf{J}\mathbf{A}_d^{-1} & -\mathbf{S}_d^{-1} \end{pmatrix}. \tag{A.1}$$

We start with the inverse of the Schur complement matrix: $\mathbf{S}_d^{-1} = (\mathbf{J}\mathbf{A}_d^{-1}\mathbf{J}^T)^{-1}$, i.e.,

$$\left(\begin{pmatrix} 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}^{-1} \begin{pmatrix} 1 \\ 1 \end{pmatrix} \right)^{-1} = \frac{1}{2}. \tag{A.2}$$

Next, $\mathbf{S}_d^{-1}\mathbf{J}\mathbf{A}_d^{-1}$ is computed, resulting in $\left(\frac{1}{2}, \frac{1}{2}\right)$. Using this result $\mathbf{A}_d^{-1}\mathbf{J}^T\mathbf{S}_d^{-1}\mathbf{J}\mathbf{A}_d^{-1}$ is computed and becomes:

$$\begin{pmatrix} \frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{1}{2} \end{pmatrix}. \tag{A.3}$$

Now the final form of the preconditioner is computed using Equation (A.1), resulting in:

$$\left(\begin{array}{cc|c} \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{1}{2} & \frac{1}{2} \\ \hline \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} \end{array}\right), \tag{A.4}$$

which is also the exact inverse of the described system. Similarly, the preconditioner can be obtained for the case with a duplicate constraint, i.e., $\mathbf{J}$ now contains two identical rows. Using the same procedure as before, the preconditioner now becomes:

$$\left(\begin{array}{cc|cc} 0 & -1 & \frac{1}{2} & \frac{1}{2} \\ -1 & 0 & \frac{1}{2} & \frac{1}{2} \\ \hline \frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & 0 \\ \frac{1}{2} & \frac{1}{2} & 0 & -\frac{1}{2} \end{array}\right). \tag{A.5}$$

Please note that $\mathbf{S}$ becomes singular in this case, but its diagonal approximation $\mathbf{S}_d$ is invertible. The preconditioner in Equation (A.5) is different compared to the one in Equation (A.4), especially the values in the upper-left block show a large deviation. When two identical constraints appear in $\mathbf{J}$, the total contribution is divided over both individual constraints. So in the case of two duplicates, their corresponding response force should be divided over the constraints such that the net force stays the same. This also suggests that the effect of the preconditioner on the rest of the problem should be invariant for the number of duplicates, i.e., the upper-left block should not change in this case. This becomes clear when duplicated constraints are removed by adding them together, i.e., $\mathbf{J}$ now becomes $(2, 2)$. Using the same procedure, the approximate preconditioner can be computed for this system, i.e.,

$$\left(\begin{array}{cc|c} \frac{1}{2} & -\frac{1}{2} & \frac{1}{4} \\ -\frac{1}{2} & \frac{1}{2} & \frac{1}{4} \\ \hline \frac{1}{4} & \frac{1}{4} & -\frac{1}{8} \end{array}\right). \tag{A.6}$$

The upper-left-block is now exactly the same as in Equation (A.4), which confirms that the upper-left block should not depend on the amount of duplicate constraints. By adding duplicate constraints together, the upper-left-block is restored. The other off-diagonal blocks are scaled by the number of duplicates, while the lower-right block is scaled by the square of this number. Since the right-hand-side of the involved constraints are also multiplied by two, the computed result is exactly the same as in the non-duplicate case, except that the computed multipliers for this system are divided by the number of duplicates. Since the scaled-down multiplier is used with two 'added' constraints, the net result is exactly the same.

A.2.1 Preconditioner Scaling

In the previous section we have shown that in case of duplicate constraints the upper-left-block of the preconditioner introduces a too large error. This can be compensated by adding duplicate constraints together. However, adding constraints together is only possible when two constraints are exactly the same. When they are similar, this procedure is not possible, therefore we propose a technique that allows one to compute a good approximation of the block-inverse as shown in Equation (A.1), when (near) duplicate constraints exist.

Similar to [180], we propose to distribute the 'mass' over the constraints. Instead of dividing the entries in $\mathbf{A}$ used in the construction of $\mathbf{S}$, we divide the entries $\mathbf{S}_d$ by the number of similar constraints $\mathbf{Z}$ for the involved constraints. The scaled instance of $\mathbf{S}_d$ is also used in the other terms appearing in the preconditioner. To illustrate this, we apply this procedure to the system in which $\mathbf{J}$ contains a duplicate row. Since both constraints have a similarity of 2, each entry on the diagonal of $\mathbf{S}_d$ is scaled by this number, i.e., the scaled instance $\mathbf{S}_{dz}$ becomes

$$\mathbf{S}_{dz} = \mathbf{S}_d\mathbf{Z} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}^{-1} = \begin{pmatrix} \frac{1}{4} & 0 \\ 0 & \frac{1}{4} \end{pmatrix}, \tag{A.7}$$

with $\mathbf{Z}$ a matrix containing for each constraint the number of similar constraints. Next, $\mathbf{S}_{dz}$ is used to replace $\mathbf{S}_d$, resulting in the following preconditioner:

$$\left(\begin{array}{cc|cc} \frac{1}{2} & -\frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ -\frac{1}{2} & \frac{1}{2} & \frac{1}{4} & \frac{1}{4} \\ \hline \frac{1}{4} & \frac{1}{4} & -\frac{1}{4} & 0 \\ \frac{1}{4} & \frac{1}{4} & 0 & -\frac{1}{4} \end{array}\right). \tag{A.8}$$

The upper-left block of the preconditioner is identical to the one in Equation (A.6). The other blocks are scaled by the number of duplicates. Now, by adding the rows and columns corresponding to duplicate constraints in $\mathbf{J}$ and $\mathbf{J}^T$, the obtained system becomes exactly the one from Equation (A.4). This implies that scaling $\mathbf{S}_d$ by $\mathbf{Z}$ yields a system that behaves exactly like the non-duplicate case, except that the magnitude of the multipliers are evenly distributed among the duplicates, with the same net result. Using this strategy the unintended scaling of the upper-left block in the preconditioner matrix is compensated by an additional matrix $\mathbf{Z}$.

A.3 Similarity Measurement

In the previous section we have shown that scaling matrix $\mathbf{S}_d$ with $\mathbf{Z}$ brings the preconditioner matrix closer to the real inverse of the problem. In case of real duplicate constraints, the scaling used is exactly the number of duplicates per constraint. However, a large number

of similar constraints can be created during a collision event and originate from close-by intersections. Many identical or similar constraints introduce an error in the upper-left block of the preconditioner. To compensate for this, we can scale the corresponding entries in $\mathbf{S}_d$ by $\mathbf{Z}$ using a similarity measure of the constraints. In order to obtain this similarity measure for a particular constraint, we have to distinguish between the type of constraints, i.e., constraints between rigid-rigid, rigid-deformable and deformable-deformable objects. By maintaining a map with all connections between individual constraints and their corresponding indices in matrix $\mathbf{A}$ and $\mathbf{J}$, a set T_a is obtained for each constraint a, containing all constraints that have at least one column in $\mathbf{A}$ in common.

A.3.1 Non-Penetration Constraints

In order to find similar constraints, let us first describe the constraints encountered in a simulation. Two bodies have collided if the signed distance between any set of two surface points is negative. Let $\mathbf{x}_o \in \Gamma_1$ and $\mathbf{x}_p \in \Gamma_2$ these surface points. The (signed) distance d_k between $\mathbf{x}_o$ and $\mathbf{x}_p$ is computed using

$$C_k(\mathbf{x}_o, \mathbf{x}_p) = (\mathbf{x}_p - \mathbf{x}_o) \cdot \mathbf{n} = d_k \geq 0, \tag{A.9}$$

with $\mathbf{n}$ the *contact normal*. Since Equation (5.11) solves for $\mathbf{v}^{i+1}$, Equation (A.9) is transformed into a velocity constraint using a first-order approximation, i.e.,

$$C_k^{i+1} = C_k + \Delta t \frac{\partial C_k}{\partial t} \geq 0, \tag{A.10}$$

which in matrix notation becomes:

$$\left(\Delta t \frac{\partial C_k}{\partial \mathbf{x}_o}, \ \Delta t \frac{\partial C_k}{\partial \mathbf{x}_p} \right) \begin{pmatrix} \mathbf{v}_o \\ \mathbf{v}_p \end{pmatrix} \geq -d_k = c_k. \tag{A.11}$$

All instances of Equation (A.11) are stored as

$$\mathbf{J}\mathbf{v}^{i+1} \geq \mathbf{c} \tag{A.12}$$

with $\mathbf{J}$ the Jacobian matrix.

Rigid-rigid contact Suppose $\mathbf{x}_o$ is a point on a rigid body, which is defined as:

$$\mathbf{x}_o = \mathbf{x}_{c,o} + \mathbf{r}_o, \tag{A.13}$$

with $\mathbf{x}_{c,o}$ the center of mass and $\mathbf{r}_o$ a vector from $\mathbf{x}_{c,o}$ to point $\mathbf{x}_o$ on the surface of the body. The signed distance between two points o and p on the surface of the rigid-bodies is defined as:

$$C_k(\mathbf{x}_o, \mathbf{x}_p) = (\mathbf{x}_{c,p} + \mathbf{r}_p - \mathbf{x}_{c,o} - \mathbf{r}_o) \cdot \mathbf{n}. \tag{A.14}$$

Similar to Equation (A.11), this constraint is transformed into a constraint on the velocity as:

$$C_k^{i+1} = C_k + \Delta t \frac{\partial C_k}{\partial \mathbf{x}_{c,o}} \frac{\partial \mathbf{x}_{c,o}}{\partial t} + \Delta t \frac{\partial C_k}{\partial \mathbf{r}_o} \frac{\partial \mathbf{r}_o}{\partial t} + \Delta t \frac{\partial C_k}{\partial \mathbf{x}_{c,p}} \frac{\partial \mathbf{x}_{c,p}}{\partial t} + \Delta t \frac{\partial C_k}{\partial \mathbf{r}_p} \frac{\partial \mathbf{r}_p}{\partial t}, \tag{A.15}$$

with $\frac{\partial \mathbf{r}_p}{\partial t} = \boldsymbol{\omega}_p \times \mathbf{r}_p$ for vector $\mathbf{r}_p$ and $\boldsymbol{\omega}_p$ its angular velocity. The term $\Delta t \frac{\partial C_k}{\partial \mathbf{r}_p} \frac{\partial \mathbf{r}_p}{\partial t}$ is then computed using:

$$\Delta t \frac{\partial C_k}{\partial \mathbf{r}_p} \frac{\partial \mathbf{r}_p}{\partial t} = \Delta t \mathbf{n}^T \boldsymbol{\omega}_p \times \mathbf{r}_p = \Delta t (\mathbf{r}_p \times \mathbf{n})^T \boldsymbol{\omega}_p. \tag{A.16}$$

The other term regarding $\mathbf{r}_o$ is computed similarly. Given Equation (A.15), the velocity constraint in matrix format is given as:

$$\left(-\Delta t\mathbf{n}^T,\ -\Delta t(\mathbf{r}_o \times \mathbf{n})^T,\ \Delta t\mathbf{n}^T,\ \Delta t(\mathbf{r}_p \times \mathbf{n})^T\right)\begin{pmatrix}\mathbf{v}_o\\ \boldsymbol{\omega}_o\\ \mathbf{v}_p\\ \boldsymbol{\omega}_p\end{pmatrix} \geq -d_k = c_k, \tag{A.17}$$

which is added to the system $\mathbf{J}\mathbf{v}^{i+1} \geq \mathbf{c}$.

Deformable-deformable contact Similarly, the signed distance of a vertex-face pair between two deformable bodies is described using:

$$C_k(\mathbf{x}_o, \mathbf{x}_p) = (\mathbf{x}_p - w_a\mathbf{x}_{o,a} - w_b\mathbf{x}_{o,b} - w_c\mathbf{x}_{o,c}) \cdot \mathbf{n}, \tag{A.18}$$

with $\mathbf{x}_p$ a vertex of object p and $\mathbf{x}_o$ a point on the surface triangle (a, b, c) of object o, with w its barycentric coordinates. Similarly the velocity constraint is obtained, i.e.,

$$\left(\Delta t\mathbf{n}^T,\ -\Delta t w_a\mathbf{n}^T,\ -\Delta t w_b\mathbf{n}^T,\ -\Delta t w_c\mathbf{n}^T\right)\begin{pmatrix}\mathbf{v}_p\\ \mathbf{v}_{o,a}\\ \mathbf{v}_{o,b}\\ \mathbf{v}_{o,c}\end{pmatrix} \geq -d_k = c_k, \tag{A.19}$$

with $\mathbf{v}$ the velocities of the involved vertices. Depending on the type of collision, face-vertex or edge-edge combinations are created. In case of collisions between rigid and deformable objects, the signed distance is defined exactly as in Equation (A.9), but either $\mathbf{x}_o$ is regarded as a point on the surface of the rigid-body, while $\mathbf{x}_p$ is a point on the surface of the deformable surface, or $\mathbf{x}_o$ is a point on the deformable surface while $\mathbf{x}_p$ is a point on the rigid surface. Given the origin of the points in contact, the derivation of the velocity constraint follows the same procedure as described above, but treats deformable and rigid points differently. Furthermore, in order to obtain the tangential friction constraints, the contact normal $\mathbf{n}$ is replaced by the tangential vectors $\mathbf{t}_1$ and $\mathbf{t}_2$, resulting in three constraints per contact-point.

A.3.2 Rigid-Rigid Constraints Scaling

A constraint between two rigid objects separates the objects at a certain location on both surfaces. This location is the point of contact, which is the same for both objects. Since this point of contact is not directly related to a vertex or face of the surface, a contact point is expressed using the degrees of freedom for each object. In case of two colliding rigid bodies, a large number of contact points can exist between the surfaces of the colliding objects. Each contact point results in a constraint that is acting on the same linear and angular velocities of both involved objects, i.e., all constraints described by Equation (A.17) have their entries at the same columns of $\mathbf{J}$.

Depending on the location of the contact point, vectors $\mathbf{r}_o \times \mathbf{n}$, $\mathbf{r}_p \times \mathbf{n}$ and $\mathbf{n}$ can be different from other constraints. To determine if two constraints are equal, the difference between vectors that coincide with the same columns in $\mathbf{J}$ are subtracted from each other, i.e.,

$$\begin{aligned}\mathbf{d}_{nb} &= \mathbf{n}_a - \mathbf{n}_b\\ \mathbf{d}_{rpb} &= \frac{\mathbf{r}_{p,a} \times \mathbf{n}_a}{\|\mathbf{r}_{p,a} \times \mathbf{n}_a\|} - \frac{\mathbf{r}_{p,b} \times \mathbf{n}_b}{\|\mathbf{r}_{p,b} \times \mathbf{n}_b\|}\\ \mathbf{d}_{rob} &= \frac{\mathbf{r}_{o,a} \times \mathbf{n}_a}{\|\mathbf{r}_{o,a} \times \mathbf{n}_a\|} - \frac{\mathbf{r}_{o,b} \times \mathbf{n}_b}{\|\mathbf{r}_{o,b} \times \mathbf{n}_b\|}\end{aligned} \tag{A.20}$$

where b indicates a 'connected' constraint stored in the set T_a. Please note that $\mathbf{r}_{p,a} \times \mathbf{n}_a$ can be zero, hence normalizing those vectors should be done with care. Next, these distances are used to compute the final similarity, i.e.,

$$s_{a,b} = \left(1 - \sqrt{\mathbf{d}_{nb}^T \mathbf{d}_{nb}}\right)^+ \left(1 - \sqrt{\mathbf{d}_{rpb}^T \mathbf{d}_{rpb}}\right)^+ \left(1 - \sqrt{\mathbf{d}_{rob}^T \mathbf{d}_{rob}}\right)^+, \tag{A.21}$$

with $()^+$ truncating negative numbers to 0. When two constraints are identical, $s_{a,b} = 1$. Contrary, when constraints are different, $s_{a,b} = 0$ and are between 0 and 1 if they are similar.

A.3.3 Rigid-Deformable and Deformable-Deformable Constraints Scaling

When a rigid and deformable body collide, many contact points can be created between the surfaces of both objects. Since constraints between the same rigid and deformable object have the same columns in $\mathbf{J}$ for the rigid body, but may have different columns for the entries corresponding with the vertices of the deformable model, we need to find constraints in set T_a with the same column indices for the rigid body and at least one similar column index for the deformable object. When the column indices for the rigid body are equal and at least one vertex of the deformable object are similar for two constraints, there is a possibility that the constraints are similar or identical. A similar approach is used for constraints describing a contact involving two deformable models. Given the set T_a of similar constraints, constraints are selected that have for both parts in the constraint at least one column index in common. Given such a pair, their similarity is computed. The following procedure computes this similarity or constraint scaling and is the same for both rigid-deformable and deformable-deformable constraints.

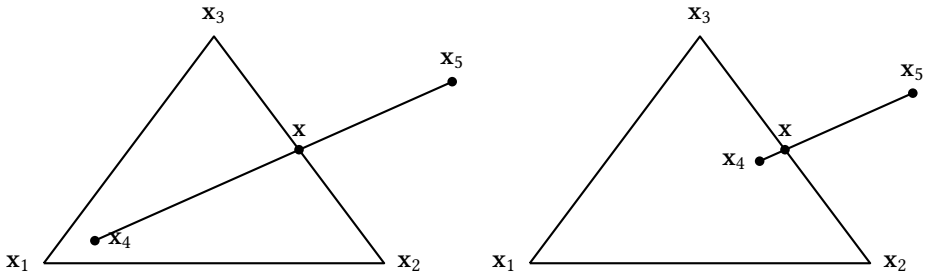
Given constraint a and b in set T_a, the computed similarity should be 1 in case the constraints are identical, 0 when they are orthogonal, and somewhere between 0 and 1 if constraints a and b act on the same sub-set of vertices, but with different weights. By computing the similarity for each constraint in set T_a, the total number of similar constraints is approximated. Since constraint a is also an element of T_a, the computed similarity is always equal or larger than one. Given constraints a and b, their barycentric weights corresponding to each vertex in the deformable models can be extracted. After that, the absolute difference between each barycentric coordinate corresponding to the same vertex in both constraint a and b is computed. Subtracting this difference from 1 will give a weight of 1 if the weights are identical. By multiplying these weights, the similarity between constraint a and b is computed, i.e.,

$$s_{a,b} = \prod_{i=1}^{\#w} \left(1 - |w_{a,i} - w_{b,i}|\right). \tag{A.22}$$

When all weights of both constraints are equal, this equation computes a similarity of 1 between constraint a and b. When the barycentric weights of both constraints are orthogonal, the similarity is 0. In all other cases when the contact points described by both constraints share a few vertices, the similarity lies between 0 and 1. When the constraints act between a rigid and deformable body, the same procedure is followed, except that the entries corresponding to the rigid body can be ignored. This reduces the number of weights to compare.

A.3.4 Total Scaling

Once all similarity values $s_{a,b}$ for each pair of constraints, are computed, the final scaling z_a for each constraint a is computed using $z_a = \sum s_{a,b}$ for all $b \in T_a$. Next z_a is stored in $\mathbf{Z}$ at the diagonal component of $\mathbf{Z}$ corresponding to constraint a.



(a) Less similar case, $s_{a,b} = 0.018$ **(b)** Similar case, $s_{a,b} = 0.52$

Figure A.1: A vertex-face and edge-edge constraint acting on the same subset of vertices. Vertex $\mathbf{x}$ is the intersection point of the edges involved in the edge-edge constraint.

A.3.5 Examples

Figure A.1a shows an example in which a vertex-face constraint is active between vertex $\mathbf{x}_4$ and face $\mathbf{x}_1\mathbf{x}_2\mathbf{x}_3$, and an edge-edge constraint between edge $\mathbf{x}_2\mathbf{x}_3$ and edge $\mathbf{x}_4\mathbf{x}_5$. The vertex-face constraint has barycentric weights $(0.8, 0.1, 0.1, -1)$ and the edge-edge constraint has barycentric weights $(0.5, 0.5, -0.5, -0.5)$. Given these weights and the corresponding indices, the similarity is computed as follows:

$$\begin{aligned}
&(1-|0.8-0|)\times(1-|0.1-0.5|)\times(1-|0.1-0.5|)\times(1-|-1--0.5|)\times(1-|0-0.5|)= \\
&(1-0.8) \quad \times(1-0.4) \quad \times(1-0.4) \quad \times(1-0.5) \quad \times(1-0.5) \quad = \\
&(0.2) \quad \times(0.6) \quad \times(0.6) \quad \times(0.5) \quad \times(0.5) \quad = \\
&0.018.
\end{aligned} \tag{A.23}$$

Since $\mathbf{x}_4$ and $\mathbf{x}$ are relatively far from each other, the similarity is low. Figure A.1b shows a similar configuration. The vertex-face constraint now has weights $(0.1, 0.45, 0.45, -1)$ and the edge-edge constraint $(0.5, 0.5, -0.8, -0.2)$. The similarity is now computed as:

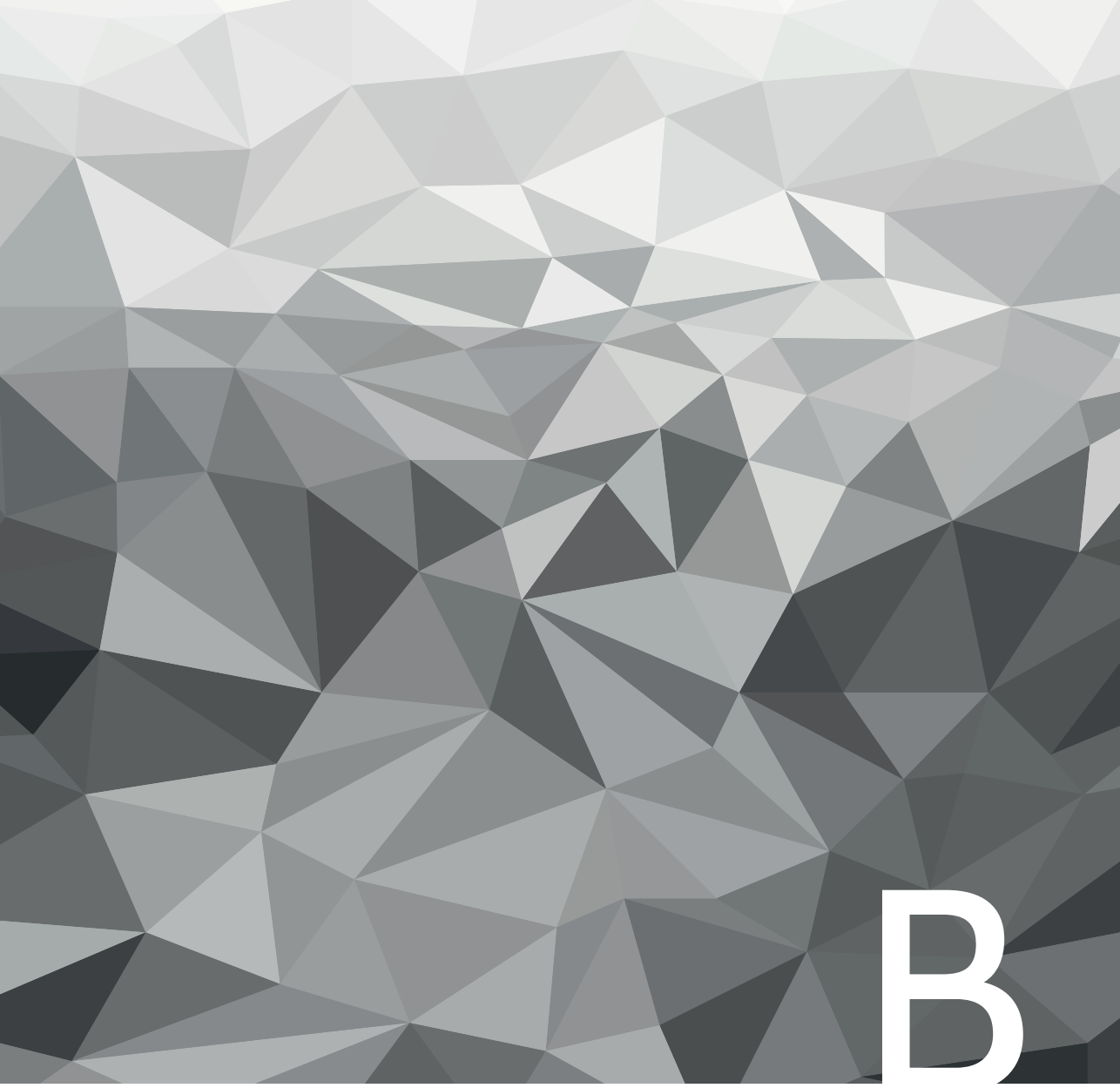
$$\begin{aligned}
&(1-|0.1-0|)\times(1-|0.45-0.5|)\times(1-|0.45-0.5|)\times(1-|-1--0.8|)\times(1-|0-0.2|)= \\
&(1-0.1) \quad \times(1-0.05) \quad \times(1-0.05) \quad \times(1-0.2) \quad \times(1-0.2) \quad = \\
&(0.9) \quad \times(0.95) \quad \times(0.95) \quad \times(0.8) \quad \times(0.8) \quad = \\
&0.52.
\end{aligned} \tag{A.24}$$

Since $\mathbf{x}$ and $\mathbf{x}_4$ are closer to each other, the similarity score is larger. By moving $\mathbf{x}_4$ towards $\mathbf{x}$ eventually results in a score of 1. Please note that when a constraint has no weight defined for a particular vertex, 0 is used to compute the difference. Furthermore, the total scaling $\mathbf{Z}$ is the total sum of the similarity scores $s_{a,b}$ for all nearby constraints stored in set T_a. Since this set also contains constraint a, each computed similarity for a constraint is at least 1.

A.4 Conclusion

The method described in this appendix reduces the error in the preconditioner caused by duplicate or similar constraints. Without this correction, the upper-left block of the Full preconditioner has a significant difference compared to cases in which no duplicate constraints exist. Due to this error, the Preconditioned Conjugate Residual method will converge much slower when many similar constraints are active between two rigid bodies. The same problem is observed with similar constraints originating from rigid-deformable or deformable-deformable

contacts, although the observed error is less prominent because most constraints act on different vertices. Due to this, the chance of creating constraints between the same degrees of freedom is smaller, but still possible. With this correction we noted that the Preconditioned Conjugate Residual method had no convergence difficulties in cases with many similar or duplicate constraints between the degrees of freedom. Although this approach worked well in the cases we have observed, a thorough study should be performed on the computation of the weights in Equations (A.21) and (A.22) and their effect on the convergence in different cases.



B

Optimization

B.1 Introduction

Mathematical Optimization is a large and long studied topic in mathematics and spans almost a century of research. Today many methods found in Computer Graphics and Computer Science in general, heavily rely on some form of optimization. Without giving a complete overview of optimization methods, we briefly introduce the methods closely related to the chapters in this dissertation. For a broader overview of Mathematical Optimization we refer to, e.g., [27, 127].

B.1.1 Unconstrained Optimization

In unconstrained optimization problems, a (global) minimum (or maximum) is searched, without putting any conditions on the solution. For example, given an objective function $f(\mathbf{x})$, the goal is to find $\mathbf{x}$ such that $f(\mathbf{x})$ is a minimum, i.e,

$$\arg\min_{\mathbf{x}} f(\mathbf{x}). \tag{B.1}$$

Depending on the type of function $f(\mathbf{x})$, different strategies are used to find minima. In general, a local minimum is found where the gradient $\nabla_{\mathbf{x}} f(\mathbf{x})$ vanishes. When $f(\mathbf{x})$ is a convex function, then the local minimum is a global minimum.

B.1.2 Method of Lagrange Multipliers

Within optimization methods the goal is to minimize or maximize a certain function $f(\mathbf{x})$ with respect to the variables $\mathbf{x}$. If the variables in $\mathbf{x}$ are constrained, *Lagrange Multipliers* can be used to restrict the feasible variables in $\mathbf{x}$. Strictly speaking, the method of Lagrange Multipliers only allows equality constraints. A constrained minimization problem can be described as:

$$\begin{aligned} &\arg\min_{\mathbf{x}} f(\mathbf{x}) \\ &\text{subject to } h(\mathbf{x}) = \mathbf{0}, \end{aligned} \tag{B.2}$$

with $f(\mathbf{x})$ the cost or objective function and $h(\mathbf{x})$ the set of equality constraints. Since the equality constraints must evaluate to zero at the optimality, the minimization can be redefined as

$$\arg\min_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = \arg\min_{\mathbf{x}} f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i h_i(\mathbf{x}), \tag{B.3}$$

with $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda})$ the *Lagrangian* and $\boldsymbol{\lambda}$ the vector with *Lagrange multipliers*. The minimum of the Lagrangian is located where its gradient vanishes, i.e.,

$$\nabla_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = \nabla_{\mathbf{x}} f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i \nabla_{\mathbf{x}} h_i(\mathbf{x}) = \mathbf{0}. \tag{B.4}$$

Furthermore, the derivative with respect to the multipliers must also vanish, i.e.,

$$\nabla_{\boldsymbol{\lambda}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}) = \frac{\partial \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda})}{\partial \lambda_i} = h_i(\mathbf{x}) = \mathbf{0}, \tag{B.5}$$

for all constraints. Since $\mathbf{x} \in \mathbb{R}^m$, $\boldsymbol{\lambda} \in \mathbb{R}^n$ and $\nabla_{\mathbf{x}} f(\mathbf{x}) \in \mathbb{R}^m$, $h(\mathbf{x}) \in \mathbb{R}^n$, there are $m + n$ equations with $m + n$ unknowns, which implies that the problem is solvable. By solving both Equations (B.4) and (B.5), an optimum will be found, assuming the problem is feasible.

B.1.3 Karush Kuhn-Tucker Conditions

Karush-Kuhn-Tucker (KKT) conditions [106] are first order necessary conditions for a non-linear programming solution to be optimal. These conditions generalize the method of Lagrange Multipliers since it also allows for inequality constraints. In general the problem that is minimized (or maximized) is:

$$\begin{aligned} &\arg\min_{\mathbf{x}} f(\mathbf{x}) \\ &\text{Subject to } \; g(\mathbf{x}) \leq 0, h(\mathbf{x}) = \mathbf{0}, \end{aligned} \tag{B.6}$$

with $g_i(\mathbf{x}) \leq 0$ the inequality constraints in addition to the method of Lagrange Multipliers in Equation (B.2). In a similar way this problem is transformed in the following minimization (or maximization) problem, i.e.,

$$\arg\min_{\mathbf{x}} f(\mathbf{x}) + \sum_{i=1}^{m} \mu_i g_i(\mathbf{x}) + \sum_{j=1}^{n} \lambda_j h_j(\mathbf{x}), \tag{B.7}$$

with now both $\boldsymbol{\mu}$ and $\boldsymbol{\lambda}$ the KKT multipliers. The minimum is located where the gradient vanishes (or the gradient of the cost function is a linear combination of the gradients of g_i and h_i), i.e.,

$$\begin{aligned} \nabla_x f(\mathbf{x}) + \sum_{i=1}^{m} \mu_i \nabla_{\mathbf{x}} g_i(\mathbf{x}) + \sum_{j=1}^{n} \lambda_j \nabla_{\mathbf{x}} h_j(\mathbf{x}) &= \mathbf{0} \\ \sum_{i=1}^{m} \nabla_{\boldsymbol{\mu}} \mu_i g_i(\mathbf{x}) = g(\mathbf{x}) &\leq \mathbf{0} \\ \sum_{i=1}^{n} \nabla_{\boldsymbol{\lambda}} \lambda_i h_i(\mathbf{x}) = h(\mathbf{x}) &= \mathbf{0}. \end{aligned} \tag{B.8}$$

When a particular constraint $g_i \leq 0$, it is contributing to the minimum in Equation (B.7). By allowing multiplier μ_i to be zero, such constraints does not contribute to the minimum of the problem. This is known also the complementarity slackness, which is $\mu_i g_i(\mathbf{x}) = 0$ for all i. If all these necessary conditions are satisfied, there exist KKT multipliers that minimize Equation (B.7). On top of these conditions, some regularity conditions can be imposed on the constraints and at the cost function $f(\mathbf{x})$. We refer to [27, 127] for a detailed overview of these conditions.

B.1.4 Quadratic Programming

Quadratic Programming (QP) is a special case of the method of Lagrange Multipliers. When the objective function $f(\mathbf{x})$ in Equation (B.2) is a Quadratic function and the constraints $g(\mathbf{x})$ are linear, the problem is called a QP problem. If also the Hessian of $f(\mathbf{x})$ is SPD, then the objective function is also convex. This class of problems can be solved using, e.g., the Conjugate Residual method.

B.1.5 Complementarity Problems

As mentioned previously, when inequality constraints are used, additional complementarity conditions are imposed. This class of problems are often named complementarity problems. In the following subsections we mention a few types of complementarity problems often found in contact mechanics.

(Mixed) Linear Complementarity Problems

The Mixed Linear Complementarity Problem (MLCP) [43] is a special class of the Karush-Kuhn-Tucker conditions. As its name suggests, it is linear, meaning that the objective function is quadratic and the constraints are linear, similar to QP problems but now with inequality constraints. Given a convex quadratic function,

$$f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T\mathbf{A}\mathbf{x} - \mathbf{x}^T\mathbf{b}, \tag{B.9}$$

a constrained minimization problem can be described as,

$$\begin{aligned} &\arg\min_{\mathbf{x}} f(\mathbf{x}) \\ &\text{Subject to } \; g_i(\mathbf{x}) \leq 0. \end{aligned} \tag{B.10}$$

The gradients of the functions are: $\nabla_{\mathbf{x}} f(\mathbf{x}) = \mathbf{A}\mathbf{x} - \mathbf{b}$ and $\nabla_{\mathbf{x}} g(\mathbf{x}) = \mathbf{J}^T$, the Jacobian of g. Furthermore, if g is a linear function, it can be rewritten using its Jacobian, i.e., $g(\mathbf{x}) = \mathbf{J}\mathbf{x} - \mathbf{c}$, with $\mathbf{c}$ containing the constant terms in g. To get the gradients to zero, the following system is solved:

$$\mathbf{0} \leq \begin{pmatrix} \mathbf{A} & \mathbf{J}^T \\ \mathbf{J} & \mathbf{0} \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \boldsymbol{\lambda} \end{pmatrix} - \begin{pmatrix} \mathbf{b} \\ \mathbf{c} \end{pmatrix} \perp \begin{pmatrix} \mathbf{1} \\ \boldsymbol{\lambda} \end{pmatrix} \geq \mathbf{0}, \tag{B.11}$$

which is known as a Mixed Linear Complementarity Problem (MLCP) and follows directly from Equation (B.8), given the gradients of the objective and constraint functions. The predicate mixed is due to the free vector $\mathbf{x}$. The $\mathbf{1}$ in the right-most vector is used to make $\mathbf{x}$ free, but also enforces that, due to the complementarity, $\mathbf{0} \leq \mathbf{A}\mathbf{x} + \mathbf{J}^T\boldsymbol{\lambda} - \mathbf{b} = \mathbf{0}$ holds. MLCPs can be solved using various projected QP solvers. When only equality constraints are used, the Conjugate Residual method can be used. When also inequality constraints are used, the CR method can be used in active-set methods. In Chapter 5 a method is presented that efficiently solves MLCPs with both equality and inequality constraints, based on active constraint switching.

Linear Complementarity Problems

Linear Complementarity Problems (LCPs) [43] are a subset of Mixed Linear Complementarity Problems. The difference between them is that LCPs do not have free variables, i.e., the variables are constrained such that $\mathbf{0} \leq \mathbf{x}$ holds. Given the quadratic function in Equation (B.9), an LCP can be described as:

$$\mathbf{0} \leq \mathbf{x} \perp \mathbf{A}\mathbf{x} - \mathbf{b} \geq \mathbf{0}. \tag{B.12}$$

Such problems can be solved using Lemke's algorithm [43] or using some projection based method, e.g., Projected Gauss-Seidel. In the latter one, the values of $\mathbf{x}$ are clamped in each iteration in order to satisfy $\mathbf{0} \leq \mathbf{x}$.

Conversion from MLCP to LCP

The MLCP in Equation (B.11) can be transformed into the following set of equations:

$$\begin{aligned} \boldsymbol{\lambda} &= (\mathbf{J}\mathbf{A}^{-1}\mathbf{J}^T)^{-1}(\mathbf{J}\mathbf{A}^{-1}\mathbf{b} - \mathbf{c}) \\ \mathbf{x} &= \mathbf{A}^{-1}(\mathbf{b} - \mathbf{J}^T\boldsymbol{\lambda}), \end{aligned} \tag{B.13}$$

with $\boldsymbol{\lambda} \geq 0$. The first equation is known as a Linear Complementarity Problem (LCP) which solves for the unknown multipliers $\boldsymbol{\lambda} \geq 0$. Furthermore, the LCP can be rewritten using slack variable $\mathbf{y}$ such that:

$$\mathbf{0} \leq \boldsymbol{\lambda} \perp \mathbf{y} = \left(\mathbf{J}\mathbf{A}^{-1}\mathbf{J}^T\right)\boldsymbol{\lambda} - \mathbf{J}\mathbf{A}^{-1}\mathbf{b} + \mathbf{c} \geq \mathbf{0}, \tag{B.14}$$

with $\boldsymbol{\lambda} \perp \mathbf{y} \Leftrightarrow \boldsymbol{\lambda}^T \mathbf{y} = 0$ the complementarity condition. A typical method for solving this LCP is using the Projected Gauss-Seidel method, see Appendix B.2.5. These (projected) multipliers are then applied in the second equation in order to obtain $\mathbf{x}$. Other methods are for example Lemke's Algorithm [43]. The advantage of converting an MLCP to an LCP is that the problem to solve becomes smaller and its dimensionality is determined only by the number of constraints. If one is able to invert matrix $\mathbf{A}$ easily, then this is an efficient approach for solving MLCPs. However, if the inverse of $\mathbf{A}$ is not easy to compute, then it is better to solve the MLCP without this conversion to an LCP.

Boxed Linear Complementarity Problems

Within standard LCPs, the multipliers are in general positive and therefore have a range $\boldsymbol{\lambda} \in [0, \infty]$. When also the upper-bound of this range is clamped by a constant, the method is known as a Boxed Linear Complementarity Problem. The purpose of this clamping can be for example found in the treatment of friction in contact mechanics. By approximating the upper-bound of the friction forces, a simple approach for friction is obtained. However, the results are not physically correct since the bounds are a rough approximation, see [37].

Non-Linear Complementarity Problems

In cases in which the bounds of the multipliers are given by the value of other free variables or the multipliers, the method becomes a Non-Linear Complementarity Problem (NCP). Examples for this type of problems are for instance a Coulomb friction model. The maximum friction force depends on the magnitude of the normal forces and is not on beforehand known. Due to this coupling of the multipliers, the problem becomes much harder to solve and is in general non-linear, despite the quadratic objective function.

B.2 Numerical Methods

In this section we describe a number of numerical methods used to solve linear problems. The steepest descent method is described since it shows how gradient based optimization methods work. Followed by Krylov subspace methods, *Conjugate Gradient* and *Conjugate Residual* methods and how they are preconditioned. Finally also the frequently used *Jacobi* and *Gauss-Seidel* methods are briefly discussed.

B.2.1 Steepest Descent

The Steepest Descent, or Gradient Descent method, is a method which can be used for solving linear and non-linear systems. Although it is not widely used for solving linear systems, it can be used to understand many related Krylov subspace methods.

Suppose a linear system $\mathbf{Ax} = \mathbf{b}$ should be solved for $\mathbf{x}$, with $\mathbf{A}$ a *Symmetric Positive Definite* (SPD) matrix, a quadratic function can be defined which has a global minimizer for the solution of that linear system, i.e.,

$$g(\mathbf{x}) = \mathbf{x}^T \mathbf{A}\mathbf{x} - 2\mathbf{x}^T \mathbf{b}, \tag{B.15}$$

with

$$\nabla_{\mathbf{x}} g(\mathbf{x}) = 2\mathbf{A}\mathbf{x} - 2\mathbf{b} = -2\mathbf{r}, \tag{B.16}$$

its gradient and $\mathbf{r}$ the residual vector $\mathbf{r} = \mathbf{b} - \mathbf{A}\mathbf{x}$. In order to move the current approximation of $\mathbf{x}$ closer to the real minimizer, a step t into some direction $\mathbf{p}$ is performed, i.e.,

$$\begin{aligned} g(\mathbf{x} + t\mathbf{p}) &= (\mathbf{x} + t\mathbf{p})^T \mathbf{A}(\mathbf{x} + t\mathbf{p}) - 2(\mathbf{x} + t\mathbf{p})^T \mathbf{b} \\ &= g(\mathbf{x}) + 2t\mathbf{p}^T \mathbf{A}\mathbf{x} + t^2 \mathbf{p}^T \mathbf{A}\mathbf{p} - 2t\mathbf{p}^T \mathbf{b}, \end{aligned} \tag{B.17}$$

with t the step-size along $\mathbf{p}$. To bring $\mathbf{x}$ closer to the minimum, t must be chosen such that $h(t) = g(\mathbf{x} + t\mathbf{p})$ is minimized. This *line search* problem has a minimizer which is located where $h'(t) = 0$. For linear problems this reduces to:

$$h'(t) = 2\mathbf{p}^T \mathbf{A}\mathbf{x} + 2t\mathbf{p}^T \mathbf{A}\mathbf{p} - 2\mathbf{p}^T \mathbf{b} = 0, \tag{B.18}$$

from which t can be directly computed, i.e.,

$$t = \frac{\mathbf{p}^T \mathbf{b} - \mathbf{p}^T \mathbf{A}\mathbf{x}}{\mathbf{p}^T \mathbf{A}\mathbf{p}} = \frac{\mathbf{p}^T \mathbf{r}}{\mathbf{p}^T \mathbf{A}\mathbf{p}}. \tag{B.19}$$

Given t, $\mathbf{x}$ can now be updated using

$$\mathbf{x}_{i+1} = \mathbf{x}_i + t_i \mathbf{p}_i. \tag{B.20}$$

Exactly at this point, vector $\mathbf{p}$ and $\nabla_{\mathbf{x}} g(\mathbf{x})$ are orthogonal. Since the gradient of $g(\mathbf{x})$ is a scaled instance of the residual vector, $\mathbf{r}$ and $\mathbf{p}$ are orthogonal. This can be directly seen from Equation (B.19). When $\mathbf{p}$ and $\mathbf{r}$ *are* orthogonal at $\mathbf{x}_{i+1}$, $t = 0$. Which results in no update of $\mathbf{x}$ in direction $\mathbf{p}$. Next, a new vector $\mathbf{p}$ can be chosen, and a new step along this direction is performed. Since $g(\mathbf{x})$ decreases the most in the direction of its gradient $\nabla_{\mathbf{x}} g(\mathbf{x})$, the new search direction $\mathbf{p}$ is set to the current residual vector. Since Equation (B.15) is a quadratic problem with a linear gradient, the residual can be updated using:

$$\begin{aligned} \mathbf{x}_{i+1} &= \mathbf{x}_i + t_i \mathbf{p}_i \\ \mathbf{A}\mathbf{x}_{i+1} &= \mathbf{A}\mathbf{x}_i + t_i \mathbf{A}\mathbf{p}_i \\ \mathbf{b} - \mathbf{A}\mathbf{x}_{i+1} &= \mathbf{b} - \mathbf{A}\mathbf{x}_i - t_i \mathbf{A}\mathbf{p}_i \\ \mathbf{r}_{i+1} &= \mathbf{r}_i - t_i \mathbf{A}\mathbf{p}_i. \end{aligned} \tag{B.21}$$

Otherwise, the residual vector must be recomputed explicitly. By repeating these steps, the procedure eventually finds an approximation for $\mathbf{x}$ which minimizes $g(\mathbf{x})$. This minimization procedure is the same for Gradient Descent (applied on non-linear problems), Conjugate Gradient, Conjugate Residual methods and many other methods. Their difference is the selection of the next search vector $\mathbf{p}$.

B.2.2 Conjugate Gradient Method

The Steepest Descent method is in general not used to solve linear problems. Steepest Descent computes at every iteration a gradient direction and then moves the current approximation in that direction. In case of a linear problem, the step-size can be computed analytically. Next, a new gradient direction is computed and a new line-search along the new gradient is performed.

For linear problems, this approach is rather inefficient. If matrix $\mathbf{A}$ is Symmetric and Positive Definite, it can be decomposed into n orthogonal n-dimensional eigenvectors, each associated with a positive eigenvalue. Furthermore, any SPD matrix can be constructed using n orthogonal n-dimensional vectors, not necessary its eigenvectors. Therefore it is sufficient to search in each of the n orthogonal directions in order to cover the space described by $\mathbf{A}$.

Within the Conjugate Gradient method [88] the search or gradient vectors are chosen to be conjugate or **A**-orthogonal, i.e.,

$$\mathbf{p}_i^T \mathbf{A} \mathbf{p}_j = 0, \text{if } i \neq j, \tag{B.22}$$

and the residual vectors are orthogonal,

$$\mathbf{r}_i^T \mathbf{r}_j = 0, \text{if } i \neq j. \tag{B.23}$$

Using these properties it can be shown that after n steps with n different search vectors and factors t_i, the residual becomes zero, assuming exact arithmetic. See for more details and proofs [35]. However, the Conjugate Gradient method is hardly used as a direct method since more efficient methods exist. When matrix **A** is sparse, the Conjugate Gradient method is used as an iterative method and terminates when the residual norm drops below a certain threshold.

Selection of the residual vectors The CG method selects residual vectors in an iterative process such that they are orthogonal. The residual vectors are updated as described in the Steepest Descent method using:

$$\mathbf{r}_{i+1} = \mathbf{r}_i - t_i \mathbf{A} \mathbf{p}_i, \tag{B.24}$$

for some t_i. The value of t_i is chosen such that $\mathbf{r}_{i+1}$ and $\mathbf{r}_i$ are orthogonal, i.e.,

$$\mathbf{r}_i^T \mathbf{r}_{i+1} = \mathbf{r}_i^T \mathbf{r}_i - t_i \mathbf{r}_i^T \mathbf{A} \mathbf{p}_i = 0, \tag{B.25}$$

and

$$\mathbf{r}_{i+1}^T \mathbf{r}_{i+1} = \mathbf{r}_{i+1}^T \mathbf{r}_i - t_i \mathbf{r}_{i+1}^T \mathbf{A} \mathbf{p}_i, \tag{B.26}$$

which gives

$$t_i = \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{r}_i^T \mathbf{A} \mathbf{p}_i} = -\frac{\mathbf{r}_{i+1}^T \mathbf{r}_{i+1}}{\mathbf{r}_{i+1}^T \mathbf{A} \mathbf{p}_i}. \tag{B.27}$$

Please note that Equation (B.27) yields the same result as the computation in Equation (B.19) since they both select a residual vector that is orthogonal to $\mathbf{p}$.

Selection of search vectors The search vectors are selected in a similar fashion and are updated using:

$$\mathbf{p}_{i+1} = \mathbf{r}_{i+1} + s_i \mathbf{p}_i. \tag{B.28}$$

The value for s_i is chosen such that $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ are conjugate, i.e.,

$$\mathbf{p}_i^T \mathbf{A} \mathbf{p}_{i+1} = \mathbf{p}_i^T \mathbf{A} \mathbf{r}_{i+1} + s_i \mathbf{p}_i^T \mathbf{A} \mathbf{p}_i = 0, \tag{B.29}$$

and

$$\mathbf{p}_{i+1}^T \mathbf{A} \mathbf{p}_{i+1} = \mathbf{p}_{i+1}^T \mathbf{A} \mathbf{r}_{i+1} + s_i \mathbf{p}_{i+1}^T \mathbf{A} \mathbf{p}_i, \tag{B.30}$$

which gives

$$s_i = -\frac{\mathbf{p}_i^T \mathbf{A} \mathbf{r}_{i+1}}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}, \tag{B.31}$$

and

$$\mathbf{p}_{i+1}^T \mathbf{A} \mathbf{p}_{i+1} = \mathbf{p}_{i+1}^T \mathbf{A} \mathbf{r}_{i+1}, \tag{B.32}$$

since $\mathbf{p}_{i+1}^T \mathbf{A} \mathbf{p}_i = 0$.

Simplification The computations of both t_i and s_i can be simplified further. Using relation Equation (B.32), t_i becomes:

$$t_i = \frac{\mathbf{r}_i^T \mathbf{r}_i}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i}. \tag{B.33}$$

Furthermore, the computation of s_i can be simplified using Equations (B.27) and (B.33), i.e.,

$$s_i = -\frac{\mathbf{p}_i^T \mathbf{A} \mathbf{r}_{i+1}}{\mathbf{p}_i^T \mathbf{A} \mathbf{p}_i} = -\frac{-\mathbf{r}_{i+1}^T \mathbf{r}_{i+1}}{t_i \mathbf{p}_i^T \mathbf{A} \mathbf{p}_i} = \frac{\mathbf{r}_{i+1}^T \mathbf{r}_{i+1}}{\mathbf{r}_i^T \mathbf{r}_i}, \tag{B.34}$$

which reuses the squared residual norm. Using s_i, search direction $\mathbf{p}_{i+1}$ is obtained. After this, the method advances to the next iteration and new approximations of $t_i, s_i, \mathbf{r}_{i+1}, \mathbf{p}_{i+1}$ and $\mathbf{x}_{i+1}$ can be computed. After performing sufficient iterations, approximation $\mathbf{x}_{i+1}$ minimizes Equation (B.15) and the residual norm $\|\mathbf{r}\|$ approaches zero.

B.2.3 Conjugate Residual Method

The Conjugate Residual method [113] is very similar to the Conjugate Gradient method. As its name suggests, the residual vectors are now conjugate (which are orthogonal for the CG method). The search vectors are chosen to be $\mathbf{A}^2$-orthogonal, i.e., $\mathbf{p}_j^T \mathbf{A}^2 \mathbf{p}_i = 0$ for $i \neq j$. Instead of minimizing the quadratic function in Equation (B.15), the following problem is minimized:

$$g(\mathbf{x})_{cr} = \|\mathbf{A}\mathbf{x} - \mathbf{b}\|^2 = (\mathbf{A}\mathbf{x} - \mathbf{b})^T (\mathbf{A}\mathbf{x} - \mathbf{b}) = \mathbf{r}^T \mathbf{r}, \tag{B.35}$$

with

$$\nabla_{\mathbf{x}} g(\mathbf{x})_{cr} = -2\mathbf{A}\mathbf{r} \tag{B.36}$$

its gradient. The update of vectors $\mathbf{x}, \mathbf{r}$ and $\mathbf{p}$ is done exactly as in the Conjugate Gradient method. The only differences are the computations of t and s.

Update of residual and search vector Given function $g(\mathbf{x})_{cr}$ and search vector $\mathbf{p}$, approximation $\mathbf{x}$ is moved along $\mathbf{p}$ such that $h(t) = g(\mathbf{x} + t\mathbf{p})_{cr}$ is minimized, i.e.,

$$\begin{aligned} g(\mathbf{x} + t\mathbf{p})_{cr} &= (\mathbf{x} + t\mathbf{p})^T \mathbf{A}^2 (\mathbf{x} + \mathbf{t}\mathbf{p}) - 2(\mathbf{x} + t\mathbf{p})^T \mathbf{A}\mathbf{b} + \mathbf{b}^T \mathbf{b} \\ &= g(\mathbf{x})_{cr} + 2t\mathbf{x}^T \mathbf{A}^2 \mathbf{p} + t^2 \mathbf{p}^T \mathbf{A}^2 \mathbf{p} - 2t\mathbf{p}^T \mathbf{A}\mathbf{b}. \end{aligned} \tag{B.37}$$

The minimum of $h(t)$ is located where $h'(t) = 0$, i.e.,

$$h'(t) = 2\mathbf{x}^T \mathbf{A}^2 \mathbf{p} + 2t\mathbf{p}^T \mathbf{A}^2 \mathbf{p} - 2\mathbf{p}^T \mathbf{A}\mathbf{b} = 0, \tag{B.38}$$

from which t can be obtained,

$$t = \frac{\mathbf{p}^T \mathbf{A}\mathbf{b} - \mathbf{p}^T \mathbf{A}^2 \mathbf{x}}{\mathbf{p}^T \mathbf{A}^2 \mathbf{p}} = \frac{\mathbf{p}^T \mathbf{A}\mathbf{r}}{\mathbf{p}^T \mathbf{A}^2 \mathbf{p}}. \tag{B.39}$$

Or alternatively, one chooses t_i such the residual vectors are chosen to be conjugate, i.e.,

$$\mathbf{r}_i \mathbf{A} \mathbf{r}_{i+1} = \mathbf{r}_i \mathbf{A} \mathbf{r}_i - t_i \mathbf{r}_i \mathbf{A}^2 \mathbf{p}_i = 0, \tag{B.40}$$

and

$$\mathbf{r}_{i+1} \mathbf{A} \mathbf{r}_{i+1} = \mathbf{r}_{i+1} \mathbf{A} \mathbf{r}_i - t_i \mathbf{r}_{i+1} \mathbf{A}^2 \mathbf{p}_i, \tag{B.41}$$

which gives

$$t_i = \frac{\mathbf{r}_i^T \mathbf{A}\mathbf{r}_i}{\mathbf{r}_i^T \mathbf{A}^2 \mathbf{p}_i} = -\frac{\mathbf{r}_{i+1}^T \mathbf{A}\mathbf{r}_{i+1}}{\mathbf{r}_{i+1}^T \mathbf{A}^2 \mathbf{p}_i}. \tag{B.42}$$

Please note that both Equations (B.39) and (B.42) compute the same value. Similarly, the next search vector $\mathbf{p}_{i+1}$ is chosen to be $\mathbf{A}^2$-orthogonal by computing s_i as follows:

$$\mathbf{p}_i^T \mathbf{A}^2 \mathbf{p}_{i+1} = \mathbf{p}_i^T \mathbf{A}^2 \mathbf{r}_{i+1} + s_i \mathbf{p}_i^T \mathbf{A}^2 \mathbf{p}_i = 0, \tag{B.43}$$

and

$$\mathbf{p}_{i+1}^T \mathbf{A}^2 \mathbf{p}_{i+1} = \mathbf{p}_{i+1}^T \mathbf{A}^2 \mathbf{r}_{i+1} + s_i \mathbf{p}_{i+1}^T \mathbf{A}^2 \mathbf{p}_i, \tag{B.44}$$

which gives

$$s_i = -\frac{\mathbf{p}_i^T \mathbf{A}^2 \mathbf{r}_{i+1}}{\mathbf{p}_i^T \mathbf{A}^2 \mathbf{p}_i}, \tag{B.45}$$

and

$$\mathbf{p}_{i+1}^T \mathbf{A}^2 \mathbf{p}_{i+1} = \mathbf{p}_{i+1}^T \mathbf{A}^2 \mathbf{r}_{i+1}, \tag{B.46}$$

since $\mathbf{p}_{i+1}^T \mathbf{A}^2 \mathbf{p}_i = 0$. A similar simplification for both t_i and s_i can be performed as shown for the CG method, i.e.,

$$t_i = \frac{\mathbf{r}_i^T \mathbf{A}\mathbf{r}_i}{\mathbf{p}_i^T \mathbf{A}^2 \mathbf{p}_i}, \tag{B.47}$$

and

$$s_i = -\frac{\mathbf{p}_i^T \mathbf{A}^2 \mathbf{r}_{i+1}}{\mathbf{p}_i^T \mathbf{A}^2 \mathbf{p}_i} = -\frac{-\mathbf{r}_{i+1}^T \mathbf{A}\mathbf{r}_{i+1}}{t_i \mathbf{p}_i^T \mathbf{A}^2 \mathbf{p}_i} = \frac{\mathbf{r}_{i+1}^T \mathbf{A}\mathbf{r}_{i+1}}{\mathbf{r}_i^T \mathbf{A}\mathbf{r}_i}. \tag{B.48}$$

For more details about the derivation and proof of the method we refer to [113].

Since the Conjugate Residual method minimizes the norm of the residual vector, see Equation (B.35), and $g(\mathbf{x})_{cr}$ contains the terms $\mathbf{x}^T \mathbf{A}^2 \mathbf{x}$ and $\mathbf{x}^T \mathbf{A}\mathbf{b}$, it actually solves the system $\mathbf{A}^2\mathbf{x} = \mathbf{A}\mathbf{b}$. Since $g(\mathbf{x})_{cr}$ is a convex function, each improved approximation of $\mathbf{x}$ monotonically decreases $\|\mathbf{r}\|$. Furthermore, since $\mathbf{A}^2$ is Symmetric and Positive Definite by definition, $\mathbf{A}$ does not need to be Positive Definite. This property makes the Conjugate Residual method interesting for solving indefinite systems. In Chapter 5 the CR method is used for solving the indefinite contact problem with inequality constraints and an additional approximation of the friction force.

Local minima When the CR method is used to solve symmetric indefinite problems, or saddle point problems, it is possible that the method breaks down. The CR method should terminate when the residual vector $\mathbf{r}$ is sufficiently small. However, it is possible that the method breaks down when the gradient $\mathbf{A}\mathbf{r}$ becomes zero when the method has found a local minimum. In this case, the computation of t_i results in zero, while the following computation of s_i results in a division by zero. Hence, the method breaks down. By also considering the norm of the gradient $\mathbf{A}\mathbf{r}$ in the termination test, the method will stop when such a local minimum is found, see [85].

B.2.4 Preconditioning

Krylov-subspace methods like the CG and CR method can converge slowly when the conditioning of matrix $\mathbf{A}$ is bad. Additionally, when an MLCP or QP is solved using the CR method, the method also converges slowly due to the different scales in the coupled subproblems. To improve the conditioning, and thus the convergence rate, a preconditioner can be applied. To apply a preconditioner, the following system is considered:

$$\mathbf{C}_A^{-1}\mathbf{A}\mathbf{C}_B^{-1}\mathbf{C}_B\mathbf{x} = \mathbf{C}_A^{-1}\mathbf{b}, \tag{B.49}$$

with $\mathbf{C}_A^{-1}\mathbf{C}_B^{-1} = \mathbf{C}^{-1}$ and $\mathbf{C}^{-1}$ the preconditioner matrix. Furthermore, $\mathbf{C}_A^{-T} = \mathbf{C}_B^{-1}$. Please note that the decomposition of $\mathbf{C}^{-1}$ is only needed in the derivation of the preconditioned methods. Eventually, matrices $\mathbf{C}_A^{-1}$ and $\mathbf{C}_B^{-1}$ are multiplied. To derive the preconditioned methods we substitute $\mathbf{A} \Rightarrow \mathbf{C}_A^{-1}\mathbf{A}\mathbf{C}_B^{-1}$, $\mathbf{x} \Rightarrow \mathbf{C}_B\mathbf{x}$, $\mathbf{b} \Rightarrow \mathbf{C}_A^{-1}\mathbf{b}$ and $\mathbf{p} \Rightarrow \mathbf{C}_B\mathbf{p}$, see [35]. Using these substitution rules, the substitution for the residual vector can be found by:

$$\begin{aligned}
\mathbf{r} &= \mathbf{b} - \mathbf{A}\mathbf{x} \\
\mathbf{r} &\Rightarrow \mathbf{C}_A^{-1}\mathbf{b} - \mathbf{C}_A^{-1}\mathbf{A}\mathbf{C}_B^{-1}\mathbf{C}_B\mathbf{x} \\
\mathbf{r} &\Rightarrow \mathbf{C}_A^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}) \\
\mathbf{r} &\Rightarrow \mathbf{C}_A^{-1}\mathbf{r}.
\end{aligned} \tag{B.50}$$

The update of $\mathbf{x}$ remains unchanged, i.e.,

$$\begin{aligned}
\mathbf{C}_B\mathbf{x}_{i+1} &= \mathbf{C}_B\mathbf{x}_i + t_i\mathbf{C}_B\mathbf{p} \\
\mathbf{x}_{i+1} &= \mathbf{x}_i + t_i\mathbf{p}.
\end{aligned} \tag{B.51}$$

Similarly, the update of the residual remains unchanged,

$$\begin{aligned}
\mathbf{C}_A^{-1}\mathbf{r}_{i+1} &= \mathbf{C}_A^{-1}\mathbf{r}_i - t_i\mathbf{C}_A^{-1}\mathbf{A}\mathbf{C}_B^{-1}\mathbf{C}_B\mathbf{p}_i \\
\mathbf{r}_{i+1} &= \mathbf{r}_i - t_i\mathbf{A}\mathbf{p}_i.
\end{aligned} \tag{B.52}$$

Contrary, the update of the search vector needs to be changed to deal with the preconditioner, i.e.,

$$\begin{aligned}
\mathbf{C}_B\mathbf{p}_{i+1} &= \mathbf{C}_A^{-1}\mathbf{r}_{i+1} + s_i\mathbf{C}_B\mathbf{p}_i \\
\mathbf{p}_{i+1} &= \mathbf{C}_B^{-1}\mathbf{C}_A^{-1}\mathbf{r}_{i+1} + s_i\mathbf{p}_i \\
\mathbf{p}_{i+1} &= \mathbf{C}^{-1}\mathbf{r}_{i+1} + s_i\mathbf{p}_i,
\end{aligned} \tag{B.53}$$

which is the same for both the CG and CR method. Due to this, the initialization of the search vector changes to $\mathbf{p}_0 = \mathbf{C}^{-1}\mathbf{r}_0$.

Preconditioned Conjugate Gradient Method For the preconditioned CG method, t_i is obtained in the same way as for the unpreconditioned version using the substitutions shown in the previous paragraph. Please note that the residual vectors are not orthogonal, but its substitutes are, i.e, $\mathbf{r}_i^T\mathbf{C}_B^{-1}\mathbf{C}_A^{-1}\mathbf{r}_j = \mathbf{r}_i^T\mathbf{C}^{-1}\mathbf{r}_j = 0$ for $i \neq j$. Contrary, the search vectors remain conjugate, i.e., $\mathbf{p}_i^T\mathbf{C}_A\mathbf{C}_A^{-1}\mathbf{A}\mathbf{C}_B^{-1}\mathbf{C}_B\mathbf{p}_j = \mathbf{p}_i^T\mathbf{A}\mathbf{p}_j = 0$ for $i \neq j$. Using these properties, the computation of t_i can be derived:

$$t_i = \frac{\mathbf{r}_i^T\mathbf{C}^{-1}\mathbf{r}_i}{\mathbf{p}_i^T\mathbf{A}\mathbf{p}_i}. \tag{B.54}$$

The computation of s_i is obtained similarly:

$$s_i = \frac{\mathbf{r}_{i+1}^T\mathbf{C}^{-1}\mathbf{r}_{i+1}}{\mathbf{r}_i^T\mathbf{C}^{-1}\mathbf{r}_i}. \tag{B.55}$$

Preconditioned Conjugate Residuals In a similar fashion the computations of t_i and s_i are derived for the CR method. The conjugacy of the preconditioned residuals become

$$\mathbf{r}_i \mathbf{C}_B^{-1} \mathbf{C}_A^{-1} \mathbf{A} \mathbf{C}_B^{-1} \mathbf{C}_A^{-1} \mathbf{r}_j = \mathbf{r}_i^T \mathbf{C}^{-1} \mathbf{A} \mathbf{C}^{-1} \mathbf{r}_j = 0, \tag{B.56}$$

for $i \neq j$. Similarly, the $\mathbf{A}^2$-orthogonality for the search vectors become

$$\mathbf{p}_i^T \mathbf{C}_A \mathbf{C}_A^{-1} \mathbf{A} \mathbf{C}_B^{-1} \mathbf{C}_A^{-1} \mathbf{A} \mathbf{C}_B^{-1} \mathbf{C}_B \mathbf{p}_j = \mathbf{p}_i^T \mathbf{A} \mathbf{C}^{-1} \mathbf{A} \mathbf{p}_j = 0, \tag{B.57}$$

for $i \neq j$. From this the computation of t_i is derived, i.e.,

$$t_i = \frac{\mathbf{r}_i^T \mathbf{C}^{-1} \mathbf{A} \mathbf{C}^{-1} \mathbf{r}_i}{\mathbf{p}_i^T \mathbf{A} \mathbf{C}^{-1} \mathbf{A} \mathbf{p}_i}. \tag{B.58}$$

Similarly, the computation of s_i is obtained, i.e.,

$$s_i = \frac{\mathbf{r}_{i+1}^T \mathbf{C}^{-1} \mathbf{A} \mathbf{C}^{-1} \mathbf{r}_{i+1}}{\mathbf{r}_i^T \mathbf{C}^{-1} \mathbf{A} \mathbf{C}^{-1} \mathbf{r}_i}. \tag{B.59}$$

Additionally, the CR method also updates vector $\mathbf{Ap}$, i.e.,

$$\mathbf{A} \mathbf{p}_{i+1} = \mathbf{A} \mathbf{C}^{-1} \mathbf{r}_{i+1} + s_i \mathbf{A} \mathbf{p}_i, \tag{B.60}$$

such that per iteration of the method only one matrix vector multiplication $\mathbf{A}\mathbf{C}^{-1}\mathbf{r}_{i+1}$ is required. Please note that due to this preconditioning the residual norm $\|\mathbf{r}\|$ does in general not decrease monotonically, but $\|\mathbf{C}_A^{-1}\mathbf{r}\|$ does.

B.2.5 Stationary Iterative Methods

Other numerical methods found in contact solver methods are for example the Jacobi and Gauss-Seidel methods. Both methods are relatively easy to implement and are widely used. A stationary iterative method has the form

$$\mathbf{x}_{i+1} = \mathbf{B}\mathbf{x}_i + \mathbf{c}, \tag{B.61}$$

and produces approximations of $\mathbf{x}$ that, when all conditions are met, converges to a stationary (fixed) point. The iteration is commonly called a fixed-point iteration. Depending on the method used, matrix $\mathbf{B}$ and vector $\mathbf{c}$ are chosen differently and depend on the splitting $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$, with $\mathbf{L}$ the strict lower triangular matrix, $\mathbf{U}$ the strict upper triangular matrix and $\mathbf{D}$ the diagonal of $\mathbf{A}$.

B.2.6 Jacobi Method

Jacobi's method is obtained by

$$\begin{aligned} (\mathbf{L} + \mathbf{D} + \mathbf{U})\mathbf{x} &= \mathbf{b} \\ \mathbf{D}\mathbf{x}_{i+1} &= \mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{x}_i \\ \mathbf{x}_{i+1} &= \mathbf{D}^{-1}(\mathbf{b} - (\mathbf{L} + \mathbf{U})\mathbf{x}_i) \end{aligned} \tag{B.62}$$

and so chooses $\mathbf{B} = -\mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})$ and $\mathbf{c} = \mathbf{D}^{-1}\mathbf{b}$. This method converges if the spectral radius of the iteration matrix $\mathbf{B}$ is smaller than one. In this case

$$\lim_{n \to \infty} \mathbf{B}^n = \mathbf{0}, \tag{B.63}$$

which implies that eventually the change in $\mathbf{x}$ vanishes and that the method has found an accurate solution. A sufficient condition is to have matrix $\mathbf{A}$ diagonally dominant [35], i.e.,

$$\|a_{jj}\| > \sum_{j \neq i} \|a_{ij}\|. \tag{B.64}$$

B.2.7 Gauss-Seidel Method

The Gauss-Seidel method is very similar to the Jacobi method and is obtained by

$$\begin{aligned} (\mathbf{L} + \mathbf{D} + \mathbf{U})\mathbf{x} &= \mathbf{b} \\ (\mathbf{D} + \mathbf{L})\mathbf{x}_{i+1} &= \mathbf{b} - \mathbf{U}\mathbf{x}_i \\ \mathbf{x}_{i+1} &= (\mathbf{D} + \mathbf{L})^{-1}(\mathbf{b} - \mathbf{U}\mathbf{x}_i). \end{aligned} \tag{B.65}$$

Due to the triangular form of $(\mathbf{L} + \mathbf{D})$, forward substitution is applied, which yields

$$x_i^* = \frac{1}{a_{ii}} \left(b_i - \sum_{j<i} a_{ij} x_j^* - \sum_{j>i} a_{ij} x_j \right), \tag{B.66}$$

with $\mathbf{x}^*$ the updated vector $\mathbf{x}_{i+1}$, x_j the j-th component of $\mathbf{x}_i$ and x_j^* the j-th component of $\mathbf{x}_{i+1}$. The Gauss-Seidel method converges when matrix $\mathbf{A}$ is Symmetric Positive Definite and/or diagonally dominant, but might also converge if these conditions are not met [35].

Projected Gauss-Seidel

Projected Gauss-Seidel is a modified version of the Gauss-Seidel method and is often used to solve Linear Complementarity Problems (LCPs), discussed in Appendix B.1.5. The method is the same as described in Equation (B.66), except that the newly computed value x_i^* is clamped or explicitly set to zero.

C

Rigid body simulations

C.1 Introduction

A rigid body is an object with an arbitrary shape that is not allowed to deform. Only its position and orientation are allowed to change during the simulation. This results in 6 degrees of freedom per object: 3 translational and 3 rotational. In contrast, each node of the deformable mesh adds 3 degrees of freedom to the total.

C.2 Rigid-Body Dynamics

The dynamics of a rigid body directly follows from *Newton's laws of motion*, i.e.

$$\mathbf{M}\mathbf{a} = \mathbf{f} \tag{C.1}$$

with $\mathbf{f}$ a function of the force. An implicit integration scheme is obtained through a Taylor expansion and a forward difference approximation of the acceleration:

$$\begin{aligned}
\mathbf{M}\left(\frac{\mathbf{v}_{i+1}-\mathbf{v}_i}{\Delta t}\right) &= \mathbf{f} + \Delta t \frac{\partial \mathbf{f}}{\partial t} \\
\mathbf{M}\left(\frac{\mathbf{v}_{i+1}-\mathbf{v}_i}{\Delta t}\right) &= \mathbf{f} + \Delta t \left(\frac{\partial \mathbf{f}}{\partial \mathbf{a}}\frac{\partial \mathbf{a}}{\partial t} + \frac{\partial \mathbf{f}}{\partial \mathbf{v}}\frac{\partial \mathbf{v}}{\partial t} + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\frac{\partial \mathbf{x}}{\partial t}\right) \\
\mathbf{M}\left(\frac{\mathbf{v}_{i+1}-\mathbf{v}_i}{\Delta t}\right) &= \mathbf{f} + \Delta t \left(\frac{\partial \mathbf{f}}{\partial \mathbf{a}}\mathbf{j} + \frac{\partial \mathbf{f}}{\partial \mathbf{v}}\left(\frac{\mathbf{v}_{i+1}-\mathbf{v}_i}{\Delta t}\right) + \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\mathbf{v}_{i+1}\right) \\
\left(\mathbf{M} - \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{v}} - \Delta t^2 \frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right)\mathbf{v}_{i+1} &= \Delta t \mathbf{f} + \left(\mathbf{M} - \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{v}}\right)\mathbf{v}_i,
\end{aligned} \tag{C.2}$$

with $\mathbf{j}$ the jerk, the derivative of the acceleration. Since linearized forces are used, this term is dropped. Similarly, the torque $\mathbf{t}$ is given by,

$$I_C \boldsymbol{\alpha} = \mathbf{t}, \tag{C.3}$$

with I_C the *moment of inertia* or *inertia tensor* and $\boldsymbol{\alpha}$ the *angular acceleration*. The implicit scheme for the angular velocity is obtained as follows:

$$\begin{aligned}
I_C\left(\frac{\boldsymbol{\omega}_{i+1}-\boldsymbol{\omega}_i}{\Delta t}\right) &= \mathbf{t} + \Delta t \frac{\partial \mathbf{t}}{\partial t} \\
I_C\left(\frac{\boldsymbol{\omega}_{i+1}-\boldsymbol{\omega}_i}{\Delta t}\right) &= \mathbf{t} + \Delta t \left(\frac{\partial \mathbf{t}}{\partial \boldsymbol{\alpha}}\frac{\partial \boldsymbol{\alpha}}{\partial t} + \frac{\partial \mathbf{t}}{\partial \boldsymbol{\omega}}\frac{\partial \boldsymbol{\omega}}{\partial t} + \frac{\partial \mathbf{t}}{\partial \boldsymbol{\phi}}\frac{\partial \boldsymbol{\phi}}{\partial t}\right) \\
I_C\left(\frac{\boldsymbol{\omega}_{i+1}-\boldsymbol{\omega}_i}{\Delta t}\right) &= \mathbf{t} + \Delta t \left(\frac{\partial \mathbf{t}}{\partial \boldsymbol{\alpha}}\boldsymbol{\zeta} + \frac{\partial \mathbf{t}}{\partial \boldsymbol{\omega}}\left(\frac{\boldsymbol{\omega}_{i+1}-\boldsymbol{\omega}_i}{\Delta t}\right) + \frac{\partial \mathbf{t}}{\partial \boldsymbol{\phi}}\boldsymbol{\omega}_{i+1}\right) \\
\left(I_C - \Delta t \frac{\partial \mathbf{t}}{\partial \boldsymbol{\omega}} - \Delta t^2 \frac{\partial \mathbf{t}}{\partial \boldsymbol{\phi}}\right)\boldsymbol{\omega}_{i+1} &= \Delta t \mathbf{t} + \left(I_C - \Delta t \frac{\partial \mathbf{t}}{\partial \boldsymbol{\omega}}\right)\boldsymbol{\omega}_i,
\end{aligned} \tag{C.4}$$

with $\boldsymbol{\zeta}$ the angular jerk, which is also dropped from the linearization. An external force working on a rigid body also generates a torque $\mathbf{t}$ on that body by:

$$\mathbf{t} = \mathbf{r} \times \mathbf{f}, \tag{C.5}$$

with $\mathbf{r}$ the vector from the center of mass to the point on the surface of the body on which force $\mathbf{f}$ is acting. Due to this, also the cross terms between the torque and force functions are derived, i.e.,

$$\begin{aligned}\frac{\partial \mathbf{f}}{\partial t} &= \frac{\partial \mathbf{f}}{\partial \boldsymbol{\omega}}\boldsymbol{\alpha} + \frac{\partial \mathbf{f}}{\partial \boldsymbol{\phi}}\boldsymbol{\omega},\\ \frac{\partial \mathbf{t}}{\partial t} &= \frac{\partial \mathbf{t}}{\partial \mathbf{v}}\mathbf{a} \quad + \frac{\partial \mathbf{t}}{\partial \mathbf{x}}\mathbf{v}.\end{aligned} \tag{C.6}$$

This results in the following linear system for rigid bodies:

$$\begin{aligned}\begin{pmatrix} \mathbf{M} - \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{v}} - \Delta t^2 \frac{\partial \mathbf{f}}{\partial \mathbf{x}} & -\Delta t \frac{\partial \mathbf{f}}{\partial \boldsymbol{\omega}} - \Delta t^2 \frac{\partial \mathbf{f}}{\partial \boldsymbol{\phi}} \\ -\Delta t \frac{\partial \mathbf{t}}{\partial \mathbf{v}} - \Delta t^2 \frac{\partial \mathbf{t}}{\partial \mathbf{x}} & I_C - \Delta t \frac{\partial \mathbf{t}}{\partial \boldsymbol{\omega}} - \Delta t^2 \frac{\partial \mathbf{t}}{\partial \boldsymbol{\phi}} \end{pmatrix} \begin{pmatrix} \mathbf{v}_{i+1} \\ \boldsymbol{\omega}_{i+1} \end{pmatrix} = \\ \Delta t \begin{pmatrix} \mathbf{f} \\ \mathbf{t} \end{pmatrix} + \begin{pmatrix} \mathbf{M} - \Delta t \frac{\partial \mathbf{f}}{\partial \mathbf{v}} & -\Delta t \frac{\partial \mathbf{f}}{\partial \boldsymbol{\omega}} \\ -\Delta t \frac{\partial \mathbf{t}}{\partial \mathbf{v}} & I_C - \Delta t \frac{\partial \mathbf{t}}{\partial \boldsymbol{\omega}} \end{pmatrix} \begin{pmatrix} \mathbf{v}_i \\ \boldsymbol{\omega}_i \end{pmatrix}.\end{aligned} \tag{C.7}$$

In cases in which rigid bodies are not connected through joints and springs, most of the partial derivatives of the forces, torques and cross terms are zero, leaving only the mass and inertia tensor. After solving this system for the unknown linear and angular velocities, the object is translated and rotated around its center of mass using $\Delta t \mathbf{v}_{i+1}$ and $\Delta t \boldsymbol{\omega}_{i+1}$.

For the following time-step, the inertia tensor must be recomputed due to the rotation of the object. Alternatively, the total rotation can be maintained by updating the total angle $\boldsymbol{\phi}_{i+1} = \boldsymbol{\phi}_i + \Delta t \boldsymbol{\omega}_{i+1}$. Using $\boldsymbol{\phi}_{i+1}$, a rotation matrix can be constructed which is used to compensate the inertia tensor for the rotation, i.e.,

$$I_{CR} = \mathbf{R} I_C \mathbf{R}^{-1}. \tag{C.8}$$

C.3 Center of Mass and Moment of Inertia

The previous subsection derived the equations of motion for rigid bodies. They solve for a linear velocity of the *center of mass* and an *angular velocity* around that center of mass. In the following paragraphs a brief overview is given about the computation of this center of mass and the momentum of inertia.

Center of mass The computation of the center of mass is straightforward. In case the object is described by triangles, each triangle forms a tetrahedron with a particular fixed point $\mathbf{x}_0$. Given the orientation of the three face vertices and the fourth fixed vertex, a tetrahedron is created with either a positive or negative volume. The total mass m of the object is the sum of all (signed) signed masses m_i of all tetrahedra. Next, the center of mass of the object is obtained by computing the weighted average of all centers of mass of each individual tetrahedron, i.e.,

$$\mathbf{x}_c = \sum_{i=1}^{n} \frac{\left(\mathbf{x}_{i,1} + \mathbf{x}_{i,2} + \mathbf{x}_{i,3} + \mathbf{x}_0\right) m_i}{4m}, \tag{C.9}$$

with $\mathbf{x}_{i,x}$ the vertices of tetrahedron i and $\mathbf{x}_0$ the common fixed vertex. At this center of mass, the weighted sum relative to $\mathbf{x}_c$ becomes zero, i.e.,

$$0 = \sum_{i=1}^{n} \frac{\left(\mathbf{x}_{i,1} + \mathbf{x}_{i,2} + \mathbf{x}_{i,3} + \mathbf{x}_0 - 4\mathbf{x}_c\right) m_i}{4m}. \tag{C.10}$$

Mass matrix The mass matrix $\mathbf{M}$ relates the linear acceleration of the rigid body with the linear forces applied on the body. This matrix is just $m\mathbf{I}_{3\times3}$ matrix with the mass of the object on the diagonal. Similarly one can relate the angular acceleration with the angular forces using the inertia tensor. This tensor depends on the shape of the object. The next paragraph describes how this inertia tensor is obtained.

Inertia tensor The computation of the inertia tensor requires the center of mass $\mathbf{x}_c$ of the object. Given each triangle of the object, a tetrahedron is created using the three vertices and the center of mass. Each tetrahedron can have a positive or negative volume. Given the volume of the tetrahedron, its mass can be computed.

Next the local inertia tensor of the tetrahedron is computed by first moving the center of mass $\mathbf{x}_{c,i}$ of the current tetrahedron to the origin. Then for each vertex of the tetrahedron a skew symmetric matrix is created, i.e.,

$$\mathrm{S}_j = \begin{pmatrix} 0 & -r_{j,z} & r_{j,y} \\ r_{j,z} & 0 & -r_{j,x} \\ -r_{j,y} & r_{j,x} & 0 \end{pmatrix}, \tag{C.11}$$

with $\mathbf{r}_j = \mathbf{x}_{i,j} - \mathbf{x}_{c,i}$ the vector between the origin and the translated vertex. Finally, the local inertia tensor is obtained using

$$I_i = -\sum_{j=1}^{4} \frac{\mathrm{S}_j \mathrm{S}_j m_i}{4} = \sum_{j=1}^{4} \frac{\mathrm{S}_j \mathrm{S}_j^T m_i}{4}, \tag{C.12}$$

with m_i the mass of the tetrahedron.

Given all local inertia tensors, the inertia tensor of the whole object is obtained using

$$I_C = \sum_{i=1}^{n} I_i + \mathbf{r}_i^T \mathbf{r}_i \mathbf{I} - \mathbf{r}_i \mathbf{r}_i^T m_i, \tag{C.13}$$

with $\mathbf{r}_i = \mathbf{x}_{c,i} - \mathbf{x}_c$ the vector between the center of mass of the tetrahedron and the center of mass of the object, $\mathbf{r}_j^T \mathbf{r}_j$ the squared length of $\mathbf{r}_j$ and $\mathbf{r}_j \mathbf{r}_j^T$ the outer product.

D

FEM Elasticity simulations

D.1 Introduction

In this chapter we provide additional background information about Linear Elasticity and its application to arbitrary objects using the Finite Element Method (FEM). First we briefly describe linear elasticity, followed by an overview of FEM as used in Chapters 4 and 5. Finally, an overview is given on methods for approximating non-linear hyper-elastic materials as used in Chapter 5.

D.2 Linear Elasticity

Strain The deformation of a body is measured by *strain*, which is defined as

$$\boldsymbol{\epsilon} = \frac{1}{2}\left((\nabla_{\mathbf{x}}\mathbf{u})^T + \nabla_{\mathbf{x}}\mathbf{u}\right), \tag{D.1}$$

with $\mathbf{u} = [u, v, w]^T$ the relative displacement of a point in the deformed body, $\nabla_{\mathbf{x}}\mathbf{u}$ the *material displacement gradient tensor* and $\boldsymbol{\epsilon}$ the dimensionless *strain tensor* describing *normal strain* and *shear strain*. Normal strain can be described by

$$\epsilon_{xx} = \frac{\partial u}{\partial x}, \epsilon_{yy} = \frac{\partial v}{\partial y}, \epsilon_{zz} = \frac{\partial w}{\partial z}, \tag{D.2}$$

and shear strain is described by

$$\epsilon_{xy} = \frac{1}{2}\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right), \epsilon_{xz} = \frac{1}{2}\left(\frac{\partial u}{\partial x} + \frac{\partial w}{\partial z}\right), \epsilon_{yz} = \frac{1}{2}\left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z}\right). \tag{D.3}$$

The strain tensor now becomes:

$$\boldsymbol{\epsilon} = \begin{pmatrix} \epsilon_{xx} & \epsilon_{xy} & \epsilon_{xz} \\ \epsilon_{yx} & \epsilon_{yy} & \epsilon_{yz} \\ \epsilon_{zx} & \epsilon_{zy} & \epsilon_{zz} \end{pmatrix}. \tag{D.4}$$

Stress The application of an external force on a body results in stress inside the body. In *continuum mechanics*, stress is described by stress vector $\mathbf{t}^{(\mathbf{n})} = \mathbf{n}.\boldsymbol{\sigma}$, across an arbitrary plane described by normal vector $\mathbf{n}$, with

$$\boldsymbol{\sigma} = \begin{pmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} \end{pmatrix} = \begin{pmatrix} \sigma_x & \tau_{xy} & \tau_{xz} \\ \tau_{yx} & \sigma_y & \tau_{yz} \\ \tau_{zx} & \tau_{zy} & \sigma_z \end{pmatrix} \tag{D.5}$$

the so called *Cauchy stress tensor*. Alternatively, the stress tensor can be described using its *normal stress* components,

$$\boldsymbol{\sigma}_{\mathbf{n}} = \frac{df_n}{dS}, \tag{D.6}$$

with df_n the normal component of $d\mathbf{f}$ to differential area dS. Furthermore, the *shear stress* is given by

$$\boldsymbol{\tau} = \frac{df_s}{dS}, \tag{D.7}$$

with df_s the tangential component of $d\mathbf{f}$, which is decomposed in two perpendicular components.

The equilibrium state follows from the principle of *conservation of linear momentum*, i.e.,

$$\begin{aligned}\frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} + \frac{\partial \sigma_{xz}}{\partial z} + f_x = 0\\ \frac{\partial \sigma_{yx}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{yz}}{\partial z} + f_y = 0\\ \frac{\partial \sigma_{zx}}{\partial x} + \frac{\partial \sigma_{zy}}{\partial y} + \frac{\partial \sigma_{zz}}{\partial z} + f_z = 0,\end{aligned} \tag{D.8}$$

which is $\nabla \cdot \boldsymbol{\sigma} + \mathbf{f} = \mathbf{0}$, with $\mathbf{f}$ the forces inside the body. The boundary conditions associated with Equation (D.8) directly follow from the definition of the stress vector $\mathbf{t^n}$, i.e.,

$$\begin{aligned}\sigma_{xx}n_x + \sigma_{xy}n_y + \sigma_{xz}n_z = t_x\\ \sigma_{yx}n_x + \sigma_{yy}n_y + \sigma_{yz}n_z = t_y\\ \sigma_{zx}n_x + \sigma_{zy}n_y + \sigma_{zz}n_z = t_z,\end{aligned} \tag{D.9}$$

with $\mathbf{t}$ the stress / force applied on the boundary of the object. Furthermore, due to the principle of *conservation of angular momentum*, the stress tensor is symmetric, i.e., $\sigma_{ij} = \sigma_{ji}$.

Hooke's law Since the normal and angular strain can be measured given the deformation, Hooke's law is used to relate the stress and strain given some material parameters of the deformed body, i.e., $\boldsymbol{\sigma} = \mathbf{D}\boldsymbol{\epsilon}$, with $\mathbf{D}$ the *material stiffness matrix*. If the unique components of the stress and strain tensors are described in *Voigt notation*, this relation becomes

$$\begin{pmatrix}\sigma_{xx}\\ \sigma_{yy}\\ \sigma_{zz}\\ \sigma_{xy}\\ \sigma_{xz}\\ \sigma_{yz}\end{pmatrix} = \begin{pmatrix}d_{11} & d_{12} & d_{12} & 0 & 0 & 0\\ d_{12} & d_{11} & d_{12} & 0 & 0 & 0\\ d_{12} & d_{12} & d_{11} & 0 & 0 & 0\\ 0 & 0 & 0 & d_{44} & 0 & 0\\ 0 & 0 & 0 & 0 & d_{44} & 0\\ 0 & 0 & 0 & 0 & 0 & d_{44}\end{pmatrix}\begin{pmatrix}\epsilon_{xx}\\ \epsilon_{yy}\\ \epsilon_{zz}\\ \epsilon_{xy}\\ \epsilon_{xz}\\ \epsilon_{yz}\end{pmatrix}, \tag{D.10}$$

with $\mathbf{D}$ the *material stiffness matrix* for isotropic materials. The unique parameters in $\mathbf{D}$ can be described using Young's modulus of elasticity E and Poisson's ratio ν, i.e.,

$$d_{11} = \frac{E(1-\nu)}{(1+\nu)(1-2\nu)}, d_{12} = \frac{E(\nu)}{(1+\nu)(1-2\nu)}, d_{44} = \frac{E}{(1+\nu)}. \tag{D.11}$$

Please note that many other types of material stiffness matrices exist which are also used for anisotropic materials. Within this dissertation we only focus on linear elasticity based on isotropic materials.

D.3 Finite Element Method

The Finite Element Method (FEM) [143] is a numerical method for approximating the solution of partial differential equations over arbitrary shaped domains. The method divides the computational domain in a large number of arbitrary shaped elements, which approximate the solution using linear or higher-order shape functions. The method uses the so called *Weak formulation* of the problem, which is a less strict version of the original PDE and has for example less restrictions on the smoothness. Using the *Galerkin form* of the weak statement, only a finite number of parameters are needed to compute in order to approximate the solution. In this section we briefly describe this process for the linear elasticity problem presented in the previous section.

Weighted residuals Suppose the computational domain is divided in a finite number of non-overlapping sub-intervals with each sub-interval a so called *element*. The endpoints of these intervals are the nodes or vertices of the elements. For each node the solution of the differential equation is approximated using a number of predefined shape functions and the same number of unknown parameters. Once the parameters are known, the approximation of the solution is also known at the nodes or vertices of the elements. However, the approximation is not exactly the true solution everywhere in the domain and some residual remains. Using a so called *weighting function* the residual is weighted such that

$$\int \phi(x) R(Z(x), x)\, dx = 0, \tag{D.12}$$

with $\phi(x)$ a weighting or test function, $R(x)$ the residual of the PDE to be weighted and $Z(x)$ the solution quantity that is approximated by the PDE. This quantity is defined over the whole domain as:

$$Z(x) \approx \sum_{i}^{n} z_i \psi_i(x), \tag{D.13}$$

with ψ the shape functions and z the unknown coefficients. The weighted residual function is then solved for the unknown coefficients z such that the weighted residual will vanish over the whole domain. The choice of this weighting function depends on the used procedure. In our case we use the *Galerkin* procedure, which uses the shape functions as weighting functions. The advantage of this procedure is that it yields a system of equations that has the same number of equations as unknowns. This guarantees the existence and uniqueness of the approximation of the solution, if the boundary conditions are specified correctly. The shape functions have a certain order, e.g., linear, cubic, quadratic or higher order, and depend on the shape and size of the element. The elements can be simple 1 dimensional rods, or 3 dimensional tetrahedra or cubes. Since the derivatives of these functions are in general not continuous at the nodes or vertices, and higher order derivatives might not exist, the differential equation is transformed into a weaker form, which has weaker requirements with respect to the derivatives of the approximation. This form is in general obtained using *integration by parts* or in higher dimensions using *Green's theorem* (2D) or *Gauss' divergence theorem* (3D). In the next paragraph we derive the weak form of the elasticity problem shown in Equations (D.8) and (D.9).

Weak form of Linear Elasticity For convenience, Equation (D.8) can be written in matrix-form, in which operator $\mathbf{P}$ contains the partial derivative operations, i.e.,

$$\mathbf{P} = \begin{pmatrix} \frac{\partial}{\partial x} & 0 & 0 & \frac{\partial}{\partial y} & \frac{\partial}{\partial z} & 0 \\ 0 & \frac{\partial}{\partial y} & 0 & \frac{\partial}{\partial x} & 0 & \frac{\partial}{\partial z} \\ 0 & 0 & \frac{\partial}{\partial z} & 0 & \frac{\partial}{\partial x} & \frac{\partial}{\partial y} \end{pmatrix}^T, \tag{D.14}$$

resulting in

$$\mathbf{P}^T \mathbf{D} \mathbf{P} \mathbf{u} - \mathbf{f} = \mathbf{0}, \tag{D.15}$$

which contains second and mixed derivatives of the solution $\mathbf{u}$. This solution is approximated using a finite number of non-overlapping elements which cover the complete computational domain. These elements have their shape functions $\mathbf{N}$, and together with the unknown nodal parameters $\hat{\mathbf{u}}$ they approximate the solution using the shape functions, i.e.,

$$\mathbf{u} = \mathbf{N}\hat{\mathbf{u}}. \tag{D.16}$$

Please note that the dimensions of $\mathbf{N}$ and $\hat{\mathbf{u}}$ depend on the shape and the order of the used elements. Next, the residuals are weighted by a test function and integrated over the domain. Using the Galerkin procedure the shape functions are also used as test functions,

$$\int_V \mathbf{N}^T \left(\mathbf{P}^T\mathbf{D}\mathbf{P}\mathbf{N}\hat{\mathbf{u}}\right) - \mathbf{N}^T\mathbf{N}\hat{\mathbf{f}}\, dV = \mathbf{0}, \tag{D.17}$$

such that the weighted residuals will vanish for the obtained solution. Finally, using Gauss' divergence theorem, Green's first identity is applied, resulting in:

$$\int_V (\mathbf{P}\mathbf{N})^T (\mathbf{D}\mathbf{P}\mathbf{N}\hat{\mathbf{u}}) + \mathbf{N}^T \left(\mathbf{P}^T\mathbf{D}\mathbf{P}\mathbf{N}\hat{\mathbf{u}}\right) dV = \int_A \left(\mathbf{N}^T\mathbf{D}\mathbf{P}\mathbf{N}\hat{\mathbf{u}}\right) \cdot \mathbf{n}\, dA = \int_A \mathbf{N}^T\mathbf{N}\hat{\mathbf{t}}\, dA. \tag{D.18}$$

This is a weak form of Equation (D.17) in which the second and mixed derivatives of the approximation vanish and additional first derivatives of the shape functions and boundary conditions appear, i.e.,

$$\int_V (\mathbf{P}\mathbf{N})^T\, \mathbf{D}\, (\mathbf{P}\mathbf{N}\hat{\mathbf{u}}) - \mathbf{N}^T\mathbf{N}\hat{\mathbf{f}}\, dV - \int_A \mathbf{N}^T\mathbf{N}\hat{\mathbf{t}}\, dA = \mathbf{0}, \tag{D.19}$$

with $\mathbf{P}\mathbf{N}$ the first derivatives of the shape functions. This system becomes

$$\mathbf{K}\hat{\mathbf{u}} = \mathbf{f} \tag{D.20}$$

in shorthand, with $\mathbf{K}$ the stiffness matrix, $\hat{\mathbf{u}}$ the nodal displacement and $\mathbf{f}$ the vector containing the body and external boundary forces. All internal boundary conditions between elements will vanish, and remain for the outer boundary of the domain.

Linear Shape functions Per element a set of shape functions are defined which, together with some unknown parameters, approximate the solution, see Equation (D.16). The shape functions depend on the shape and order of the element. In this dissertation we use three-dimensional elements (tetrahedra) on which we define linear shape functions. Per tetrahedron four nodes are used and possibly shared with neighboring elements. The shape functions define the (piece-wise) interpolation of a quantity ϕ inside the element at position (x, y, z), i.e.,

$$\phi = \alpha_1 + x\alpha_2 + y\alpha_3 + z\alpha_4, \tag{D.21}$$

with α the coefficients of the linear function. At each vertex node we have

$$\begin{aligned}
\phi_1 &= \alpha_1 + x_1\alpha_2 + y_1\alpha_3 + z_1\alpha_4 \\
\phi_2 &= \alpha_1 + x_2\alpha_2 + y_2\alpha_3 + z_2\alpha_4 \\
\phi_3 &= \alpha_1 + x_3\alpha_2 + y_3\alpha_3 + z_3\alpha_4 \\
\phi_4 &= \alpha_1 + x_4\alpha_2 + y_4\alpha_3 + z_4\alpha_4.
\end{aligned} \tag{D.22}$$

At (x_1, y_1, z_1), ϕ should evaluate to ϕ_1, whereas the other ϕ's evaluate to zero. This generalizes to

$$\boldsymbol{\phi} = \mathbf{C}\boldsymbol{\alpha}, \tag{D.23}$$

with $\mathbf{C}$ containing the coordinates of the nodes of the element. Using

$$\boldsymbol{\alpha} = \mathbf{C}^{-1}\boldsymbol{\phi}, \tag{D.24}$$

the unknown constant and linear coefficients of the element are obtained. In order to obtain the shape functions $\mathbf{N}$, quantity $\boldsymbol{\phi}$ is transformed into a function containing the nodal values ϕ_1, ϕ_2, ϕ_3 and ϕ_4 such that

$$\phi(x,y,z) = N_1(x,y,z)\phi_1 + N_2(x,y,z)\phi_2 + N_3(x,y,z)\phi_3 + N_4(x,y,z)\phi_4 \tag{D.25}$$

gives the interpolated value of ϕ at (x,y,z), with $N_i(x,y,z)$ the shape functions at the corresponding node. In order to obtain these shape functions, the following relation is used:

$$\begin{aligned}
\phi(x,y,z) &= \begin{pmatrix} 1 & x & y & z \end{pmatrix} \boldsymbol{\alpha} \\
\phi(x,y,z) &= \begin{pmatrix} 1 & x & y & z \end{pmatrix} \mathbf{C}^{-1} \boldsymbol{\phi} \\
\phi(x,y,z) &= \mathbf{N}(x,y,z)\boldsymbol{\phi},
\end{aligned} \tag{D.26}$$

with

$$\mathbf{N}(x,y,z) = \begin{pmatrix} 1 & x & y & z \end{pmatrix} \mathbf{C}^{-1} = \begin{pmatrix} N_1 & N_2 & N_3 & N_4 \end{pmatrix}, \tag{D.27}$$

which is a vector containing the values of all shape functions at (x,y,z). Furthermore, if (x,y,z) represents the coordinate of a particular node of the element, Equation (D.25) returns the corresponding nodal value of ϕ. Please note that ϕ can represent any quantity with any dimensionality. In case of Equation (D.16), in which ϕ represents the three dimensional deformation, the interpolation becomes:

$$\begin{aligned}
u(x,y,z) &= N_1 u_1 + N_2 u_2 + N_3 u_3 + N_4 u_4 \\
v(x,y,z) &= N_1 v_1 + N_2 v_2 + N_3 v_3 + N_4 v_4 \\
w(x,y,z) &= N_1 w_1 + N_2 w_2 + N_3 w_3 + N_4 w_4,
\end{aligned} \tag{D.28}$$

which in matrix form becomes

$$\mathbf{u} = \begin{pmatrix} u \\ v \\ w \end{pmatrix} = \begin{pmatrix} N_1 & 0 & 0 & & N_4 & 0 & 0 \\ 0 & N_1 & 0 & \dots & 0 & N_4 & 0 \\ 0 & 0 & N_1 & & 0 & 0 & N_4 \end{pmatrix} \begin{pmatrix} u_1 \\ v_1 \\ w_1 \\ \vdots \\ u_4 \\ v_4 \\ w_4 \end{pmatrix}. \tag{D.29}$$

Shape-function derivatives When the problem is discretized using FEM, the shape functions are not used directly. Instead, their derivatives and integrals at the nodes are used. For example, the derivative of ϕ becomes

$$\frac{\partial \phi(x,y,z)}{\partial x} = \frac{\partial N_1}{\partial x}\phi_1 + \frac{\partial N_2}{\partial x}\phi_2 + \frac{\partial N_3}{\partial x}\phi_3 + \frac{\partial N_4}{\partial x}\phi_4. \tag{D.30}$$

Since the shape functions depend on x, see Equation (D.27), and ϕ is constant at the nodes, the derivative becomes

$$\frac{\partial \phi(x,y,z)}{\partial x} = \frac{\partial \begin{pmatrix} 1 & x & y & z \end{pmatrix}}{\partial x} \mathbf{C}^{-1} \boldsymbol{\phi} = \begin{pmatrix} 0 & 1 & 0 & 0 \end{pmatrix} \mathbf{C}^{-1} \boldsymbol{\phi}, \tag{D.31}$$

which boils down to a row in matrix $\mathbf{C}^{-1}$. Similarly, the derivatives of the shape functions to y and z are obtained.

Volume integration As shown in Equation (D.19), the shape functions (and their derivatives) are integrated over the volume of the element. In case of linear shape functions, the derivatives result in constant terms. This make the integration of the first term of Equation (D.19) trivial. In case the shape functions are used directly, the shape functions need to be integrated over the volume. When a volume coordinate-system is used instead, the integration becomes easier. For linear elements the four volume coordinates are related by:

$$\begin{aligned} x &= L_1x_1+L_2x_2+L_3x_3+L_4x_4, \\ y &= L_1y_1+L_2y_2+L_3y_3+L_4y_4, \\ z &= L_1z_1+L_2z_2+L_3z_3+L_4z_4, \\ 1 &= L_1 \quad +L_2 \quad +L_3 \quad +L_4, \end{aligned} \tag{D.32}$$

which simply give the shape functions $N_1 = L_1, N_2 = L_2, N_3 = L_3, N_4 = L_4$. These volume coordinates are similar to the Barycentric coordinates of a tetrahedron. For example, the integral $\int_V \mathbf{N}^T\mathbf{N}\, dV$ expands to

$$\int_V \begin{pmatrix} L_1^2 & L_1L_2 & L_1L_3 & L_1L_4 \\ L_2L_1 & L_2^2 & L_2L_3 & L_2L_4 \\ L_3L_1 & L_3L_2 & L_3^2 & L_3L_4 \\ L_4L_1 & L_4L_2 & L_4L_3 & L_4^2 \end{pmatrix} dV. \tag{D.33}$$

Using numerical integration, the individual terms in Equation (D.33) can be integrated over the volume. In case of simple expressions (linear shape functions), the integral can be evaluated directly using:

$$\int_V L_1^a L_2^b L_3^c L_4^d \, dV = \frac{a!b!c!d!}{(3+a+b+c+d)!} 6V_e, \tag{D.34}$$

with V_e the volume of the tetrahedron. The expression in Equation (D.33) therefore becomes

$$\begin{pmatrix} 0.10 & 0.05 & 0.05 & 0.05 \\ 0.05 & 0.10 & 0.05 & 0.05 \\ 0.05 & 0.05 & 0.10 & 0.05 \\ 0.05 & 0.05 & 0.05 & 0.10 \end{pmatrix} V_e, \tag{D.35}$$

which only depends on the volume of the element. In case of higher-order elements or when the number of terms are large, a numerical integration scheme, such as Gaussian Quadrature, must be used instead. For more details we refer to [143]. Similarly, $\int_V \mathbf{N}^T\, dV$ becomes

$$\int_V \begin{pmatrix} L_1 & L_2 & L_3 & L_4 \end{pmatrix}^T dV = \begin{pmatrix} 0.25 & 0.25 & 0.25 & 0.25 \end{pmatrix}^T V_e. \tag{D.36}$$

In case of higher dimensional problems, like the elasticity problem in Equation (D.19), the shape functions (and their derivatives) are integrated over the volume of the elements, resulting in 12×12 and 3×12 matrices per element, see Equation (D.29).

Time integration The system described in Equation (D.20) computes the displacement of the nodes, given some external load and other boundary conditions. The system itself does not account for time and dynamics, therefore it computes the deformation after an infinite amount of time. To cope with the change of the material over time, Equation (D.20) is time-integrated using a semi-implicit scheme. In general, a body is in a natural rest state if

$$m\mathbf{a} + c\mathbf{v} + \mathbf{P}^T\mathbf{DPu} = \mathbf{f}, \tag{D.37}$$

with m the mass of the object, $\mathbf{a}$ the acceleration, c a damping constant, $\mathbf{v}$ the velocity, $\mathbf{P}^T\mathbf{DPu}$ the elasticity forces and $\mathbf{f}$ the external forces. First, the acceleration and displacement are rewritten in terms of the velocity, i.e., $\mathbf{a} = \frac{\mathbf{v}_{i+1}-\mathbf{v}_i}{\Delta t}$ and $\mathbf{u} = \Delta t\mathbf{v}_{i+1} + \mathbf{x}_i - \mathbf{x}_0$. This results in the following system:

$$m\frac{\mathbf{v}_{i+1}-\mathbf{v}_i}{\Delta t} + c\mathbf{v}_{i+1} + \mathbf{P}^T\mathbf{DP}\left(\Delta t\mathbf{v}_{i+1} + \mathbf{x}_i - \mathbf{x}_0\right) = \mathbf{f}, \tag{D.38}$$

with $\mathbf{v}_{i+1}$ the unknown velocity, $\mathbf{v}_i$ the velocity of the previous time-step, $\mathbf{x}_0$ the initial positions and $\mathbf{x}_i$ the nodal positions at the previous time-step. This can be rearranged as

$$\left(m + \Delta tc + \Delta t^2\mathbf{P}^T\mathbf{DP}\right)\mathbf{v}_{i+1} = \mathbf{m}\mathbf{v}_i + \Delta t\mathbf{P}^T\mathbf{DP}\left(\mathbf{x}_i - \mathbf{x}_0\right) + \Delta t\mathbf{f}, \tag{D.39}$$

which is the semi-implicit time integrated form of Equation (D.19). Next, the Galerkin procedure is performed and finally the weak-form is obtained, as described previously, i.e.,

$$\begin{aligned}\int_V \mathbf{N}^T\left(m + \Delta tc + \Delta t^2\mathbf{P}^T\mathbf{DP}\right)\mathbf{N}\hat{\mathbf{v}}_{i+1}\,dV = \\ m\int_V \mathbf{N}^T\mathbf{N}\hat{\mathbf{v}}_i\,dV + \Delta t\int_V (\mathbf{PN})^T\,\mathbf{DPN}\left(\hat{\mathbf{x}}_i - \hat{\mathbf{x}}_0\right)\,dV + \int_V \Delta t\mathbf{N}^T\mathbf{N}\hat{\mathbf{f}}\,dV,\end{aligned} \tag{D.40}$$

which can be reordered as:

$$\begin{aligned}m\int_V \mathbf{N}^T\mathbf{N}\hat{\mathbf{v}}_{i+1}\,dV + \Delta tc\int_V \mathbf{N}^T\mathbf{N}\hat{\mathbf{v}}_{i+1}\,dV + \Delta t^2\int_V (\mathbf{PN})^T\,\mathbf{DPN}\hat{\mathbf{v}}_{i+1}\,dV = \\ m\int_V \mathbf{N}^T\mathbf{N}\hat{\mathbf{v}}_i\,dV + \Delta t\int_V (\mathbf{PN})^T\,\mathbf{DPN}\left(\hat{\mathbf{x}}_i - \hat{\mathbf{x}}_0\right)\,dV + \int_V \Delta t\mathbf{N}^T\mathbf{N}\hat{\mathbf{f}}\,dV,\end{aligned} \tag{D.41}$$

with m and c the element mass and damping factor respectively. Please note that also a nodal representation of the velocity, positions and forces is used, i.e., $\hat{\mathbf{v}}$, $\hat{\mathbf{x}}$ and $\hat{\mathbf{f}}$. This results in:

$$\left(\mathbf{M} + \Delta t\mathbf{C} + \Delta t^2\mathbf{K}\right)\hat{\mathbf{v}}_{i+1} = \mathbf{M}\hat{\mathbf{v}}_i + \Delta t\mathbf{K}\left(\hat{\mathbf{x}}_i - \hat{\mathbf{x}}_0\right) + \Delta t\mathbf{f}, \tag{D.42}$$

with $\mathbf{M} \in \mathbb{R}^{12\times12}$ the element mass matrix, $\mathbf{C} \in \mathbb{R}^{12\times12}$ the element damping matrix, $\mathbf{K} \in \mathbb{R}^{12\times12}$ the element stiffness matrix, $\mathbf{f} \in \mathbb{R}^{1\times12}$ the integrated nodal external load. Per element the mass matrix is computed using $m\int_V \mathbf{N}^T\mathbf{N}\,dV$, see Equation (D.35). Similarly, the damping matrix is obtained. In the rest of this chapter only the nodal values of all quantities are used, therefore we drop the $\hat{\cdot}$ notation in order to improve readability.

Assembled system For each element in the discretized model, a small linear system is obtained. Since nodes are shared between elements, a global linear system is assembled based on all local systems. Each local linear system acts on a shared set of nodes, which are part of the global system. The values of the global system are obtained by adding all local systems to the global system, respecting the unique ids of the nodes. The obtained system can then be solved and gives an approximation that minimizes the residual with respect to all nodes. In the rest of this dissertation we do not consider the individual elements but work with the global vectors and matrices instead.

D.3.1 Corotational Finite Elements

The system described in the previous paragraphs describes the deformation of a body over time. At each time-step the configuration of the mesh (with its vertex positions stored in $\mathbf{x}$) changes shape and orientation. Since this change in orientation also changes the reference

frame for the computation of the elastic forces, we must compensate for this rotation, as described by Muller et al.[124]. The main idea is to compute the elastic forces using the orientation of the element described at $\mathbf{x}_0$. Given the orientation corresponding to the configuration at $\mathbf{x}_i$, the deformation $\mathbf{u}$ is rotated back using $\mathbf{r}^T$ to the initial orientation of the element. After computing the elastic force in the initial orientation, the elastic force is rotated to the current orientation of the element using $\mathbf{r}$. The rotation of a particular element with respect to its initial configuration is stored per element in $\mathbf{r}$, which is an orthogonal 12×12 matrix with 4 instances of the same 3×3 rotation matrix on the main diagonal. This gives the following co-rotational elastic force:

$$\mathbf{f} = \mathbf{RK}(\mathbf{R}^T\mathbf{x}_i - \mathbf{x}_0), \tag{D.43}$$

with $\mathbf{R}$ a global matrix assembled using all local rotations $\mathbf{r}$. Please note that $\mathbf{x}_0$ reflects the configuration of the element at their initial configuration, therefore no additional rotation is required for $\mathbf{x}_0$.

Rotation retrieval Rotation matrix $\mathbf{r}$ in the previous paragraph describes the rotation of the element with respect to its initial configuration. The rotation, together with the deformation, form the complete transformation $\mathbf{F}$ from the initial configuration to its current one. Matrix $\mathbf{F}$ is also known as the *Deformation Gradient Tensor*, see next section. In order to obtain the rotational component of the transformation, a Polar Decomposition can be used. The Polar Decomposition decomposes the transformation $\mathbf{F}$ into a unitary (orthogonal) and a Hermitian (symmetric) matrix, and can be obtained using a Singular Value Decomposition (SVD) of the transformation matrix $\mathbf{F}$. The rotation is obtained using $\mathbf{r} = \mathbf{UV}^T$, with $\mathbf{U}$ and $\mathbf{V}$ the matrices of the SVD. If the element undergoes an inversion, the rotation matrices may contain a reflection. To deal with these situations, a correction is applied, see [95] and the next section for more details.

Implicit time-integration The rotation matrix $\mathbf{R}$ is included in Equation (D.42) resulting in:

$$\left(\mathbf{M} + \Delta t\mathbf{C} + \Delta t^2\mathbf{RKR}^T\right)\mathbf{v}_{i+1} = \mathbf{M}\mathbf{v}_i + \Delta t\mathbf{RK}\left(\mathbf{R}^T\mathbf{x}_i - \mathbf{x}_0\right) + \Delta t\mathbf{f}, \tag{D.44}$$

which becomes

$$\mathbf{A}\mathbf{v}_{i+1} = \mathbf{b}, \tag{D.45}$$

in short, with $\mathbf{A}$ a symmetric and positive definite matrix. The system is solved for $\mathbf{v}_{i+1}$, followed by an update of $\mathbf{x}_{i+1}$ using $\mathbf{x}_{i+1} = \mathbf{x}_i + \Delta t\mathbf{v}_{i+1}$.

The system in Equation (D.44) also contains an approximation of the elastic force derivatives

$$\frac{\partial \mathbf{f}}{\partial t} \approx \mathbf{RKR}^T\mathbf{v}_{i+1}. \tag{D.46}$$

However, since the forces depend on the rotations, and the deformation changes the rotation of the elements, also the derivatives of the rotation matrices must be included, as mentioned in [12], i.e.,

$$\frac{\partial \mathbf{f}}{\partial t} = \frac{\partial \mathbf{RK}(\mathbf{R}^T\mathbf{x}_{i+1} - \mathbf{x}_0)}{\partial \mathbf{x}}\frac{\partial \mathbf{x}}{\partial t}. \tag{D.47}$$

Instead of computing the derivatives $\frac{\partial \mathbf{R}}{\partial \mathbf{x}}$ directly, the derivatives with respect to the deformation gradient tensor are used: $\frac{\partial \mathbf{R}}{\partial F_{i,j}}\frac{\partial F_{i,j}}{\partial \mathbf{x}}$. Since the rotation is obtained using the polar decomposition of $\mathbf{F}$, also the gradients of the SVD operation are required. In the next section a summary is given on the computation of hyper-elastic material, which includes a description of the SVD derivatives.

D.4 Non-Linear (Hyper) Elasticity

The elasticity model described in Appendix D.2 is a widely used approach in Computer Graphics for deforming elastic objects. Since the underlying elasticity model is linear, the model is not accurate in cases with large deformations; it is possible that elements can invert, resulting in an incorrect visualization of the simulation, an incorrect force computation and possibly oscillations in the simulation. A discussion on the behavior of energy models when elements invert can be found in [168]. Inversions are possible due to the constant stiffness of the material. In cases of large compressions, the (constant) stiffness of the material is not sufficient to prevent a collapse of the elements. This suggests that the stiffness should increase due to compression of the material, i.e., the stiffness should in theory approach infinity while the volume of the elements approach zero. These non-linear materials can be modeled using Piola-Kirchhoff stress which is the derivative of some strain energy density function, which can be defined for any kind of material.

Deformation gradient tensor Considering a point $\mathbf{X}$ in the undeformed object (reference configuration), there exists a map ϕ to the same point $\mathbf{x}$ in the deformed object, as $\mathbf{x} = \phi(\mathbf{X})$. The total displacement of a point X to x can be described by:

$$\begin{aligned} \mathbf{u} &= \mathbf{x} - \mathbf{X} + \mathbf{b} \\ \mathbf{u} &= \phi(\mathbf{X}) - \mathbf{X} + \mathbf{b}, \end{aligned} \tag{D.48}$$

with $\mathbf{b}$ a linear translation. The deformation gradient tensor is then obtained through the *material displacement gradient tensor* of $\nabla_{\mathbf{X}}\mathbf{u}$, i.e.,

$$\begin{aligned} \nabla_{\mathbf{X}}\mathbf{u} &= \nabla_{\mathbf{X}}\phi(\mathbf{X}) - \nabla_{\mathbf{X}}\mathbf{X} + \nabla_{\mathbf{X}}\mathbf{b} \\ \nabla_{\mathbf{X}}\mathbf{u} &= \nabla_{\mathbf{X}}\phi(\mathbf{X}) - \mathbf{I} \\ \nabla_{\mathbf{X}}\mathbf{u} &= \mathbf{F} - \mathbf{I}, \end{aligned} \tag{D.49}$$

with $\mathbf{F} = \frac{\partial\phi(\mathbf{X})}{\partial\mathbf{X}}$ the *deformation gradient tensor*. When the space is discretized using, e.g., tetrahedral elements, the deformation gradient tensor relates the undeformed tetrahedron $\mathbf{X}$ with the deformed configuration $\mathbf{x}$, i.e., $\mathbf{D}_s = \mathbf{F}\mathbf{D}_m$, with

$$\mathbf{D}_m = \begin{pmatrix} X_{1,x} - X_{0,x} & X_{2,x} - X_{0,x} & X_{3,x} - X_{0,x} \\ X_{1,y} - X_{0,y} & X_{2,y} - X_{0,y} & X_{3,y} - X_{0,y} \\ X_{1,z} - X_{0,z} & X_{2,z} - X_{0,z} & X_{3,z} - X_{0,z} \end{pmatrix} \tag{D.50}$$

the matrix containing the edges of the undeformed tetrahedron. Similarly, $\mathbf{D}_s$ is defined as:

$$\mathbf{D}_s = \begin{pmatrix} x_{1,x} - x_{0,x} & x_{2,x} - x_{0,x} & x_{3,x} - x_{0,x} \\ x_{1,y} - x_{0,y} & x_{2,y} - x_{0,y} & x_{3,y} - x_{0,y} \\ x_{1,z} - x_{0,z} & x_{2,z} - x_{0,z} & x_{3,z} - x_{0,z} \end{pmatrix}. \tag{D.51}$$

Using $\mathbf{F} = \mathbf{D}_s\mathbf{D}_m^{-1}$ the deformation gradient tensor is obtained for a single tetrahedron.

Finite volume method The finite volume method is very similar to the finite element method and also relies on the *divergence theorem*. The divergence of a vector field in a closed volume equals the outward flux of the same vector field through the closed surface of the same volume, i.e.,

$$\int_V (\nabla \cdot \mathbf{Z})\, dV = \int_S \mathbf{Z} \cdot \mathbf{n}\, dS, \tag{D.52}$$

for some vector field $\mathbf{Z}$ and $\mathbf{n}$ the surface normal. Given the Cauchy stress tensor $\boldsymbol{\sigma}$, the surface traction $\mathbf{t}$ on boundary S is expressed as $\mathbf{t} = \boldsymbol{\sigma}\mathbf{n}$, with $\mathbf{n}$ the surface normal. Furthermore, the force at a specific location, associated with a certain volume, is computed as:

$$\mathbf{f} = \frac{\partial}{\partial t} \iiint_x \rho \mathbf{v}\, d\mathbf{x} = \int_S \boldsymbol{\sigma}\mathbf{n}\, dS = \int_S \mathbf{t}\, dS, \tag{D.53}$$

with ρ the density of the material. If this volume coincides with one tetrahedron, the Cauchy stress tensor $\boldsymbol{\sigma}$ is constant, hence, the net traction and force are zero and the mass is constant, i.e., traction $\mathbf{t}$ on the surface is divergence-free. From this observation, the traction on all surfaces of one element can be related to each other by:

$$\sum_{i=1}^{4} \int_{S_i} \boldsymbol{\sigma}\mathbf{n}_i\, dS_i = \mathbf{0}, \tag{D.54}$$

for all surfaces S_i of a tetrahedron. To compute the force acting on a certain vertex j of the tetrahedron, the average normal stress of the three incident faces connected to vertex j is computed by:

$$\mathbf{f}_j = -\frac{1}{3}\boldsymbol{\sigma}\left(a_1\mathbf{n}_1 + a_2\mathbf{n}_2 + a_3\mathbf{n}_3\right), \tag{D.55}$$

with a_i the area of the associated face incident to vertex j. By accumulating all forces applied on vertex j resulting from all stresses in the connected tetrahedra, the net force $\mathbf{f}_j$ is obtained.

Piola-Kirchhoff stress Where the Cauchy stress tensor measures stress at the current (deformed) configuration of the object, the First Piola-Kirchhoff stress tensor measures the stress relative to some reference configuration of the object. Using the deformation gradient tensor $\mathbf{F}$, quantities can be transformed from the reference configuration to the current deformed configuration. Similarly, the change in volume of the reference configuration to the current configuration is expressed by $dv = JdV$, with $J = \det(\mathbf{F})$ the change in volume, dv the deformed volume and dV the reference volume. Eventually, using Nanson's relation, $da\ \mathbf{n} = JdA\mathbf{F}^{-T}\mathbf{N}$, the reference normal $\mathbf{N}$ and area dA can be used to obtain the current configuration of the normal $\mathbf{n}$ and area da through the deformation gradient tensor and the change in volume J.

Since the traction $\mathbf{t}$ on the surface is computed using $\mathbf{t} = \boldsymbol{\sigma}\mathbf{n}$ using the current configuration of the material, we can use Nanson's relation to obtain the same traction using the reference configuration, i.e.,

$$\begin{aligned} \mathbf{t} &= J\boldsymbol{\sigma}\mathbf{F}^{-T}\mathbf{N} \\ \mathbf{t} &= \mathbf{P}\mathbf{N}, \end{aligned} \tag{D.56}$$

with $\mathbf{P} = J\boldsymbol{\sigma}\mathbf{F}^{-T}$ the First Piola-Kirchhoff stress tensor, which computes a stress given a reference configuration and the deformation gradient tensor between the reference and current configuration. Since a rotation of the object is encoded into the deformation gradient tensor, this stress tensor is not invariant to rotations. Furthermore, since $\mathbf{P}$ relates two configurations, the tensor is a *two-point tensor* and is therefore not symmetric.

The Second Piola-Kirchhoff stress tensor $\mathbf{S}$ computes stress only for a given reference configuration. The Second Piola-Kirchhoff stress tensor is defined as $\mathbf{S} = J\mathbf{F}^{-1}\boldsymbol{\sigma}\mathbf{F}^{-T}$. Since $\boldsymbol{\sigma}$ is symmetric, $\mathbf{S}$ is also symmetric, which implies that the Second Piola-Kirchhoff stress tensor is

rotation invariant. Given the definition of the First and Second Piola-Kirchhoff stress tensors, the First Piola-Kirchhoff stress tensor can be expressed in terms of the Second Piola-Kirchhoff stress tensor, i.e., $\mathbf{P} = \mathbf{FS}$.

Given the force computation shown in Equation (D.55), a similar computation can be obtained using a reference configuration through Nanson's relation, i.e.,

$$\begin{aligned}
\mathbf{f}_j &= -\frac{1}{3}\boldsymbol{\sigma}(a_1\mathbf{n}_1 + a_2\mathbf{n}_2 + a_3\mathbf{n}_3) \\
\mathbf{f}_j &= -\frac{1}{3}J^{-1}\mathbf{F}\mathbf{S}\mathbf{F}^T(a_1\mathbf{n}_1 + a_2\mathbf{n}_2 + a_3\mathbf{n}_3) \\
\mathbf{f}_j &= -\frac{1}{3}J^{-1}\mathbf{F}\mathbf{S}\mathbf{F}^T(JA_1\mathbf{F}^{-T}\mathbf{N}_1 + JA_2\mathbf{F}^{-T}\mathbf{N}_2 + JA_3\mathbf{F}^{-T}\mathbf{N}_3) \\
\mathbf{f}_j &= -\frac{1}{3}\mathbf{F}\mathbf{S}(A_1\mathbf{N}_1 + A_2\mathbf{N}_2 + A_3\mathbf{N}_3) \\
\mathbf{f}_j &= -\frac{1}{3}\mathbf{P}(A_1\mathbf{N}_1 + A_2\mathbf{N}_2 + A_3\mathbf{N}_3),
\end{aligned} \tag{D.57}$$

which measures a force based on the initial configuration of the surface and the First Piola-Kirchhoff stress tensor.

The First Piola-Kirchhoff stress tensor can be obtained by taking the derivative of the strain energy density function W with respect to the deformation gradient tensor, i.e,

$$\mathbf{P}(\mathbf{F}) = \frac{\partial W(\mathbf{F})}{\partial \mathbf{F}}, \tag{D.58}$$

this makes it easier to define various constitutive models based on the underlying energy density function $W(\mathbf{F})$. For isotropic materials W can be expressed in terms of invariants of $\mathbf{F}$, i.e., its determinant or eigenvalues. Furthermore, since the First Piola-Kirchhoff measures forces based on a reference configuration, this model better fits cases in which larger deformations occur, see, e.g., [176] for more information.

Diagonalization using Singular Value Decomposition The stress inside a volume should be invariant to additional rotations of the volume. Given the definition of the Piola-Kirchhoff stress, see Equation (D.58), a diagonalization through the Singular Value Decomposition (SVD) can be obtained. Using a SVD, the deformation gradient tensor $\mathbf{F}$ can be decomposed as $\mathbf{F} = \mathbf{U}\hat{\mathbf{F}}\mathbf{V}^T$, with $\hat{\mathbf{F}}$ a diagonal matrix representing the pure deformations in each dimension. Matrices $\mathbf{U}$ and $\mathbf{V}$ are two orthogonal rotation matrices which describe the rotations of the deformation gradient tensor. Since the invariants of $\mathbf{F}$ are also invariant to rotations, $\hat{\mathbf{F}}$ has the same invariants of $\mathbf{F}$. Therefore, the following relation holds: $\mathbf{P}(\mathbf{F}) = \mathbf{U}\mathbf{P}(\hat{\mathbf{F}})\mathbf{V}^T$, which also shows that $\mathbf{P}$ is rotation variant, as mentioned in the previous paragraph. This allows one to define and use constitutional models for materials based on only the singular values of the deformation gradient tensor.

Force Jacobian There exist several methods for approximating the force Jacobian, see, e.g., [176]. Due to its purity we follow the procedure using the Singular Value Decomposition and its derivatives as presented in [161]. The Jacobian of the force on a single node of a single tetrahedron is given by

$$\frac{\partial \mathbf{f}_i}{\partial \mathbf{x}} = \sum_{j,k} \frac{\partial \mathbf{f}_i}{\partial F_{j,k}} \frac{\partial F_{j,k}}{\partial \mathbf{x}} = \sum_{j,k} \left(-\frac{\partial \mathbf{P}(\mathbf{F})}{\partial F_{j,k}} \mathbf{b}_i \right) \frac{\partial F_{j,k}}{\partial \mathbf{x}}, \tag{D.59}$$

with $\mathbf{b}_i = (A_1\mathbf{N}_1 + A_2\mathbf{N}_2 + A_3\mathbf{N}_3)/3$.

Given the Singular Value Decomposition (SVD) of the deformation gradient tensor $\mathbf{F} = \mathbf{U}\hat{\mathbf{F}}\mathbf{V}^T$, and $\mathbf{P}$ describing an isotropic material, $\mathbf{P}$ can be defined as $\mathbf{P}(\mathbf{F}) = \mathbf{U}\mathbf{P}(\hat{\mathbf{F}})\mathbf{V}^T$. The Jacobian of $\mathbf{P}(\mathbf{F})$ becomes:

$$\frac{\partial \mathbf{P}(\mathbf{F})}{\partial \mathbf{F}} = \frac{\partial \mathbf{U}}{\partial \mathbf{F}}\mathbf{P}(\hat{\mathbf{F}})\mathbf{V}^T + \mathbf{U}\frac{\partial \mathbf{P}(\hat{\mathbf{F}})}{\partial \hat{\mathbf{F}}}\frac{\partial \hat{\mathbf{F}}}{\partial \mathbf{F}}\mathbf{V}^T + \mathbf{U}\mathbf{P}(\hat{\mathbf{F}})\frac{\partial \mathbf{V}^T}{\partial \mathbf{F}}, \tag{D.60}$$

with $\partial \mathbf{P}(\hat{\mathbf{F}})/\partial \hat{\mathbf{F}}$ the Hessian of the strain energy density function $W(\hat{\mathbf{F}})$. Given the used constitutional model, its Hessian can be obtained analytically. The other terms needed to compute the Jacobian of the elasticity force requires the derivatives of the Singular Value Decomposition, see [141, 161]. Here we shall give a brief summary of its computation. In order to omit a tensor notation, the derivative of the SVD is computed per component of the deformation gradient tensor $\mathbf{F}$, i.e., for each $F_{j,k}$.

The derivative of the SVD of the deformation gradient tensor $\mathbf{F}$ with respect to itself can be obtained as follows. First the derivative is pre and post multiplied by $\mathbf{U}^T$ and $\mathbf{V}$ respectively.

$$\begin{aligned}
\frac{\partial \mathbf{F}}{\partial F_{j,k}} &= \frac{\partial \mathbf{U}}{\partial F_{j,k}}\hat{\mathbf{F}}\mathbf{V}^T + \mathbf{U}\frac{\partial \hat{\mathbf{F}}}{\partial F_{j,k}}\mathbf{V}^T + \mathbf{U}\hat{\mathbf{F}}\frac{\partial \mathbf{V}^T}{\partial F_{j,k}} \\
\mathbf{U}^T\left(\frac{\partial \mathbf{F}}{\partial F_{j,k}}\right)\mathbf{V} &= \mathbf{U}^T\left(\frac{\partial \mathbf{U}}{\partial F_{j,k}}\hat{\mathbf{F}}\mathbf{V}^T + \mathbf{U}\frac{\partial \hat{\mathbf{F}}}{\partial F_{j,k}}\mathbf{V}^T + \mathbf{U}\hat{\mathbf{F}}\frac{\partial \mathbf{V}^T}{\partial F_{j,k}}\right)\mathbf{V} \\
&= \underbrace{\left(\mathbf{U}^T\frac{\partial \mathbf{U}}{\partial F_{j,k}}\right)}_{\omega_U^{j,k}}\hat{\mathbf{F}} + \frac{\partial \hat{\mathbf{F}}}{\partial F_{j,k}} + \hat{\mathbf{F}}\underbrace{\left(\frac{\partial \mathbf{V}^T}{\partial F_{j,k}}\mathbf{V}\right)}_{\omega_{V^T}^{j,k}}.
\end{aligned} \tag{D.61}$$

This gives the matrices $\omega_U^{j,k}$ and $\omega_{V^T}^{j,k}$. Furthermore, it should be noted that $\frac{\partial F_{l,m}}{\partial F_{j,k}} = 0$ for all $(l,m) \neq (j,k)$ and 1 otherwise. Since $\mathbf{U}$ is orthogonal, it can be shown that matrices $\omega_U^{j,k}$ and $\omega_{V^T}^{j,k}$ are antisymmetric, i.e,

$$\frac{\partial \mathbf{U}^T\mathbf{U}}{\partial F_{j,k}} = \frac{\partial \mathbf{I}}{\partial F_{j,k}} \implies \frac{\partial \mathbf{U}^T}{\partial F_{j,k}}\mathbf{U} + \mathbf{U}^T\frac{\partial \mathbf{U}}{F_{j,k}} = \omega_{U^T}^{j,k} + \omega_U^{j,k} = \mathbf{0}. \tag{D.62}$$

The same holds for $\mathbf{V}^T$ and $\omega_{V^T}^{j,k}$. Due to this, and the fact that $\hat{\mathbf{F}}$ is a diagonal matrix, $\frac{\partial \hat{\mathbf{F}}}{\partial F_{j,k}}$ yields exactly the diagonal of the matrix on the left hand side of Equation (D.61). In order to compute the components (l,m) of both $\omega_U^{j,k}$ and $\omega_{V^T}^{j,k}$, the antisymmetric properties of these matrices are used. Hence, the following set of equations are solved for each component in the lower-triangular parts of $\omega_U^{j,k}$ and $\omega_{V^T}^{j,k}$:

$$\begin{aligned}
\left(\mathbf{U}^T\left(\frac{\partial \mathbf{F}}{\partial F_{j,k}}\right)\mathbf{V}\right)_{l,m} &= +\left(\omega_U^{j,k}\right)_{l,m}\left(\hat{\mathbf{F}}\right)_{m,m} + \left(\hat{\mathbf{F}}\right)_{l,l}\left(\omega_{V^T}^{j,k}\right)_{l,m} \\
\left(\mathbf{U}^T\left(\frac{\partial \mathbf{F}}{\partial F_{j,k}}\right)\mathbf{V}\right)_{k,l} &= -\left(\omega_U^{j,k}\right)_{l,m}\left(\hat{\mathbf{F}}\right)_{l,l} - \left(\hat{\mathbf{F}}\right)_{m,m}\left(\omega_{V^T}^{j,k}\right)_{l,m},
\end{aligned} \tag{D.63}$$

with $(.)_{l,m}$ component (l,m) of the involved matrix. Solving this linear system gives one component in $\omega_U^{j,k}$ and $\omega_{v^T}^{j,k}$. When the difference between the components in $\hat{\mathbf{F}}$ is small, the system in Equation (D.63) becomes ill-conditioned, which requires a Tikhonov regularization in or-

der to solve it properly. After solving 3 of these 2×2 systems, the derivatives of the rotation matrices with respect to $F_{j,k}$ are obtained using

$$\frac{\partial \mathbf{U}}{\partial F_{j,k}} = \mathbf{U}\omega_U^{j,k}$$
$$\frac{\partial \mathbf{V}^T}{\partial F_{j,k}} = \omega_{V^T}^{j,k}\mathbf{V}^T. \tag{D.64}$$

In order to obtain the Jacobian of the elastic force $\frac{\partial \mathbf{f}}{\partial \mathbf{u}} \in \mathbb{R}^{12\times12}$, the individual forces $\mathbf{f}_i \in \mathbb{R}^{1\times3}$ acting on each vertex i of the tetrahedron are decomposed. Next, the Jacobian is further decomposed into $\frac{\partial \mathbf{f}_i}{\partial x_l}$ for each component in $\mathbf{x} \in \mathbb{R}^{1\times12}$, i.e.,

$$\frac{\partial \mathbf{f}_i}{\partial x_l} = \sum_{j,k} \frac{\partial \mathbf{f}_i}{\partial F_{j,k}} \frac{\partial F_{j,k}}{\partial x_l} = \sum_{j,k} \left(-\frac{\partial \mathbf{P}(\mathbf{F})}{\partial F_{j,k}} \mathbf{b}_i \right) \frac{\partial F_{j,k}}{\partial x_l} \in \mathbb{R}^{3\times12}, \tag{D.65}$$

which computes the component of the symmetric and positive definite force Jacobian associated with the force at node i. The term $\frac{\partial \mathbf{P}(\mathbf{F})}{\partial F_{j,k}}$ is obtained as discussed previously. The remaining computation is $\frac{\partial F_{j,k}}{\partial x_l}$.

Deformation gradient tensor $\mathbf{F}$ is obtained from the deformed configuration $\mathbf{D}_s$ and undeformed configuration $\mathbf{D}_m$ of the vertices, i.e., $\mathbf{F} = \mathbf{D}_s\mathbf{D}_m^{-1}$. Therefore, the derivative can be transformed into

$$\frac{\partial \mathbf{F}}{\partial x_l} = \frac{\partial \mathbf{D}_s}{\partial x_l}\mathbf{D}_m^{-1} + \mathbf{D}_s\frac{\partial \mathbf{D}_m^{-1}}{\partial x_l} = \frac{\partial \mathbf{D}_s}{\partial x_l}\mathbf{D}_m^{-1}, \tag{D.66}$$

since $\mathbf{D}_m$ is constant, its derivative is zero. Furthermore, $\mathbf{D}_s$ contains the deformed configuration $\mathbf{x}$, i.e.,

$$\mathbf{D}_s = \begin{pmatrix} x_3 - x_0 & x_6 - x_0 & x_9 - x_0 \\ x_4 - x_1 & x_7 - x_1 & x_{10} - x_1 \\ x_5 - x_2 & x_8 - x_2 & x_{11} - x_2 \end{pmatrix}, \tag{D.67}$$

hence, $\frac{\partial \mathbf{D}_s}{\partial x_l}$ results in a matrix with only one ±1.

Singular Value Decomposition correction The Singular Value Decomposition of the deformation gradient tensor $\mathbf{F}$ is in general not unique. The SVD computes matrices $\mathbf{U}$ and $\mathbf{V}$ which can have a negative determinant, while the determinant of matrix $\hat{\mathbf{F}}$ is always positive. Since $\mathbf{U}$ and $\mathbf{V}$ are pure rotations, their determinants must be 1. As a result, the determinant of $\hat{\mathbf{F}}$ must correspond to the determinant of the deformation gradient tensor. In other words, if a particular element inverts, i.e., it undergoes a reflection, then this inversion will result in a negative determinant for both $\mathbf{F}$ and $\hat{\mathbf{F}}$. To achieve this, a commonly used 'convention' is used, presented in [95], which will be briefly discussed here.

When the determinant of the deformation gradient tensor is negative, the element is inverted. In this case the assumption is made that the smallest component of diagonal matrix $\hat{\mathbf{F}}$ had changed sign. By negating the smallest component in $\hat{\mathbf{F}}$, its determinant becomes negative and now matches the determinant of the deformation gradient tensor. Next, if the determinant of $\mathbf{V}$ is negative, its column associated with the smallest component in $\hat{\mathbf{F}}$ is negated. This makes its determinant positive. Once $\mathbf{V}$ and/or $\hat{\mathbf{F}}$ are corrected, $\mathbf{U}$ is corrected using

$$\mathbf{U} = \mathbf{F}\mathbf{V}\hat{\mathbf{F}}^{-1}. \tag{D.68}$$

Please note that when an element becomes degenerate, $\hat{\mathbf{F}}$ is not invertible. So extra care must be taken in order to compute $\mathbf{U}$.

Neo-Hookean Hyper elasticity Instead of using a linear model for elasticity, also non-linear models can be used. In general, linear models are more prone to collapses of the elements since the stiffness is fixed, so the force that is generated for a given compression might not be sufficient. Using non-linear materials, like a Neo-Hookean model for elasticity, in principle an infinite amount of stress can be measured due to the compression of an element. This prevents the material from collapsing under heavy load. The Neo-Hookean energy density function is given by,

$$\Psi(\hat{\mathbf{F}}) = \frac{\mu}{2}\left(\hat{F}_1^2 + \hat{F}_2^2 + \hat{F}_3^2 - 3\right) - \mu \ln J + \frac{\lambda}{2}(\ln J)^2, \tag{D.69}$$

with μ and λ the Lamé parameters for a particular material, and J the determinant of $\hat{\mathbf{F}}$, i.e., $J = \hat{F}_1\hat{F}_2\hat{F}_3$. The corresponding stress is given by its derivative with respect to $\hat{\mathbf{F}}$, i.e,

$$\mathbf{P}(\hat{\mathbf{F}}) = \frac{\partial \Psi(\hat{\mathbf{F}})}{\partial \hat{\mathbf{F}}} = \mu\hat{\mathbf{F}} - \frac{\mu}{\hat{\mathbf{F}}} + \frac{\lambda \ln J}{\hat{\mathbf{F}}}. \tag{D.70}$$

From this derivation it is easy to see that when J (which is the relative volume with respect to the initial configuration) becomes very small, the stress approaches infinity. However, when an element collapses, J can be negative, and thus Ψ and $\mathbf{P}$ are not defined. When an infinite small time-step is used, these problems will not arise, but for large time-steps it is possible that stress $\mathbf{P}$ linearized at the beginning of the time-step is not sufficient to prevent a collapse. To deal with these situations, $\hat{\mathbf{F}}$ is often clamped such that it is still possible to compute a proper stress and force. Another approach, presented in [168], is to linearly extrapolate the energy density function Ψ after a certain amount of deformation is reached. This model is used in Chapter 5 in which highly deformable objects under high load are simulated. In the following paragraph this model is briefly described and we show how it is used in the Singular Value Decomposition Gradients framework.

Energy density extrapolation In order to robustly simulate hyper-elastic materials that can undergo large deformations, one must handle degenerate cases properly. It is always possible that an element accidentally inverts. When it does, the computed forces and force derivatives for such an element should always work in the direction that eventually recovers the element. In [168] an extrapolation method is presented that extrapolates the energy density function into the inverted region of the volume. When a certain amount of compression is reached, i.e., $J < \epsilon$, the energy density function is linearly extrapolated, i.e.,

$$\Psi^{ext}(\hat{\mathbf{F}}) = \Psi(\hat{\mathbf{q}}) + h\Psi'(\hat{\mathbf{q}})\mathbf{u} + \frac{h^2}{2}\mathbf{u}^T\Psi''(\hat{\mathbf{q}})\mathbf{u}, \tag{D.71}$$

which is a second order Taylor expansion of Ψ around $\hat{\mathbf{q}}$, using first and second order directional derivatives in direction $\mathbf{u}$. To do so, first the direction $\mathbf{u}$ is determined. This is the vector from the rest configuration of the material $\mathbf{r} = (1,1,1)^T$ to the current deformation $\hat{\mathbf{F}}$. Next, the intersection $\hat{\mathbf{q}}$ at $\mathbf{u}$ with surface $J = \epsilon$ must be determined. For this a root-finding problem is solved for s, such that $q_1q_2q_3 = \epsilon$, with

$$\hat{\mathbf{q}} = \mathbf{r} + s(\hat{\mathbf{F}} - \mathbf{r}). \tag{D.72}$$

Given $\hat{\mathbf{q}}$, the location is known where Ψ is evaluated and extrapolated into direction $\mathbf{u}$. If $J > \epsilon$, then the Ψ is evaluated at $\hat{\mathbf{F}}$ and not extrapolated. Step size h is the distance between $\hat{\mathbf{q}}$ and $\hat{\mathbf{F}}$. Since the force is computed using the first order derivatives of Ψ and the force derivative using the second order derivatives of Ψ, also the first and second order derivatives of Ψ^{ext}

with respect to $\hat{\mathbf{F}}$ must be computed. Since, $\hat{\mathbf{q}}, \mathbf{u}, h$ and s all depend on $\hat{\mathbf{F}}$, also their first and second order derivatives with respect to $\hat{\mathbf{F}}$ are required. In order to find the derivatives of s, an implicit differentiation must be performed on $q_1 q_2 q_3 = \epsilon$. Furthermore, also the third and fourth order derivatives of Ψ are required. See the appendix of Stomakhin's PhD thesis [170] for the complete derivation of the first and second derivative of Ψ^{ext}.

In Chapter 5 the extrapolated energy density function, based on a Neo-Hookean energy density function, is used in combination with the Singular Value Decomposition Gradients method described in the previous paragraphs. Given the Jacobian and Hessian of Ψ^{ext}, they can be applied directly in Equation (D.65).

D.5 Conclusion

In this appendix we have described the methods used in Chapters 4 and 5 for simulating elastically deformable models using FEM. Linear elasticity in combination with corotational FEM [124] is commonly used in computer graphics for simulating deformable objects in which the deformations are relatively small. Since the element stiffness matrices are computed once, only the rotation of the elements needs to be computed at each simulation step. This makes a GPU implementation more straightforward as shown in Chapter 4. When the deformations are larger, corotational FEM is less accurate due to the absence of the derivatives of the element rotations. Furthermore, since the stiffness is linear, the response forces may be too small to prevent element inversions. When elements invert, they may not recover when the external load is removed and could introduce oscillations, degrading the realism of such simulations. Using Neo-Hookean elasticity models, the 'stiffness' of the material changes with to the amount of compression, and approaches infinity when the volume approaches zero. In discrete time-stepping methods, elements still can invert, which could result in undefined stresses and energies for Neo-Hookean models. To overcome this, the underlying energy density function can be extrapolated into the inverted region of the elements, as described in [168], resulting in consistent forces for inverted elements. Due to this, inverted elements will always recover when external loads are removed. This extrapolated energy density function is then used in the computation of the elastic forces. Using this in combination with the SVD and their derivatives [141, 161] of the deformation gradient tensor, results in an accurate and robust simulation of Neo-Hookean materials. By taking also a robust treatment of element inversion into account [95], such simulations will produce robust and smooth results when the objects undergo large deformations under a large external load as shown in Chapter 5.

Bibliography

[1] Alduán, I. and Otaduy, M. A. SPH Granular Flow with Friction and Cohesion. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Computer Animation* (2011), SCA '11, ACM, pp. 25–32.

[2] Allard, J., Faure, F., Courtecuisse, H., Falipou, F., Duriez, C., and Kry, P. G. Volume Contact Constraints at Arbitrary Resolution. *ACM Trans. Graph. 29*, 4 (2010), pp. 82:1–82:10.

[3] Amestoy, P. R., Davis, T. A., and Duff, I. S. Algorithm 837: AMD, an Approximate Minimum Degree Ordering Algorithm. *ACM Trans. Math. Softw. 30*, 3 (2004), pp. 381–388.

[4] An, S. S., Kim, T., and James, D. L. Optimizing Cubature for Efficient Integration of Subspace Deformations. *ACM Trans. Graph. 27*, 5 (2008), pp. 165:1–165:10.

[5] Anitescu, M. and Hart, G. D. A fixed-point iteration approach for multibody dynamics with contact and small friction. *Math. Program. 101*, 1 (2004), pp. 3–32.

[6] Aristidou, A., Lasenby, J., Chrysanthou, Y., and Shamir, A. Inverse kinematics techniques in computer graphics: A survey. *Comput. Graph. Forum 37*, 6 (2018), pp. 35–58.

[7] Baraff, D. Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies. In *Proc. of the 16th Annual Conf. on Computer Graphics and Interactive Techniques* (1989), SIGGRAPH '89, ACM, pp. 223–232.

[8] Baraff, D. Fast Contact Force Computation for Nonpenetrating Rigid Bodies. In *Proc. of the 21st Annual Conf. on Computer Graphics and Interactive Techniques* (1994), SIGGRAPH '94, ACM, pp. 23–34.

[9] Baraff, D. Linear-Time Dynamics using Lagrange Multipliers. In *Proc. of the 23rd Annual Conf. on Computer Graphics and Interactive Techniques* (1996), SIGGRAPH '96, ACM, pp. 137–146.

[10] Baraff, D. and Witkin, A. Dynamic Simulation of Non-penetrating Flexible Bodies. In *Proc. of the 19th Annual Conf. on Computer Graphics and Interactive Techniques* (1992), SIGGRAPH '92, ACM, pp. 303–308.

[11] Baraff, D. and Witkin, A. Large Steps in Cloth Simulation. In *Proc. of the 25th Annual Conf. on Computer Graphics and Interactive Techniques* (1998), SIGGRAPH '98, ACM, pp. 43–54.

[12] Barbič, J. Exact Corotational Linear FEM Stiffness Matrix. Tech. rep., University of Southern California, 2012.

[13] Barbič, J. and James, D. Time-critical distributed contact for 6-DoF haptic rendering of adaptively sampled reduced deformable models. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Computer Animation* (2007), SCA '07, Eurographics Association, pp. 171–180.

[14] Barbič, J. and James, D. L. Real-Time Subspace Integration for St. Venant-Kirchhoff Deformable Models. *ACM Trans. Graph. 24*, 3 (2005), pp. 982–990.

[15] Barbič, J. and Zhao, Y. Real-time Large-deformation Substructuring. *ACM Trans. Graph. 30*, 4 (2011), pp. 91:1–91:8.

[16] Bargteil, A. W., Goktekin, T. G., O'brien, J. F., and Strain, J. A. A semi-Lagrangian contouring method for fluid simulation. *ACM Trans. Graph. 25*, 1 (2006), pp. 19–38.

[17] Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J. M., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and van der Vorst, H. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, 2nd ed. SIAM, 1994.

[18] Bell, N. and Garland, M. Efficient Sparse Matrix-Vector Multiplication on CUDA. Tech. Rep. NVR-2008-004, NVidia, 2008.

[19] Bell, N. and Garland, M. *CUSP library*, 2011. Available at `http://code.google.com/p/cusp-library`.

[20] Bender, J., Erleben, K., and Trinkle, J. Interactive Simulation of Rigid Body Dynamics in Computer Graphics. *Comput. Graph. Forum 33*, 1 (2014), pp. 246–270.

[21] Bender, J., Müller, M., Otaduy, M. A., Teschner, M., and Macklin, M. A survey on position-based simulation methods in computer graphics. *Comput. Graph. Forum 33*, 6 (2014), pp. 228–251.

[22] van den Bergen, G. SOLID Software Library for Interference Detection. `http://solid.sourceforge.net`. September 2018.

[23] van den Bergen, G. *Collision Detection in Interactive 3D Computer Animation*. PhD thesis, Eindhoven University of Technology, 1999.

[24] van den Bergen, G. *Collision Detection in Interactive 3D Environments*. CRC Press, 2003.

[25] Bertails-Descoubes, F., Cadoux, F., Daviet, G., and Acary, V. A Nonsmooth Newton Solver for Capturing Exact Coulomb Friction in Fiber Assemblies. *ACM Trans. Graph. 30*, 1 (2011), pp. 6:1–6:14.

[26] Bolz, J., Farmer, I., Grinspun, E., and Schröder, P. Sparse Matrix Solvers on the GPU: Conjugate Gradients and Multigrid. *ACM Trans. Graph. 22*, 3 (2003), pp. 917–924.

[27] Boyd, S. and Vandenberghe, L. *Convex Optimization*. Cambridge University Press, 2004.

[28] Brandt, C., Eisemann, E., and Hildebrandt, K. Hyper-reduced projective dynamics. *ACM Trans. Graph. 37*, 4 (2018), pp. 80:1–80:13.

[29] Breen, D. E., House, D. H., and Wozny, M. J. Predicting the Drape of Woven Cloth Using Interacting Particles. In *Proc. of the 21st Annual Conf. on Computer Graphics and Interactive Techniques* (1994), SIGGRAPH '94, ACM, pp. 365–372.

[30] Bridson, R., Fedkiw, R., and Anderson, J. Robust Treatment of Collisions, Contact and Friction for Cloth Animation. *ACM Trans. Graph. 21*, 3 (2002), pp. 594–603.

[31] Brodtkorb, A. R., Dyken, C., Hagen, T. R., Hjelmervik, J. M., and Storaasli, O. O. State-of-the-art in heterogeneous computing. *Sci. Program. 18*, 1 (2010), pp. 1–33.

[32] Brodtkorb, A. R., Sætra, M. L., and Altinakar, M. Efficient shallow water simulations on GPUs: Implementation, visualization, verification, and validation. *Computers and Fluids 55* (2012), pp. 1–12.

[33] Brogliato, B. *Nonsmooth Mechanics*, 3rd ed. Springer, Cham, 2016.

[34] Buatois, L., Caumon, G., and Lévy, B. Concurrent Number Cruncher: An Efficient Sparse Linear Solver on the GPU. In *High Performance Computation Conference* (2007), HPCC'07, pp. 358–371.

[35] Burden, R. L. and Faires, J. D. *Numerical Analysis*, 8th ed. PWS Publishing Co., 2005.

[36] Casas, D. and Otaduy, M. A. Learning nonlinear soft-tissue dynamics for interactive avatars. *Proc. ACM Computer Graphics Interactive Techniques 1*, 1 (2018), pp. 10:1–10:15.

[37] Catto, E. Iterative Dynamics with Temporal Coherence. *Game Developer Conference* (2005).

[38] Cevahir, A., Nukada, A., and Matsuoka, S. Fast Conjugate Gradients with Multiple GPUs. In *Int. Conf. on Computational Science* (2009), ICCS, pp. 893–903.

[39] Cevahir, A., Nukada, A., and Matsuoka, S. High performance conjugate gradient solver on multi-GPU clusters using hypergraph partitioning. *Computer Science - Research and Development 25*, 1-2 (2010), pp. 83–91.

[40] Choi, J. W., Singh, A., and Vuduc, R. W. Model-driven autotuning of sparse matrix-vector multiply on GPUs. *SIGPLAN Not. 45*, 5 (2010), pp. 115–126.

[41] Cirio, G., Lopez-Moreno, J., Miraut, D., and Otaduy, M. A. Yarn-level Simulation of Woven Cloth. *ACM Trans. Graph. 33*, 6 (2014), pp. 207:1–207:11.

[42] Corrigan, A., Camelli, F. F., Löhner, R., and Wallin, J. Running unstructured grid-based CFD solvers on modern graphics hardware. *Int. J. Numer. Methods Fluids 66*, 2 (2011), pp. 221–229.

[43] Cottle, R., Pang, J.-S., and Stone, R. E. *The Linear Complementarity Problem.* Academic Press, 1992.

[44] Cottle, R. W. and Dantzig, G. B. Complementary pivot theory of mathematical programming. *Linear Algebra and its Applications 1*, 1 (1968), pp. 103–125.

[45] Coumans, E. Bullet Physics Library. `https://www.bulletphysics.org`. September 2018.

[46] Cuthill, E. and McKee, J. Reducing the bandwidth of sparse symmetric matrices. In *Proc. of the 24th nat. conf.* (1969), ACM '69, pp. 157–172.

[47] Daviet, G., Bertails-Descoubes, F., and Boissieux, L. A hybrid iterative solver for robustly capturing coulomb friction in hair dynamics. *ACM Trans. Graph. 30*, 6 (2011), pp. 139:1–139:12.

[48] Davis, T. A. and Hu, Y. The university of florida sparse matrix collection. *ACM Trans. Math. Softw. 38*, 1 (2011), pp. 1:1–1:25.

[49] Dick, C., Georgii, J., and Westermann, R. A Real-Time Multigrid Finite Hexahedra Method for Elasticity Simulation using CUDA. *Simulation Modelling Practice and Theory 19*, 2 (2011), pp. 801–816.

[50] Duriez, C., Dubois, F., Kheddar, A., and Andriot, C. Realistic Haptic Rendering of Interacting Deformable Objects in Virtual Environments. *IEEE Trans. Vis. Comput. Graph. 12*, 1 (2006), pp. 36–47.

[51] Dutre, P., Bala, K., Bekaert, P., and Shirley, P. *Advanced Global Illumination.* AK Peters Ltd, 2006.

[52] Eisenstat, S. C., Elman, H. C., and Schultz, M. H. Variational Iterative Methods for Nonsymmetric Systems of Linear Equations. *SIAM J. Numer. Anal. 20*, 2 (1983), pp. 345–357.

[53] van den Elzen, S., Holten, D., Blaas, J., and van Wijk, J. J. Reducing snapshots to points: A visual analytics approach to dynamic network exploration. *IEEE Trans. Vis. Comput. Graph. 22*, 1 (2016), pp. 1–10.

[54] Enright, D., Fedkiw, R., Ferziger, J., and Mitchell, I. A hybrid particle level set method for improved interface capturing. *J. Comput. Phys. 183*, 1 (2002), pp. 83–116.

[55] Enright, D., Marschner, S., and Fedkiw, R. Animation and Rendering of Complex Water Surfaces. *ACM Trans. Graph. 21*, 3 (2002), pp. 736–744.

[56] Enright, D., Nguyen, D., Gibou, F., and Fedkiw, R. Using the Particle Level Set Method and a Second Order Accurate Pressure Boundary Condition for Free Surface Flows. In *Proc. of the 4th ASME-JSME Joint Fluids Engineering Conf.* (2003), FEDSM2003, pp. 337–342.

[57] Epic Games. Unreal Engine. `https://www.unrealengine.com`. September 2018.

[58] Erleben, K. Velocity-Based Shock Propagation for Multibody Dynamics Animation. *ACM Trans. Graph. 26*, 2 (2007), pp. 12:1–12:20.

[59] Erleben, K. Numerical Methods for Linear Complementarity Problems in Physics-based Animation. In *ACM SIGGRAPH 2013 Courses* (2013), SIGGRAPH '13, ACM, pp. 8:1–8:42.

[60] Everts, M., Bekker, H., and Roerdink, J. Visualizing white matter structure of the brain using dijkstra's algorithm. In *Proc. of 6th Int. Symp. on Image and Signal Processing and Analysis* (2009), pp. 569–574.

[61] Fedkiw, R., Stam, J., and Jensen, H. W. Visual Simulation of Smoke. In *Proc. of the 28th Annual Conf. on Computer Graphics and Interactive Techniques* (2001), SIGGRAPH '01, ACM, pp. 15–22.

[62] Filippone, S., Cardellini, V., Barbieri, D., and Fanfarillo, A. Sparse Matrix-Vector Multiplication on GPGPUs. *ACM Trans. Math. Softw. 43*, 4 (2017), pp. 30:1–30:49.

[63] Fong, D. C. and Saunders, M. CG versus MINRES: An empirical comparison. Tech. rep., Stanford University, 2011.

[64] Foster, N. and Fedkiw, R. Practical Animation of Liquids. In *Proc. of the 28th Annual Conf. on Computer Graphics and Interactive Techniques* (2001), SIGGRAPH '01, ACM, pp. 23–30.

[65] Foster, N. and Metaxas, D. Realistic Animation of Liquids. *Graphical models and image processing: GMIP 58*, 5 (1996), pp. 471–483.

[66] Galoppo, N., Otaduy, M. A., Mecklenburg, P., Gross, M., and Lin, M. C. Fast Simulation of Deformable Models in Contact Using Dynamic Deformation Textures. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Computer Animation* (2006), SCA '06, Eurographics Association, pp. 73–82.

[67] Gao, J., Qi, P., and He, G. Efficient CSR-Based Sparse Matrix-Vector Multiplication on GPU. *Math. Probl. Eng.* (2016).

[68] Gao, J., Wang, Y., Wang, J., and Liang, R. Adaptive Optimization Modeling of Preconditioned Conjugate Gradient on Multi-GPUs. *ACM Trans. Parallel Comput. 3*, 3 (2016), pp. 16:1–16:33.

[69] Georgii, J. and Westermann, R. A multigrid framework for real-time simulation of deformable bodies. *Computers & Graphics 30*, 3 (2006), pp. 408–415.

[70] Gibson, S. and Mirtich, B. A Survey of Deformable Modeling in Computer Graphics. Tech. Rep. TR-97-19, MERL, Cambridge, MA, 1997.

[71] Gilbert, E. G., Johnson, D. W., and Keerthi, S. S. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal on Robotics and Automation 4*, 2 (1988), pp. 193–203.

[72] Göddeke, D., Strzodka, R., Mohd-Yusof, J., McCormick, P., Buijssen, S., Grajewski, M., and Turek, S. Exploring weak scalability for FEM calculations on a GPU-enhanced cluster. *Parallel Comput. 33*, 19-11 (2007), pp. 685–699.

[73] Göddeke, D., Strzodka, R., and Turek, S. Accelerating Double Precision FEM Simulations with GPUs. In *Proc. 18th Symp. on Simul. Technique* (2005), ASIM.

[74] Göddeke, D., Strzodka, R., and Turek, S. Performance and accuracy of hardware-oriented native-, emulated- and mixed-precision solvers in FEM simulations. *Int. Journal of Parallel, Emergent and Distributed Systems 22*, 4 (2007), pp. 221–256.

[75] Golub, G. H. and Van Loan, C. F. *Matrix Computations*, 4th ed. Johns Hopkins University Press, 1996.

[76] Goumas, G., Kourtis, K., Anastopoulos, N., Karakasis, V., and Koziris, N. Understanding the Performance of Sparse Matrix-Vector Multiplication. In *Proc. of the 16th Euromicro Conf. on Parallel, Distributed and Network-Based Processing* (2008), PDP '08, pp. 283–292.

[77] Guendelman, E., Bridson, R., and Fedkiw, R. Nonconvex Rigid Bodies with Stacking. *ACM Trans. Graph. 22*, 3 (2003), pp. 871–878.

[78] Guo, P. and Zhang, C. Performance Optimization for SpMV on Multi-GPU Systems Using Threads and Multiple Streams. In *Int. Symp. on Computer Architecture and High Performance Computing Workshops* (2016), SBAC-PADW, pp. 67–72.

[79] Harada, T., Koshizuka, S., and Kawaguchi, Y. Smoothed Particle Hydrodynamics on GPUs. *Proc. of Computer Graphics International* (2007), pp. 63–70.

[80] Harlow, F. H. and Welch, J. E. Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface. *Physics of Fluids 8* (1965), pp. 2182–2189.

[81] Harris, M., Sengupta, S., and Owens, J. D. Parallel Prefix Sum (Scan) with CUDA. In *GPU Gems 3*. Addison Wesley, 2007, pp. 851–876.

[82] Harris, M. J. *Real-time Cloud Simulation and Rendering*. PhD thesis, The University of North Carolina at Chapel Hill, 2003.

[83] Hassan, A. and Fluke, C. J. Scientific visualization in astronomy: Towards the petascale astronomy era. *Publications of the Astronomical Society of Australia 28*, 2 (2011), pp. 150–170.

[84] Hauth, M. and Straßer, W. Corotational Simulation of Deformable Solids. *Journal of WSCG 12*, 1-3 (2004), pp. 137–144.

[85] Hayami, K. On the Behaviour of the Conjugate Residual Method for Singular Systems. In *Proc. of the Fifth China-Japan Seminar on Num. Math.* (2002), Science Press, pp. 117–126.

[86] He, G. and Gao, J. A Novel CSR-Based Sparse Matrix-Vector Multiplication on GPUs. *Math. Probl. Eng.* (2016).

[87] Heidelberger, B., Teschner, M., Keiser, R., Müller, M., and Gross, M. Consistent penetration depth estimation for deformable collision response. In *Proc. of Vision, Modeling, Visualization* (2004), VMV'04, pp. 339–346.

[88] Hestenes, M. R. and Stiefel, E. Methods of Conjugate Gradients for Solving Linear Systems. *Journal of Research of the National Bureau of Standards 49* (1952), pp. 409–436.

[89] Heyn, T., Anitescu, M., Tasora, A., and Negrut, D. Using Krylov Subspace and Spectral Methods for Solving Complementarity Problems in Many-Body Contact Dynamics Simulation. *Int. J. Numer. Meth. Eng. 95*, 7 (2013), pp. 541–561.

[90] Higham, N. J. Computing the Polar Decomposition – with Applications. *SIAM J. Sci. Stat. Comput. 7* (1986), pp. 1160–1174.

[91] Holten, D. Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data. *IEEE Trans. Vis. Comput. Graph. 12*, 5 (2006), pp. 741–748.

[92] Holten, D. and van Wijk, J. J. Force-directed edge bundling for graph visualization. In *Proc. of the 11th Eurographics / IEEE - VGTC Conf. on Visualization* (2009), EuroVis'09, The Eurograpics Association and John Wiley & Sons Ltd., pp. 983–998.

[93] van der Hulst, J. M., Punzo, D., and Roerdink, J. B. T. M. 3-D interactive visualisation tools for Hi spectral line imaging. *International Astronomical Union Symp. 325* (2016), pp. 305–310.

[94] Irving, G., Schroeder, C., and Fedkiw, R. Volume Conserving Finite Element Simulations of Deformable Models. *ACM Trans. Graph. 26*, 3 (2007).

[95] Irving, G., Teran, J., and Fedkiw, R. Invertible finite elements for robust simulation of large deformation. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Computer Animation* (2004), SCA '04, Eurographics Association, pp. 131–140.

[96] Irving, G., Teran, J., and Fedkiw, R. Tetrahedral and Hexahedral Invertible Finite Elements. *Graph. Models 68*, 2 (2006), pp. 66–89.

[97] James, D. L. and Pai, D. K. Artdefo: Accurate real time deformable objects. In *Proc. of the 26th Annual Conf. on Computer Graphics and Interactive Techniques* (1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., pp. 65–72.

[98] Jiang, C., Schroeder, C., Selle, A., Teran, J., and Stomakhin, A. The Affine Particle-in-cell Method. *ACM Trans. Graph. 34*, 4 (2015), pp. 51:1–51:10.

[99] Judice, J. and Pires, F. A block principal pivoting algorithm for large-scale strictly monotone linear complementarity problems. *Computers and Operations Research 21*, 5 (1994), pp. 587–596.

[100] Kajiya, J. T. The Rendering Equation. *SIGGRAPH Comput. Graph. 20*, 4 (1986), pp. 143–150.

[101] Kaufman, D. M., Sueda, S., James, D. L., and Pai, D. K. Staggered Projections for Frictional Contact in Multibody Systems. *ACM Trans. Graph. 27*, 5 (2008), pp. 164:1–164:11.

[102] Kavan, L., Collins, S., Žára, J., and O'Sullivan, C. Geometric Skinning with Approximate Dual Quaternion Blending. *ACM Trans. Graph. 27*, 4 (2008), pp. 105:1–105:23.

[103] Kessenich, J., Sellers, G., and Shreiner, D. *OpenGL®Programming Guide: The Official Guide to Learning OpenGL®, Version 4.5 with SPIR-V*, 9th ed. Addison-Wesley Professional, 2016.

[104] King, I. P. An automatic reordering scheme for simultaneous equations derived from network problems. *Int. J. Numer. Meth. Eng. 2* (1970), pp. 523–533.

[105] Krüger, J. and Westermann, R. Linear Algebra Operators for GPU Implementation of Numerical Algorithms. *ACM Trans. Graph. 22*, 3 (2003), pp. 908–916.

[106] Kuhn, H. W. and Tucker, A. W. Nonlinear Programming. In *Proc. 2nd Berkeley Symp. on Math. Statist. and Prob.* (1950), Univ. of Calif. Press, pp. 481–492.

[107] Leine, R., Brogliato, B., and Nijmeijer, H. Periodic motion and bifurcations induced by the Painlevé paradox. *European Journal of Mechanics - A/Solids 21*, 5 (2002), pp. 869–896.

[108] Li, S., Pan, Z., Huang, J., Bao, H., and Jin, X. Deformable Objects Collision Handling with Fast Convergence. *Comput. Graph. Forum 34*, 7 (2015), pp. 269–278.

[109] Lindholm, E., Nickolls, J., Oberman, S., and Montrym, J. NVidia Tesla: A Unified Graphics and Computing Architecture. *IEEE Micro 28*, 2 (2008), pp. 39–55.

[110] Liu, G. R. and Liu, M. B. *Smoothed Particle Hydrodynamics: a Meshfree Particle Method.* World Scientific Publishing, 2003.

[111] Liu, Y., Jiao, S., Wu, W., and De, S. GPU accelerated fast FEM deformation simulation. In *IEEE Asia Pacific Conference on Circuits and Systems* (2008), APCCAS 2008, pp. 606–609.

[112] Lobo, D., Saraç, M., Verschoor, M., Solazzi, M., Frisoli, A., and Otaduy, M. A. Proxy-based haptic rendering for underactuated haptic devices. In *IEEE World Haptics Conference* (2017), WHC '17, pp. 48–53.

[113] Luenberger, D. G. The Conjugate Residual Method for Constrained Minimization Problems. *SIAM J. Numer. Anal. 7*, 3 (1970), pp. 390–398.

[114] Luenberger, D. G. An Approach to Nonlinear Programming. *J. of Optimiz. Theory App. 11*, 3 (1973), pp. 219–227.

[115] Macklin, M. and Müller, M. Position Based Fluids. *ACM Trans. Graph. 32*, 4 (2013), pp. 104:1–104:12.

[116] Magnenat-Thalmann, N., Laperrière, R., and Thalmann, D. Joint-dependent Local Deformations for Hand Animation and Object Grasping. In *Proc. on Graphics Interface* (1988), Canadian Information Processing Society, pp. 26–33.

[117] Manavski, S. A. Cuda compatible gpu as an efficient hardware accelerator for aes cryptography. In *IEEE Int. Conf. on Signal Processing and Communications* (2007), ICSPC '07, pp. 65–68.

[118] Mazhar, H., Heyn, T., Negrut, D., and Tasora, A. Using Nesterov's Method to Accelerate Multibody Dynamics with Friction and Contact. *ACM Trans. Graph. 34*, 3 (2015), pp. 32:1–32:14.

[119] Miguel, E. and Otaduy, M. A. Efficient Simulation of Contact between Rigid and Deformable Objects. In *ECCOMAS - Multibody Dynamics* (2011).

[120] Monakov, A. and Avetisyan, A. Implementing Blocked Sparse Matrix-Vector Multiplication on NVIDIA GPUs. In *Proc. of the 9th Int. Workshop on Embedded Computer Systems: Architectures, Modeling, and Simulation* (2009), SAMOS '09, pp. 289–297.

[121] Moore, M. and Wilhelms, J. Collision Detection and Response for Computer Animation. In *Proc. of the 15th Annual Conf. on Computer Graphics and Interactive Techniques* (1988), SIGGRAPH '88, ACM, pp. 289–298.

[122] Moreau, J. J. Unilateral Contact and Dry Friction in Finite Freedom Dynamics. In *Nonsmooth Mechanics and Applications. International Centre for Mechanical Sciences (Courses and Lectures)*, J. J. Moreau and P. D. Panagiotopoulos, Eds., vol. 302. Springer, Vienna, 1988.

[123] Müller, M., Charypar, D., and Gross, M. Particle-based Fluid Simulation for Interactive Applications. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Computer Animation* (2003), SCA '03, Eurographics Association, pp. 154–159.

[124] Müller, M. and Gross, M. Interactive virtual materials. In *Proc. of Graphics Interface 2004* (2004), GI '04, Canadian Human-Computer Communications Society, pp. 239–246.

[125] Müller, M., Heidelberger, B., Hennix, M., and Ratcliff, J. Position Based Dynamics. *J. Vis. Comun. Image Represent. 18*, 2 (2007), pp. 109–118.

[126] Nealen, A., Müller, M., Keiser, R., Boxerman, E., and Carlson, M. Physically Based Deformable Models in Computer Graphics. *Comput. Graph. Forum 25*, 4 (2006), pp. 809–836.

[127] Nocedal, J. and Wright, S. *Numerical Optimization*, 2nd ed. Springer series in operations research and financial engineering. Springer, 2006.

[128] NVidia Corporation. NVidia Physx. `https://www.nvidia.com/object/physx_faq.html`. September 2018.

[129] NVidia Corporation. *NVidia GeForce 8800 GPU Architecture Overview*, 2006. Available at `http://www.nvidia.com/page/8800_tech_briefs.html`.

[130] NVidia Corporation. *NVidia nForce 790i SLI Chipsets, Reducing Latencies and Bandwidth Utilizations*, 2008.

[131] NVidia Corporation. *NVidia's Next Generation CUDA Compute Architecture Fermi*, 2009. Available at `http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf`.

[132] NVidia Corporation. *Compute Unified Device Architecture programming guide*, 2010. Available at `http://developer.nvidia.com/cuda`.

[133] NVidia Corporation. *CUBLAS library*, 2010. Available at `http://developer.download.nvidia.com/compute/cuda/2_0`.

[134] NVidia Corporation. *CUDA Occupancy Calculator*, 2010. Available at `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[135] NVidia Corporation. *cuSPARSE library*, 2010. Available at `http://developer.download.nvidia.com/compute/cuda/2_0`.

[136] NVidia Corporation. *NVidia GPUDirect*, 2010. Available at `http://developer.nvidia.com/gpudirect`.

[137] Osher, S. and Sethian, J. A. Fronts Propagating with Curvature-Dependent Speed: Algorithms Based on Hamilton-Jacobi Formulations. *J. Comput. Phys. 79* (1988), pp. 12–49.

[138] Otaduy, M. A., Tamstorf, R., Steinemann, D., and Gross, M. Implicit Contact Handling for Deformable Objects. *Comput. Graph. Forum 28*, 2 (2009), pp. 559–568.

[139] Paige, C. C. and Saunders, M. A. Solution of Sparse Indefinite Systems of Linear Equations. *SIAM J. Numer. Anal. 12*, 4 (1975), pp. 617–629.

[140] Pan, Z., Bao, H., and Huang, J. Subspace Dynamic Simulation Using Rotation-strain Coordinates. *ACM Trans. Graph. 34*, 6 (2015), pp. 242:1–242:12.

[141] Papadopoulo, T. and Lourakis, M. I. A. Estimating the Jacobian of the Singular Value Decomposition: Theory and Applications. In *Proc. of the 6th European Conf. on Computer Vision-Part I* (2000), ECCV '00, Springer-Verlag, pp. 554–570.

[142] Pauly, M., Pai, D. K., and Guibas, L. J. Quasi-Rigid Objects in Contact. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Computer Animation* (2004), SCA '04, Eurographics Association, pp. 109–119.

[143] Pepper, D. W. and Heinrich, J. C. *The Finite Element Method: Basic Concepts and Applications.* Taylor and Francis, 2005.

[144] Pharr, M., Jakob, W., and Humphreys, G. *Physically Based Rendering: From Theory to Implementation*, 3rd ed. Morgan Kaufmann Publishers Inc., 2016.

[145] Press, W. H., Flannery, B. P., Teukolsky, S. A., and Vetterling, W. T. *Numerical Recipes: The Art of Scientific Computing*, 2nd ed. Cambridge University Press, 1992.

[146] Raghupathi, L. and Faure, F. QP-Collide: A New Approach to Collision Treatment. In *Journées du groupe de travail Animation et Simulation* (2006), GTAS 06.

[147] Ramage, A. and Wathen, A. J. Iterative Solution Techniques for the Stokes and Navier-Stokes Equations. *Int. J. Numer. Methods Fluids 19* (1994), pp. 67–83.

[148] Redon, S., Kheddar, A., and Coquillart, S. Gauss' Least Constraints Principle and Rigid Body Simulations. In *Proc. of IEEE Int. Conf. on Robotics and Automation* (2002), pp. 11–15.

[149] Renouf, M. and Alart, P. Conjugate gradient type algorithms for frictional multi-contact problems: applications to granular materials. *Comput. Methods Appl. Mech. Eng. 194*, 18–20 (2005), pp. 2019–2041.

[150] Reynolds, C. W. Flocks, Herds and Schools: A Distributed Behavioral Model. *SIGGRAPH Comput. Graph. 21*, 4 (1987), pp. 25–34.

[151] Rost, R. J., Licea-Kane, B., Ginsburg, D., Kessenich, J. M., Lichtenbelt, B., Malan, H., and Weiblen, M. *OpenGL Shading Language*, 3rd ed. Addison-Wesley Professional, 2009.

[152] Rumpf, M. and Strzodka, R. Using Graphics Cards for Quantized FEM Computations. In *Proc. IASTED Vis., Imaging and Image Proc.* (2001), pp. 193–202.

[153] Saad, Y. *Iterative Methods for Sparse Linear Systems*, 2nd ed. Society for Industrial and Applied Mathematics, 2003.

[154] Saad, Y. and Schultz, M. H. GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM J. Sci. Stat. Comput. 7*, 3 (1986), pp. 856–869.

[155] Scheepens, R., Hurter, C., van de Wetering, H., and van Wijk, J. J. Visualization, selection, and analysis of traffic flows. *IEEE Trans. Vis. Comput. Graph. 22*, 1 (2016), pp. 379–388.

[156] Schvartzman, S. C., Perez, A. G., and Otaduy, M. A. Star-Contours for Efficient Hierarchical Self-Collision Detection. *ACM Trans. Graph. 29*, 3 (2010), pp. 80:1–80:8.

[157] Shinar, T., Schroeder, C., and Fedkiw, R. Two-way Coupling of Rigid and Deformable Bodies. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Computer Animation* (2008), SCA '08, Eurographics Association, pp. 95–103.

[158] Signorini, A. Questioni di elasticità non linearizzata e semilinearizzata. *Rend. Mat. Appl., V. Ser. 18* (1959), pp. 95–139.

[159] Silcowitz-Hansen, M., Abel, S. M. N., and Erleben, K. Projected gauss-seidel subspace minimization method for interactive rigid body dynamics: improving animation quality using a projected gauss–seidel subspace minimization method. In *Proc. of the Int. Conf. on Computer Graphics Theory and Applications* (2010), GRAPP 2010, SCITEPRESS Digital Library, pp. 38–45.

[160] Silcowitz-Hansen, M., Niebe, S., and Erleben, K. A nonsmooth nonlinear conjugate gradient method for interactive contact force problems. *Vis. Comput. 26*, 6-8 (2010), pp. 893–901.

[161] Sin, F., Zhu, Y., Li, Y., and Schroeder, D. Invertible isotropic hyperelasticity using SVD gradients. In *Proc. of the ACM SIGGRAPH / Eurographics Symp. on Computer Animation (Posters)* (2011), SCA '11, Eurographics Association.

[162] Singh, N. and Singh, S. Virtual reality: A brief survey. In *2017 International Conference on Information Communication and Embedded Systems* (2017), ICICES, pp. 1–6.

[163] van der Sluis, A. and Vorst van der, H. A. The rate of convergence of Conjugate Gradients. *Numerische Mathematik 48*, 5 (1986), pp. 543–560.

[164] Spillmann, J., Becker, M., and Teschner, M. Non-iterative Computation of Contact Forces for Deformable Objects. *Journal of WSCG 15*, 1-3 (2007), pp. 33–40.

[165] Springel, V. Smoothed Particle Hydrodynamics in Astrophysics. *Annual Review of Astronomy and Astrophysics 48*, 1 (2010), pp. 391–430.

[166] Stam, J. Stable fluids. In *Proc. of the 26th Annual Conf. on Computer Graphics and Interactive Techniques* (1999), SIGGRAPH '99, ACM Press/Addison-Wesley Publishing Co., pp. 121–128.

[167] Stewart, D. and Trinkle, J. C. An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Coulomb Friction. *Int. J. Numer. Meth. Eng. 39* (1996), pp. 2673–2691.

[168] Stomakhin, A., Howes, R., Schroeder, C., and Teran, J. M. Energetically Consistent Invertible Elasticity. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Computer Animation* (2012), SCA '12, Eurographics Association, pp. 25–32.

[169] Stomakhin, A., Schroeder, C., Chai, L., Teran, J., and Selle, A. A Material Point Method for Snow Simulation. *ACM Trans. Graph. 32*, 4 (2013), pp. 102:1–102:10.

[170] Stomakhin, A. D. *Part I: Reconstruction of Missing Data in Social Networks Based on Temporal Patterns of Interactions Part II: Constitutive Modeling in Solid Mechanics for Graphics Applications.* PhD thesis, University of California, 2013.

[171] Sulsky, D., Zhou, S.-J., and Schreyer, H. L. Application of a particle-in-cell method to solid mechanics. *Computer Physics Communications 87*, 1 (1995), pp. 236–252.

[172] Sutherland, I. E. Sketchpad: A man-machine graphical communication system. In *Proc. of the May 21-23, 1963, Spring Joint Computer Conference* (1963), AFIPS '63 (Spring), ACM, pp. 329–346.

[173] Tang, M., Kim, Y. J., and Manocha, D. C2A: Controlled conservative advancement for continuous collision detection of polygonal models. In *IEEE International Conference on Robotics and Automation* (2009), pp. 849–854.

[174] Tang, M., Manocha, D., Otaduy, M. A., and Tong, R. Continuous Penalty Forces. *ACM Trans. Graph. 31*, 4 (2012), pp. 107:1–107:9.

[175] Teng, Y., Otaduy, M. A., and Kim, T. Simulating Articulated Subspace Self-contact. *ACM Trans. Graph. 33*, 4 (2014), pp. 106:1–106:9.

[176] Teran, J., Blemker, S., Hing, V. N. T., and Fedkiw, R. Finite volume methods for the simulation of skeletal muscle. In *Proc. of the ACM SIGGRAPH/Eurographics Symp. on Computer Animation* (2003), SCA '03, Eurographics Association, pp. 68–74.

[177] Terzopoulos, D., Platt, J., Barr, A., and Fleischer, K. Elastically deformable models. *SIGGRAPH Comput. Graph. 21*, 4 (1987), pp. 205–214.

[178] Teschner, M., Kimmerle, S., Heidelberger, B., Zachmann, G., Raghupathi, L., Fuhrmann, A., p. Cani, M., Faure, F., Magnenat-thalmann, N., Strasser, W., and Volino, P. Collision Detection for Deformable Objects. In *Comput. Graph. Forum* (2004), pp. 61–81.

[179] Tompson, J., Schlachter, K., Sprechmann, P., and Perlin, K. Accelerating eulerian fluid simulation with convolutional networks. In *Proc. of the 34th Int. Conf. on Machine Learning* (2017), ICML, JMLR.org, pp. 3424–3433.

[180] Tonge, R., Benevolenski, F., and Voroshilov, A. Mass Splitting for Jitter-Free Parallel Rigid Body Simulation. *ACM Trans. Graph. 31*, 4 (2012), pp. 105:1–105:8.

[181] von Tycowicz, C., Schulz, C., Seidel, H.-P., and Hildebrandt, K. An Efficient Construction of Reduced Deformable Objects. *ACM Trans. Graph. 32*, 6 (2013), pp. 213:1–213:10.

[182] Unity Technologies. Unity Game Engine. `https://unity3d.com`. September 2018.

[183] Verschoor, M. and Jalba, A. C. Analysis and Performance Estimation of the Conjugate Gradient Method on Multiple GPUs. *Parallel Comput. 38*, 10-11 (2012), pp. 552–575.

[184] Verschoor, M. and Jalba, A. C. Elastically Deformable Models based on the Finite Element Method Accelerated on Graphics Hardware using CUDA. *Journal of WSCG 20*, 3 (2012), pp. 179–188.

[185] Verschoor, M. and Jalba, A. C. Efficient and Accurate Collision Response for Elastically Deformable Models. *ACM Trans. Graph. 38*, 2 (2019), pp. 17:1–17:20.

[186] Verschoor, M., Lobo, D., and Otaduy, M. A. Soft-Hand Simulation for Smooth and Robust Natural Interaction. In *Proc. of the IEEE Virtual Reality Conference* (2018), VR, pp. 183–190.

[187] Volino, P. and Magnenat-Thalmann, N. Collision and Self-Collision Detection: Robust and Efficient Techniques for Highly Deformable Surfaces. In *Proc. Eurographics Workshop on Animation and Simulation* (1995), Springer, pp. 105–108.

[188] Volino, P. and Thalmann, N. M. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Comput. Graph. Forum 13*, 3 (1994), pp. 155–166.

[189] van der Vorst, H. A. BI-CGSTAB: a fast and smoothly converging variant of BI-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput. 13*, 2 (1992), pp. 631–644.

[190] van Wijk, J. J. Image based flow visualization. *ACM Trans. Graph. 21*, 3 (2002), pp. 745–754.

[191] van Wijk, J. J. The value of visualization. In *Proc. of the 16th IEEE Visualization Conf.* (2005), VIS, pp. 79–86.

[192] Williams, S., Oliker, L., Vuduc, R., Shalf, J., Yelick, K., and Demmel, J. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *Proc. of the ACM/IEEE Conf. on Supercomputing* (2007), pp. 38:1–38:12.

[193] Wriggers, P. *Computational Contact Mechanics.* Wiley, 2002.

[194] Wriggers, W. A., Bakker, V., Kokkeler, A. B. J., and Smit, G. J. M. Implementing the conjugate gradient algorithm on multi-core systems. In *Proc. of the Int. Symp. on System-on-Chip* (2007), SoC, pp. 11–14.

[195] Xia, S., Gao, L., Lai, Y.-K., Yuan, M.-Z., and Chai, J. A survey on human performance capture and animation. *Journal of Computer Science and Technology 32*, 3 (2017), pp. 536–554.

[196] Xu, H., Zhao, Y., and Barbič, J. Implicit Multibody Penalty-Based Distributed Contact. *IEEE Trans. Vis. Comput. Graph. 20*, 9 (2014), pp. 1266–1279.

[197] Yang, S., Liang, J., and Lin, M. C. Learning-based cloth material recovery from video. In *IEEE International Conference on Computer Vision* (2017), ICCV, pp. 4393–4403.

[198] Zhu, Y. and Bridson, R. Animating Sand As a Fluid. *ACM Trans. Graph. 24*, 3 (2005), pp. 965–972.

Summary

Simulation and Animation of Deformable Solids

An animation is an illusion of motion that is created by showing a sequence of still images. This illusion is obtained when consecutive images have a minimal difference and are displayed at a sufficient high rate. Using this principle, an animator is able to bring the objects and characters in these images to life. Formerly, these images were mainly drawn by hand, nowadays computers are important tools for creating such animations. Thanks to animation software, animators are able to animate complex objects and scenes. However, when it comes to realistic animations of complex objects, or complex scenes containing many objects, some physical principles should be obeyed. These are for example gravity, deformation and contact between the surfaces of the involved objects. Each can depend on the shape, material and other properties of the involved objects. Creating realistic animations of such systems manually can be very challenging and time consuming. Therefore, one approach to improve the animation process for this kind of animations is by using computer simulations.

Using computer simulations, the motion and deformation of an object can be computed given the external and internal forces working on the body. Additionally, a simulation can compute this for many objects while computing the contact and friction forces between the objects. However, this kind of simulations often require a large amount of computational resources. The direct and indirect influence of each object onto their motions makes this a difficult problem to solve, especially when objects can deform. Computer simulations can also be applied in interactive animation applications, like VR and gaming. For those applications it is important that the simulation runs fast and stable.

In this dissertation the following research question is addressed:

"How can we accurately and efficiently simulate rigid and deformable solids that can collide, in a fast and stable way for computer animation applications?"

In particular, this dissertation presents methods that can improve the simulation process of elastically deformable objects, applied to (interactive) computer animations, while taking the accuracy into account. We address how to efficiently use graphics processors (GPUs) for assembling and solving the numerical problems related to such simulations. These numerical problems can be in general described by linear systems having large sparse matrices. These matrices have to be processed efficiently by a GPU, while solving the linear system. Apart from simulating the internal physics of deformable models, this dissertation presents methods for solving contact and friction between the surfaces of the simulated objects. By solving the coupled problem containing contact, friction, motion and deformation, one can obtain accurate results faster. Here the accurate detection of collisions, while taking possible deformations into account, is an important aspect.

In Chapter 3 we address how to store large sparse matrices in the memory of graphics processors. These matrices need to be stored in memory such that each individual processor can perform optimal during a matrix-vector multiplication. Because this is a key operation in many iterative methods for solving linear systems, we demonstrate the use of this operation for the conjugate gradient method, executed on one or multiple GPUs. To measure the efficiency of these methods, we apply an extensive analysis of the method and used hardware.

Chapter 4 describes methods for simulating elastically deformable models, executed on a GPU. The linear system obtained from the finite element method (FEM) and time-integration method, is solved using the conjugate gradient method, executed on a single GPU, as described in Chapter 3. The computations regarding individual elements are distributed among all available processors of the GPU.

In Chapter 5 a method is presented for solving coupled problems containing contact, friction, dynamics and deformation, instead of solving several sub-problems. The problem of such a strategy applied to complex deformable models is that setting up the individual sub-problems usually require a significant amount of memory and/or computation time. In this chapter we show that we can avoid this by solving the coupled problem as one. Friction and contact is modeled using Lagrange multipliers that directly constrain the degrees of freedom of the simulated objects. This allows us to accurately compute normal and friction forces, which is crucial regarding realism.

Chapter 6 focuses om the detection of collisions between deformable objects. Given a few primitives, we can detect intersections between edges, faces and vertices. Using these intersections, eventually collisions between objects (or between parts of the same object) are correctly detected. For deformable objects it is necessary that one can distinguish between normal and internal collisions and that they are processed correctly. Because the response of a collision affects the shape of an object, which in turn affects the normal and friction forces, the non-linear problem needs to be solved instead. If this non-linear problem is not accurately solved, the result may contain oscillations that negatively affects the realism.

Chapter 7 concludes this dissertation with the main conclusions and a reflection. Finally, other applications for the described methods are presented and possible future lines of research are explored.

Samenvatting

Simulatie en Animatie van Vervormbare Vaste Stoffen

Een animatie is een illusie van beweging welke wordt gecreëerd door het weergeven van een reeks stilstaande beelden. Deze illusie wordt bereikt wanneer de opeenvolgende beelden minimaal van elkaar verschillen en deze snel genoeg worden weergegeven. Met behulp van dit principe is een animator in staat de objecten en karakters in deze beelden tot leven te wekken. Waar deze beelden vroeger veelal met de hand werden getekend, is de computer vandaag de dag een onmisbaar hulpmiddel voor het creëren van animaties. Dankzij animatiesoftware is een animator in staat om complexe objecten en scenes te animeren. Echter, als het gaat om het realistisch animeren van complexe objecten of complexe scenes bestaande uit vele objecten, dan zal de animator rekening moeten houden met een aantal natuurkundige principes om de animatie er realistisch uit te laten zien. Dit zijn onder andere zwaartekracht, vervorming en contact tussen objecten. Deze kunnen afhankelijk zijn van de vorm, het materiaal en andere eigenschappen van de objecten. Het realistisch animeren van dergelijke systemen kan lastig zijn en veel tijd kosten. Eén manier om het animatieproces voor dergelijke animaties te verbeteren is met behulp van computersimulaties.

Met behulp van een computersimulatie kan de vervorming en de beweging van een object berekend worden aan de hand van de externe en interne krachten. Daarnaast kan een simulatie dit voor meerdere objecten tegelijk berekenen en rekening houden met eventuele normaal- en schuifkrachten tussen de objecten. Echter, dergelijke simulaties vereisen de nodige rekentijd. De directe en indirecte invloed van de objecten op elkaars beweging maakt het probleem lastig om op te lossen, met name als de objecten ook kunnen vervormen door deze externe invloeden. Computersimulaties kunnen ook worden gebruikt in interactieve animatietoepassingen, zoals in VR en games. Voor dergelijke applicaties is het van belang dat het simulatieproces snel en stabiel verloopt.

In dit proefschrift staat de volgende onderzoeksvraag centraal:

"Hoe kunnen we, zo nauwkeurig en zo efficiënt mogelijk, rigide en vervormbare objecten, welke kunnen botsen, zo snel mogelijk op een stabiele manier simuleren, voor animatie toepassingen?"

Dit proefschrift presenteert methodes die het gehele simulatieproces van elastisch vervormbare objecten, toegepast in (interactieve) computeranimaties, kunnen verbeteren en versnellen, terwijl er ook rekening wordt gehouden met de nauwkeurigheid. We gaan in op het zo efficiënt mogelijk gebruiken van grafische processoren (GPUs) voor het opstellen en oplossen van de numerieke problemen behorende bij dergelijke simulaties. Deze numerieke problemen kunnen in het algemeen worden beschreven door lineaire systemen die uit grote ijle matrices bestaan. Deze matrices dienen zo efficiënt en zo snel mogelijk door een GPU te worden verwerkt wanneer een oplossing voor het lineaire systeem wordt berekend. Naast het simuleren van de interne fysica, gaat dit proefschrift in op methoden voor het oplossen van contact en wrijving tussen de oppervlakten van de gesimuleerde objecten. Door contact, wrijving,

beweging en vervorming als één probleem te beschouwen, kan men sneller tot nauwkeurige resultaten komen. Een belangrijk aspect hierin is het correct detecteren van botsingen en tegelijkertijd rekening houden met de mogelijke vervorming van de objecten.

In Hoofdstuk 3 gaan we specifiek in op de opslag van grote ijle matrices in het geheugen van grafische processoren. Deze matrices worden zodanig opgeslagen dat de individuele processoren optimaal benut kunnen worden tijdens een matrix-vector vermenigvuldiging. Omdat dit een belangrijke operatie is in vele iteratieve methodes voor het oplossen van lineaire systemen, demonstreren we het gebruik van deze operatie in de geconjugeerde-gradiënt-methode, uitgevoerd op één en meerdere GPUs. Om de effectiviteit van deze methodes te meten, wordt er een uitgebreide analyse van de methode en gebruikte hardware uitgevoerd.

Hoofdstuk 4 gaat dieper in op het simuleren van elastische materialen, uitgevoerd op één GPU. Het lineaire systeem dat wordt verkregen door de eindige-elementenmethode (FEM) en tijd-integratiemethode, wordt opgelost met de geconjugeerde-gradiënt-methode, uitgevoerd op één GPU, zoals beschreven in Hoofdstuk 3. De berekeningen van de individuele elementen worden over alle beschikbare processoren van de GPU verdeeld.

In Hoofdstuk 5 wordt een methode gepresenteerd om contact, wrijving, beweging en vervorming als één probleem op te lossen, in plaats van het probleem in kleinere deelproblemen op te delen. Het probleem van een dergelijke strategie voor complexe vervormbare objecten is dat het opstellen van deze deelproblemen meestal veel geheugen en/of rekentijd vereisen. In dit hoofdstuk tonen we aan dat dit voorkomen kan worden door het probleem als één geheel op te lossen. Wrijving en contact worden gemodelleerd door middel van Lagrange-multiplicatoren, die direct de vrijheidsgraden van de gesimuleerde objecten begrenzen. Dit stelt ons in staat de normaal- en schuifkrachten nauwkeurig te berekenen, wat cruciaal is als het om realisme gaat.

Hoofdstuk 6 gaat dieper in op het detecteren van botsingen tussen vervormbare objecten. Door middel van een aantal primitieven kunnen intersecties van randen, driehoeken en punten worden gevonden. Met behulp van deze intersecties kunnen uiteindelijk botsingen tussen objecten (en tussen delen van hetzelfde object) correct worden gedetecteerd. Voor vervormbare objecten is het noodzakelijk dat er onderscheid gemaakt kan worden tussen normale en interne botsingen en dat deze correct worden verwerkt. Omdat ook de reactie van een botsing de vorm van het object verandert, welke invloed heeft op de normaal- en schuifkrachten, moet een niet-lineair probleem worden opgelost. Wordt dit niet of niet nauwkeurig gedaan, dan zal het resultaat ongewenste oscillaties kunnen bevatten, wat het realisme negatief beïnvloedt.

Hoofdstuk 7 sluit het proefschrift af met de belangrijkste bevindingen en een reflectie. Tot slot worden andere toepassingen voor de beschreven methodes gepresenteerd en wordt mogelijk toekomstig onderzoek verkend.

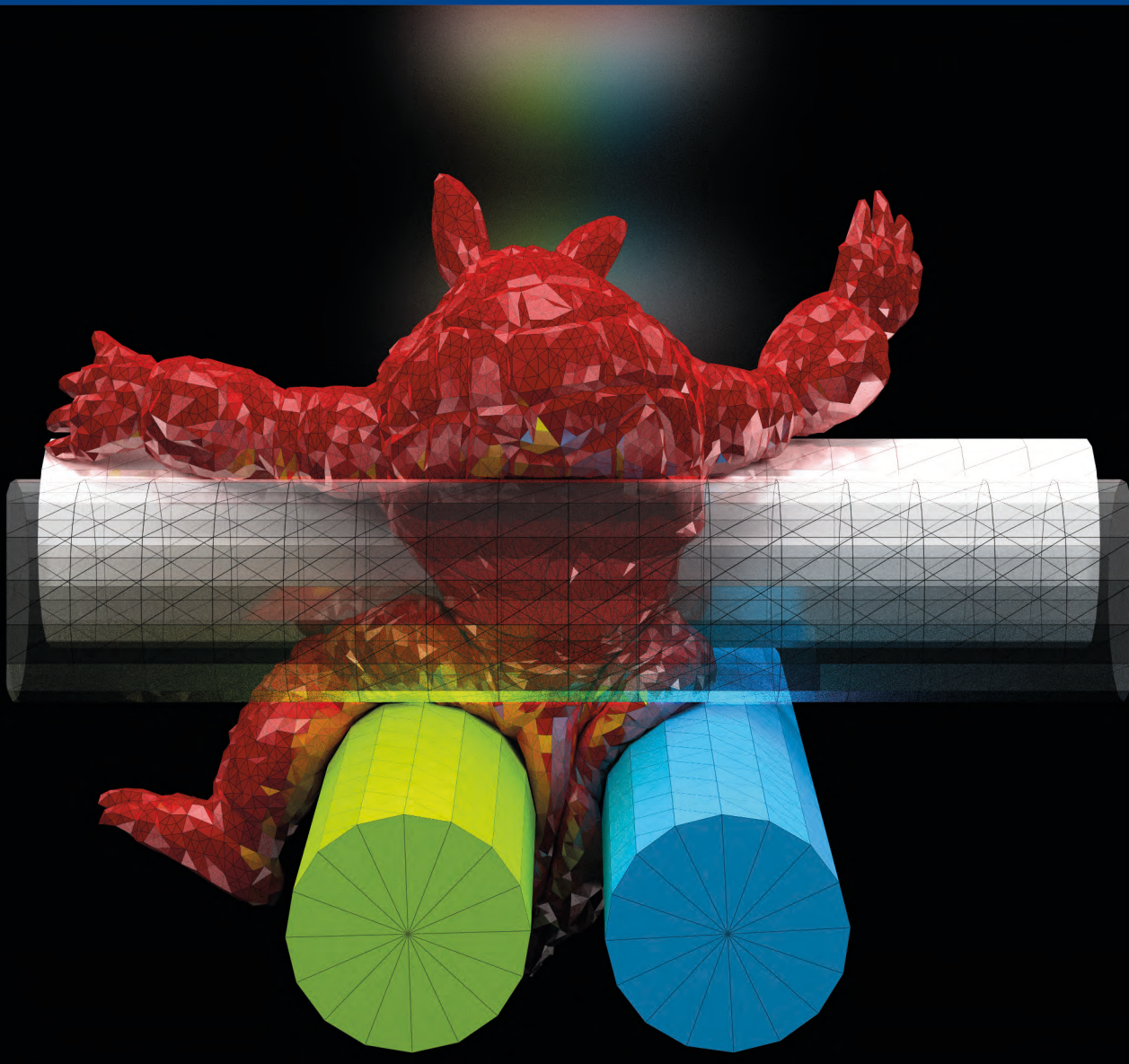
Biography

Mickeal Verschoor was born on September 20 1980 in Dordrecht, The Netherlands. After completing his secondary pre-vocational and vocational education in electrical engineering at S.G. de Grienden (Sliedrecht 1996) and Da Vinci College (Dordrecht 2000), he studied 'Media technology' at University of Applied Sciences Utrecht. Here he got interested in computer graphics by following courses on rendering techniques and participated in computer graphics and gaming projects. He obtained his Bachelor of Engineering degree in 2004 with honors and continued at the University of Groningen where he obtained in 2005 his Bachelor of Science degree in Computing Science. In the same year Mickeal continued with the master Computing Science at the same university. During this master he was particularly interested in numerical mathematics, computer graphics and simulation methods. In 2008 he finished his master thesis 'Fluid Simulations for Computer Graphics', supervised by dr. A.C. Jalba in the Scientific Visualization and Computer Graphics group of prof. dr. J.B.T.M. Roerdink, and he obtained his Master of Science degree with honors.

In 2009 he started as a PhD candidate at the Mathematics and Computer Science department of Eindhoven University of Technology under supervision of prof. dr. ir. J.J. van Wijk and dr. A.C. Jalba. During his PhD he developed efficient methods for accelerating numerical methods on GPUs[183], worked on GPU simulations of deformable models[184] and worked on coupled simulations of deformable and rigid models together with accurate treatment of contact and friction[185].

In 2015 Mickeal started to work part-time for Indicia in Tilburg on GPU based rendering methods for procedurally generated shapes. Since 2016 he is a researcher at the Multimodal Simulation Lab of dr. M.A. Otaduy at Universidad Rey Juan Carlos in Madrid, Spain, where he participated in the FP7 WEARHAP project, working on simulation-based hand interaction methods for virtual reality[186], and haptic and tactile rendering methods[112].

In this dissertation we improve upon methods for simulating elastically deformable objects applied to computer animation, while taking accuracy and stability into account. We address how to efficiently use Graphics Processing Units (GPUs) for assembling and solving numerical problems related to such simulations. Furthermore, we present methods for solving contact and friction between the surfaces of the simulated objects. Here accurate collision detection is essential, when also accounting for possible deformations.



TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

ISBN: 978-90-386-4736-4

"54 68 65 72 65 20 69 73 20 6E 6F 20 61 72 6D 61 64 69 6C 6C 6F 2E 2E" - M.Verschoor