Numerical prediction of the keel resistance of first-year pressure ice ridges on offshore structures

Master's Thesis

Departement of Naval Architecture and Ocean Engineering

Submitted by:	Johannes Seidel
Enrolment Number:	31 84 09
1st Reviewer:	Prof. Dr.-Ing. Andrés Cura Hochbaum
2nd Reviewer:	Dipl.-Ing. Peter Jochmann

Eidesstattliche Erklärung

Die selbständige und eigenhändige Anfertigung versichert an Eides statt

Berlin, den 28. Januar 2016

.....

Johannes Seidel

Abstract

In this thesis the development of a simulation tool based on the Discrete Element Method for approximation of keel resistance in first-year pressure ice ridges on offshore structures in model scale is discussed. To achieve this an overview of ice ridges in nature and in the model basin, their characteristics and the challenge for arctic engineers is given and subsequently different potential numerical modelling techniques are presented. The complete calculation algorithm and its implementation written in the programming language Fortran is explained in detail and validated by running simple test cases by re-creation, simulation and evaluation of suitable model tests, which were performed at the Hamburg Ship Model Basin. In addition further improvement suggestions are made.

Zusammenfassung

Die vorliegende Arbeit behandelt die Entwicklung eines auf Diskreten Elementen basierenden Simulationstools zur Approximation des Widerstands des Kiels eines sogenannten First-Year Pressure Ice Ridges auf Offshore Strukturen im Modellmaßstab. Dahinführend wird ein Überblick zu Ridges in der Natur und im Modell gegeben, ihre besonderen Merkmale sowie die Herausforderungen für Ingenieure beleuchtet und unterschiedliche mögliche Ansätze zur numerischen Modellierung vorgestellt. Im Hauptteil wird der Berechnungsalgorithmus und seine Implemetierung mittels der Programmiersprache Fortran detailliert behandelt und anschließend durch einfache Tests validiert. Den Schluss dieser Arbeit bilden mehrere Simulationen geeigneter Modellversuche, die bei der Hamburgischen Schiffbau-Versuchsanstalt durchgeführt wurden, und deren Auswertung sowie darauf basierend Vorschläge zur Verbesserung und Erweiterung des entwickelten Tools.

Keywords: Numerical Ridge, Keel Resistance, Discrete Element Method, Fortran

Contents

List of Figures

List of Tables

Nomenclature

Acronyms

S^b	Body-fixed coordinate system
S^c	Co-moving coordinate system
S^f	Frozen coordinate system
S^s	Space-fixed coordinate system
AABB	Axis Aligned Bounding Boxes
BDF	Backward Differentiation Formulas
CFD	Computational Fluid Dynamics
DEM	Discrete Element Method
FEM	Finite Element Method
HSVA	Hamburgische Schiffbau-Versuchsanstalt
ODE	Ordinary Differential Equation
OOP	Object Orientated Programming
SAP	Sweep And Prune

Symbols

$\boldsymbol{\omega}$	Angular velocity	rad/s
$\boldsymbol{\tau}$	Torque	$\mathrm{N} \cdot \mathrm{m}$
$\boldsymbol{c}$	Centroid	m

$\boldsymbol{F}$	Face	-
$\boldsymbol{F}$	Total force acting on an element	N
$\boldsymbol{f}_b$	Buoyancy force	N
$\boldsymbol{f}_c$	Contact force	N
$\boldsymbol{f}_d^n$	Damping force component, normal direction	N
$\boldsymbol{f}_f^t$	Frictional force component, tangential direction	N
$\boldsymbol{f}_g$	Gravitational force	N
$\boldsymbol{f}_h$	Hydrodynamic forces	N
$\boldsymbol{f}_{coh}^n$	Cohesive force component, normal direction	N
$\boldsymbol{f}_{el}^n$	Elastic force component, normal direction	N
$\boldsymbol{I}$	Inertia tensor in S^s	$\mathrm{kg} \cdot \mathrm{m}^2$
$\boldsymbol{I}^b$	Inertia tensor in S^b	$\mathrm{kg} \cdot \mathrm{m}^2$
$\boldsymbol{L}$	Angular momentum	$\mathrm{N} \cdot \mathrm{m/s}$
$\boldsymbol{n}$	Normal vector	-
$\boldsymbol{n}_c$	Contact area normal vector	-
$\boldsymbol{Pl}$	Plane	-
$\boldsymbol{P}$	Polygon	-
$\boldsymbol{q}$	Quaternion	-
$\boldsymbol{R}$	Rotation matrix	-
$\boldsymbol{r}$	Center of mass	m
$\boldsymbol{r}_i$, $\boldsymbol{r}_j$	Vector connecting two centroids	m
$\boldsymbol{v}$	Vertex	m
$\boldsymbol{v}_t$	Tangential contact velocity	m/s
Δt	time step	s
γ^n	Normal damping coefficient	-
λ	Nondimensional length parameter	-
μ	Friction coefficient	-

ρ	Density	kg/m^3
A_o	Overlap area	m
d	distance	m
g	Gravitational constant	m/s^2
L_c	Characteristic length	-
m	Mass	kg
M_{red}	Reduced mass	kg
Re	Reynolds number	-
t	time	s
V	Volume	m^3
V_o	Overlap volume	m^3
Y	Young's Modulus	N/m^2

Chapter 1

Introduction

1.1 Background

Due to climate change and technical improvement the interest in opening-up the arctic regions has been increasing throughout the last decades, but the harsh environment still remains one of the biggest challenges for arctic engineers. Ships and offshore structures need to withstand the environmental forces to ensure the safety of crew, environment and the structure itself and as well as manoeuvring capabilities. One of the main factors for a successful operation in the Arctic is good performance of a vessel in so-called *ice ridges*. Ice ridges are an important, common feature in arctic regions because they add a significant amount to the overall forces acting on a vessel. It is believed that 40 % of the overall sea ice is ridged ice [see 4], therefore there is much interest in research and development of this issue.

In the context of increasing activies in arctic regions, the present work focuses on the estimation of the ridge breaking ability of offshore structures. This thesis explores the working principle of a newly developed simulation software based on the Discrete Element Method (DEM).

1.2 Ice ridges at the HSVA

The Hamburg ship model basin (HSVA) contributes to this subject by providing test facilities for model testing of ships and offshore structures in arctic environments as well testing facilities for fundamental research of ice mechanics and dynamics.

The HSVA's arctic technology department is capable of building ice ridges of many types and sizes in model scale for testing with ships and offshore structures. The scope of these tests is to determine the forces and torques, vibrations and motions in general during the penetration of such ridge. Some fundamental experimental studies for prediction of the induced resistance during ridge breaking of ships have been performed by Ehle [5] in 2011. This work enables calculating a ship's velocity and resistance during ridge breaking, taking into account the bow shape and ship's power. A numerical implementation has been performed by Hisette [6] in 2014. The aforementioned method is based on available analytical formulae and results of model tests in ice tank but does not allow visualisation of ship behaviour in the ridge and simulate different possible interaction scenarios.

Ridges also effect offshore structures due to current and wind induced movements of the ice mass. Floating structures are mostly affected, but because of the possible enormous draught, submerged structures and even the sea bottom can come into contact with ridges. In the context of the international research project *Hydralab III* (2008) Serré et al. [7] investigated the interaction between a ridge keel and a submerged offshore structure.

Chapter 2

Fundamentals of pressure ice ridges

2.1 Formation of pressure ridges

Pressure ridges are piles of ice rubble that crisscross the arctic ice pack, created from the rubble of ice broken during compressive deformation. Hopkins [8] suggests to divide the ridging process into four stages, in which the first stage starts with two intact ice sheets of lead ice coming into contact and ends when the sail reaches its maximum height. In the second stage the ridge keel grows wider and deeper until the maximum keel draft is reached. The third stage, in which leeward growth creates a rubble field of more or less uniform thickness, ends when the supply of thin ice is exhausted. In the fourth stage the rubble field is compressed between converging floes.

Ridges typically have two distinct parts, which are shown in figure 2.1.

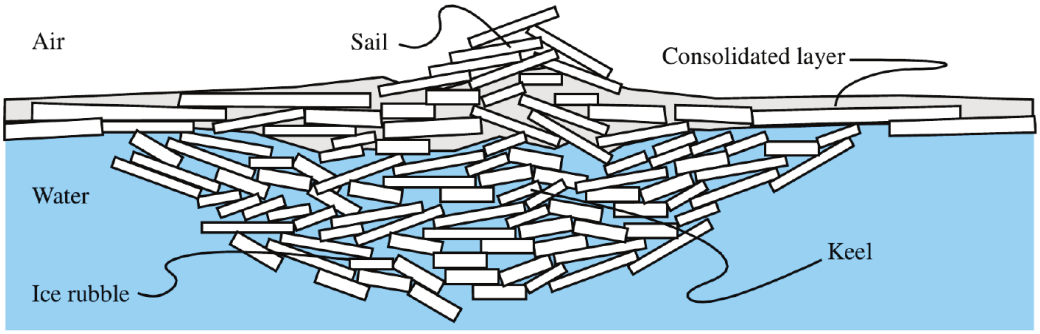


Figure 2.1: Cross section of a ridge (from [1])

The visible part above the waterline is called the sail and the underwater part the keel, which can be further divided into the consolidated layer and the ice rubble. The porosity, which is the fraction of the volume of voids over the total volume, varies from 10 and 40 %, depending on whether they are a result of compression or shearing (see [9]). Pictures 2.2 and 2.3 show, how the visible part of ridges could occur in the Arctic.

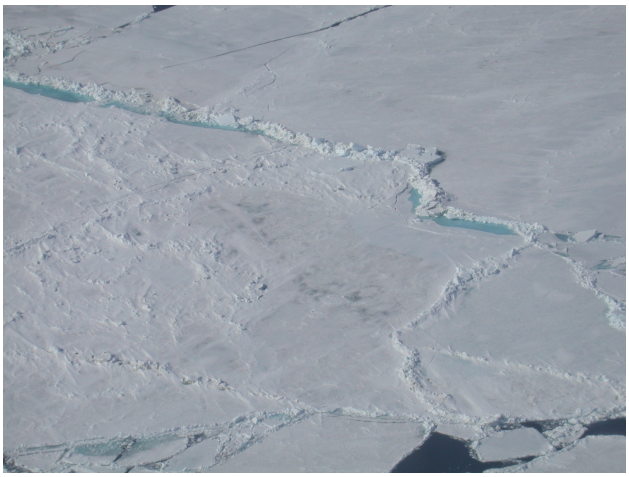


Figure 2.2: Ridges in the Arctic

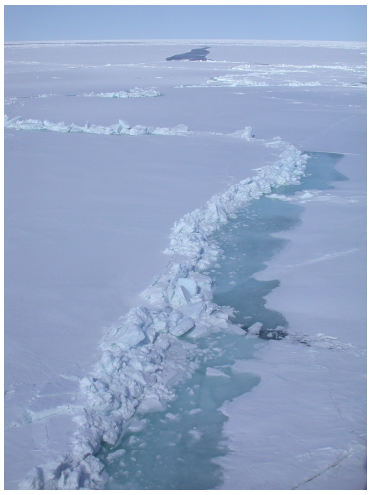


Figure 2.3: Ridges in the Arctic

One also has to distinguish between first-year and older (second-year and multi-year) ice ridges, where the consolidation of the ice rubble increases during its lifetime. The keels of first-year ice ridges are mostly loose or slightly bonded together by freeze bonds and over the years melting and freezing in turn lead to the formation of an almost completely coherent

ice block, which is shown in figure 2.4. Obviously old ridges lead to higher loads on structures than first-year ridges.

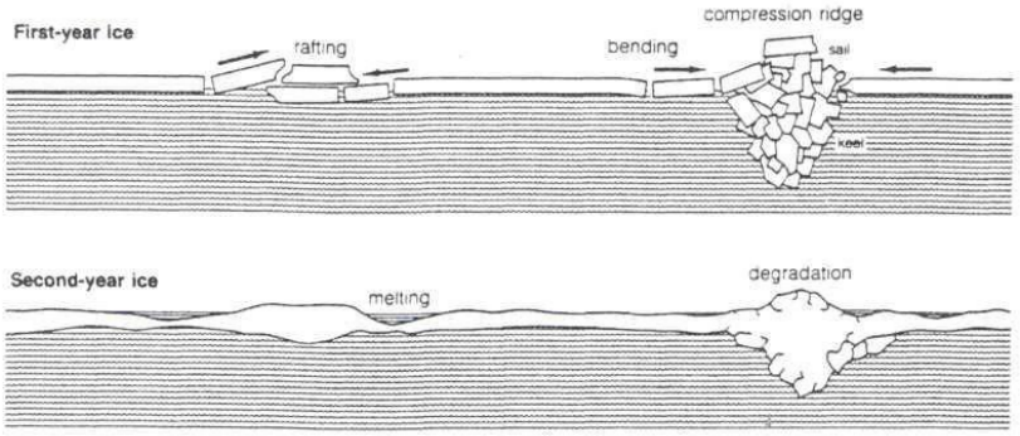


Figure 2.4: First-year and second-year ridges (from [2])

The following attributes, proposed by Timco [10], are commonly used to describe a ridge:

- Keel length L, which is measured along the contact edge of the colliding floes.
- Keel width W_k, which is measured along the consolidated layer
- Keel bottom width BW_k, which is measured at the keel bottom
- Keel depth H_k, which is the draught of the ridge
- Keel porosity P_k, which describes the fraction of enclosed water inside the ridge
- Sail width W_s, is the width measured on the consolidated layer
- Sail height H_s
- Sail porosity P_s

The largest recorded ridges (see Weeks [11]) had a keel depth of 45 m and a sail height of 12 m.

2.2 Principles of load prediction

Alongside the geometrical attributes the mechanical properties of ridges have to be known for comparability and semi-empirical formulae. Therefore so-called *punch tests*, introduced in 2008 at the HSVA, are executed in order to assess the internal shear strength of the ice keel rubble by measuring the ridge action on a penetrating cylinder. The cylinder (mass = 122 kg, diameter = 25 cm) attached to a crane through a load cell is lowered into the ridge with a constant speed $v = 6.7\,mm/s$ (see figure 2.2)

The cylinder displacement is recorded together with the load exerted by the ridge.

Figure 2.5: Initial punch test setup

Figure 2.6: Punch test underwater

A typical result of a punch test is shown in figure 2.7 (after substracting its own buoyancy).

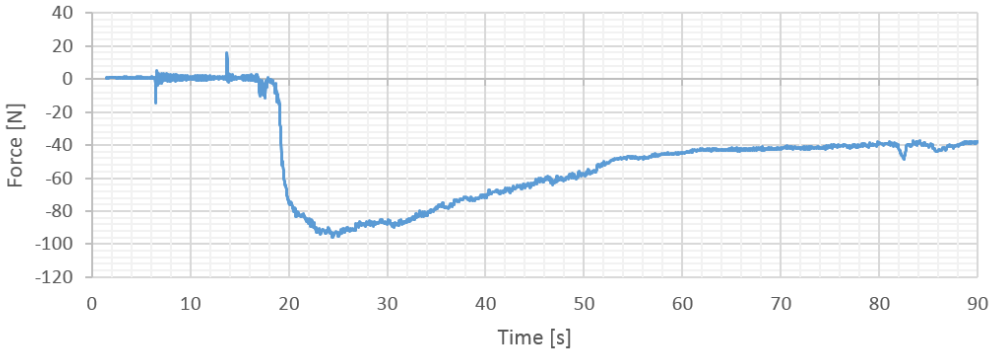


Figure 2.7: Typical evolution of z-forces acting on a punch device

Chapter 3

Literature research

3.1 Modeling techniques

3.1.1 Finite Element Method

The Finite Element Method approach for ice related simulations has been state of the art for a long time due to the long history and well-engineered algorithms of this method. It is capable of modelling ice pressure and global ice load for the level ice, icebergs and ice ridges. Nowadays the use of the ice-related FEM is decreasing but is still in use. Serré [12] for example used the FEM for the simulation of ice ridges which are driven against a submerged offshore structure. For this kind of application the FEM produces satisfying and quick results. But when it comes to the breaking of ice and the prediction of movements of single floes, for example to analyse whether a design of a ship hull prevents the rubble to be transported into the propeller disc area, the FEM is not suitable.

3.1.2 Discrete Element Method

The Discrete Element Method was introduced as the *Distinct Element Method* by Cundall [13] in 1979 in order to solve problems from rock mechanics. In this method the domain is represented by a discontinuous lagrangian treatment of discrete particles which match real particle

sizes and preferably their shapes. The computational focus lays in the detection of particle collisions and the calculation of contact forces.

One of the first applications of discrete elements for ice-related simulations can be found in the simulation of the sea ice ridging process in two dimensions with rectangular elements, presented by Hopkins in [8]. Subsequent big scale simulations tend to model ice floes as circular disks in 3D (Hansen and Løset [14]) or as sharp-edged polygons in two dimensions, as well by Hopkins [15] in 2004, as a compromise between computational complexity and a realistic reproduction of the floes.

In small scale investigations the level of detail increases. Puntigliano [16] e.g. reproduces a significantly more detailed computer model, implementing a complex ship geometry and a countable set of pre-broken ice floes. An even more complex combined Finite-Discrete Element Method is introduced by Munjiza [17], which is used by Polojärvi in [18] as a numerical approach for simulating a punch test.

Nowadays, as a consequence of the advantages over the FEM in the physical representation of the ice behaviour, the use of discrete elements has been established as common practice. The DEM shows intrinsic advantages in the ice load calculation because of its ability to describe the discrete nature of ice structure on macro-scale and its potential to model reasonably the ice breakup during the ice-vessel/ice-ice interaction on micro-scale.

3.2 Conclusion

The advantages of the Discrete Element Method for ice-related simulations are the reasons, why the author decided to develop a tool based on discrete elements. To accomplish this, the thesis “Discrete Element Method for 3D simulations of mechanical systems of non-spherical granular materials” by Chen [3] and the related book “Understanding the Discrete Element Method” by Matuttis and Chen [19] are the main sources of information for implementing the algorithms and general understanding of the DEM.

As shown in chapter 2.1 a ridge consists of a large heap of ice elements. To ensure the conservation of this formation during time progression , the algorithm of Matuttis and Chen succeeds “to construct a three dimensional heap on a smooth ground with a realistically high angle re-

pose. This outperforms any existing DEM codes which need manipulations of boundary conditions or unrealistically large friction parameters." [3, p.147].

Chapter 4

Kinematics and time integration

The fundamental of the Discrete Element Method is the resolution of the time evolution of every element in the domain. Without considering the inter-particle force laws, the motion of a single particle is treated as that of a rigid body subjected to external forces. In this chapter the basics of rigid-body mechanics and numerical algorithms for solving ordinary differential equations are introduced.

4.1 Coordinate systems

Since translation and rotation don't have to be described as compulsory in the same coordinate system, different coordinate systems must be carefully distinguished. The *space-fixed system* S^s is particle-independent. It is an arbitrarily defined right-handed, time-independent, Cartesian inertial system. This is the native system of the simulation. Ultimately, all coordinates and other values describing the particles' state have to be expressed in this system.

To separate the conceptually simple translational motion from the more complicated rotation the *co-moving system* S^c is introduced. Its origin is located in the centre of mass of the particle, whilst its axes are parallel to the axes of the space-fixed system. In this system the particle performs a pure rotation. The motion of S^c itself with respect to S^s is completely described by the motion of the centre of mass of the particle. Newton's equation of motion for this translational motion can be solved in precisely

the same way as for two-dimensional systems, therefore it will not be discussed here.

The integration of the rotational degrees of freedom is more complicated: Newton's equation of motion is usually formulated in the space-fixed coordinate system S^s. In this coordinate system, however, the moments of inertia of the particles are time-dependent and contain off-diagonal elements. This complication is overcome by introducing the *body-fixed* coordinate system S^b whose origin coincides with the origin of the System S^c and whose axes are aligned along the particle's principal axes. In this system the moment of inertia tensor $\boldsymbol{I}$ adopts its simplest form, i.e., it is constant and diagonal:

$$\boldsymbol{I}^b = \begin{bmatrix} \boldsymbol{I_{xx}} & 0 & 0 \\ 0 & \boldsymbol{I_{yy}} & 0 \\ 0 & 0 & \boldsymbol{I_{zz}} \end{bmatrix}. \tag{4.1}$$

Unfortunately the system S^b is not inertial since it moves with yet unknown acceleration due to the interaction with other particles. Therefore, equations of motion cannot be formulated in this system. To resolve this problem a fourth coordinate system, the *frozen* system S^f, which is identical to the body-fixed coordinate system at time t, is defined.

To compute the coordinates, orientation, and the corresponding velocities of a particle i at time $t + \delta t$, the following procedure is implemented in the current tool:

1. Compute the forces and torques which act on particle i due to interaction with other particles and the wall. These values are represented in the space-fixed system S^s.
2. Transform all relevant variables describing particle i into the comoving system S^c where the centre of mass position is $(0, 0, 0)^T$.
3. Formulate the equation of motion for the orientation in S^c and compute the new orientation and the according velocities by integrating Euler's equation over the time interval δt.
4. Compute the new position of the centre of mass of i in S^s by integrating Newton's equation of motion for the translational degrees of freedom.

This procedure is repeated for all particles of the system. Note that the body-fixed system is only used during the initial calculation of the inertia tensor for each element. Its practical use is indeed small since it is not

inertial - it is only needed as an abstract concept to define the frozen system. Please note also that the frozen system as well is not used at all. Other algorithms however propose to transform the external forces into S^f, to resolve the equations in S^f and afterwards to transform the displacements back into S^s (see [20]).

4.2 Equations of motion

The representation of a rigid body in three dimensional Euclidian space at a certain time t is done by using a fixed Cartesian frame. The motion of such element is decomposed into a translational displacement of the centres of mass and the rotational motion around the centres of mass. The translational motion of a particle at time t follows Newton's second law

$$\ddot{\boldsymbol{r}}(t) = \frac{\boldsymbol{F}(t)}{m}, \tag{4.2}$$

where m is the mass, $\boldsymbol{r}(t)$ the vector of the centre of mass and $\boldsymbol{F}(t)$ the external force vector of one element. The external force

$$\boldsymbol{F}(t) = \boldsymbol{f}_g + \boldsymbol{f}_b + \boldsymbol{f}_h + \sum_{j=1}^{l} \boldsymbol{f}_{c_j}(t) \tag{4.3}$$

is the sum of the gravitational force $\boldsymbol{f}_g$, buoyancy force $\boldsymbol{f}_b$, hydrodynamic forces $\boldsymbol{f}_h$ and contact forces $\boldsymbol{f}_{c_j}(t)$, where l is the total number of contacts per element. A contact force itself

$$\boldsymbol{f}_c = \boldsymbol{f}_{el}^n + \boldsymbol{f}_d^n + \boldsymbol{f}_{coh}^n + \boldsymbol{f}_f^t \tag{4.4}$$

can be divided into a normal component, which is the sum of a repulsive force $\boldsymbol{f}_{el}^n$, a dissipative force $\boldsymbol{f}_d^n$ and a cohesive force $\boldsymbol{f}_{coh}^n$, and a dissipative tangential friction force $\boldsymbol{f}_f^t$, which are shown in figure 4.1.

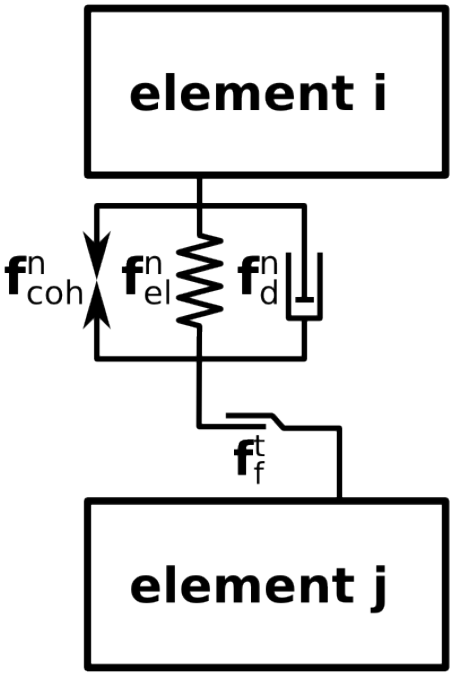


Figure 4.1: Contact force model

The essential difference between translation and rotation is that the order of rotation doesn't commute. Torques are usually calculated in the space-fixed coordinate system, therefore in the following, the superscript $()^s$ for all variables is neglected. Hence, the torques are denoted by $\boldsymbol{\tau}$. They equal the rate of change with respect to time of the moment of inertia:

$$\dot{\boldsymbol{L}} = \boldsymbol{\tau}, \tag{4.5}$$

in which $\boldsymbol{L}$ is the angular momentum and $\boldsymbol{\tau}$ is the external torque introduced by the contact force. Substituting $\boldsymbol{L} = \boldsymbol{I}\boldsymbol{\omega}$ gives

$$\dot{\boldsymbol{I}}\boldsymbol{\omega} + \boldsymbol{I}\dot{\boldsymbol{\omega}} = \boldsymbol{\tau}. \tag{4.6}$$

Here $\boldsymbol{I}$ is the inertia tensor and $\boldsymbol{\omega}$ the angular velocity, both in space-fixed coordinates.

To obtain the inertia tensor in S^s, one has to explicitly transform $\boldsymbol{I}^b$ at every new time step, using

$$\boldsymbol{I}^s = \boldsymbol{R}\boldsymbol{I}^b\boldsymbol{R}^T, \tag{4.7}$$

where $\boldsymbol{R}$ is the rotation matrix of following structure

$$\boldsymbol{R} = \begin{bmatrix} r_{xx} & r_{yx} & r_{zx} \\ r_{xy} & r_{yy} & r_{zy} \\ r_{xz} & r_{yz} & r_{zz} \end{bmatrix}. \tag{4.8}$$

Each column of $\boldsymbol{R}$ represents the direction of the $x'-, y'-, z'-$axis of the body-fixed system in the space-fixed coordinate system. The rotation could be done by using Euler angles, but as shown in [20] they can become singular in the transformation between the angle variables and the component of rotation matrix.

A solution to this is the use of quaternions. A quaternion $\boldsymbol{q}$ is a three-dimensional representation of a complex number, which consists out of a scalar part s and a triple (x, y, z) called the vector part $\boldsymbol{v}$. The linear combination expression for a quaternion can be written in the following form:

$$\begin{aligned} \boldsymbol{q} &= w + x\boldsymbol{I} + y\boldsymbol{J} + z\boldsymbol{K} && \text{(4.9a)} \\ &= [w, x, y, z] && \text{(4.9b)} \\ &= (s, \boldsymbol{v}), && \text{(4.9c)} \end{aligned}$$

where $\boldsymbol{I}, \boldsymbol{J}, \boldsymbol{K}$ are imaginary numbers, w is the representation of the scalar part and $\boldsymbol{v}$ is the representation of the vector part. However in contrast to linear algebra the multiplication of two quaternions, say $\boldsymbol{q_1} = (s_1, \boldsymbol{v_1})$ and $\boldsymbol{q_2} = (s_2, \boldsymbol{v_2})$, turns out to be

$$\boldsymbol{q_1} \cdot \boldsymbol{q_2} = (s_1 s_2 - \boldsymbol{v_1} \cdot \boldsymbol{v_2}, s_1 \boldsymbol{v_2} + s_2 \boldsymbol{v_2} + \boldsymbol{v_1} \times \boldsymbol{v_2}), \tag{4.10}$$

with the usual cross product and dot product for three-dimensional vectors. Due to the anti-commutativity of the cross product the quaternion product itself is anti-commutative.

For an equation of the rotative motion one additionally needs time derivatives of the quaternions and their relationship to the angular velocities $\boldsymbol{\omega}$. The first time derivative with respect to time of a quaternion can be expressed by

$$\dot{\boldsymbol{q}} = \frac{1}{2}\boldsymbol{\omega}\boldsymbol{q}. \tag{4.11}$$

Therefore, the second time derivative, which is needed to write the equation of motion like Newton's law, is

$$\ddot{\boldsymbol{q}} = \frac{1}{2}(\dot{\boldsymbol{\omega}}\boldsymbol{q} + \boldsymbol{\omega}\dot{\boldsymbol{q}}) \tag{4.12}$$

This equation and equation (4.2) are differential equations of second order, which form the equations to describe the motions of a rigid body. They can be solved numerically as any other ODE for which in the following chapter a popular solving algorithm is introduced.

To receive $\boldsymbol{\omega}$ and $\dot{\boldsymbol{\omega}}$ the following auxiliary equations are needed:

$$\boldsymbol{\omega} = 2\dot{\boldsymbol{q}}\boldsymbol{q}^*, \tag{4.13a}$$

$$\dot{\boldsymbol{\omega}} = \boldsymbol{I}^{-1}(\boldsymbol{L} \times \boldsymbol{\omega} + \boldsymbol{\tau}), \tag{4.13b}$$

$$\boldsymbol{L} = \boldsymbol{I}\boldsymbol{\omega}, \tag{4.13c}$$

$$\boldsymbol{I} = \boldsymbol{R}\boldsymbol{I}^b\boldsymbol{R}^T, \tag{4.13d}$$

$$\boldsymbol{I}^{-1} = \boldsymbol{R}(\boldsymbol{I}^b)^{-1}\boldsymbol{R}^T. \tag{4.13e}$$

where the angular velocity $\boldsymbol{\omega}$, angular momentum $\boldsymbol{L}$, torque $\boldsymbol{\tau}$ are computed in the space-fixed coordinate system. Therefore, the quaternion $\boldsymbol{q}$ and its time derivatives are represented in S^s as well. The moment of inertia $\boldsymbol{I}$ is obtained from the moment of inertia in the body-fixed system, $\boldsymbol{I}^b$, by the principal axis transformation shown in subequation (4.13d). The transformation matrix $\boldsymbol{R}$ is obtained from a quaternion $\boldsymbol{q} = (w, x, y, z)$ by

$$\boldsymbol{R} = \begin{bmatrix} w^2 + x^2 - y^2 + z^2 & 2(xy + wz) & 2(xz - wy) \\ 2(xy - wz) & w^2 - x^2 + y^2 - z^2 & 2(yz + wx) \\ 2(xz + wy) & 2(yz - wx) & w^2 - x^2 - y^2 + z^2 \end{bmatrix}. \tag{4.14}$$

Coordinate transformations of a pure vector $\boldsymbol{r}$ with quaternions $\boldsymbol{q}$ are obtained by

$$\tilde{\boldsymbol{r}} = \boldsymbol{q}\boldsymbol{r}\boldsymbol{q}^*, \tag{4.15}$$

in which $\boldsymbol{q}^* = w - x\boldsymbol{I} - y\boldsymbol{J} - z\boldsymbol{K}$ is the transposed quaternion and $\boldsymbol{r} = [0, r_x, r_y, r_z]$ the quaternion-represention of a pure vector.

Using the translational vector $\boldsymbol{r}_c$ obtained by Newton's equation and the quaternion $\boldsymbol{q}$ every point of a rigid body can be expressed in the space-fixed coordinate system. Let $\boldsymbol{p}^b$ be a vector in the body-fixed coordinate system then its vector $\boldsymbol{p}$ in the space-fixed coordinate system is

$$\boldsymbol{p} = \boldsymbol{r}_c + \boldsymbol{q}\boldsymbol{p}^b\boldsymbol{q}^*. \tag{4.16}$$

[19] also note that the unit quaternion $\boldsymbol{q}$ should be normalised in frequent intervals after calling the integrator to preserve its unit length. Otherwise

it would not only rotate the elements but also change their shape. Secondly they noticed in their research that the first time derivative of $\boldsymbol{q}$ should be constrained by an frequent orthogonalisation of $\boldsymbol{q}$ and $\dot{\boldsymbol{q}}$, see equation (4.13a). By this orthogonalisation, since $\boldsymbol{\omega}$ is a pure vector, it is ensured that the scalar part of the product $\boldsymbol{q}\dot{\boldsymbol{q}}$ vanishes, i.e.

$$\boldsymbol{q}\dot{\boldsymbol{q}} = (s_q s_{\dot{q}} - \boldsymbol{v}_q \cdot \boldsymbol{v}_{\dot{q}}, \dots) \stackrel{!}{=} (0, \dots) \tag{4.17}$$

For the complete discourse on quaternions please refer to [21]. Further in detail explanations in the DEM-related context can be found in [19] or [20].

4.3 Time integration

For the numerical approximation of the equations of motion, eq. (4.2) and eq. (4.12), a backward-difference formula in the form of Gear's predictor-corrector algorithm is used. As it is an implicit method its advantages are its stability and efficiency, which additionally needs neither a matrix inversion nor the solution of a non-linear system of equations.

4.3.1 Explicit and implicit methods

To explain the the principle of the implicit predictor-corrector methods the first order Euler method is used as example. Since one always can rewrite a higher-order ODE as a system of first-order ODEs, we start with following general form of first-order differential equations

$$\dot{y}(t) = f(t). \tag{4.18}$$

The lowest order discretisation of this equation with finite differences can be done with two different ways. The first one uses the old gradient

$$\frac{y(t+\delta t) - y(t)}{\delta t} = f(t) \tag{4.19}$$

which is called the "Euler forward method", since the tangent is pointing in the forward direction. Alternatively the "new" gradient

$$\frac{y(t+\delta t) - y(t)}{\delta t} = f(t+\delta t) \tag{4.20}$$

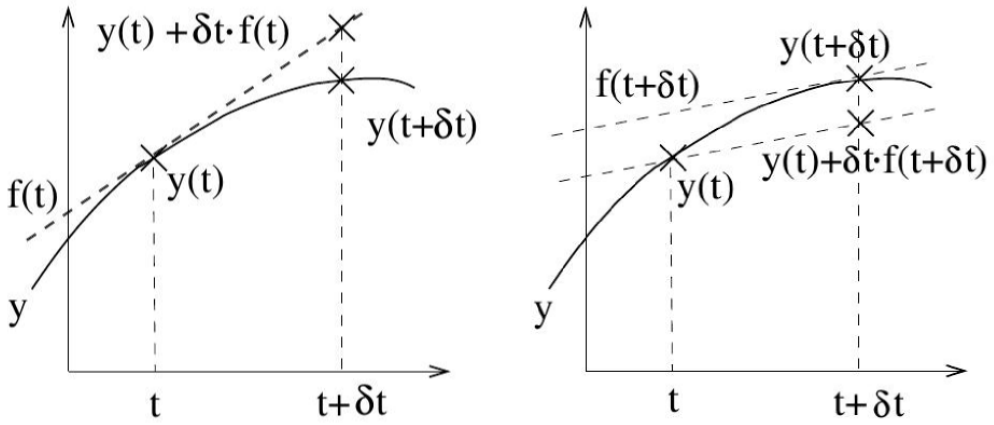


Figure 4.2: left: explicit Euler, right: implicit Euler (from [3])

can be used to approximate the time derivative. This commonly known as the "Euler backward method".

Rewriting these two equations, an expression of the quantity of y in the next time step is received

$$y(t+\delta t) = y(t) + f(t)\delta t \tag{4.21a}$$
$$y(t+\delta t) = y(t) + f(t+\delta t)\delta t, \tag{4.21b}$$

where the equation (4.21a) is the explicit expression, because all terms necessary to compute $y(t+\delta t)$, namely $y(t)$ and $f(t)$, are known at time t. In contrast the equation (4.21b) makes use of the not yet known $f(t+\delta t)$. Therefore, it's called an implicit expression.

To obtain an estimate for the yet unkown $f(t+\delta t)$ at $t+\delta t$ in the current implementation, $f(t+\delta t)$ is approximated via the explicit Euler, which is called the "predictor step"

$$y^p(t+\delta t) = y(t) + f(t)\delta t, \tag{4.22}$$

One can see from 4.2 that the numerical approximation via the explicit method overestimates the correct solution, while the implicit method leads to an underestimation of the solution value. Therefore, in the predictor-corrector scheme the predicted values are corrected using an implicit correction step. This is done in such way that the error introduced by prediction is to be compensated by the correction error. This leads to an overall smaller error and good stability compared to using explicit or implicit Euler alone.

For better results, higher-order predictors can be used which do not only use values from the current time step, $y(t)$ and $f(t)$, but also from previous time steps, $y(t-\delta t)$, $f(t-\delta t)$, ... To obtain a higher-order corrector formula the old right-hand sides $f(t)$, $f(t-\delta t)$, $f(t-2\delta t)$, ... together with the old function values $y(t)$, $y(t-\delta t)$, $y(t-2\delta t)$, ... are used, which is procedure proposed by Gear [22]. The advantage of Gear's predictor-corrector formulation over other multi-step methods, like the Adamas-Moulton family, are its efficiency and the stability at evaluating stiff differential equations, i.e. the ability to neglect oscillations due to high gradients in the characteristics of the solved ODEs. Moreover, it's an implicit method which doesn't need a solution of a non-linear system of equations nor a matrix inversion which is the case for implicit Runge-Kutta methods.

4.3.2 Gear's predictor-corrector scheme

Since most solvers in the field of numerical mathematics are developed to solve first-order ordinaray differential equations (ODE), the common practice in the solution of higher-order ODEs is to rewrite them into a system of first-order ODEs. E.g., in the current case of Newton's equation of motion, equation (4.2), the system of coupled first-order differential equations would result to

$$\begin{cases} \dot{\boldsymbol{r}} &= \boldsymbol{v} \\ \dot{\boldsymbol{v}} &= \frac{\boldsymbol{F}(t)}{m}, \end{cases} \tag{4.23}$$

which then are solved simultaneously. A feature of Gear's algorithm is its possibility to be applicable directly on second-order ODEs, like the above introduced equations of motion. Let

$$\boldsymbol{r}_n = \frac{\delta t^n}{n!} \frac{d^n \boldsymbol{r}_0}{dt^n} \tag{4.24}$$

be the n-th time derivative of $\boldsymbol{r}_0$, which is the vector of the center of mass, rescaled by a factor $\frac{\delta t^n}{n!}$, then the six-value prediction for $\boldsymbol{r}_i$ $(i = 0, 1, \ldots, 5)$

at time $t + \delta t$ starting from t is a simple application of the Taylor series

$$\begin{bmatrix} \boldsymbol{r}_0^p(t+\delta t) \\ \boldsymbol{r}_1^p(t+\delta t) \\ \boldsymbol{r}_2^p(t+\delta t) \\ \boldsymbol{r}_3^p(t+\delta t) \\ \boldsymbol{r}_4^p(t+\delta t) \\ \boldsymbol{r}_5^p(t+\delta t) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 0 & 1 & 3 & 6 & 10 \\ 0 & 0 & 0 & 1 & 4 & 10 \\ 0 & 0 & 0 & 0 & 1 & 5 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \boldsymbol{r}_0(t) \\ \boldsymbol{r}_1(t) \\ \boldsymbol{r}_2(t) \\ \boldsymbol{r}_3(t) \\ \boldsymbol{r}_4(t) \\ \boldsymbol{r}_5(t) \end{bmatrix}. \tag{4.25}$$

The corrected values $\boldsymbol{r}_i^c$ are calculated using

$$\begin{bmatrix} \boldsymbol{r}_0^c(t+\delta t) \\ \boldsymbol{r}_1^c(t+\delta t) \\ \boldsymbol{r}_2^c(t+\delta t) \\ \boldsymbol{r}_3^c(t+\delta t) \\ \boldsymbol{r}_4^c(t+\delta t) \\ \boldsymbol{r}_5^c(t+\delta t) \end{bmatrix} = \begin{bmatrix} \boldsymbol{r}_0^p(t+\delta t) \\ \boldsymbol{r}_1^p(t+\delta t) \\ \boldsymbol{r}_2^p(t+\delta t) \\ \boldsymbol{r}_3^p(t+\delta t) \\ \boldsymbol{r}_4^p(t+\delta t) \\ \boldsymbol{r}_5^p(t+\delta t) \end{bmatrix} + \begin{bmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \end{bmatrix} \Delta \boldsymbol{r} \tag{4.26}$$

in which $\Delta \boldsymbol{r} = \boldsymbol{r}_2^c(t+\delta t) - \boldsymbol{r}_2^p(t+\delta t)$ is the difference between the predicted value $\boldsymbol{r}_2^p$ and the corrected value $\boldsymbol{r}_2^c$ and the c_i are the Gear corrector coefficients shown in table 4.1, see references [19], [23] or [22].

Order	c_0	c_1	c_2	c_3	c_4	c_5
2	0	1	1			
3	1/6	5/6	1	1/3		
4	16/90	3/4	1	1/2	1/12	
5	3/16	251/360	1	11/18	1/6	1/60

Table 4.1: Gear corrector coefficients for second-order ODEs

One can easily show that the corrected "acceleration" $\boldsymbol{r}_2^c(t+\delta t)$ is obtained by simply evaluating the equation of translational motion (4.2)

$$\begin{aligned} \boldsymbol{r}_2^c(t+\delta t) &= \boldsymbol{r}_2^p(t+\delta t) + c_2 \cdot \delta \boldsymbol{r} \\ &= \boldsymbol{r}_2^p(t+\delta t) + 1 \cdot [\boldsymbol{r}_2^c(t+\delta t) - \boldsymbol{r}_2^p(t+\delta t)] \\ &= \boldsymbol{r}_2^c(t+\delta t) \\ &= \frac{\boldsymbol{F}(t)}{m} \end{aligned} \tag{4.27}$$

Analogously the equation of rotative motion (4.15) is solved exactly the same way since it is a second-order ODE as well. Matuttis states to choose

the same integrator for the angular as for the rectilinear degrees of freedom, so that noise from one integrator cannot feed instabilities in the other integrator.

In conclusion, the sequence of the Gear predictor-corrector algorithm for second-order ODEs is

1. Predict the new values for all the time derivatives
2. Using the predicted coordinates calculate the forces and torques acting on each element
3. Calculate $\Delta \boldsymbol{r}$ and $\Delta \boldsymbol{q}$
4. Use $\Delta \boldsymbol{r}$ and $\Delta \boldsymbol{q}$ to correct the current time derivatives
5. Next time step and start from 1.

4.3.3 Stability and efficiency

Matuttis in [19] proposes, that if high precision in the positions is needed (e.g. for simulating stable heaps of particles), second-order BDF will be more accurate than the higher-order methods, which may create numerical drift, due to averaging of positions and their higher-order derivatives over several time-steps, in configurations that should be static. Therefore, relating to table 4.1 it is sufficient just to calculate the time derivatives up to second order.

Since we use an implicit integration procedure in the time domain, the program will be stable. But to get an idea how big the time steps should be for efficiency and accuracy a first reference value is given by Cundall in his pioneering work [24]

$$\delta t = \text{frac} \cdot 2 \cdot \sqrt{\frac{m_{min}}{2\,Y_{max}}}, \tag{4.28}$$

where m_{min} is the smallest mass in the domain, Y_{max} is the largest normal or shear contact stiffness in the problem and `frac` is a user-defined factor which is chosen by Cundall to be equal to 0.1. The normal stiffness is usually bigger than the shear contact stiffness and therefore Y_{max} is nothing more than the element's Young's Modulus.

Despite Cundall's implementation of a simple explicit Euler method, this formula is implemented in the current implicit calculation method with

`frac` as an input parameter from *Simulation_Settings.txt*. Later on in section 10.2 a detailed analysis of different `frac`-values and their impact on the outcome of the simulation is conducted, where it it will be shown that appropriate time step sizes for current ice simulations appears to be of order $10^{-4}\,s$.

Chapter 5

Geometry and contact detection

In all discrete element programs an element occupies a certain space and preserves a certain shape. While most studies of discrete elements use disks in two dimensions or spheres in three dimensions, it is obvious that realistic particles are more polyhedral than roundish. As shown in section 2.1, the shape of an ice element can be simplified to a cuboid shape. Therefore, in the current program, the author implements a polyhedral representation of the elements, where each element has $n_f = 8$ faces and overall $n_v = 8$ vertices. Each face itself is bounded by $n_{v,f} = 4$ vertices.

In this chapter the representation of polyhedral particles and their geometric and physical properties is discussed. The overlap detection algorithm, which is closely related to the particle geometry is also looked at.

5.1 Particle geometry

The focus in this section is, how features like vertices and faces have to be represented for an efficient computation of the overlap polyhedron.

5.1.1 Basic concepts and data structure

To represent a polyhedron one needs to specify two kinds of information: geometric and topological. The geometric information refers to

the boundary of the polyhedron, that is the surface which separates the bounded inward from the unbounded outward region. The surface of a polyhedron is composed of a set of flat faces, which are defined by their vertices and a normal vector which points outwards. An edge is the intersection of two adjacent faces and a vertex is the intersection of several edges. Additionally, information is needed to assign exactly the connectivity of vertices, edges and faces.

A vertex is a point in the three dimensional Euclidian space and it's represented by its coordinates $\boldsymbol{v} = (v_x, v_y, v_z)^T$ in Cartesian coordinates. For a polyhedron with n_v vertices the array $\texttt{vert\_coord}(1:3,\, 1:n_v)$ is used to store the coordinates of the vertices, where the indices of the first dimension represent the coordinate axes and the second dimension represents the vertex index.

For the current purpose it is convenient to characterise a face by the point normal form

$$\boldsymbol{n} \cdot \boldsymbol{p} - d = 0, \tag{5.1}$$

where $\boldsymbol{n} = (n_x, n_y, n_z)^T$ is the unit normal vector of that plane and $\boldsymbol{p} = (p_x, p_y, p_z)^T$ is an arbitrary point on the plane and d is the distance from the origin to that plane. Then $\cdot$ indicates the regular vector scalar product. Therefore, a plane can be described using the following four parameters $(n_x, n_y, n_z, -d)$. Therefore a face can be represented by a table with its vertex indices and by the coefficients of the plane equation. For n_f faces of a polyhedron the array $\texttt{face\_vertex\_table}(1:4,\, 1:n_f)$ is used to store the vertex indices for each face and $\texttt{face\_equation}(1:4,\, 1:n_f)$ stores the information for each plane equation.

Let a face be bounded by (at least) three known vertices $\boldsymbol{v}_1, \boldsymbol{v}_2, \boldsymbol{v}_3$, then the unit normal vector of this face can be computed by

$$\boldsymbol{n} = \frac{(\boldsymbol{v}_2 - \boldsymbol{v}_1) \times (\boldsymbol{v}_3 - \boldsymbol{v}_1)}{|(\boldsymbol{v}_2 - \boldsymbol{v}_1) \times (\boldsymbol{v}_3 - \boldsymbol{v}_1)|}. \tag{5.2}$$

The distance d can then be obtained by resolving equation (5.1) substituting e.g. $\boldsymbol{p} = \boldsymbol{v}_1$

$$d = \boldsymbol{n} \cdot \boldsymbol{v}_1. \tag{5.3}$$

An essential prerequisite for the success of this program is the direction of the normal vectors. They always have to point outwards. If equation

(5.1) is fulfilled for a point $\boldsymbol{p}_1$ then this point is situated on this plane. If

$$\boldsymbol{n}_i \cdot \boldsymbol{p}_1 - d_i < 0 \tag{5.4}$$

is obtained for all n_f faces, $\boldsymbol{p}_1$ is inside the polyhedron. To check if the normal vector of a face points outwards one could solve (5.4) after substituting the centre of mass of current element $\boldsymbol{r}_c$ (which is by definition an interior point). If the solution is greater than zero, the sign of the normal vector has to be flipped.

Since we want to compute the overlap of two polyhedra to determine the contact force, it's convenient to construct a `vertex_face_table`$(1 : n_{v,f_{max}}, 1 : n_v)$ array, in which for each vertex the faces on which it is located are stored. Here $n_{v,f_{max}}$ is the maximum number of faces a vertex is connected to. This table can be considered as a reverse table of `face_vertex_table`.

To summarise, the necessary information for representing a polyhedron consists of the geometric information, stored in `vert_coord`, and the topological information, stored in the `face_vertex_table`. The additional auxilliary information for further computations in `face_equation` can be derived from the two previously mentioned tables.

An example of how those arrays would appear in a simplified case, where the element is a tetrahedron (four faces and three vertices per face), is shown in the figure 5.1 and table 5.1.

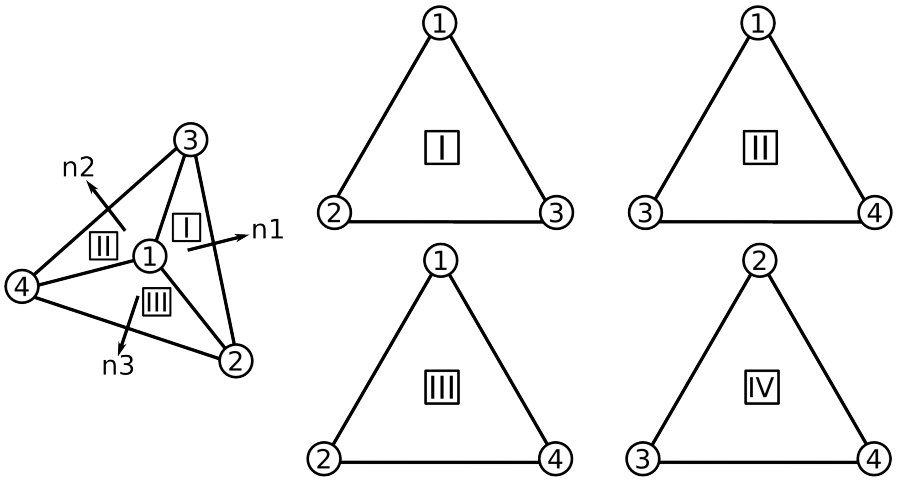


Figure 5.1: Tetrahedron

v_{1x}	v_{2x}	v_{3x}	v_{4x}
v_{1y}	v_{2y}	v_{3y}	v_{4y}
v_{1z}	v_{2z}	v_{3z}	v_{4z}

(a) `vert_coord`

n_{1x}	n_{2x}	n_{3x}	n_{4x}
n_{1y}	n_{2y}	n_{3y}	n_{4y}
n_{1z}	n_{2z}	n_{3z}	n_{4z}
d_1	d_2	d_3	d_4

(b) `face_equation`

1	1	1	2
2	3	4	4
3	4	2	3

(c) `face_vertex_table`

1	1	1	2
2	3	2	3
3	4	4	4

(d) `vertex_face_table`

Table 5.1: Array examples with reference to figure 5.1

5.1.2 Particle generation and geometry update

All elements in this program share common attributes. Each particle is a cuboid. That means it has six sides and eight vertices, and four vertices for each face. An important attribute can derived by this: All elements are convex. This means that each randomly picked point inside the element can be connected to every other inside point via a straight line, without leaving that element. This condition is necessary for a successful overlap computation.

The elements in this program can be divided into three subgroups, where the numbering of the vertices and faces is shown in figure 5.2:

1. Ice elements: This is the majority of elements in a simulation. They form the ridge and can move about freely in the domain.
2. Boundary elements: These elements are fixed in space and edge the domain. In the current program there are just three boundary elements: the left basin wall, the right basin wall and the consolidated layer (see 9.2) .
3. Structure: This element (or several elements) represents the model which is moving with a predefined velocity and direction through the domain.

Ice rubble The size of the ice rubble elements is defined by the input file *Simulation_Settings.txt*. Based on the input a stochastic distribution of the rubble length, width and thickness for every element is made by the

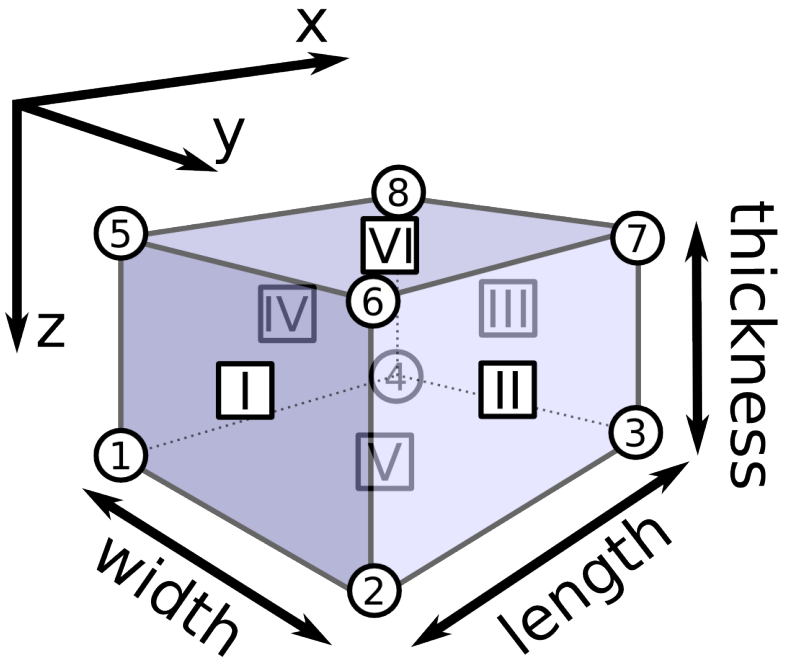


Figure 5.2: Element geometry

program to reproduce real distribution of rubble sizes, which occur in the HSVA's model basin during ridge building. According to P. Jochmann, Head of the Arctic Technology Department, a variation of 50% of the length, 10% of the width and 5% of the thickness is reasonable.

Boundary elements The placement and size of the boundary elements depend on the size of the ridge. The basin walls per default are $0.5\,m$ wide. This dimension is relatively unimportant, it just has to be big enough to ensure that ice elements aren't able to penetrate through it. The thickness of the consolidated layer is chosen to be $5\,cm$. Its size is relatively unimportant as well, since it just prevents the ridge elements to breach the water surface.

Structure The structure element represents the model, which is towed through the domain. As mentioned before, the elements have to be convex. Therefore, complex geometries, like a ship hull, have to be divided into multiple convex elements. In this program the structure is a submerged cuboid and therefore just one element is sufficient.

For each ice rubble element at its initial generation an additional array `vert_coord_body` is created in which the vertex coordinates in the body-fixed coordinate system are stored. The origin of each body-fixed system is located in the centre $\boldsymbol{r}_c$ of each element.

The geometry update of such elements, like calculating its new vertex coordinates after a new time step, is done by using the equation (4.16), which is a combination of a space-fixed component $\boldsymbol{r}_c$ and body-fixed components $\boldsymbol{p}^b$. In this case the $\boldsymbol{p}^b$ are the coordinates of the vertices stored in the array `vert_coord_body`. This formula then allows us to calculate easily `vert_coord_space`, by only using the solution of Newton's equation and the solution for the quaternion.

5.1.3 Physical properties

To represent a polyhedron as a rigid body with the Newton-Euler equations of motion, we need to know the physical properties volume, mass, center of mass and moment of inertia. These physical properties are computed during the initialisation of the particles.

Volume, mass and center of mass

Since all elements are of cuboid shape, the calculation of volume, mass and centre of mass is easily done and not explained further here. The density of model ice, which is a variable input parameter, is around $\rho_{ice} = 800\,kg/m^3$.

Differnet to the ice rubble, the mass of the boundary elements has to be set to a high value for reasons explained later in chapter 7, e.g. it is randomly set to $mass_{boundary} = 10^9\,kg$.

Moment of inertia

The moment of inertia has to be calculated only for the rubble elements using the standard formula for cuboid elements with their centre of mass located in the centre of the volume

$$I = \begin{bmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{bmatrix} = \begin{bmatrix} \frac{m}{12}(w^2+t^2) & 0 & 0 \\ 0 & \frac{m}{12}(t^2+l^2) & 0 \\ 0 & 0 & \frac{m}{12}(l^2+w^2) \end{bmatrix}, \quad (5.5)$$

where m is the mass, l the length, w the width and t the thickness of that element.

5.2 Contact detection

The contact detection algorithm checks, if there is a possible overlap between two elements. The simplest algorithm would calculate for a element if there is an overlap with every other element in the domain. Since this procedure is very time consuming (it is proportional to n^2, where n is the total number of elements) and would result for most of the tests in no overlap, some effort is put into the developing of a fast contact algorithm. A contact algorithm usually consists of three phases:

1. **Broadphase** Here the collisions which might happen are detected.
2. **Midphase** The mid-phase can be seen as a level 2 broad -phase for complex geometries, where a refinement of the contact list is done.

3. **Narrowphase** In the narrow-phase the actual collision detection is computed, based on the list of potential contact pairs created during the two previous phases

Promising basic approaches for the broad-phase can be found in physics engines developed for computer games, where the contacts of plenty of elements can be detected in realtime. One of the most popular algorithms is the so -called *sweep and prune* algorithm, which is explained well by Terdiman in [25]. Nearly all broad phase systems use the axis aligned bounding boxes (aabb) of the objects as their basis. An aabb is the box, whose edges are aligned with the global coordinate axes and its minimum and maximum values correspond to the minimum and maximum values of the object it surrounds. An example of overlapping aabbs in two dimensions is shown in figure 5.3. One can see, that even if there is an overlap of the bounding boxes, it is not given, that the elements themselves overlap.

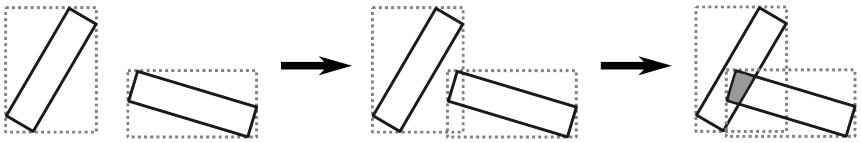


Figure 5.3: Principle of aabb overlap

The computation of such bounding boxes is fast and the checking for an intersection of two boxes is also done in a few lines of code. The key to a quick check if two bounding boxes start or stop to overlap, is to implement a suitable sorting algorithm. For this purpose some important points have to be considered. Each bounding box has a minimum and a maximum value in each dimension (see figure 5.4).

For each dimension a list, called `endPoints`, is set up, in which the minimum and maximum values of all elements are stored in sorted order. After each time step a sorting algorithm is applied to the current lists, and every time two neighbouring entries swap their places, there is possibly a beginning or an end of the overlap of two bounding boxes. The check whether two bounding boxes overlap in all three dimensions is performed immediately after the recognition of the position swap in one dimension. Therefore, an additional array in which the contact pairs are listed, has to be maintained.

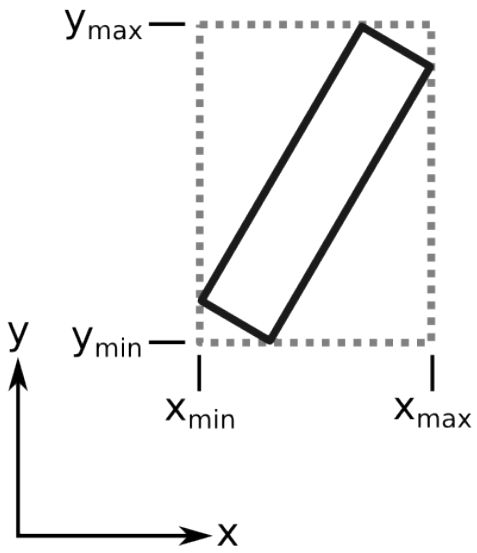


Figure 5.4: Bounding box

Before a detailed explanation of how sweep and prune work is given, it is necessary to introduce the used arrays and how they are connected to each other. In this program the Fortran feature of so-called *derived types* is implemented.

AABB The axis aligned bounding boxes have their own derived type, called `aabb`. An `aabb` has two accessible variables, namely `mMin(3)` and `mMax(3)`, which represent its minimum and maximum value in three dimensions, and which are updated right after the element itself has been updated. The array `boundingBoxes` is an array of type `aabb` with one entry for every AABB in the domain.

endPoint An `endPoint` is strongly connected to one bounding box, its "parent", and it has two accessible variables, which are `mData` and `mValue`. `mData` is an integer with two lots of information: 1) the absolute value of `mData` represents the ID of its bounding box and 2) its sign indicates, if this `endPoints` is a minimum or a maximum of that bounding box. The variable `mValue` is a pointer, which points to `mMin` respectively to `mMax` of its "parent". There are three `endPoint`-arrays in the program, called `xEndPoints`, `yEndPoints` and `zEndPoints`, each for one dimension in the

three-dimensional space. The size of each array is twice the number of elements or more specifically number of bounding boxes, because each AABB has two `endPoint`.

The correlation between two bounding boxes and their representation in the sorted `endPoint`-arrays is shown in figure 5.5, where

$$\texttt{boundingBoxes} = \begin{bmatrix} \texttt{mMin} = (x_1, y_2)^T & \texttt{mMin} = (x_3, y_1)^T \\ \texttt{mMax} = (x_2, y_4)^T & \texttt{mMax} = (x_4, y_2)^T \end{bmatrix}$$

$$\texttt{xEndPoints} = \begin{bmatrix} \texttt{mData} = -1 & \texttt{mData} = 1 & \texttt{mData} = -2 & \texttt{mData} = 2 \\ \texttt{mValue} = x_1 & \texttt{mValue} = x_2 & \texttt{mValue} = x_3 & \texttt{mValue} = x_4 \end{bmatrix}$$

$$\texttt{yEndPoints} = \begin{bmatrix} \texttt{mData} = -2 & \texttt{mData} = -1 & \texttt{mData} = 2 & \texttt{mData} = 1 \\ \texttt{mValue} = y_1 & \texttt{mValue} = y_2 & \texttt{mValue} = y_3 & \texttt{mValue} = y_4 \end{bmatrix}$$

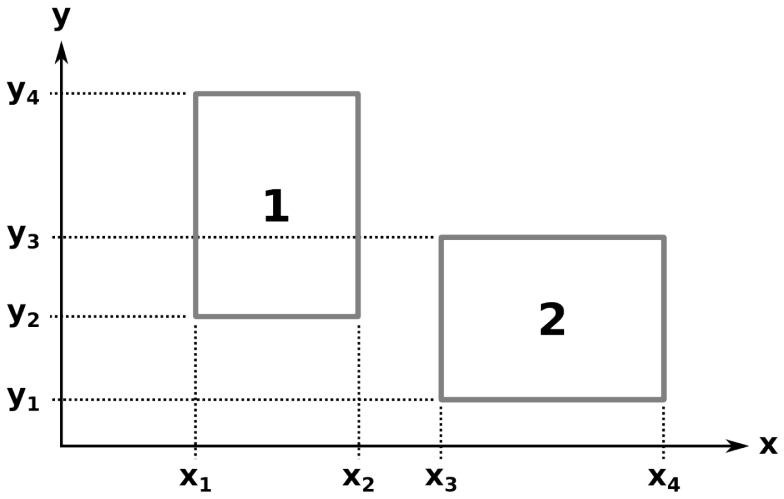


Figure 5.5: endPoints

The subroutines used for this neighbourhood algorithm are all located in the file *mod_Neighbour.f08*. The main subroutine `CalculateNeighbours` is called by the main program after each time step. `CalculateNeighbours` itself calls the subroutine `SweepAndPrune` three times; each time with another `endPoints`-list. If `SweepAndPrune` detects a swap of two list entries and the swapper is the maximum of its bounding box plus the other entry is the minumum of its bounding box, it immediately calls `AABBoverlap`, which checks an overlap of the two corresponding bounding boxes in all

three dimensions. If the overlap check is positive then the next clause checks, if the element IDs are already stored in `overlapPairs`. This can happen if this pair is already known from a previous `SweepAndPrune`-check with another coordinate axis. If this is not the case, a new `overlapPair` is found.

The ending of an aabb overlap is found under the condition that the swapper is a minimum and the other value a maximum of their bounding box. Then the current pair is deleted from `overlapPairs`.

The sweep and prune procedure is shown below.

Procedure SweepAndPrune

Input: `endPoints`-array
Output: List `overlapPairs` of overlapping bounding boxes
for *i = 2 to length(endPoints)* **do**
    j = i - 1;
    `tmp = endPoints(i);`
    while *j >= 1 and endPoints(j) > tmp* **do**
        `swapper = endPoints(j);`
        if *swapper is max(AABB(j-1)) and tmp is min(AABB(j))* **then**
            Check for three-dimensional overlap of these two AABBs;
            if *they overlap and are not already registered* **then**
                Create new entry in `overlapPairs`;
            end
        else if *swapper is min(AABB(j-1)) and tmp is max(AABB(j))* **then**
            Remove that pair from `overlapPairs`;
        else
            do nothing;
        end
        `endPoints(j+1) = endPoints(j);`
        j = j - 1;
    end
    `endPoints(j+1) = tmp;`
end

This is the end of the broadphase. The midphase is skipped and the detailed overlap computation for two element pairs in the narrowphase is explained in the next chapter.

Chapter 6

Overlap computation

In this DEM simulation of polyhedral particles, the full overlap geometry with its volume, center of volume and the contact area of two colliding particles is needed. This is computationally different, more complex and needs more effort than approaches where just the penetration depth is evaluated to compute the contact force. Other approaches even don't need penetration information. Puntigliano in [16] for example just adds a repulsive force to two close enough elements, depending on their velocities and added mass matrices.

An overlap polyhedron itself consists of vertices, faces of irregular shape for which the scope of the implemented subroutines is

- to calculate the vertices of the overlap polyhedron, resulting from face-face intersections and resulting from penetrated vertices of one element into the other,
- to match these vertices to faces of the overlap volume,
- to characterize the faces with the earlier discussed point-normal representation.

6.1 Face intersection computation

To compute the majority of the vertices of the overlap polyhedron between two elements $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$, the intersections of each face of $\boldsymbol{P}_1$ with each face of $\boldsymbol{P}_2$ have to be analysed.

6.1.1 Intersection algorithm using the point-normal form

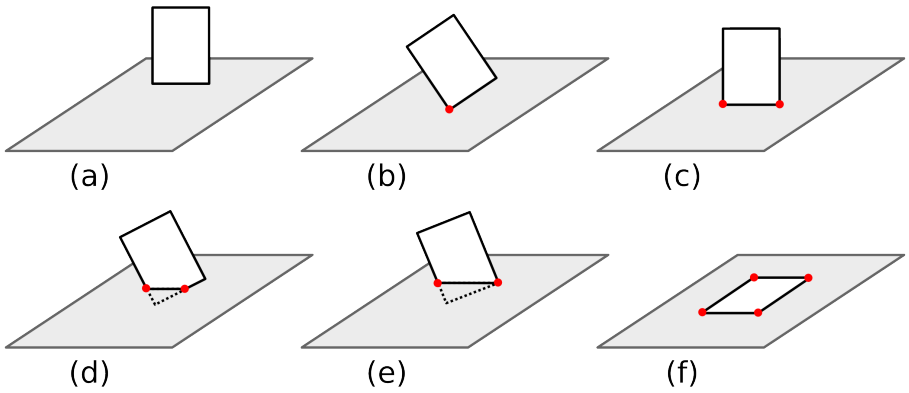


Figure 6.1: Possible face-plane intersection points

To explain the following algorithm, first the face-plane intersection computation has to be looked at. The possible relative positions of a face and a plane are shown in figure 6.1, where the red dots indicate the intersection points. The six possible cases are

(a) Separation: no intersection

(b) Touching: one intersection point which is a vertex

(c) Colinear: two intersection point, of which both are vertices

(d) Intersection: two intersection points of which one is a vertex

(e) Intersection: two intersection points of which none is a vertex

(f) Coplanar: three intersection points of which all are vertices

The cases (b), (c) and (f) are special cases, since they could simply be determined by a point on a plane test, but nevertheless they are detected by the following alogorithm.

In chapter 5 the point-normal form of a plane was introduced

$$\boldsymbol{n} \cdot \boldsymbol{r} - d = 0.. \tag{6.1}$$

Additionally, the point-parameter form of a line, which passes through two vertices v_1 and v_2, which can be written as

$$\boldsymbol{r} = \boldsymbol{v}_1 + \lambda \cdot \overline{v_1 v_2}, \tag{6.2}$$

where $\overline{v_1v_2} = v_2 - v_1$ and the scalar parameter λ determines the position of a point on the line. Substituting equation (6.2) into equation (6.1) and solving for the parameter λ an expression for the intersection is obtained

$$\lambda = \frac{d - \boldsymbol{n} \cdot v_1}{\boldsymbol{n} \cdot \overline{v_1v_2}}, \tag{6.3}$$

when $\boldsymbol{n} \cdot \overline{v_1v_2} \neq 0$.

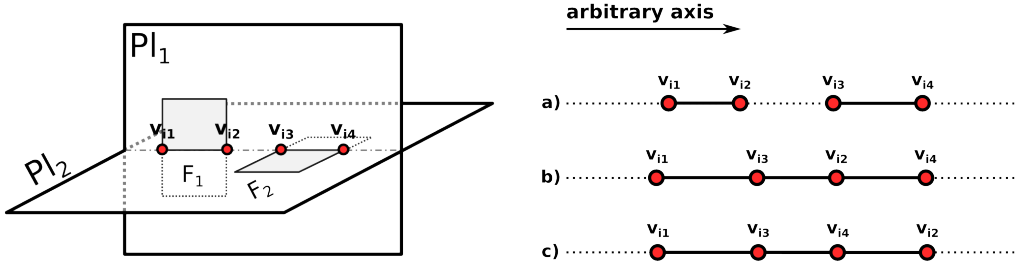
Now λ indicates if

- there is an intersection for $0 < \lambda < 1$,
- v_1 or v_2 lay on the plane for $\lambda = 0$, respectiveley $\lambda = 1$,
- there is no intersection for $\lambda < 0$ and $\lambda > 1$,
- or there is no solution at all if $\overline{v_1v_2}$ is parallel to the plane, as shown in (c) or (f).

For a rectangular face with four vertices v_1, v_2, v_3 and v_4 and a given plane we now can obtain the λ for all edges $\overline{v_1v_2}$, $\overline{v_2v_3}$, $\overline{v_3v_4}$ and $\overline{v_4v_1}$ and apply the criteria above to check for intersection points.

6.1.2 Intersection of two faces

After performing the face-plane intersection of face F_1 and plane P_2 and there are intersection points found (let them be v_{i1} and v_{i2}) the next step is to check for intersection points of face F_2 and plane P_1, noted as v_{i3} and v_{i4}. In figure 6.2a it is shown, how this constellation could look like.



(a) Arbitrary face-face intersection (b) Intersection patterns

Figure 6.2: Face-face intersection

To determine whether these two faces intersect, choose an arbitrary global coordinate axis, where all vertices don't have the same value. Then sort $v_{i1}v_{i2}$ and $v_{i3}v_{i4}$ in ascending order according to their projections on that chosen axis and swap their values if necessary, so that $v_{i1}v_{i2}$, respectiveley $v_{i3}v_{i4}$ is the correct order. Lastly sort all vertices along this axis. Every outcome of this sorting has the pattern shown in figure 6.2b, where pattern b) and c) represent an intersection of the two faces.

In procedure FindFaceIntersection the basic approach for finding two intersection points from a face-face intersection using the equation (6.3) is shown.

Procedure FindFaceIntersection
Input: $\boldsymbol{F}_i$ of $\boldsymbol{P}_k$ and $\boldsymbol{F}_j$ of $\boldsymbol{P}_l$
Output: Intersection points v_{int1} and v_{int2}
Compute the intersection points for $\boldsymbol{F}_i$ and $\boldsymbol{Pl}_j$ using equation (6.3);
if *two intersection points* v_{i1} *and* v_{i2} *are found* **then**
  Compute the intersection points for $\boldsymbol{F}_j$ and $\boldsymbol{Pl}_i$ using equation (6.3);
  if *two intersection points* v_{i3} *and* v_{i4} *are found* **then**
    Find a non-zero corrdiante axis for v_{i1}, v_{i2}, v_{i3} and v_{i4};
    Sort v_{i1} and v_{i2} in ascending order;
    Sort v_{i3} and v_{i4} in ascending order;
    if $max(v_{i1}, v_{i3}) < min(v_{i2}, v_{i4})$ **then**
      $[v_{int1}, v_{int2}] = [max(v_{i1}, v_{i3}), min(v_{i2}, v_{i4})]$
    end
  end
end

6.2 Vertex computation

For vertex computation one first hast to distinguish between *inherited* and *generated* vertices of an overlap polyhedron. By *inherited* vertices these vertices are meant, which penetrate from one element into the other element. *Generated* vertices are those vertices, which are formed during the face-face intersections. For clarification a simple two-dimensional drawing of two colliding elements can be seen in figure 6.3.

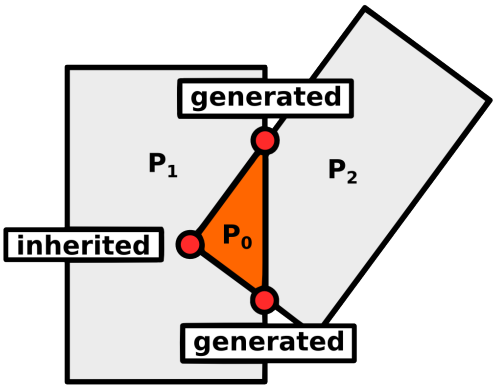


Figure 6.3: Difference between inherited and generated vertices

In this section it is discussed how to compute the coordinates of the inherited and generated vertices of an overlap polyhedron $\boldsymbol{P}_0$.

6.2.1 Inherited vertices

The oriented distance of a point $\boldsymbol{p}$ to a plane represented in the point-normal form is

$$d' = \boldsymbol{n} \cdot \boldsymbol{p} - d. \tag{6.4}$$

The sign of the distance indicates whether the point $\boldsymbol{p}$ is situated on the side of the plane in which the normal vector points or not. Remembering that the definiton of the plane equations for each face requires the normal vector to point outwards, one now can develop an algorithm, which checks if a point lays behind every face of the other element. This is

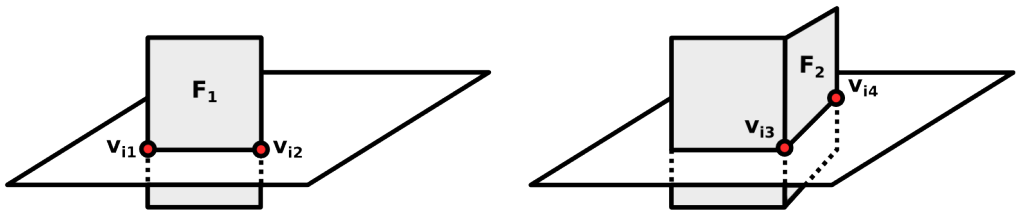
shown in procedure FindInheritedVertices.

Procedure FindInheritedVertices
Input: Colliding polyhedrons $\boldsymbol{P}_x$ and $\boldsymbol{P}_y$
Output: Inherited vertices of $\boldsymbol{P}_x$ inside of $\boldsymbol{P}_y$, stored in `inherited_vert`
forall the *vertices* $\boldsymbol{v}_i$ *of* $\boldsymbol{P}_x$ **do**
  forall the *faces* $\boldsymbol{F}_k$ *of* $\boldsymbol{P}_y$ **do**
    if $\boldsymbol{n}_k \cdot \boldsymbol{v}_i - d_k > -\epsilon$ **then**
      cycle outer loop, since $\boldsymbol{v}_i$ is outside $\boldsymbol{F}_k$;
    end
  end
  Add $\boldsymbol{v}_i$ to `inherited_vert`;
end

This algorithm has to be run twice with swapped input elements, so that first the inherited vertices from $\boldsymbol{P}_1$ are computed and then analogously the inherited vertices from $\boldsymbol{P}_2$. A point whose distance to this plane is between the chosen pecision $\epsilon = 10^{-14}\,m$ will be regarded as on the plane. This point is of no interest, since it will later on be detected as an generated vertex. As a result of this part a list `inherited_vert` is created, which stores the coordinates of each vertex and the index of its parent element ($\boldsymbol{P}_1$ or $\boldsymbol{P}_2$).

6.2.2 Generated vertices

To obtain all generated vertices of $\boldsymbol{P}_0$ the procedure FindFaceIntersection has to be run for every face of $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$. However, in picture 6.4 it is shown how every generated vertex will be recorded twice in the loop to compute the face intersections of two polyhedra. In this example the recorded vertices $\boldsymbol{v}_{i2}$ and $\boldsymbol{v}_{i3}$ are the same, therefore, one of those has to be eliminated afterwards.



(a) Intersection points for face 1 (b) Intersection points for face 2

Figure 6.4: Double computation of the same vertex

The whole subroutine to detect all generated vertices during a collision of two elements, using the introduced procedure FindFaceIntersection, is

shown in the procedure ComputeGeneratedVertices below.

Procedure ComputeGeneratedVertices

Input: Colliding polyhedrons $\boldsymbol{P}_x$ and $\boldsymbol{P}_y$
Output: A list of all generated vertices (`vert_gen`) and a list with all intersection point pairs, which form the edges of the overlap polyhedron (`intersect_point_pair_idx`)

forall the *Faces* $\boldsymbol{F}_{1i}$ *of* $\boldsymbol{P}_1$ **do**
  forall the *Faces* $\boldsymbol{F}_{2k}$ *of* $\boldsymbol{P}_2$ **do**
    call `FindFaceIntersection`($\boldsymbol{F}_{1i}$, $\boldsymbol{F}_{2k}$) (see algorithm FindFaceIntersection);
    if *two intersection points* $\boldsymbol{v}_{int1}$ *and* $\boldsymbol{v}_{int2}$ *are found* **then**
      `num_int_pair` = `num_int_pair` + 1;
      `contact_face_pair`(1:2, `num_int_pair`) = ($\boldsymbol{F}_{1i}$, $\boldsymbol{F}_{2k}$);
      `intersect_point_pair`(1:2, `num_int_pair`) = ($\boldsymbol{v}_{int1}$, $\boldsymbol{v}_{int2}$);
    end
  end
end
`vert_gen(1:2)` = `intersect_point_pair(1:2, 1)`;
`intersect_point_pair_idx(1:2, 1)` = [1, 2];
set `vert_idx` = 2, to initialize the number of generated vertices;
for $i = 2$ *to num_int_pair* **do**
  for $j = 1$ *to* 2 **do**
    $\boldsymbol{v}_{check}$ = `intersect_point_pair(j, i)`;
    if $\boldsymbol{v}_{check}$ *is not in vert_gen* **then**
      `vert_idx` = `vert_idx` + 1;
      `vert_gen(vert_idx)` = $\boldsymbol{v}_{check}$;
      `intersect_point_pair_idx(j, i)` = `vert_idx`;
    else
      Save index of $\boldsymbol{v}_{check}$ in `vert_gen` into `intersect_point_pair_idx(j, i)`;
    end
  end
end

6.3 Face and contact area determination

When all vertices of the overlap polyhedron $\boldsymbol{P}_0$ have been computed, the next step is to clarify their topological relations to obtain the faces of $\boldsymbol{P}_0$. As soon as these information are available, the computations of the `overlap_volume`, `overlap_centroid`, `contact_area` and the `force_direction` as the basis for the force model (see chapter 7) can be executed.

6.3.1 Face determination

Similar to the inherited and generated vertices the faces of the overlap polyhedron also have to be differentiated between inherited and generated faces. Inherited faces are those faces whose vertices are all inherited vertices. The generated faces are part of the original faces of $\boldsymbol{P}_1$ and $\boldsymbol{P}_2$, bounded by generated and inherited vertices. Different to generated vertices, the generated faces are not something totally new, because their face equations for example are already known from the original polyhedrons faces.

The byproduct `contact_face_pair` from the computation of the generated vertices can now be used to determine the generated faces of the overlap polyhedron $\boldsymbol{P}_0$, since a part of every intersection face automatically is a face of $\boldsymbol{P}_0$. Additionally the corresponding entries in the array `intersect_point_pair_idx` give the vertices, which form an edge of that new face. The subroutine FindGeneratedFaces clarifies how the compu-

tation of generated faces and the detection of their vertices is done.

Procedure FindGeneratedFaces
Input: Colliding polyhedrons $\boldsymbol{P}_x$ and $\boldsymbol{P}_y$
Output: A list of all generated faces and their corresponding vertices `overlap_face_verts` and the face equations for each face `overlap_face_equation`
forall the *the faces* $\boldsymbol{F}_i$ *of* $\boldsymbol{P}_x$ **do**
  if $\boldsymbol{F}_i$ *is intersecting with* $\boldsymbol{P}_y$ **then**
    Register $\boldsymbol{F}_i$ as a face of $\boldsymbol{P}_0 \rightarrow \boldsymbol{F}_k$;
    Find the array index of $\boldsymbol{F}_i$ in `contact_face_pair`;
    Register all the vertex indices from the corresponding entries in `intersect_point_pair_idx` for the new face $\boldsymbol{F}_k$;
  end
end

For finding the inherited faces the array `vertx_face_table` comes into use. In this table for every vertex the faces on which it is located are listed. Therefore, every face, which is assigned to an inherited vertex $\boldsymbol{v}_i^{inh}$, is also a face of the overlap polyhedron. The procedure for the determination of the inherited faces is shown below (see subroutine FindInheritedFaces).

Procedure FindInheritedFaces
Input: Colliding polyhedrons $\boldsymbol{P}_x$ and $\boldsymbol{P}_y$
Output: A completed list of all faces of and their corresponding vertices `overlap_face_verts` and the face equations for each face `overlap_face_equation`
forall the *the inherited vertices* $\boldsymbol{v}_i^{inh}$ *from* $\boldsymbol{P}_x$ **do**
  forall the *the faces* $\boldsymbol{F}_i$ *in vertex_face_table of* $\boldsymbol{v}_i^{inh}$ **do**
    if $\boldsymbol{F}_i$ *is already known as* $\boldsymbol{F}_k$ *in the list of faces of* $\boldsymbol{P}_0$ **then**
      Add $\boldsymbol{v}_i^{inh}$ to `overlap_face_verts` for face $\boldsymbol{F}_k$;
    else
      Register $\boldsymbol{F}_i$ as a new entry in the face list of $\boldsymbol{P}_0$;
      Add $\boldsymbol{v}_i^{inh}$ to `overlap_face_verts` for face $\boldsymbol{F}_i$;
    end
  end
end

These two algorithms are merged with the subroutine `FindFaces`, since many of the used variables are used by both algorithms. This merging reduces the overhead during the paralelisation a bit (see chapter 8). This

new subroutine then has to be run twice, for $\boldsymbol{P}_1$ and then for $\boldsymbol{P}_2$, since it only detects the faces and vertices resulting from one of the two colliding elements.

Though all elements are of cuboid shape the overlap polyhedron is not necessary a cuboid and neither is it a given that each face has four vertices. Since we are interested in the computation of the volume and centroid of that polyhedron a triangulation of that new body has to be performed. In order to make a triangulation the vertices of each generated face have to be brought into order to identify neighbouring vertices.

A simple sorting algorithm uses an edge to determine the relative orientations of the other vertices and sorts them counterclockwise. To start with the sorting we can use the first two registered vertices, $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, for a face, since these two vertices where acquired from the `intersect_point_pair`-array and therefore form definetively an edge. The vector connecting $\boldsymbol{v}_1$ with $\boldsymbol{v}_2$ is

$$\boldsymbol{v}_b = \frac{\boldsymbol{v}_2 - \boldsymbol{v}_1}{|\boldsymbol{v}_2 - \boldsymbol{v}_1|}. \tag{6.5}$$

Then the angles between $\boldsymbol{v}_b$ and the vectors connecting $\boldsymbol{v}_1$ with $\boldsymbol{v}_i$, where $i = 3, \ldots, k$ and k is the number of vertices for that specific face, can be computated via

$$\cos(\theta_i) = \frac{(\boldsymbol{v}_i - \boldsymbol{v}_1) \cdot \boldsymbol{v}_b}{|\boldsymbol{v}_i - \boldsymbol{v}_1|}. \tag{6.6}$$

Depending on these values in ascending order the corresponding vertices are sorted, which is illustrated in figure 6.5. Please note that this procedure is just applicable for angles below 180°, since the consinus above 180° again grows. Therefore this sorting just works for convex faces, which only result from an overlap of convex elements, since the overlap volume of convex elements is convex itself.

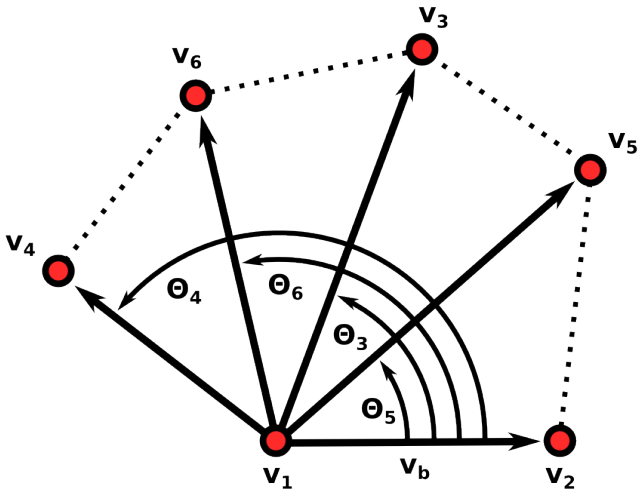


Figure 6.5: Sorting procedure of the vertices of a face

Now, knowing all the faces of the overlap polyhedron and having sorted the vertices, the volume and the centre of mass via triangulation can be calculated. The volume of a tetrahedron can be calculated as follows

$$V' = \frac{1}{6}|[(v_1 - v_c) \times (v_2 - v_c)] \cdot (v_3 - v_c)|, \tag{6.7}$$

where v_1, v_2 and v_3 are the three vertices of the triangulated face and v_c is the origin, which has to be inside the polyhedron and which is going to be the reference point for all sub -pyramids of that overlap polyhedron. The total volume V of a polyhedron is the sum over its sub-pyramids

$$V = \sum V_i'. \tag{6.8}$$

The centroid of a tetrahedron is located at

$$c' = \left(\frac{x_1 + x_2 + x_3 + x_c}{4}, \frac{y_1 + y_2 + y_3 + y_c}{4}, \frac{z_1 + z_2 + z_3 + z_c}{4} \right)^T, \tag{6.9}$$

where the terms of the sums are the coordinate components of the four vertices in the three-dimensional space. Knowing the centroid of each subpyramid and its volume, the centroid of the full overlap polyhedron, the `overlap_centroid` c_0, can be calculated

$$c_0 = \frac{\sum c_i' V_i'}{V}. \tag{6.10}$$

6.3.2 Contact area and contact normal determination

The last computation in the overlap computation is the determination of the contact area and hence the force direction. Therefor the two arrays `intersect_point_pair` and `intersect_point_pair_idx`, which were calculated in the subroutine ComputeGeneratedVertices, are needed. Each generated vertex appears twice in `intersect_point_pair`, since it is connected to two line segments. The contact area is an area where all generated vertices are connected in such a way that they form a closed `contact_line`, which is the border of the contact area. Every vertex of this contact line is then connected to the `overlap_centroid` and a set of triangles is obtained, for which the normal vectors are calculated.

The normal vector for the i-th triangle with the vertices $\boldsymbol{c}_0$, $\boldsymbol{v}_{i1}$ and $\boldsymbol{v}_{i1}$ is calculated as

$$\boldsymbol{n}_i = 0.5 \cdot (\boldsymbol{v}_{i1} - \boldsymbol{c}_0) \times (\boldsymbol{v}_{i2} - \boldsymbol{c}_0). \tag{6.11}$$

Note that this is a normal vector, whose absolute value additionally represents the area of that triangle ($|\boldsymbol{n}_1| = A_i$), which comes in handy for the calculation of the total area. The sign of the normal directions has to point to one of the intersection polyhedra. Here it is chosen to point towards the centroid of $\boldsymbol{P}_1$. Therefore, if

$$\boldsymbol{n}_i \cdot (\boldsymbol{c}_{P_1} - \boldsymbol{c}_0) < 0, \tag{6.12}$$

then

$$\boldsymbol{n}_i = -\boldsymbol{n}_i. \tag{6.13}$$

The `force_direction` then is a weighted vector resulting from all normal vectors multiplied by the area of their face

$$\boldsymbol{n}_c = \frac{\sum_k A_i \boldsymbol{n}_i}{|\sum_k A_i \boldsymbol{n}_i|}. \tag{6.14}$$

An example is illustrated in figures 6.6 and 6.7.

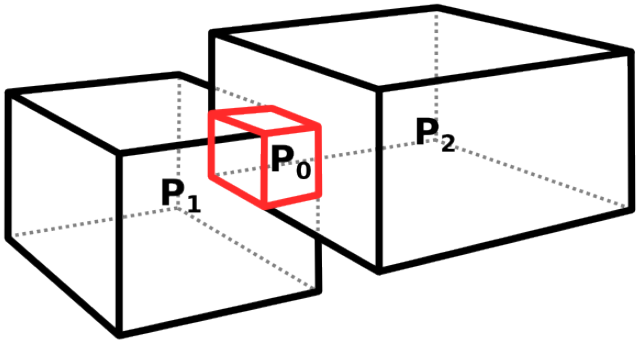
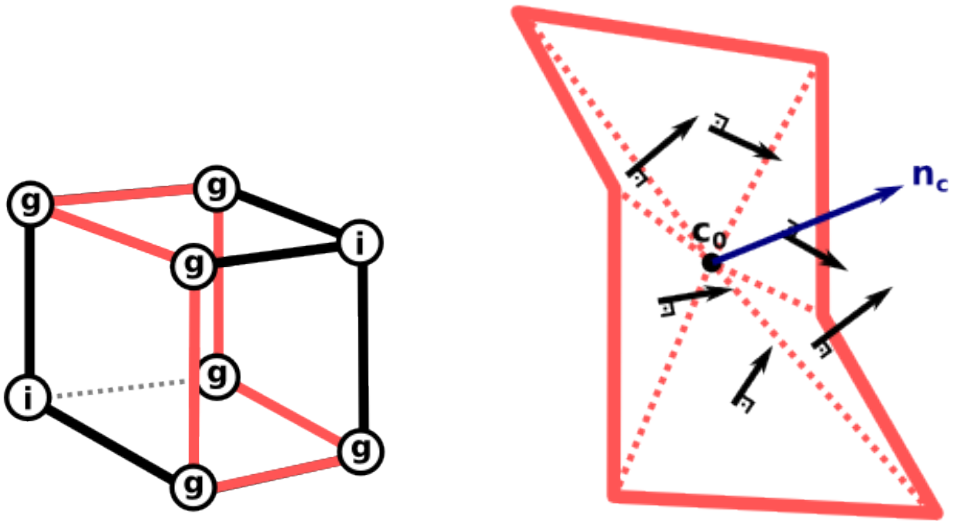


Figure 6.6: Overlap of two elements



(a) (g)enerated and (i)nherited vertices (b) Calculation of `force_direction`

Figure 6.7: Detail of overlap polyhedron and contact line

The pseudo code of the subroutine ComputeContactArea is shown below.

Procedure ComputeContactArea

Input: `Intersect_point_pair_idx` and `vert_gen`
Output: The `overlap_area` in $[m^2]$, the 3d-vector `force_direction` and the overlap centroid `overlap_centroid`
Register the first point pair of `intersect_point_pair_idx(1:2, 1)` into `contact_line(1:2)`;
Clear this entry $\rightarrow$ `intersect_point_pair_idx(1:2, 1)` = 0;
$\boldsymbol{v}_{end}$ = `contact_line(2)`;
for *i = 2 to num_int_pair* **do**
    Find $\boldsymbol{v}_{end}$ in the k-th pair in `intersect_point_pair_idx`;
    Add the other vertex $\boldsymbol{v}_{partner}$ to `contact_line(i+1)`;
    $\boldsymbol{v}_{end} = \boldsymbol{v}_{partner}$;
    Clear the k-th entry $\rightarrow$ `intersect_point_pair_idx(1:2, k)` = 0;
end
for *i = 1 to num_int_pair* **do**
    Calculate triangle area A_i using `contact_line(i)`, `contact_line(i+1)` and $\boldsymbol{c}_0$;
    Calculate triangle normal $\boldsymbol{n}_i$ using `contact_line(i)`, `contact_line(i+1)` and $\boldsymbol{c}_0$;
    Make sure $\boldsymbol{n}_i$ points towards centroid of polhyhedron $\boldsymbol{P}_1$;
end
Calculate total `overlap_area` = $\sum_i A_i$;
Calculate resulting `force_direction` using an area-weighted summation of all triangle normals $\boldsymbol{n}_i$;

It is assumed that the penetrations are small relative to the element sizes, the result of which is, that one element doesn't pierce through another element and hence just one contact area is established.

Chapter 7

Force modeling

7.1 Force directions

While the degrees of freedom for granular particles are given by the equations of the classical mechanics (see chapter 4) various approaches for the modelling of the forces between two particles can be found in the literature already touched upon in chapter 3. Three basic properties for the modelling of the forces in this DEM code have to be known:

1. The magnitude of the force,
2. The force direction and
3. The action point to define the torques.

Additionally a smooth evolution of the forces and torques during the collision is required to prevent instabilities during the solving of the equations of motion.

7.2 The normal and tangential direction and force point

For the normal force components the "deformation" and the collision speed are taken into account, while sliding of the two colliding elements

leads to Coulomb friction. Therefore, a mathematical definition of the normal and tangential directions is necessary (see also figure 7.1).

The direction of the normal force is identical to the normal vector $\boldsymbol{n}_c$ calculated in the subroutine ComputeContactArea. Along this direction all normal forces act on the centroid of the overlap polyhedron.

The orthogonal plane to the normal vector is the basis for the tangential force components whose exact directions depend on the relative collision velocity and relative displacement.

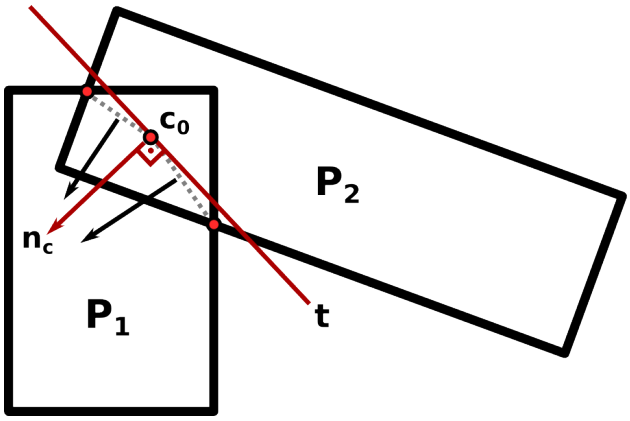


Figure 7.1: Force directions

Translational and rotational excitation The resulting force vector (normal + tangential component) acting on the centroid of the colliding elements to obtain translational movement and a component, which leads to torques

$$\tau = \boldsymbol{f} \times r_{i/j}, \tag{7.1}$$

where $r_{i/j}$ is the vector connecting the overlap centroid with the centroid of the intersecting elements (figure 7.2). The contact velocity and its direction is then obtained by

$$\boldsymbol{v}_c = (\boldsymbol{v}_i + \boldsymbol{\omega}_i \times r_i) - (\boldsymbol{v}_j + \boldsymbol{\omega}_j \times r_j). \tag{7.2}$$

7.3 Modeling of the normal force

7.3.1 Elastic force

The elastic force is a repulsive force, which is adapted from elasticity models by Hook and Hertz. The assumption for this model is, that there is an elastic deformation of the particles during collision and the magnitude of this force is proportional to the hypothetical deformation of both elements.

Since linear elastic behaviour is demanded to apply Hook's law, the Young's modulus Y of the elements can serve as a spring constant in the force law. The corresponding dimension to the spring displacement is here the volume of the overlap polyhedron. And while the interaction should locally reproduce the contact laws depending on the shape of the particles, the units have also to be taken into account. The unit of the Young's modulus is $[N/m^2]$ multiplied by the overlap volume, measured in $[m^3]$, there is one length L in $[m]$ still missing.

The sound velocity in bulk solids is a physical property which is used to find this length factor. Sound waves in space-filling packings in DEM simulations should be independent from the sizes of the particles, while depending only on the density and the Young's modulus of that bulk.

To make the sound velocity independent the *characteristic length* L_c for two intersecting particles i and j is introduced. For element i and respectively j defining two radii r_i and r_i (see also figure 7.2), which are given by connecting the overlap centroid and the centroid of the overlapping elements

$$\boldsymbol{r}_{i/j} = \boldsymbol{c}_o - \boldsymbol{c}_{i/j} \tag{7.3}$$

the characteristic length can be expressed as

$$L_c = 4\frac{|\boldsymbol{r}_i|\,|\boldsymbol{r}_j|}{|\boldsymbol{r}_i| + |\boldsymbol{r}_j|}. \tag{7.4}$$

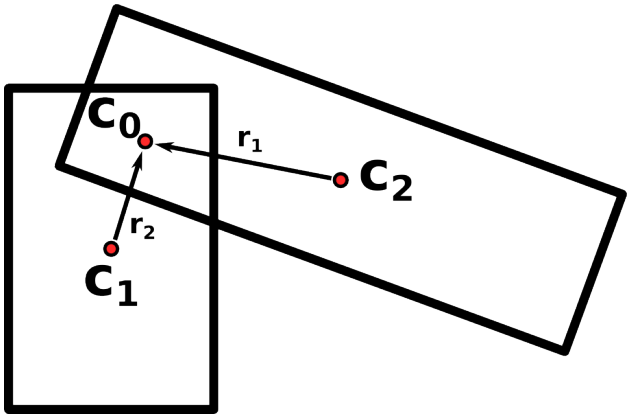


Figure 7.2: Radii of both intersecting elements

The characteristic length completes the elastic force component, which results to

$$|\boldsymbol{f}_{el}| = Y \cdot V_o / L_c, \tag{7.5}$$

where Y is the mean Young's modulus of element i, V_o the volume of the overlap polyhedron and L_c the characteristic length.

7.3.2 Damping force

In analogy to the harmonic oscillator

$$\ddot{x} = -kx - \gamma\dot{x}, \tag{7.6}$$

where $-kx$ is the elastic spring force and $-\gamma\dot{x}$ is a dissipative force, the damping force in this code is modelled to be proportional to the overlap volume change $dV/\delta t$. To achieve a correct unit for the resulting force, and keeping in mind, that the dissipation has to be equivalent for both colliding and potentially different masses, the damping force in normal direction is chosen to be

$$f_d^n = \gamma^n \sqrt{\frac{Y\, M_{red}}{L_c^3}}\, \frac{\delta V_o}{\delta t}, \tag{7.7}$$

with γ^n as a dimensionless damping coefficient for normal direction and M_{red} as the reduced mass

$$M_{red} = \frac{m_i\, m_j}{m_i + m_j}, \tag{7.8}$$

where m_i and m_j are the masses of element $\boldsymbol{P}_i$ and $\boldsymbol{P}_j$ respectively. Since δV_o can be positive and negative the damping force f_d^n is a signed quantity which has to be treated carefully. This force would add an pulling component to the sum of all normal forces during separation. For fast separations the damping force could even be greater than the elastic component, which would lead to unphysical attractive force. Therefore, the following cut-off condition is implemented

$$\text{if } f_e + f_d^n < 0 \quad \Rightarrow \quad f^n = -f_e, \tag{7.9}$$

Then, for a contact pair of particles $\boldsymbol{P}_i$ and $\boldsymbol{P}_j$ with the calculated force direction $\boldsymbol{n}_c$ the total normal force acting on element $\boldsymbol{P}_i$ is computed by

$$\boldsymbol{f}^n = (f_e + f_d^n) \cdot \boldsymbol{n}_c. \tag{7.10}$$

The third of Newton's laws states that *actio = reactio*, which easily leads to the force acting on element $\boldsymbol{P}_j \Rightarrow -\boldsymbol{f}^n$.

7.3.3 Cohesive force

The cohesive force represents the bonding between the ice elements, which in contrast with the repulsive elastic force, is calculated using the contact area

$$f_{coh} = Y_{coh} \cdot A_o, \tag{7.11}$$

where Y_{coh} is an attractive constant which has the same dimension as the Young's modulus and which is implemented as a fraction of Y. A_o is the contact area, calculated in the subroutine ComputeContactArea. The fraction coefficient is an additional variable, which can be defined by the user in *Simulation_Settings.txt*.

Please note, that the friction force, which will be discussed in the next section, depends on the normal force without cohesive effects. Therefore, only after the calculation of the tangential forces the cohesive force can be added to the sum of normal forces

$$\boldsymbol{f}^n = \boldsymbol{f}^n - f_{coh} \cdot \boldsymbol{n}_c. \tag{7.12}$$

The advantage of using the contact area as the magnitude of the attractive force is its depending on the relative orientation of the two colliding

elements, which is physical justifiable. The cohesive force between two elements is greater when they overlap along their long sides, than when they collide along their short sides.

Please note also, that this cohesive effect is only implemented in the normal force direction. Shearing is not effected by this force model.

7.4 Modeling of the tangential force

The correlation between normal force and tangential forces in general is given by sliding Coulomb friction

$$\boldsymbol{f}_f^t = -\frac{\boldsymbol{v}_t}{|\boldsymbol{v}_t|}\mu|\boldsymbol{f}_n|. \tag{7.13}$$

However, this force law is complicated to implement numerically. Cundall and Strack in [13] introduced a "breaking spring" model, in which at the initial contact an imaginary spring is attached between the two elements, which gets stretched until a maximum force (the dynamic friction above) is exceeded. Exceeding this point the spring acts as a constant force, whose magnitude is the above calculated force.

Other authors model this force via adaptive viscous damping or even worse, they neglect this component, as Chen in [3] states.

7.4.1 Cundall-Strack friction

The procedure to implement the Cundall-Strack friction in three dimensions is introduced by Matuttis in [19].

1. Projection onto the new tangential plane As a result of the advance in time, the steady change in the contact normal $\boldsymbol{n}_c$ and tangential velocity $\boldsymbol{v}_t$ leads to a correction of the friction force from the earlier time step, by projecting it onto the new tangential plane:

$$\boldsymbol{f}_f^t(t-\delta t)^p = \boldsymbol{f}_f^t(t-\delta t) - [\boldsymbol{f}_f^t(t-\delta t)\cdot\boldsymbol{n}_c]\boldsymbol{n}_c \tag{7.14}$$

2. Rescaling the old magnitude Then $\boldsymbol{f}_f^t(t-\delta t)^p$ has to be recaled to the magnitude of the previous tangential force $|\boldsymbol{f}_f^t(t-\delta t)|$:

$$\boldsymbol{f}_f^t(t-\delta t)^r = |\boldsymbol{f}_f^t(t-\delta t)| \cdot \frac{\boldsymbol{f}_f^t(t-\delta t)^p}{|\boldsymbol{f}_f^t(t-\delta t)^p|} \tag{7.15}$$

3. Vectorial addition of the new increment The rescaled projection $\boldsymbol{f}_f^t(t-\delta t)^r$ is then incremented to the new tangential force:

$$\boldsymbol{f}_f^t(t) = \boldsymbol{f}_f^t(t-\delta t)^r - k_t \boldsymbol{v}_t(t)\delta t \tag{7.16}$$

4. If necessary, application of a cut-off If the result from the previous vector addition exceeds the dynamic friction, a cut-off is applied:

$$\text{if } |\boldsymbol{f}_f^t| > \mu|\boldsymbol{f}_n| \quad \text{then} \quad \boldsymbol{f}_f^t(t) = \frac{\boldsymbol{f}_f^t(t)}{|\boldsymbol{f}_f^t(t)|} \cdot \mu|\boldsymbol{f}_n| \tag{7.17}$$

7.4.2 Dissipative force in tangetial direction

Chen [3] suggests to include another additional dissipative force in tangential direction, in the form of a damping component

$$\boldsymbol{f}_d^t = -\gamma^t \sqrt{Y_t M_{red}^t L_c} \boldsymbol{v}_t, \tag{7.18}$$

in which M_{red}^t is the tangential reduced mass, which can be chosen to be equal to M_{red}. However Matuttis [**?**] states that tangential damping hardly effects the outcome of the simulation, therefore it is neglected.

7.5 Drag and other hydrodynamic forces

The complex hydrodynamic interaction between the elements is not modeled in this code, since this is a problem, which only can be solved satisfactorily by using computational fluid dynamics methods. Nevertheless

the drag as an importand hydrodynamic force and should be taken account of. The general formulation of the drag force is

$$f_{drag} = -0.5 \cdot \rho \cdot v^2 \cdot C_d \cdot A, \tag{7.19}$$

where ρ is the density of the fluid, v the relative velocity of that body in the fluid, C_d a dimensionless coefficient and A the cross sectional area. The drag coefficient takes pressure and drag effects depending on the element shape into account. Additionally it depends on the Reynolds number, so this is not a fixed value, hence, it is approximated for each element at each time step. The maximum expected ice element velocity corresponds to the velocity of the towed structures, which is $v_{max} = 0.045\,m/s$ for the offshore model. Using the edge length $l \approx 0.1\,m$ and the recommended kinematic viscosity from the ITTC [26] for salt water at $1°C$ $\nu = 1.7926 \cdot 10^{-6}\,m^2/s$ the Reynolds Number lays in the range

$$Re \leq \frac{0.045 \cdot 0.1}{1.7926 \cdot 10^{-6}} \tag{7.20}$$

$$\leq 2300. \tag{7.21}$$

This is a very small value, compared to usual model tests, wherefore the the drag coefficient is going to be relatively big. Since the implementation of the calculation of a polyhedron's drag coefficient would take its orientation in the relative velocity into account, which is leaves too many unknown parameters, a more simple approach is chosen, where the drag coeffient of a sphere is appoximated. Morrison [27] proposes to calculate a sphere's drag coefficient for $Re \leq 10^6$ using

$$C_d = \frac{24}{Re} + \frac{2.6\,\left(\frac{Re}{5.0}\right)}{1+\left(\frac{Re}{5.0}\right)^{1.52}} + \frac{0.411\,\left(\frac{Re}{263\,000}\right)^{-7.94}}{1+\left(\frac{Re}{263\,000}\right)^{-8.0}} + \left(\frac{Re^{0.8}}{461\,000}\right), \tag{7.22}$$

which is applied in the current force calculation.

The cross sectional area as well is going to be approximated, since an algorithm to calculate this area for each element at each step would be too expensive. Therefore a mean area is calculated for each element at the initialisation, by adding up all element faces and dividing this value by the total number i of faces:

$$A_{mean} = \frac{\sum_i A_i}{i}. \tag{7.23}$$

This takes into account the arbitrary orientation of the great number of elements, and which therefore will reproduce satisfactory results.

While for ships the wave making resistance has to be considered, this force component for the ice elements is neglibible, since they are submerged and additionally the consolidated layer anyway prevents waves from building.

The low pressure phenomenon between adjacent objects, which was examined by Puntigliano in [16], is not modeled.

7.6 Gravity and buoyancy

The gravitational and buoyancy forces are the last forces to be calculated in the subroutine `ForceComputation`

$$f_g = V_{elem} \cdot g, \tag{7.24}$$
$$f_{buoy} = -\rho_w \cdot V_{elem} \cdot g, \tag{7.25}$$

which are added to the z-component of each element.

Chapter 8

Parallelization

Since modern workstations usally provide more than one processor, and since common simulations with millions of time steps and thousands of elements are very time consuming, parallel computing is used for big loops, where potential speed up is expected.

Without parallelization it can be shown, that the force computation subroutine (including the overlap computation) occupies over 95 % of the processing time. Therefore, some effort has been put into this subroutine by using the parallel processing interface *OpenMP*.

8.1 OpenMP

OpenMP is a shared memory processing interface, which was developed for the programming languages Fortran, C and C++ and whose current stable release is from November 2015[1]. Its main feature is its default usage of shared memory. This means, every variable in a parallel region is shared by all processors, unless its otherwise declared.

This principle corresponds to the architecture of modern workstations with many processors which all access the same memory. Having in mind, that this program should be used on a workstation (contrary to a cluster), this interface is the most efficient and most easy to set up solution.

[1] at the time of this assignment

The basic feature of a parallel region is the distribution of loops to different cores. At the beginning of a parallel the region the master thread creates a fork, according to the number of available cores, and distributes the work onto these processors.

8.2 Parallelization features

This section just gives a quick overview of how the paralleliszation is implemented in the subroutine `ForceComputation`.

Before the multithreading all potential contact pairs are calculated, calling `CalculateNeighbours`. The resulting list `overlapPairs` is the input for the force computation, which is distributed among all available processors.

Since the calculation time of the overlap geometry varies from case to case, some processors will reach the end of the loop before others do. In the default settings of OpenMP, these processors will be idle until the last processor has done its work. This can be avoided using the *dynamic* setting, which assigns every processor just one pair at a time and when a processor reaches the bottom of its fork the next uncalculated pair is assigned to it.

Furthermore, a solution to the following problem has to be found. Each element can have more than one contact. Therefore, it is impracticable to let each processor add the calculated forces to a shared array, which obviously would be `elements(i)%forces`. A problem occurs, when two processors add their result to `elements(i)%forces` at the same time, since the result of the processor, which accesses this entry slightly later, would get lost. A solution to this is to set up two local arrays `forces` and `torques`, which are shared, but have the attribute *reduction*. This attribute aims at creating a private variable for each thread, but sums it up afterwards. Thus the processors don't go amiss.

Chapter 9

Installation and usage of NumRidge

In this chapter a short overview over the general functional principle of a DEM-code is drafted and an detailed explanation of the different subroutines, how they work, how they are connected to each other and where they are found is given and in the last section the compiling of the source code and the usage of the executable file will be explained.

9.1 Functional principle

The flowchart of a typical DEM-program is shown in fig. 9.1.

9.2 Setup

The source code is devided into three types of files.

1. The **main file** *Main.f08*.
2. **Module files**, identifiable by their beginning *mod_XXX.f08*. These files contain the subroutines and procedures, which are linked by their purpose. For example *mod_Particle.f08* contains all subroutines, which are used for the initialization and updating of all elements.

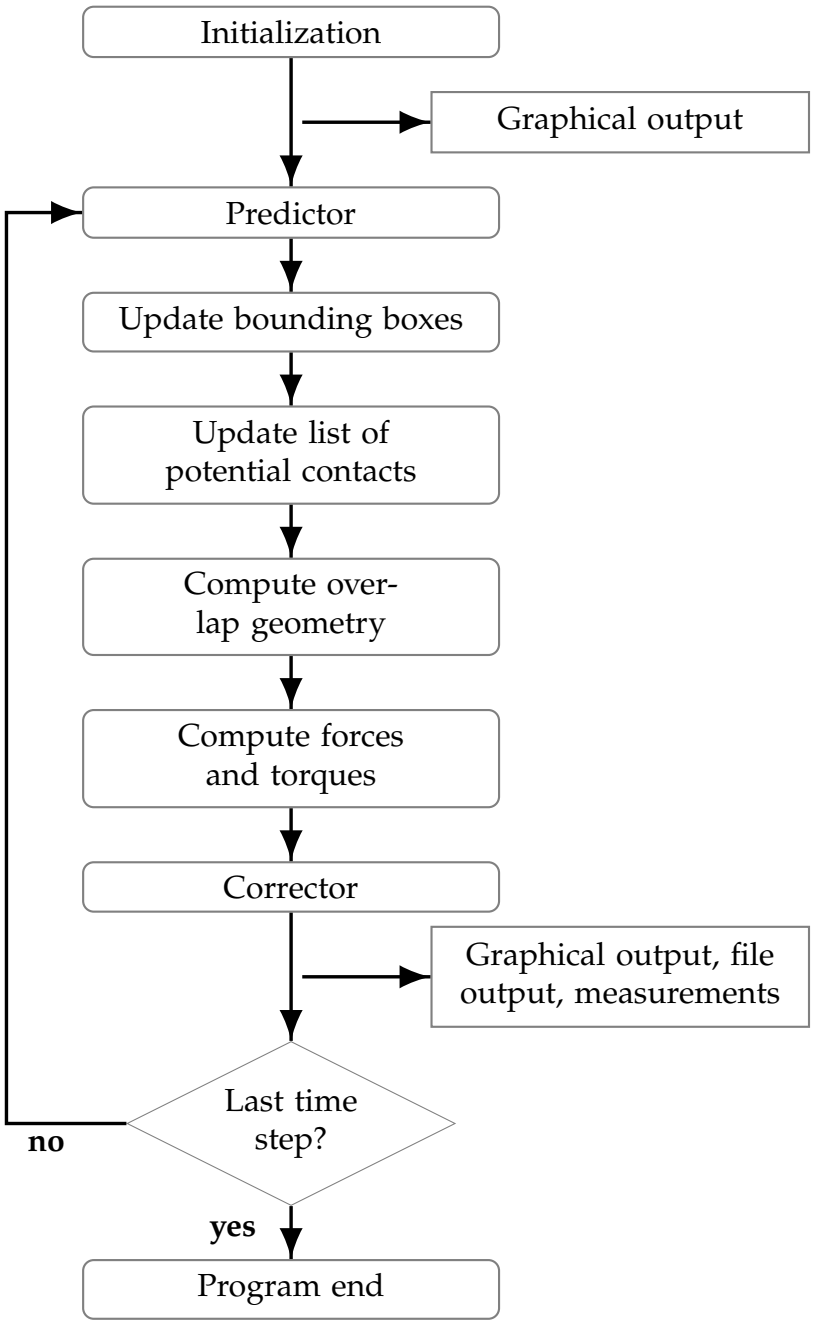


Figure 9.1: Typical flow in DEM algorithms

3. **Type files**, indentifiable by their beginning *type_XXX.f08*. In *NumRidge* so called derived types are used, which is a feature introduced in Fortran 95, that allows the programmer, to structure the code in respect of modern *object orientated programming* (OOP) technique. Each derived type has its own file and is included in those subroutines, where it's needed.

Main.f08 This is the main file, in which the input is evaluated and depending on which option the user chooses different initialization steps are performed. Then some precalculations are conducted, for example calculation of the time step size, using equation (4.28) and `frac` from *Simulation_Settings*. And after the output csv-files are allocated, the main loop starts calling succesive the subroutines `Predict`, `UpdateParticles`, `ForceComputation`, `Corrector`, `TerminationCheck` and in a defined interval `Graphics` (since we are not interested in a graphical output file for each time step). After the main loop an extra output file *results.txt* is created, in which among others the total calculation time is saved. At the end and if the ridge creation option was chosen, this ridge is exported using `ExportDomain` for further use.

mod_Force.f08 This module contains the subroutine `ForceComputation`, which firstly calls `CalculateNeighbours` and then starts a parallel loop for all neighbouring elements, in which at the beginning the geometrical overlap information are calculated by calling `OverlapComputation` and then based on these information the actual forces are determined.

mod_Functions.f08 This module contains several self written functions, for which no intrinsic Fortran function exists, and which are used more than one time. These functions are

- `Cross`: This is the cross function for two vectors
- `Swap_vertices`: This function swaps two vertices
- `Bubble_Sort`: This is a sorting algorithm, which sorts array columns according to a user defined row
- `M33INV`: This function inverts a 3x3 matrix
- `init_random_seed`: This subroutine creates a seed for the random number generator

mod_Global.f08 Many variables are used by most of the subroutines, and to avoid long transfer lists from subroutine to subroutine, the most used variables, such as the `elements`-array are stored there. This module of course has to be *used* by the subroutine, which needs the information from these variables.

mod_ImportExport.f08 In *mod_ImportExport.f08* the two subroutines `ImportDomain` and `ExportDomain` are located. The subroutine `ExportDomain` is used to export all ridge information, which are needed to run a subsequent simulation. These information are

- **rubble%q.txt** stores the quaternion for each rubble element.
- In **rubble%r.txt** the centers of volume of each rubble element a stored.
- **rubble%wltA.txt** stores the **w**idth, **l**ength, **t**hickness and mean cross-sectional **A**rea of each ice element.
- In **simulation.txt** the used simulation parameters (which were defined in *Simulation_Settings.txt*) are stored.
- **overlapPairs.txt** stores all relevant information of each overlapping bounding boxes, which are
  - indices of both elements i and j,
  - the old tangential friction force $\boldsymbol{f}_{f,old}$,
  - the overlap volume V_o,
  - the overlap area A_o,
  - the overlap centroid $\boldsymbol{c}_o$ and
  - the force direction $\boldsymbol{n}$.
- In **endPoints.txt** the sorted `xEndPoints`, `yEndPoints` and `zEndPoints` are saved. This is necessary, because the sorting algorithm in the contact detection algorithm doesn't work when the elements overlap in the first time step. Therefore these array entries have to be in already sorted order during *import*.

`ImportDomain` basically is an inverted `ExportDomain`. All ridge information are read in to start a new simulation with the exact topological and kinematical configuration of the last time step of ridge creation.

mod_Neighbour.f08 This module contains five subroutines, batched into two nested trees. The first subroutine `CalculateNeigbours` calls the procedure `SweepAndPrune` three times, each time with another `endPoints`-array as the only argument. `SweepAndPrune` itself calls `AABBOverlap` if during the sorting of an `endPoints`-array a potential overlap of the bounding boxes is detected. The exact contact finding algorithm is explained in the section 5.2. The second batch of subroutines is called by the procedure `InitialSort`, which calles `InsertionSort` three times (analog to `CalculateNeighbours`. `InsertionSort` is the basis for the above mentioned procedure `SweepAndPrune`, and which is used just unce during the initialization to sort the `endPoints`-arrays without checking for AABB-overlaps.

mod_Output.f08 The module *mod_Output* contains all subroutines, which are called to generate graphical and measurement output files. The contained subroutine `Graphics` creates consecutively numbered vtk-files in 1/20 seconds intervals. Hence, if the user wants to create an animation out of this out, the clip would be rendered with 20 frames per second. This could of course be adjusted by modifying the variable `fps` in *Main.f08*. The second subroutine `Output`, which is called at the end of the main program, creates the output file *results.txt*, in which the essential information from the current simulation are saved. This subroutine, when running on Linux, as well compresses all output files and moves the archive to *./output/save*.

mod_Particle.f08 In *mod_Particle.f08* the following five subroutines are located:

- `InitializeParticles` initializes the domain, containing the rubble elements, the consolidated layer and the two basin walls. For each element its physical properties are set and the pointers for the `elements`-array and the bounding boxes are allocated.
- `MakeStructure` initializes the structure, which is in case (1) the pushing bars from the ridge creation, in case (2) the punch device from the punch test or the offshore structure, which is tested in option 3).
- In `UpdateParticles` the new positions of the ice elements are calculated, depending on their predicted velocities. Additionally the structure is displaced in the domain, according to its predefined

velocity. For all elements (excluding the boundaries) their `face_equations` are updated and in the end the subroutine `UpdateBoundingBoxes` is called.

- `UpdateBoundingBoxes` is used to update the bouding boxes of all elements.
- `TerminationCheck` checks if conditions are fulfilled to terminate the main loop.

mod_PredCorr.f08 This module contains the three subroutines `Predict` and `Correct`, which form Gear's backwards-difference algorithm and the the third subroutine `OutputForces`. This procedure is called in `Correct` and it exports the total kinetic energy in the domain and the sum of the forces acting on the structure.

mod_OverlapComputation.f08 This module contains all subroutines, which were already explained in the chapter 6. The input information are basically just the indices of two potentially overlapping elements leading to the output information, which are the `overlap_volume`, the `overlap_area`, the `overlap_centroid` and the `force_direction`. For clarity of the connection of all the subroutines, the following enumeration can be consulted. The main subroutine `OverlapCalculation` calls succesively

1. `Compute_inherited_vertices`, which calls
   (a) FindInheritedVertices for $\boldsymbol{P}_1$ inside of $\boldsymbol{P}_2$
   (b) FindInheritedVertices for $\boldsymbol{P}_2$ inside of $\boldsymbol{P}_1$
2. ComputeGeneratedVertices, which calls
   (a) FindFaceIntersection, which calls
      i. `Face_plane_intersect` for $\boldsymbol{F}_i(\boldsymbol{P}_k)$ and $\boldsymbol{Pl}_j(\boldsymbol{P}_l)$[1]
      ii. `Face_plane_intersect` for $\boldsymbol{Pl}_i(\boldsymbol{P}_k)$ and $\boldsymbol{F}_j(\boldsymbol{P}_l)$
3. `Compute_overlap_polyhedron_faces`, which calls
   (a) `FindFaces` from element i
   (b) `FindFaces` from element j

[1] Face i of Polygon k and Plane j of Polygon l

4. `Compute_overlap_volume`
5. ComputeContactArea

9.3 Usage of NumRidge

The calculation domain of the numerical ridge is shown in figure 9.2. The origin of the global space-fixed coordinate system is chosen to be on the upper-aft-left corner of the ridge and three boundary elements (two basin walls and one consolidated layer) are sufficient to prevent the ridge particles to leave the domain. The mechanical properties of the basin walls are chosen to be equal to the ice properties (Young's Modulus and friction coefficient) and additionally during the force computation cohesive forces are calculated between ice and basin wall elements, to imitate a continuation of the ridge beyond the boundary. This allows us to decrease the ridge length and therefore to decrease the computation time.

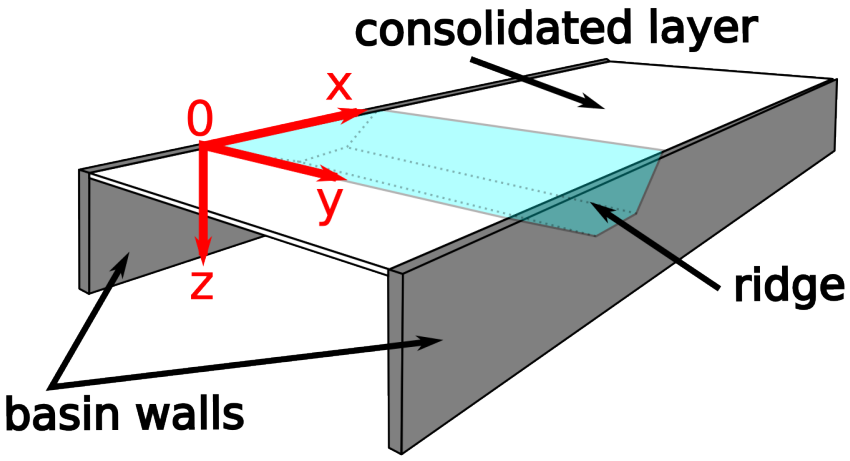


Figure 9.2: Sketch of the domain

The first thing to do is to run the compiling script *optimize.sh* for linux or *optimize.bat* for windows, which uses *gfortran* from the *GNU Compiler Collection*[2]. Optional scipts for Linux and Windows are provided for debugging purposes. All scripts additionaly create, besides the executable

[2]freely aviable from `https://gcc.gnu.org/wiki/GFortranBinaries`

file *Numridge* in the main folder, the required folders, in which during execution the output files are stored.

Before running the executable file, the user has to be sure that the following folders and initial files are in place:

- **NumRidge** and **Simulation_Settings.txt** should be in the main directory.
- the following output folders have to be created, if they were not created automatically by the compiling script:
  - *./output/csv*
  - *./output/vtk*
  - *./output/ridge*
  - *./output/save*

In the **Simulation_Settings.txt** four different kinds of input information have to be set:

1. The ridge dimensions (for reference see section 2.1)
2. Information about the rubble dimensions and mechanical properties
3. Coefficients for the force model
4. and time integration parameters

After starting the executable file, the user is asked to choose between three options (see figure 9.3): 1) To create a new ridge, 2) to perform a punch test with an already existing ridge or 3) to perform a model test with an already existing ridge. After each succesfull simulation, when running on Linux, the whole output is compressed and saved to *./output/save*. The compression and moving of the files to be saved when operating Windows has to be done manually afterwards.

After choosing an option and pressing *enter*, every time a graphical output file is created, a message will be printed, in which the real calculation time is given. This allows the user to estimate the overall progress of the simulation.

Option 1) creates a ridge, depending on the input settings and saves afterwards all relevant information to *./output/ridge*. Since the forming of a ridge in the nature can take several months, an artifical way for creating

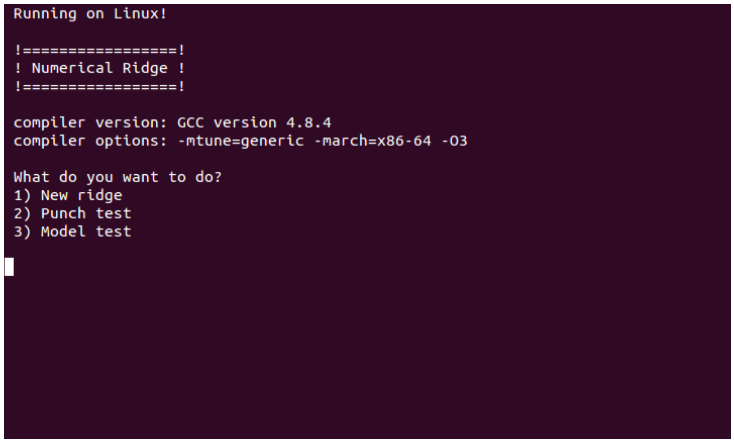


Figure 9.3: Startscreen

the desired ridge is implemented here. At the initialization of all elements the rubble particles are placed in the water, not touching each other. In addition two rotated pushing plates, which in the code are handled as structures, are placed in the water and also not touching the ice. Once the simulation starts, the rubble elements ascend due to their bouyancy and the two pushing bars move towards each other, pushing the heap of rubble into the desired formation. Finally the velocity of the push bars is reversed to relieve the stresses in the ridge, and twenty seconds after the reversing the simulation stops and the ridge is saved to *./output/ridge*. For convergency analysis purposes additionally a csv-file in *./output/csv* containing the sum of the kinetic energy of all rubble elements over time is saved. The typical progression during the ridge creation is shown in figure 9.4.

As shown in figure 9.5, the numerical result of the so called *floating-up techniqe* represents satisfactorily a model ridge created at the HSVA.

Option 2) loads the ridge from *./output/ridge* and places a punch device in the middle of that ridge. Once the simulation starts, the punch device is lowered with a fixed velocity into the ridge until it has penetrated the whole keel. The measuring frequenzy for all three force components acting on the cylinder is set to be 50 Hz. During the simluation four csv-files in *./output/csv* are created, three for each force component over time and one for the total kinetic energy of the ice elements over time. The simulation is terminated after the punch device has penetrated the ridge and is displaced another 20 centimeters.

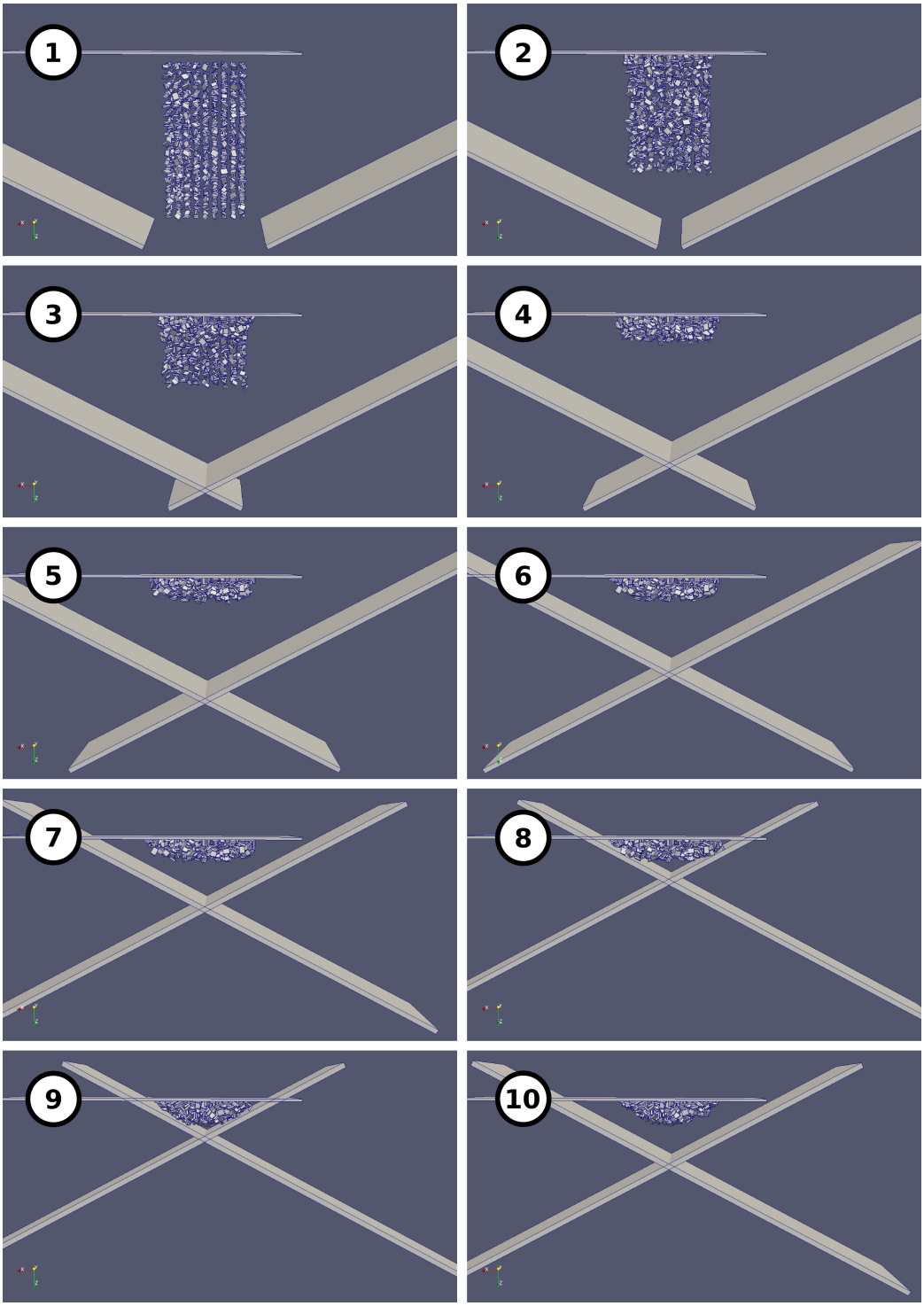


Figure 9.4: Stages of ridge creation

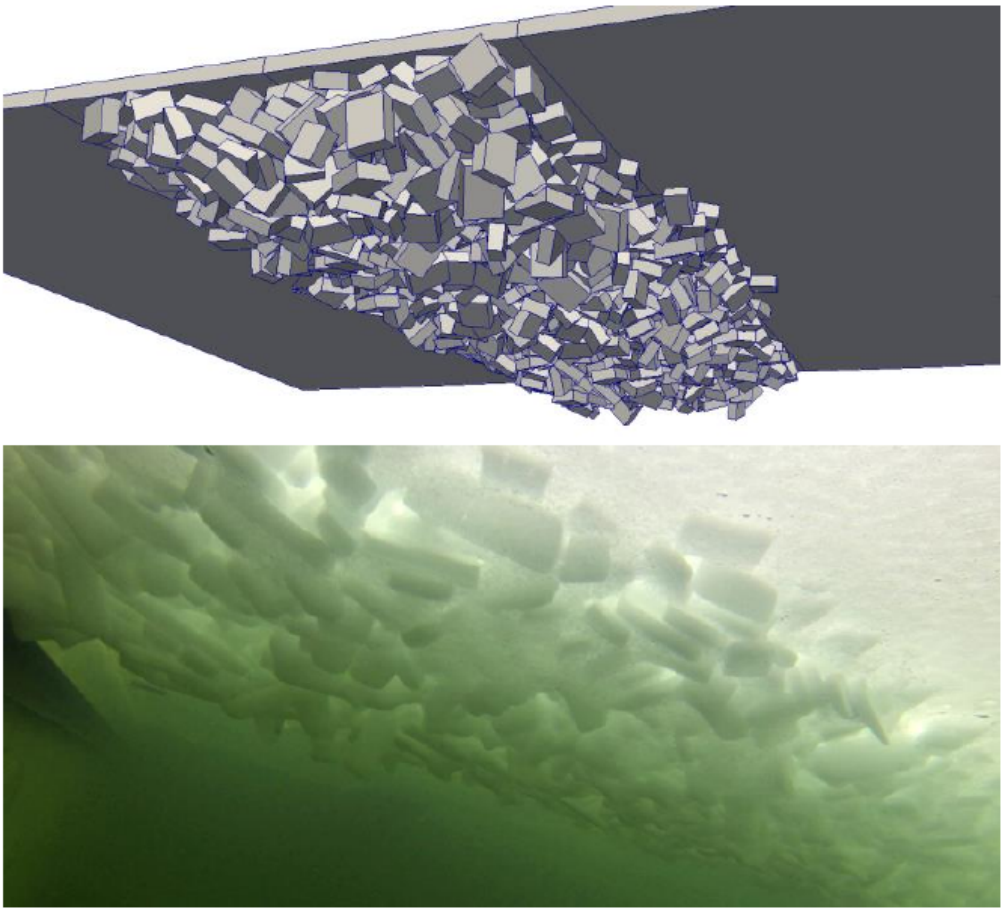


Figure 9.5: Comparison of numerical ridge vs. model ridge at HSVA

Option 3) is very similar to option 2), since the model is a structure like the punch device, only with a different starting position and a different velocity. The output files are the same like in 2). The termination condition as well is similar to the punch test, only the displacement of the structure in x-direction is compared to the ridge width.

Postprocessing For the postprocessing the open source program *Paraview*[3] can be used. The translational speed of every element is saved to the output files and can be visualized in Paraview selecting the desired component from drop-down menu, shown in figure 9.6.

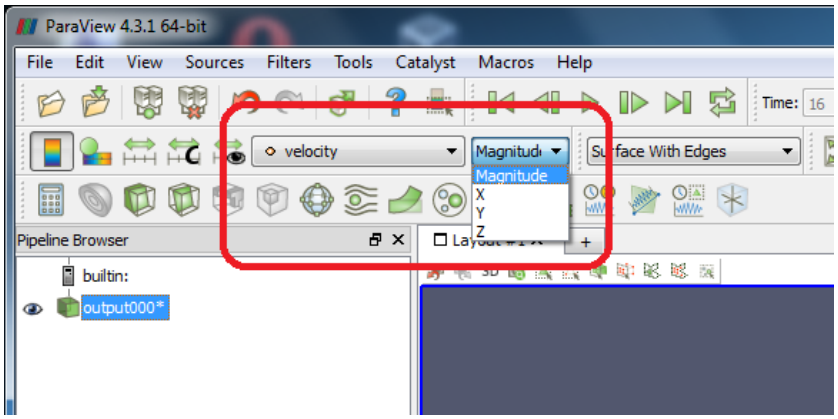


Figure 9.6: Visualization of velocity in Paraview

Some useful stackable filters are *Elevation* and *Clip*. These can be found in the list of filters from the drop-down menu located in the menu bar. Applying both filters gives the user the ability to cut free parts of the ridge to examine the movements of embedded elements and to have a good visualization option for publications. An application of both filters for a random frame of a punch test is shown in the following figure 9.7.

[3] Available from `http://www.paraview.org/`

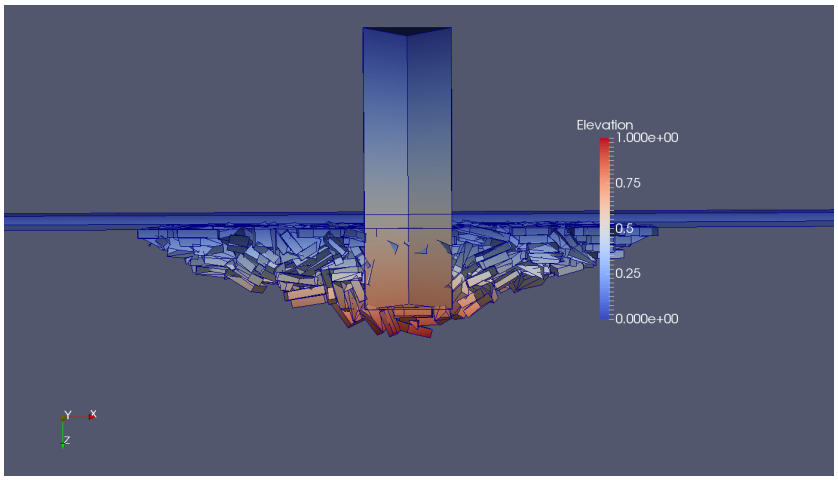


Figure 9.7: Applied clip and elevation filters

Chapter 10

Simulations and validation

10.1 Correctness of the overlap computation

To ensure the correctness of the `overlap_volume`, the center of the overlap volume, the `force_direction` and the `contact_area`, which are obtained by the subroutine-tree nested in `OverlapComputation`, three simple test cases are set up, for which the above mentioned output can easily be verified.

One inherited vertex from each element (overlap polyhedron is a cuboid) The dimensions (in [m]) of the two intersecting elements (see figure 10.1) are:

- Element 1: length, width and thickness = 2 with its center of volume at $c_1 = (0, 0, 0)^T$
- Element 2: length, width and thickness = 2 with its center of volume at $c_2 = (1, 1, 1)^T$

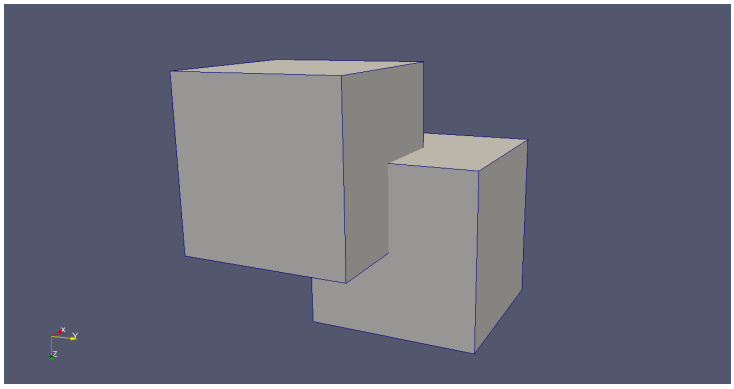


Figure 10.1: Validation case 1

The output parameters from `OverlapComputation` are,

- $\boldsymbol{n} = \begin{pmatrix} -0.57735026918962584 \\ -0.57735026918962584 \\ -0.57735026918962584 \end{pmatrix}$
- $\boldsymbol{c}_o = \begin{pmatrix} 0.50000000000000000 \\ 0.50000000000000000 \\ 0.50000000000000000 \end{pmatrix}$
- $\boldsymbol{V}_o = 1.0000000298023224$
- $\boldsymbol{A}_o = 1.7320508075688772$

which are correct.

Four inherited vertices and four generated vertices The aim of this test case (see figure 10.2) is to check, whether the inherited face is correctly detected. The two intersecting elements are:

- Element 1: length, width and thickness = 2 with its center of volume at $\boldsymbol{c}_1 = (0, 0, 0)^T$
- Element 2: length, width and thickness = 1 with its center of volume at $\boldsymbol{c}_2 = (0, 0, 1)^T$

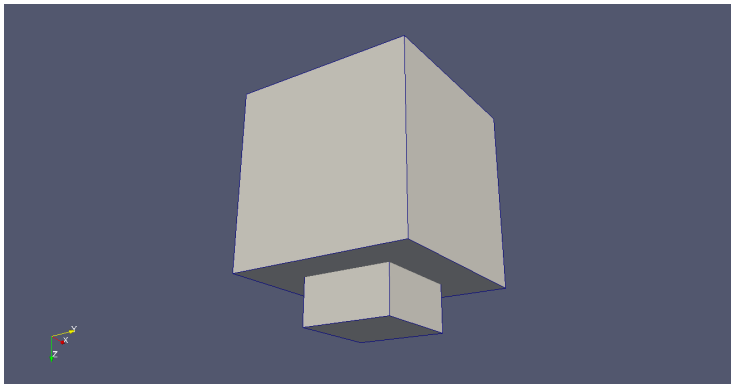


Figure 10.2: Validation case 2

The output parameters from `OverlapComputation` are,

- $\boldsymbol{n} = \begin{pmatrix} 0.0000000000000000 \\ 0.0000000000000000 \\ -1.0000000000000000 \end{pmatrix}$
- $\boldsymbol{c}_o = \begin{pmatrix} 0.00000000000000000 \\ 0.00000000000000000 \\ 0.75000000000000000 \end{pmatrix}$
- $\boldsymbol{V}_o = 0.50000001490116119$
- $\boldsymbol{A}_o = 1.0000000000000000$

as expected.

No inherited vertices at all An edge-edge contact leads to no inherited vertices at all. To ensure the correctness of the overlap computation the validation test case three from figure 10.3 has been established. The element dimensions are:

- Element 1: length, width and thickness = 1 with its center of volume at $\boldsymbol{c}_1 = (0, 0, 0)^T$
- Element 2: length = 1, width and thickness = 0.5 and it's rotated by 45 degrees around its y-axis. The center of volume is located at $\boldsymbol{c}_2 = (0.5, 0, 0.5)^T$

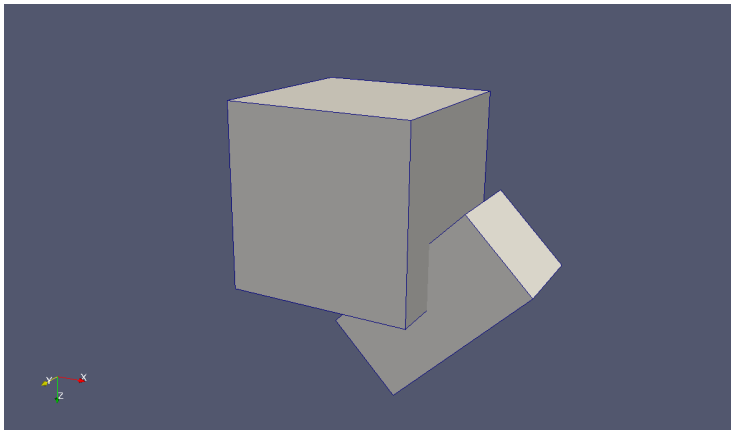


Figure 10.3: Validation case 3

The correct output parameters from `OverlapComputation` are:

- $\boldsymbol{n} = \begin{pmatrix} -0.70710678118654757 \\ -3.4694469519536148E-017 \\ -0.70710678118654757 \end{pmatrix}$
- $\boldsymbol{c_o} = \begin{pmatrix} 0.38214886980224205 \\ 0.00000000000000000 \\ 0.38214886980224205 \end{pmatrix}$
- $\boldsymbol{V_o} = 3.1250000931322575E-002$
- $\boldsymbol{A_o} = 0.24999999999999997$

Other cases, like an element piercing through another or beeing totally inherited don't have to be checked, since the overlaps are very small (an ice element in contact with the consolidated layer comes to an rest with an overlapping volume $V_o \approx 7.2 \cdot 10^{-8}\, m^3$).

10.2 Punch test

As explained in section 2.2 the purpose of the punch test is the determination of a ridge's mechanical properties. The HSVA has conducted many punch tests with different ridges, wherefore a numerical punch test can be used to validate the implemented force model and to find correct ranges for the input force parameters. Since the allowed element shape in the current tool is just a cuboid, the punch device is represented

by a box with bottom edge lengths $l = \sqrt{\pi \cdot r_{cyl}^2}$, where $r_{cyl} = 0.125\,m$, to obtain the same bottom area as the original punch cylinder.

The chosen ridge dimensions and element size correspond to ridges, which were built at the HSVA in 2014, and for which punch tests have been performed. Target dimensions were:

- Keel width $W_K = 1.8\,m$
- Bottom keel width $BW_K = 0.3\,m$
- Keel draught $H_K = 0.4\,m$
- Ridge length $L = 2\,m$

with mean ice elements dimension $0.1\,m \times 0.1\,m.03\,m$ and an ice density $\rho_{ice} = 900\,kg/m^3$. The resulting ridge has been saved and used for every subsequent simulation by modifying the file *Settings.txt* in the folder *./output/ridge*. Whilst this procedure didn't lead to significant complications during the the last time step of ridge creation and the first time step of the punch test, it is recommended to create an entirely new ridge for changing force parameters.

10.2.1 Time step convergency

First a time step convergency analysis is conducted, to find adequate time step sizes for the subsequent simulations. Therefore random but reasonable input parameters are chosen and five simulations with varying `frac` are run.

The used input parameters are:

- Young's Modulus of ice: $Y = 10\,mPa$
- Normal damping coefficient: $\gamma_n = 0.5$
- Friction coefficient: $\mu = 0.75$

To start with a conservative choice for `frac`, Cundall in [24] proposes to use $frac = 0.1$ for explicit one-step integrators. Subsequently it is doubled, and from then on it is increased by 0.2.

To compare the results the kinetic energies of every simulation, saved to *kinetic_energy.csv*, are plotted. Increasing the time step size, it is expected that the kinetic energy in the domain rises.

As from figure 10.4 can be seen, fraction values up to 0.4 lead to exacly the same kinetic energy in the domain. Increasing `frac` further, the simulation gets more unstable until, at frac = 0.8, it crashes when the pushing bars start to compress the ridge.

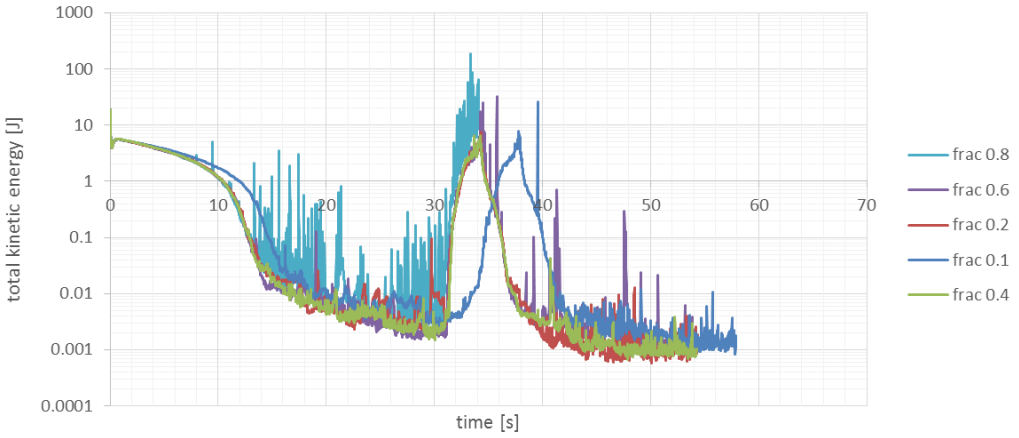


Figure 10.4: Kinetic energy during ridge creation

Additional potential in increasing the time step could be obtained by decreasing the Young's Modulus. Referencing to section 4.3.3 the minimum time step size increases with decreasing Young's Modulus, since the overlaps get bigger and hence the total overlapping time increases. This however is not examined here, since in the used force model the normal repulsive force scales with Y but on the other hand the normal damping with $\sqrt{Y}$, which leads to a different composition in the forces.

10.2.2 Parameter finding

The two important force parameters, which have to be determined using the numerical punch test, are

- the friction coefficient μ and
- the cohesion coefficient c_{coh},

Friction and cohesion should represent the freezing bonds of the ice, wherefore especially the friction is expected to be unphysically large, since it is the only force component which prevents the elements from tangential relative movement. The determination of the normal damping

coefficient is going to be neglected, since the slow velocity of the punch device will not lead to different outcome for changing damping coefficients. It is to be mentioned, that the resulting z-force on the cylinder is the sum of shear forces between its sides and the surrounding elements, which is obtained by using a friction coefficient $\mu_{cyl} = 0.1$, and the pressure acting on its bottom. Bouyancy is not calculated for the cylinder.

Using $Y = 1\,mPa$ for the ice, different values for the ridge porosity have shown, that a porosity around 0.45 gives the correct keel height during ridge creation.

Impact of friction

Tested friction coefficients were

- $\mu = 0.25$
- $\mu = 0.50$
- $\mu = 0.75$
- $\mu = 1.00$
- $\mu = 1.50$

where cohesion and normal damping coefficient were kept constant. Using $c_{coh} = 0.0001$ and $\gamma_n = 0.5$, the following results (see figure 10.5) were obtained. As expected increasing the friction coefficient increases the force acting on the punch cylinder during ridge penetration.

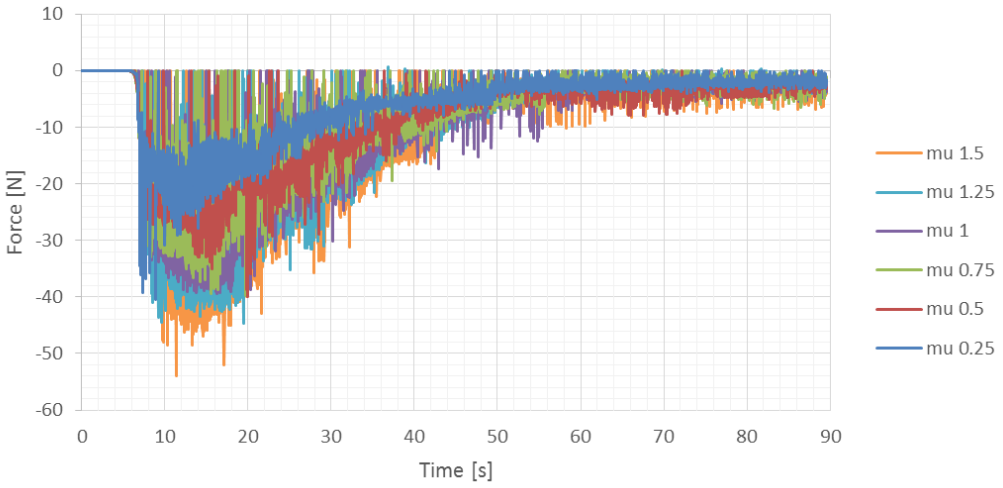


Figure 10.5: Z-forces on punch device with varying friction

The kinetic energy in the domain is not effected significantly during the punch, which can be seen in figure 10.6.

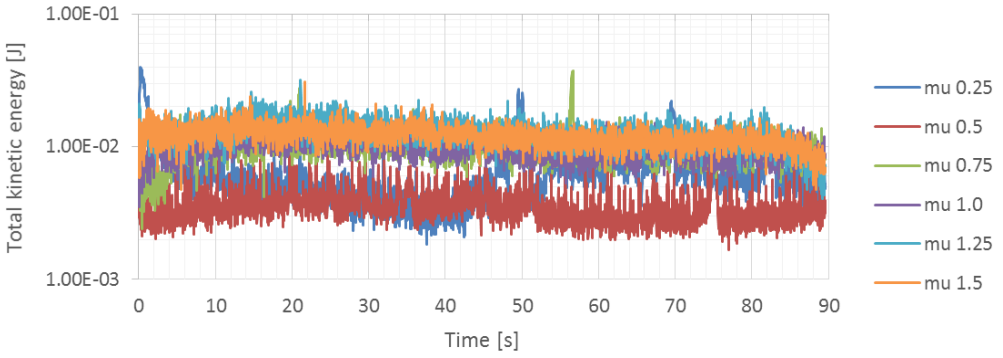


Figure 10.6: Kinetic energy during punch test with varying friction

Angle of repose It is also noticable, that the angle of repose for $\mu = 0.25$ is smaller, than for the heaps with greater friction. In [28] an approximation formula for the angle of repose θ, depending on the static friction coefficient, is given

$$\tan(\theta) \approx \mu. \tag{10.1}$$

For $\mu = 0.25$ the calculated angle of repose is approximately $\theta \approx 25°$, which corresponds to the visual output from figure 10.11. Therefore,

this tool allows realistic (at least friction depending) predictions of the topological outcome of simulations.

Impact of cohesion

Punch tests with five different cohesion coefficients have been simulated as well. The used parameters were

- $c_{coh} = 0.1$
- $c_{coh} = 0.01$
- $c_{coh} = 0.001$
- $c_{coh} = 0.0001$
- $c_{coh} = 0.00001$

with $mu = 0.5$ and $\gamma_n = 0.5$ as constants. From figure 10.7 one can see, that during the simulation with $coh = 0.1$ after approximately 42 seconds for at least one element the forces resulted in NaN, which leads to NaN in the velocities and therefore to NaN in the total kinetic energy, which cannot be plotted. Despite this occurence, the simulation kept on running with the remaining elements and the result of the punch test is shown in the figure 10.8.

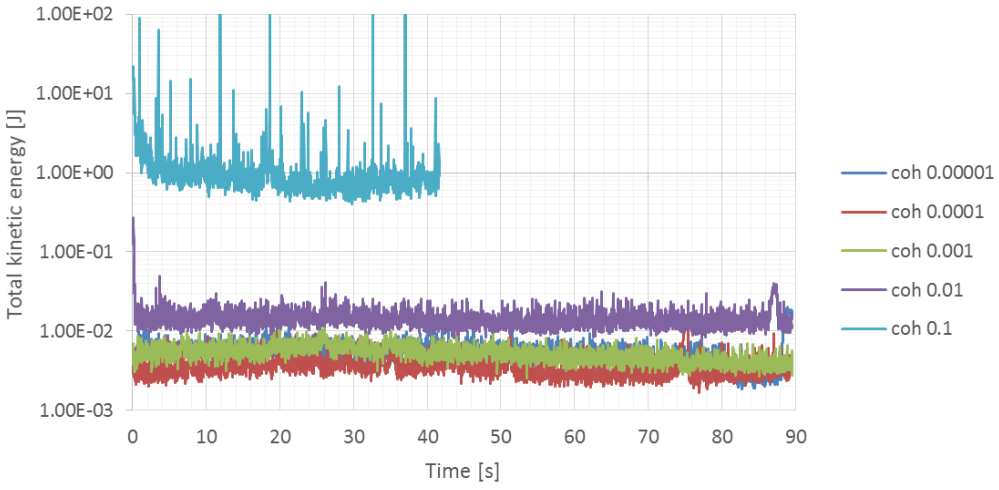


Figure 10.7: Kinetic energy during punch test with varying cohesion

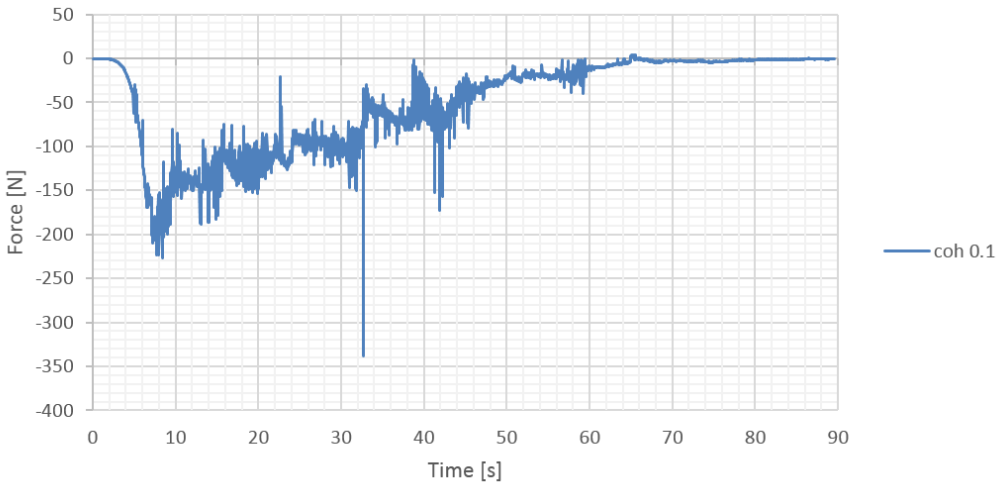


Figure 10.8: Z-forces on punch device with $coh = 0.1$

Apart from that, even by varying this coefficient by a factor up to 1000 and having stable kinetic energy over time, the outcome in the forces doesn't change significantly, which can be seen in figure 10.9.

This is easily explained by looking at the computation of the cohesive force component. Cohesion is purely a normal contact force, which is calculated after the tangential friction. Therefore, the maximum friction doesn't depend on cohesive effects, and since during the punch no pull-like actions on the elements are applied, it is just the regular friction, which effects the outcome of the physical results.

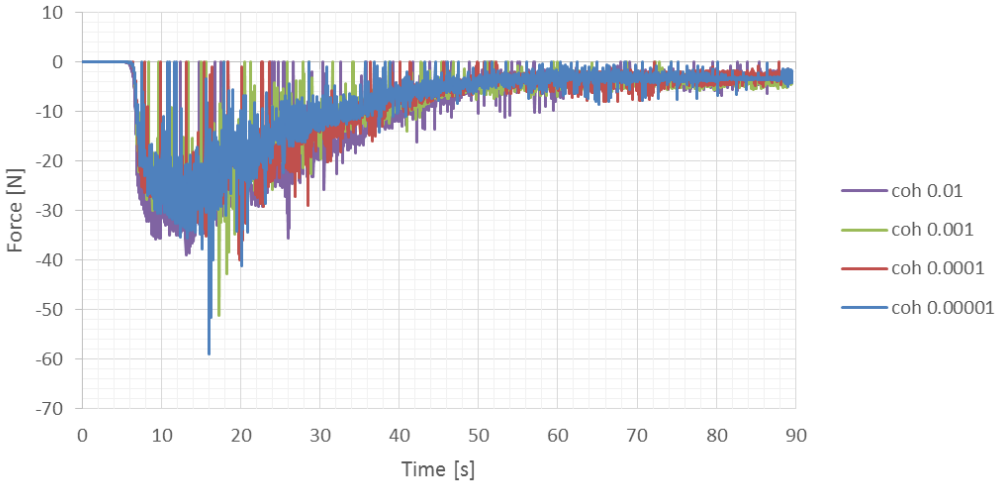


Figure 10.9: Resulting z-forces on punch device

On the other hand, when looking at the visual output one can clearly see the formation of ice clusters. This permits the hypothesis that for applications where besides physical load prediction a kinemtatic analysis of the ice movements is of interest, cohesion gains influence.

Comparison with original punch test Using the force parameters $\mu = 1.0$ and $c_{coh} = 0.001$ and comparing the measured forces with simulation results reveals that the evolution of the force over time (a big gradient in the beginning and a smooth decrease after the peak) represents the original tests very well (see figure 10.10). But on the other hand the simulated values are constantly about 40 to 45 Newton too low.

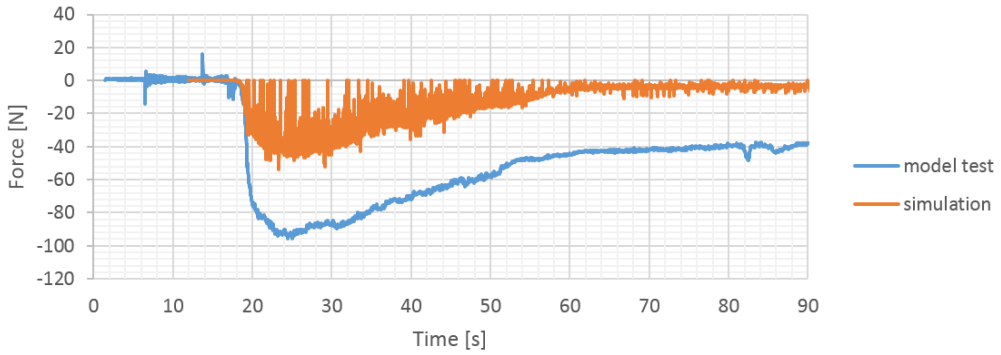


Figure 10.10: Comparison punch test forces

Looking at the end configuration of some of the punch tests (figures 10.11 to 10.13) and comparing them with an original test from figure 2.6, reveals that just a small heap of ice elements (around four to five elements) remains on the bottom of the device on contrast to a large conical structure in the model basin. A possible explanation for this discovery is the incapability of the current tool to allow breaking of the ice. Having a closer look on figure 2.6 shows that the size distribution of the elements is much bigger, than implemented numerically. Additionally it looks like, that potential void spaces are filled with slushy ice, which could support the pressed out heap as well.

Due to this missing ice cluster, the force component resulting from the buoyancy of the rubble is effectively nonexistent. The force difference beginning from approximately second 60 is exacly the difference between the bouyancy of the simulated heap and original outcome, since from there on the punch device has penetrated the ridge and just the bouyancy and a small amount of friction between ice and the device's sides is measured.

On the enclosed DVD two real time animations of a puch test with $\mu = 1.0$ and $c_{coh} = 0.001$ are located.

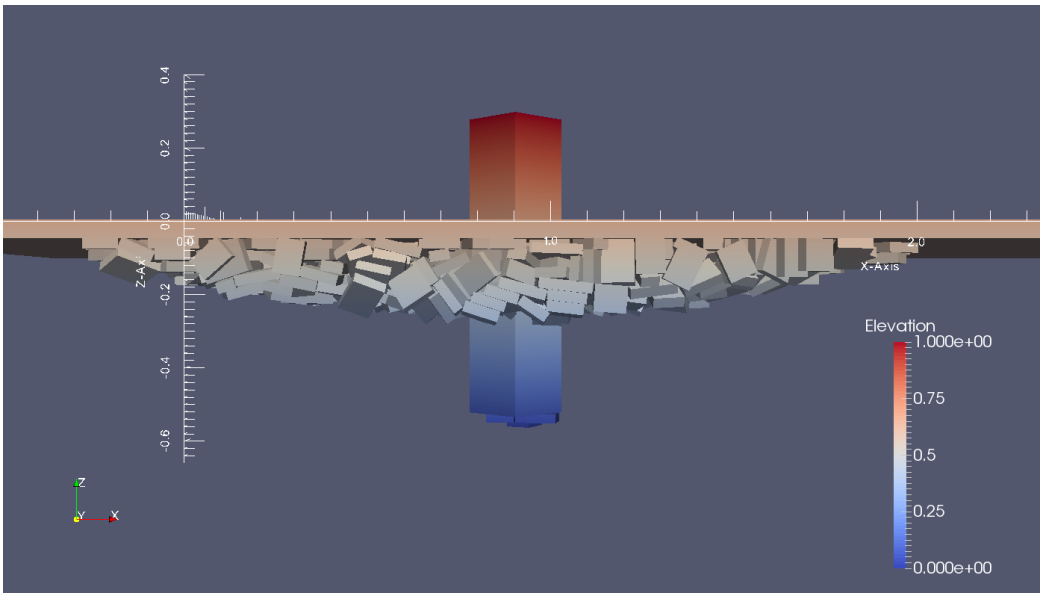


Figure 10.11: End configuration for $\mu = 0.25$

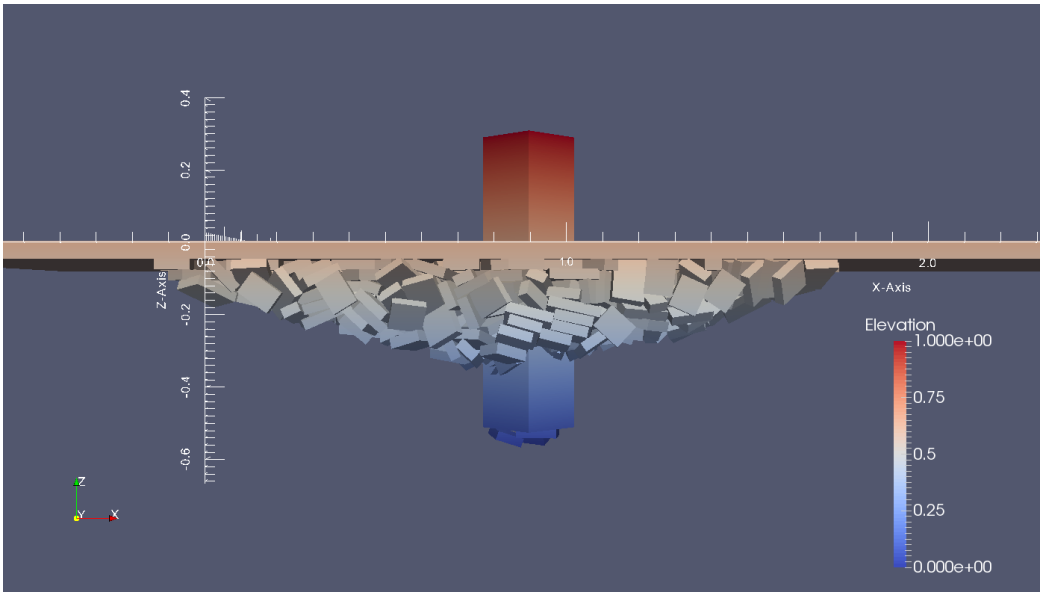


Figure 10.12: End configuration for $\mu = 0.75$

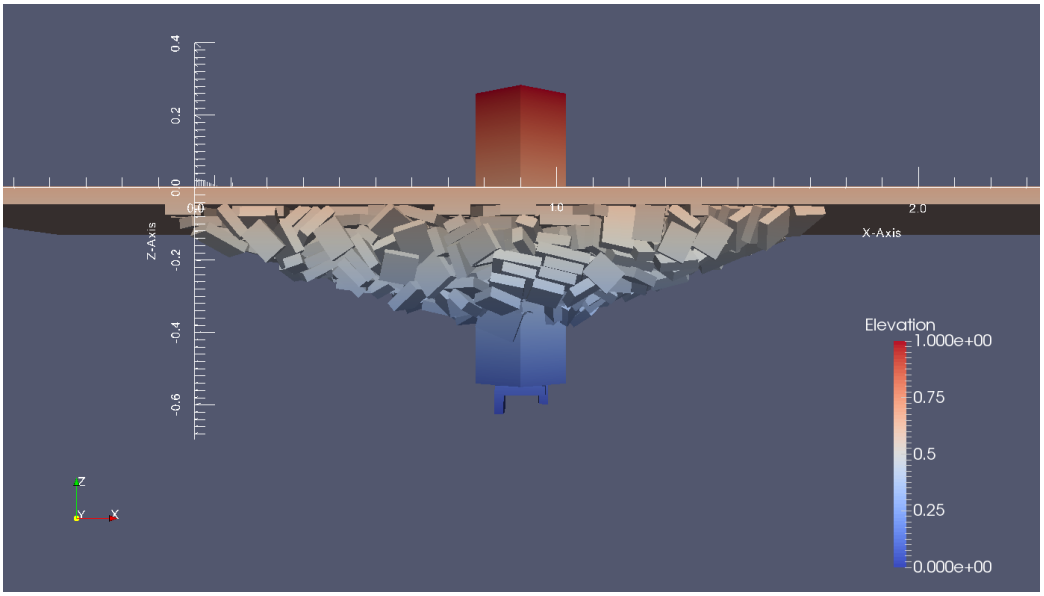


Figure 10.13: End configuration for $\mu = 1.25$

10.3 Hydralab III

Serré and Evers [7] conducted ridge breaking model tests with a submerged model in the context of the international hydraulic research program *Hydralab III* in 2008. The current simulation tool *NumRidge* is capable of simulating this model test, since no breaking of ice elements (especially the level ice sheet) is necessary.

The target dimension for the chosen ridge had been

- Ridge width 5 m
- Keel width 3 m
- Keel height 0.5 m

and measured properties of the ice elements were

- Ice length 0.08 m
- Ice width 0.11 m
- Ice thickness 0.033 m
- Ice density 795 kg/m^3.

The tested subsea structure is a simple cube $(0.7 \times 0.7\,m)$, submerged 0.22 m below water line and towed with 4.5 cm/s. During ridge breaking the acting forces in x-, y- and z-direction were measured. A drawing of the test setup is given in figure 10.14 and the corresponding numerical domain in figure 10.15.

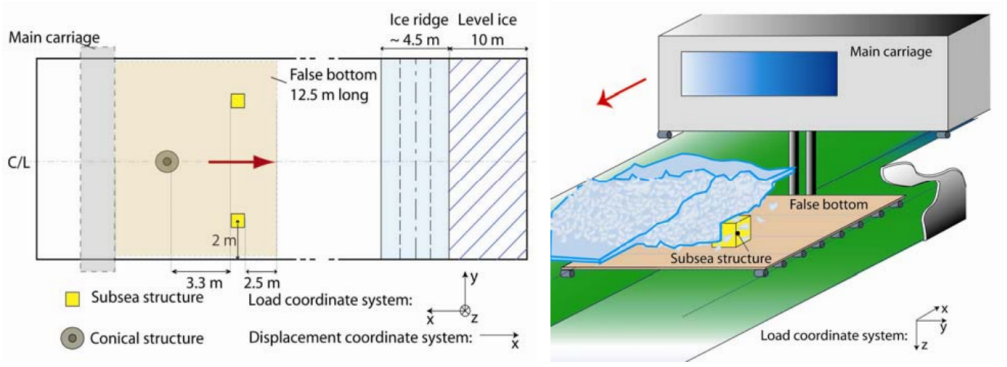


Figure 10.14: Test setup for Hydralab III

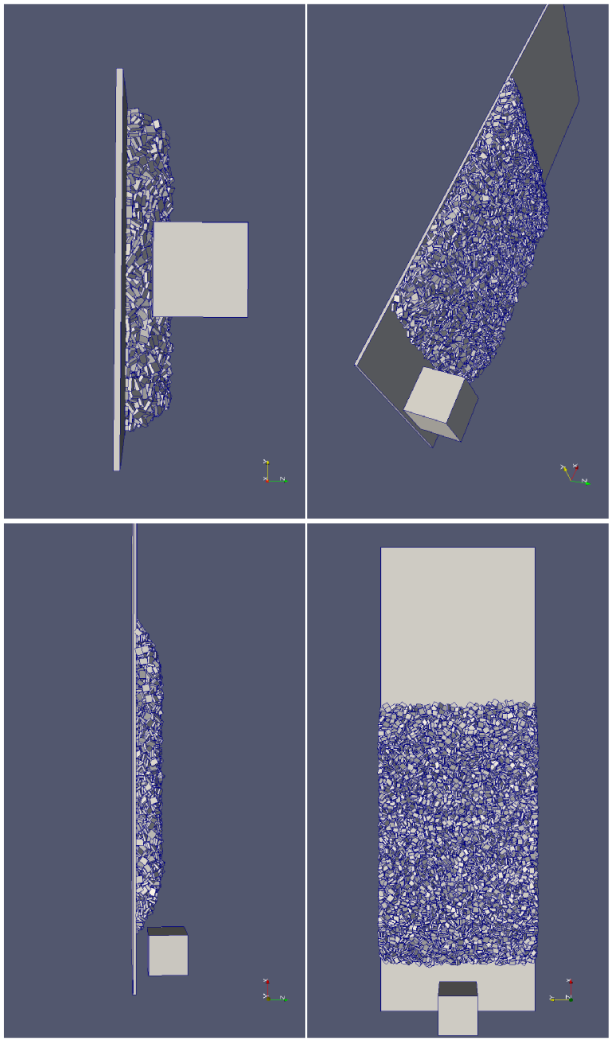


Figure 10.15: Initial set-up for simulation

Using the force parameters $\mu = 1.0$ and $c_{coh} = 0.001$ the following results were obtained.

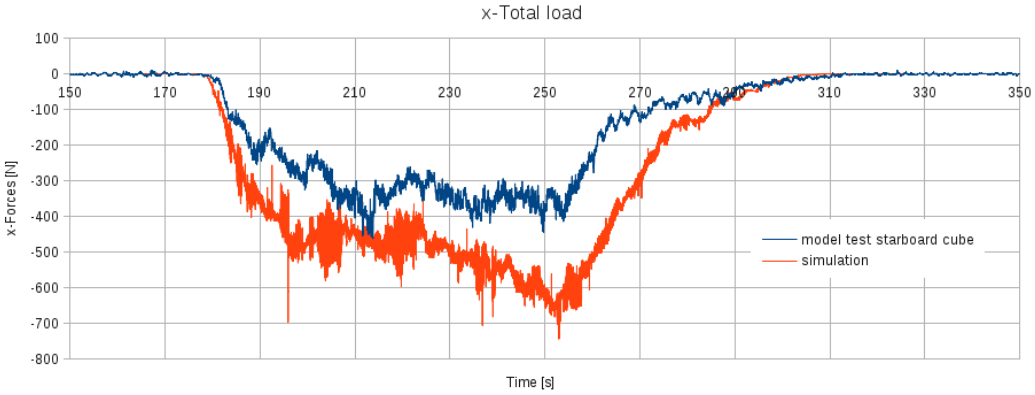


Figure 10.16: Measured x-force on structure

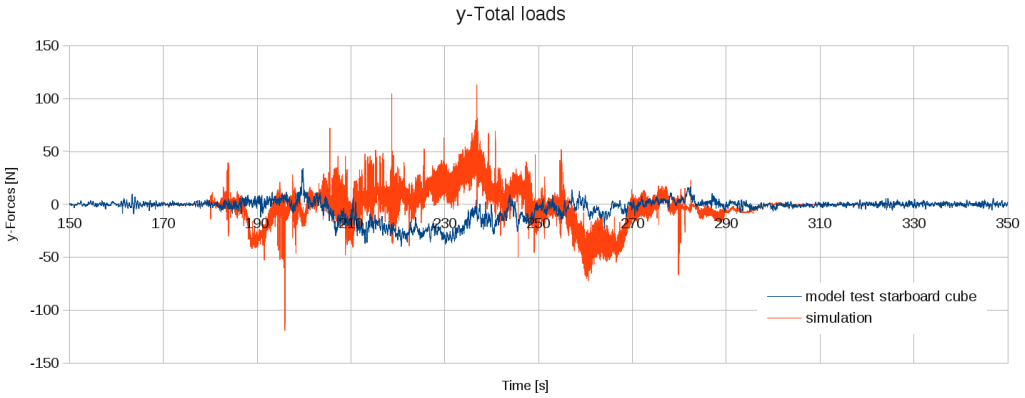


Figure 10.17: Measured y-force on structure

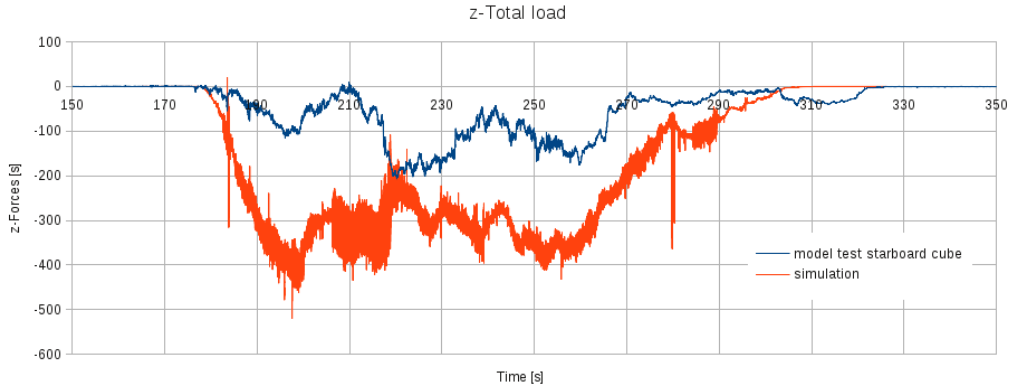


Figure 10.18: Measured z-force on structure

One can clearly see, that the approximation of x- and y-forces acting on the strcuture is very good. The time history of both forces is represented very well, they are just a little bit overvalued, which could be corrected by using a lower friction coefficient in subsequent simulations. Only the quality of the calculated z-force is not convincing, since it is much too high.

A simple explanation for this discrepancy could be the the program's inability to model ice breaking and plastic deformation of the particles. During the model test many rubble elements get compressed between the structure's roof and the consolidated layer. In the original test on the one hand the consolidated layer would yield to these forces and on the other hand the elements themselves would break and be deformed due to this compression. In the simulation no yield of the consolidated layer and no compression of the particles can be modeled, which leads to unrealistic high repulsive forces due to the bigger overlap volumes.

In the following pictures four snap shots taken during the model test are shown. Additional visual decumentation is provided on the enclosed DVD, on which three real time animations of the current simulation from different angles are available.

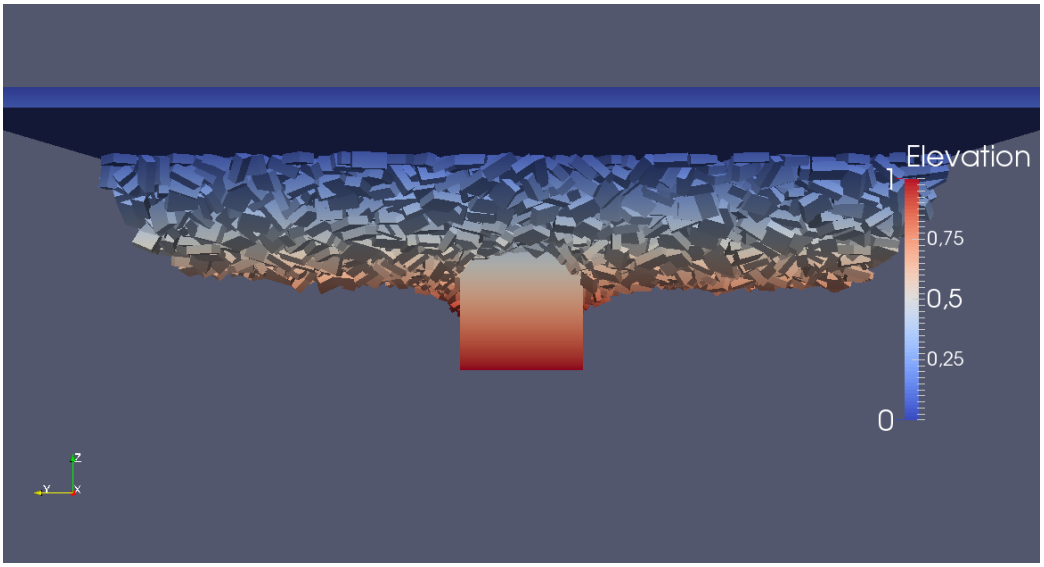


Figure 10.19: Hydralab III: View from behind

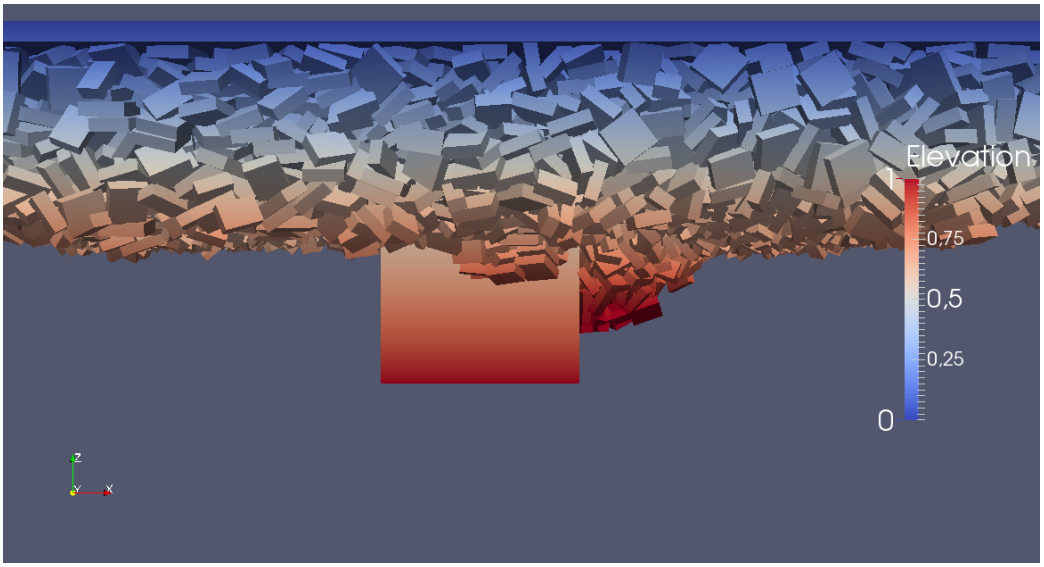


Figure 10.20: Hydralab III: View from the side

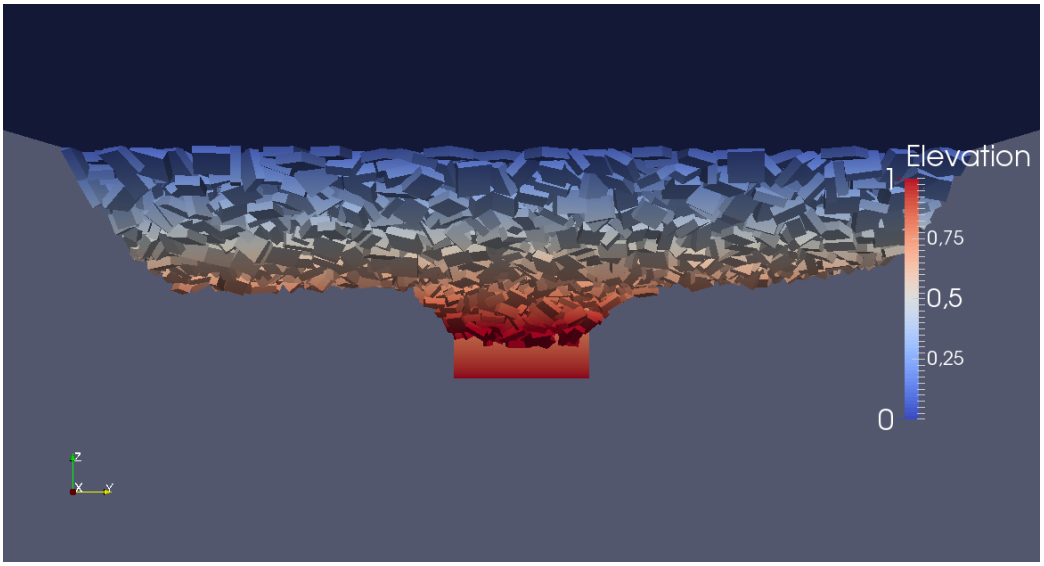


Figure 10.21: Hydralab III: View from front

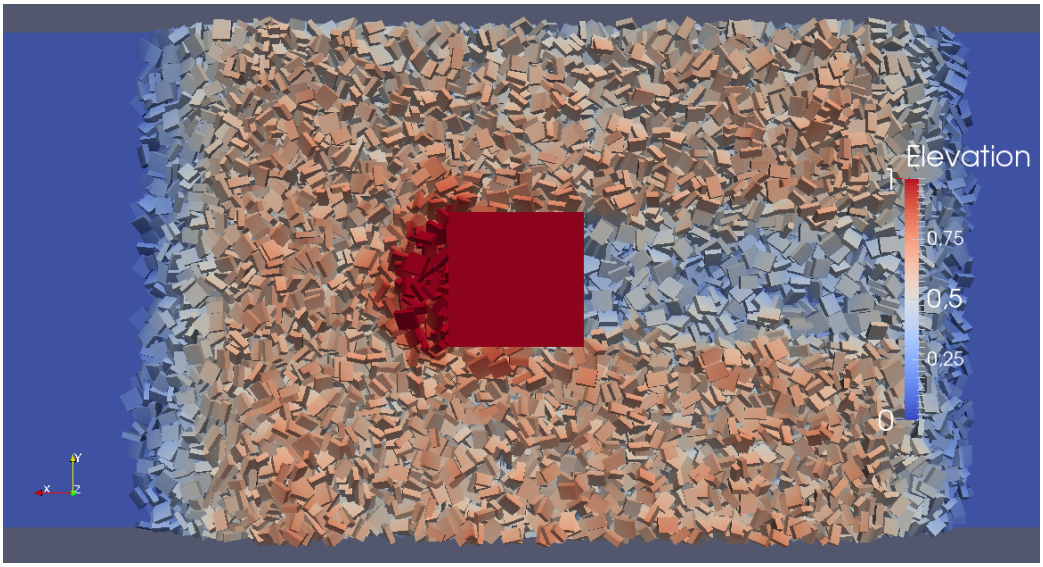


Figure 10.22: Hydralab III: View from bottom

Chapter 11

Conclusion and improvement suggestions

The author has developed a simulation tool based on a successful implementation of the Discrete Element Method for the approximation of the keel resistance of first-year pressure ice ridges on offshore structures. The important part in the current tool is a robust `OverlapComputation`-routine, which gives correct values for the overlap polyhedrons features as a basis for the force computation.

Since the implemented force model is orginally used for granular materials in the open air, it could need further enhancements. Especially when it comes to the calcultation of the hydrodynamic forces it is just rudymentary implemented.

11.1 Improvement suggestions for punch test

As discovered in section 10.2 no buoyancy force components act on the device, since the pushed-out heap collapses and leaves just a few elements on the device's bottom. Apart from this, the simulation tool represents the evolution of the force very well. Two improvement suggestions can be given:

Greater variation in element sizes A greater variation in the element sizes could support the heap of the pushed out pile, since then the void spaces are filled with smaller elements and hence a more stable packing could be obtained.

Adding bouyancy afterwards The easier way is to add afterwards the neglected buoyancy of the ice. This is calculated easily, since the shape and hence the volume of the pushed-out cluster can be estimated, and depending on the density of ice and water the resulting force is obtained.

11.2 Improvement suggestions for model test

The simulation represents the model test (besides the already discussed too high z-force) very well, wherefore it is only recommended to conduct more simulations to obtain better knowledge about the force parameters.

11.3 Outlook

Besides the already mentioned necessity of conducting more simulations to obtain tables for selection of force parameters (μ, c_{coh} and eventually γ_n), the main scope for subsequent improvement will be the implementation of more complex geometries.

With more complex geometries than just cubes, it is recommended to modify the source code to that effect, that the basis of each particle are triangle faces. This includes modifying mainly the data structure, the initialization and particle updating routines and a little bit the overlap computation. All other procedures like the force computation, neighbour finding and the prediction-correction subroutines will not be effected by this modification.

Having established more complex geometries, the breaking of the consolidated layer by surface piercing models could be taken into account by implementing empirical formulae, e.g. Lindqvist's approximation in [29].

After accomplishing this task, an important milestone in the development of the current tool is achieved, since then the majority of current model tests performed at the HSVA can be simulated.

For further improvement the programmer has to choose between two forks:

1. Coupling with Finite Element Methods to model breaking and bending of ice, or
2. coupling with Computational Fluid Dynamics to achieve better results for the hydrodynamic interaction of the discrete elements.

Bibliography

[1] K.V. Høyland. Physical modeling of first-year ice ridges - part 1: Production, consolidation and physical properties. 2011.

[2] Araon. Rv araon operational guidelines. 2004.

[3] J. Chen. *Discrete Element Method for 3D simulations of mechanical systems of non-spherical granular materials*. PhD thesis, The University of Electro-Communications, 2012.

[4] P. Wadhams. *Ice in the Ocean*. Gordon and Breach Science Publishers, 2000.

[5] D. Ehle. Analysis of breaking through sea ice ridges for development of a prediction method. Master's thesis, Universität Duisburg Essen, 2011.

[6] Q. Hisette. Simulation of ice management operations. In *Proc. of the 24th International Ocean and Polar Engineering Conference*, 2014.

[7] N. Serré and K.-U. Evers. Model testing of ridge keel loads on structures part i: Test set up and main results. In *Proc. of the 20th International Conference on Port and Ocean Engineering under Arctic Conditions*, 2009.

[8] M.A. Hopkins. Four stages of pressure ridging. *Journal of Geophysical Research: Oceans*, 103(C10):21883–21891, 1998. ISSN 2156-2202. doi: 10.1029/98JC01257. URL `http://dx.doi.org/10.1029/98JC01257`.

[9] A. Kovacs and M. Mellor. Sea ice morphology and ice as a geologic agent in the southern beaufort sea. In John C. Reed and John E. Sater, editors, *The coast and shelf of the Beaufort Sea: proceedings of*

a Symposium on Beaufort Sea Coast and Shelf Research. Arctic Insitute of North America, 1974. URL `https://books.google.de/books?id=HMbGAAAAIAAJ`.

[10] G.W. Timco and R.P. Burden. An analysis of the shapes of sea ice ridges. *Cold Regions Science and Technology*, 25(1):65 – 77, 1997. ISSN 0165-232X. doi: http://dx.doi.org/10.1016/S0165-232X(96)00017-1. URL `http://www.sciencedirect.com/science/article/pii/S0165232X96000171`.

[11] W.F. Weeks. *On Sea Ice*. University of Alaska Press, 2010.

[12] N. Serré. Numerical modelling of ice ridge keel action on subsea structures. *Cold Regions Science and Technology*, 67(3):107 – 119, July 2011.

[13] P.A. Cundall and O.D.L. Strack. A computer model for simulating progressive, large-scale movements in block rock systems. *Géotechnique*, 29(1):47 – 65, March 1979.

[14] E.H. Hansen and S. Løset. Modelling floating offshore units moores in broken ice: model description. *Cold Regions Science and Technology*, 29(2):97–106, August 1999.

[15] M.A. Hopkins. A discrete element lagrangian sea ice model. *Engineering Computations*, 21(2/3/4):409 – 421, 2004. ISSN 2156-2202. doi: 10.1029/98JC01257. URL `http://dx.doi.org/10.1029/98JC01257`.

[16] F.M. Puntigliano Etchart. *Experimental and Numerical Research on the Interaction Between Ice Floes and a Ship's Hull During Icebreaking*. PhD thesis, Technical University Hamburg-Harburg, 2003.

[17] A. Munjiza. *The Combined Finite-Discrete Element Method*. John Wiley & Sons, Ltd., 2004.

[18] A. Polojärvi. *Sea ice ridge keel punch through experiments: model experiments and numerical modeling with discrete and combined finite-discrete element methods*. PhD thesis, Aalto University, 2013.

[19] H.-G. Matuttis and J. Chen. *Understanding the Discrete Element Method: Simulation of Non-Spherical Particles for Granular and Multi-Body Systems*. John Wiley & Sons, Singapore Pte. Ltd., 2014.

[20] T. Pöschel and T. Schwager. *Computational Granular Dynamics*. Springer-Verlag Berlin Heidelberg, 2005.

[21] William Rowan Hamilton. On Quaternions; or on a new System of Imaginaries in Algebra, 1844-1850. URL `http://www.emis.de/classics/Hamilton/OnQuat.pdf`.

[22] C.W. Gear. *Numerical initial value problems in ordinary differential equations*. Prentice-Hall series in automatic computation. Prentice-Hall, 1971. URL `https://books.google.de/books?id=e9QQAQAAIAAJ`.

[23] M.P. Allen and D.J. Tildesley. *Computer simulation of liquids*. Oxford science publications. Clarendon Press, 1987.

[24] P.A. Cundall. Formulation of a three-dimensional distinct element model - part ii. mechanical calculations for motion and interaction of a system composed of many polyhedral blocks. *International Journal of Rock Mechanics and Mining Sciences & Geomechanics Abstracts*, 25(3): 117 – 125, 1988.

[25] Pierre Terdiman. sweep-and-prune, 2007. URL `http://www.codercorner.com/SAP.pdf`.

[26] ITTC. Fresh water and seawater properties. In *ITTC - Recommended Procedures*, 2011.

[27] F.A. Morrison. *An Introduction to Fluid Mechanics*. Cambridge University Press, 2013.

[28] E.L. Nichols and W.S. Franklin. *The Elements of Physics*. Macmillan, 1898.

[29] G. Lindqvist. A straightforward method for calculation of ice resistance of ships. In *Port and Ocean Engineering under Arctic Conditions*, 1989.