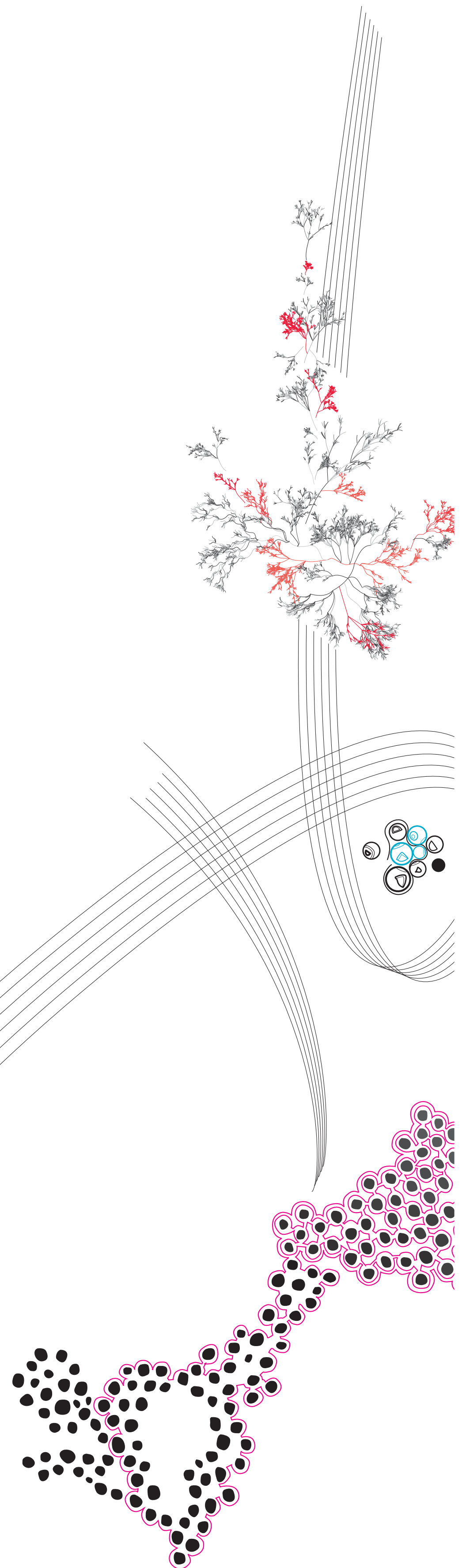


MSc Mechanical Engineering
Thesis

Simulating evolving surfaces due to wear in granular processes

Jan-Willem Bisschop

Supervisors: dr. T. Weinhart
prof.dr. A.R. Thornton

September, 2024

Department of Thermal and Fluid Engineering
Faculty of Engineering Technology
University of Twente

UNIVERSITY OF TWENTE.

Preface

I would like to thank my supervisors Thomas Weinhart and Anthony Thornton for the weekly meetings, the fruitful discussions, and the helpful comments. I further want to express my gratitude to David Pinson, who, together with Anthony, came up with the idea for this project. Due to time constraints the number of meetings was less than I would have liked (and also sort of abruptly ended), but they were very useful nonetheless and gave great inspiration on how to approach things. The final simulations were made using a case study David provided, unfortunately missing the validation with real life, due to the same time constraints.

Summary

Granular materials are used in many industries around the globe and in many cases the machinery will wear down significantly over time. On time inspections are needed to prevent catastrophic failure, but these are often costly due to hard to reach places and the need to shut down (parts of) a continuous production line. Any insights in understanding or possibly even predicting the wear is valuable.

Particle simulations have been proven successful in studying granular materials. They have also been used to study surface wear, however, so far this has been limited to predicting the amount of wear while keeping the surface geometry unchanged. That is, the surface does not actively evolve due to the wear and this project aims to change that. The open-source particle simulation code MercuryDPM, developed at the University of Twente, is used for this. MercuryDPM uses the Discrete Particle Method (DPM), also known as the Discrete Element Method (DEM), to simulate the particles at discrete steps in time. Each time step Newton's second law of motion is solved for each particle, where the sum of the forces is calculated from the interactions with other particles and walls. From this the acceleration is obtained and using a time integration scheme the velocity and position of each particle is updated.

For this project the Archard wear model is used, which calculates the debris due to abrasive wear. Other forms of wear exist, such as erosive and cavitation wear, but the focus was kept to one. The implementation is not dependent on the type of wear, so other wear models can easily be added in the future.

A surface that is able to evolve must have some sort of underlying grid to allow for local control of the shape. For each particle-wall interaction, the debris is calculated and assigned to one or multiple points on the grid. When all interactions have been handled the surface shape is updated.

MercuryDPM has two types of surfaces which are able to locally evolve, namely the NurbsWall and the TriangleWall. The latter is a single triangle, which can be part of a full mesh of triangles forming a surface. The vertices of the triangle mesh can then be moved to evolve it. The former uses Non-Uniform Rational B-Splines (NURBS), of which the basic idea is that a number of points in 3D space, called control points, pull on the surface and in that way shape it. The control points themselves do not lie on the surface. They do have a weight property, which increases or decreases the amount the surface is pulled towards it. By moving the control points, the shape of the surface can be controlled. The choice was made to base the wearable surface on the NurbsWall, as it able to create smooth surfaces and is already fully implemented in MercuryDPM. As opposed to triangle meshes, which are not smooth, have a small numerical error on particle contact on a shared triangle edge, and require more development in MercuryDPM to make the TriangleWall part of a evolvable mesh.

Unfortunately, it turned out the MercuryDPM implementation of NURBS was lacking in a lot of aspects and needed quite a bit of improvements. After that, some results were obtained by simplifying the problem down to a quasi-2D surface made from a grid of control points in xy-plane at $z = 0$ and which only moved down in z-direction. The change in volume due to the surface evolving must be equal to the total debris, which was calculated by triangulating the surface and calculating the volume under the surface with the $z = 0$ plane. The reason for triangulation is that no analytical method for calculating the volume under a NURBS surface was found, nor were we able to derive it ourselves. Extension to proper 3D was therefore very hard as well and might not even be possible, because the volume difference between two NURBS surfaces is an even harder problem to solve. All in all, NURBS turned out to not work as well as anticipated and together with the fact that the MercuryDPM implementation, although improved a lot, still was not where

it should be, the choice was made to try triangle meshes instead.

A mesh made from TriangleWalls was developed and the implementation of the wear was quite straightforward by assigning the debris to the three vertices of a triangle by means of barycentric interpolation. Moving the vertices to account for a change in volume was harder than anticipated, as it turned out to not be analytically possible. Instead, the vertices were moved iteratively until the correct volume change was obtained. The simulated wear was visually very comparable to real life, with a higher number of particle-wall interactions and a higher load resulting in more severe wear. Still, many improvements can be made, for example, changing the wall description from a (triangulated) surface mesh to a (tetrahedral) volumetric mesh, which by default would have a certain thickness to it. It would be able to form holes naturally, as opposed to the infinitely thin triangle mesh, which can wear down infinitely in both directions.

Wear in real life is usually very slow and we prefer to have results within a reasonable time span. It is therefore needed to accelerate the wear, however there is a limit to how high the wear coefficient can be. The amount of wear relative to the wear coefficient used, is lower for a higher wear coefficient. This has most likely to do with the normal force of an interaction being reduced more, the more the surface is moved away from it. Furthermore, for the triangle mesh implementation, at some point the iterative process of moving the vertices to match a change in volume simply fails to converge, or is very slow.

It can be concluded that there is potential in using DEM to simulate surface wear with actively evolving surfaces. This can be used to predict and understand the wear in more detail, as well as how the granular flow is affected by it, which in turn can lead to more wear at the same or at other places. NURBS were not as suitable as initially thought. Triangle meshes seem like the best way to go forward, but there are still things to improve. A lot of verification and validation is still needed.

Contents

Preface i

Summary ii

List of acronyms vi

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Research goal and questions . . . 1
- 1.3 Report structure . . . 2

2 Discrete Element Method 3

3 Method 5
- 3.1 Wear model . . . 5
- 3.2 Calculating and processing wear . . . 5
- 3.3 Geometries . . . 6

4 NURBS 7
- 4.1 Introduction to NURBS . . . 7
  - 4.1.1 Splines . . . 7
  - 4.1.2 B-splines . . . 8
  - 4.1.3 Rational B-splines . . . 13
  - 4.1.4 Non-uniform rational B-splines . . . 13
  - 4.1.5 NURBS surface . . . 16
- 4.2 Improving MercuryDPM implementation . . . 17
- 4.3 Wearing . . . 19
- 4.4 Limitations . . . 21

5 Triangles 22
- 5.1 TriangleMeshWall . . . 22
  - 5.1.1 Mesh . . . 22
  - 5.1.2 Periodic boundaries . . . 23
  - 5.1.3 Contact detection . . . 24
  - 5.1.4 Bounding box . . . 25
- 5.2 Wearing . . . 26
  - 5.2.1 Computing the wear . . . 26
  - 5.2.2 Evolving the mesh . . . 26
- 5.3 Limitations . . . 27
- 5.4 Possible improvements . . . 28

6 Simulations 30
- 6.1 Number of recursive calls . . . 31
- 6.2 Number of warning messages . . . 32
- 6.3 The wear over time . . . 32

6.4 Conclusion 35
6.5 Recommendation 35

7 Conclusions and recommendations **36**
7.1 Conclusions 36
7.2 Recommendations 36

References **37**

A Source code **38**

B NURBS **40**
B.1 piecewise_polynomials.py 40
B.2 basis_functions.py 41
B.3 basis_splines.py 45
B.4 rational_basis_splines.py 49
B.5 non_uniform_rational_basis_splines.py 52

C Wall details VTK **56**
C.1 Vtu file structure 56
C.2 VTKData class 57
C.3 General VTK writers structure and usage 58
C.4 Wall details 58
C.5 NURBS control points wall details 60

D TriangleMeshWall **63**
D.1 Writing to VTK 63
D.2 Moving vertices to match a change in volume 63
D.3 Creating a parallelogram mesh 64
D.4 Creating a four point mesh 65
D.5 Adding a mesh to a mesh 65

List of acronyms

DEM Discrete Element Method
DPM Discrete Particle Method
NURBS Non-Uniform Rational B-Splines

Chapter 1

Introduction

1.1 Background

Granular materials are essential in many industries, such as mining, steelmaking, food, pharmaceutical, etc. In many applications, the machinery will wear down over time due to the granular material interacting with it. It is important to keep track of this and to replace (parts of) the machinery in time. It is not always easy to inspect the machinery, due to hard to reach places and the need for complete shut downs. The latter especially is not preferred in a time of continuous production lines. One turned off machine means the complete production line is out the running, making it a very costly process. Waiting too long before doing inspections, however, can be even more costly, as it can lead to catastrophic failures. Getting a better understanding of how, where, and how fast wear occurs can help in decision making. Particle simulations, which have been used to study and improve many granular processes in the past, can help with that.

There are numerous particle simulation codes in existence today. Some of them are proprietary and others are open-source. One highly regarded particle simulation code is the open-source code MercuryDPM [10]. It is developed at the University of Twente and has been proven very successful in many projects since its development started in 2009. It is still actively maintained and developed, with many new features regularly being added, some of which contributed by this project.

1.2 Research goal and questions

The goal of this project is to actively evolve a surface, due to the wear of particles interacting with it. The *active feedback* is key here, as it can impact the granular flow and thus possibly the production process. A changing granular flow could also cause the wear to increase, decrease, or start at areas otherwise unaffected. Particle simulations have been used in the past by others to study surface wear and its effects, important properties, etc. [13][1][2][4]. Actively evolving the surface has, to the author's knowledge and at the time of this project, rarely been tried before. Kalala, Bwalya, and Moys [5] show the potential of evolving surfaces by simulating the evolution of mill liner profiles.
The main research question is:

- Can particle simulations be used to simulate evolving surfaces due to the wear caused by granular materials?

With the following sub research questions:

- How can a surface capable of evolution be implemented in MercuryDPM?
- How can the simulation of wear be accelerated, in order to get results within a reasonable time span?
- How can the implementation be validated?

1.3 Report structure

Chapter 2 gives a brief explanation on the basics of the Discrete Element Method (DEM), the fundamental concept in particle simulations.
Chapter 3 tells a bit about wear models and what the general structure of an implementation will look like. It then discusses the different options for evolving a wall in MercuryDPM and the decision on which option to use.
Chapter 4 goes in depth about the implementation of a wearable surface using Non-Uniform Rational B-Splines (NURBS). First, a brief introduction of the mathematical concept of NURBS is given. Then, a number of improvements made to the existing NURBS implementation in MercuryDPM are shown. Next, the implementation of the wear and the evolution of the surface. Finally, the limitations that this approach brings.
Chapter 5 shows how a wearable surface can be implemented using triangle meshes. It starts by showing the implementation of a triangle mesh. Then, it talks about the wear and surface evolution. Finally, the limitations and possible improvements of this approach are shown.
Chapter 6 shows the result of a few simulations, using the triangle mesh as a wearable surface, and does some analysis of the results.
Chapter 7 makes conclusions based on this work and gives recommendations for future research.

Appendix A gives instructions on how to find all publicly available MercuryDPM coding work of this project. It also includes instructions on how to install, build, and run the code.
Appendix B shows all python scripts used to make the NURBS plots in Chapter 4.
Appendix C gives detailed information about the implementation of a new tool in MercuryDPM, useful for visualisation of certain additional wall properties, such as the NURBS control points.
Appendix D explains a few useful functions of the new triangle mesh in MercuryDPM.

Chapter 2

Discrete Element Method

Here a very simple explanation of the Discrete Element Method (DEM), also known as the Discrete Particle Method (DPM), is given. It is meant to be simple and intuitive to understand, and does not go in depth about mathematical details.

DEM simulates particles by taking discrete steps through time. For every time step the forces reacting on each particle are calculated and the particle's velocity and position are updated. This is done using Newton's second law of motion, where the sum of the forces equals the mass times the acceleration: $\sum F = ma$. The mass of the particle is known and the sum of the forces has just been calculated, which gives us the acceleration. The velocity and position can be calculated using a time integration scheme, which at its core takes just a few simple steps

$$\begin{aligned} a_i &= \frac{\sum F}{m} \\ v_i &= v_{i-1} + a_i \Delta t \\ x_i &= x_{i-1} + v_i \Delta t \end{aligned} \tag{2.1}$$

where Δt is the time step, i indicates the value at the current time step, and $i-1$ indicates the value at the previous time step. More involved time integration schemes exist; differing in numerical stability and accuracy. MercuryDPM uses the Velocity-Verlet algorithm[9].

A fundamental assumption used in DEM is that of infinitely rigid particles. When two particles collide, rather than deforming, they are allowed to have some overlap. The overlapping distance, denoted by δ, is then used to calculate the reaction force. See Figure 2.1 for an illustration of two particles overlapping, as well as a particle overlapping with a wall. The linear spring dashpot model, for example, sees the overlap as the amount a spring is compressed and calculates the reaction force as $F = k\delta$, where k is the spring constant or particle stiffness. The damping force is calculated as $F = \gamma\dot{\delta}$, where γ is the damping coefficient and $\dot{\delta}$ is the relative normal velocity between the two particles.

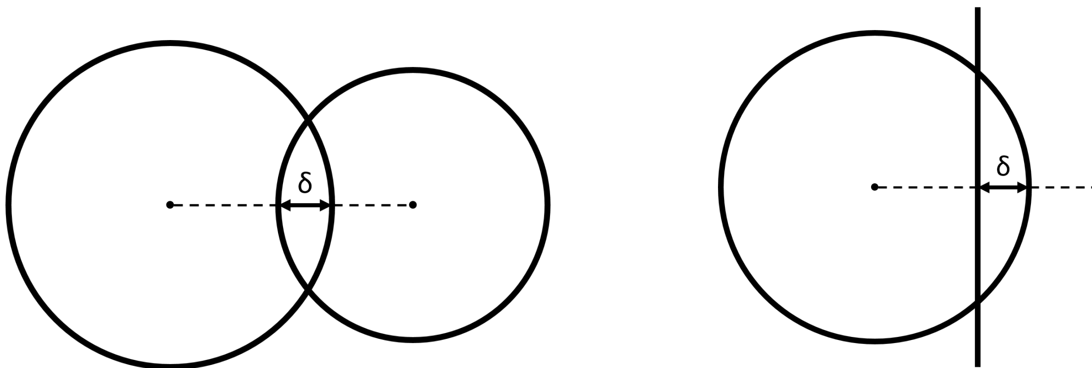


Figure 2.1: The overlap between two particles (left) and between a particle and a wall (right).

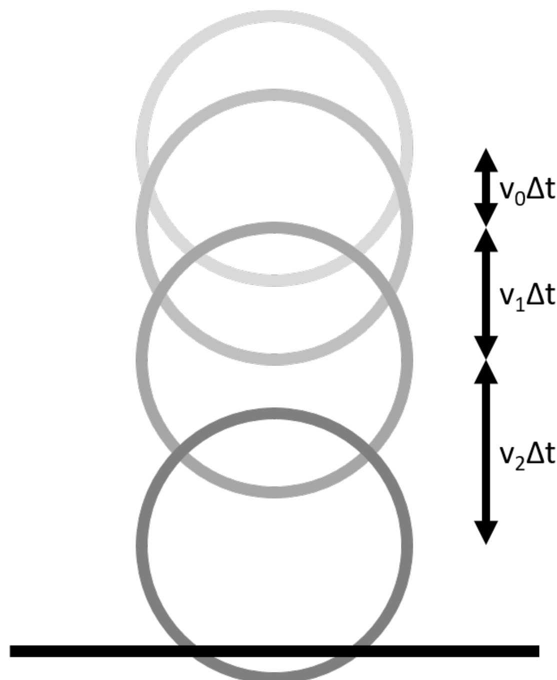


Figure 2.2: Example illustration of a particle falling due to gravity. The discrete steps through time are using the basic time integration scheme from Eq. 2.1.

Figure 2.2 illustrates a particle taking discrete steps through time, using the basic time integration scheme from Eq. 2.1. As one can imagine, the value for the time step is very important. When it is too big, it will cause unreasonably high overlaps and therefore unreasonably high reaction forces. This will cause the particles to chaotically shoot off to infinity, which is obviously not an accurate representation of a real life system. Setting it too small will make sure that it is accurate, but it will also slow the simulation down. This quickly becomes significant when simulating hundreds of thousands, or even millions, of particles at once. We want to set the time step as high as possible, while still being physically accurate.

Not too much detail will be shown here, but it can be derived that a good value for the time step is the collision time divided by 50. The collision time is the total duration of the contact for a simple head-on collision of two particles. In other words, the collision is resolved in 50 time steps. Initially the overlap will be small, but it increases with every time step. So does the reaction force, increasing the acceleration in the direction opposite to which the particle travels. The acceleration will reduce the velocity by an increasing amount, until it flips direction. The particles start moving away from each other, in turn reducing the overlap, reaction force, and acceleration. Finally, the particles are not in contact anymore and move away from each other with constant velocity.

Chapter 3

Method

3.1 Wear model

There are multiple different types of wear, such as abrasive wear, where particles slide over a surface; erosive wear, caused by the impact of particles on a surface; and cavitation wear, caused by the shock wave from collapsing bubbles formed by local pressure differences in a fluid [7]. This project will focus on abrasive wear, using the Archard wear model [11], however it is coded in such a way that other wear models can be implemented as well. The Archard wear model states that

$$Q = \frac{KWL}{H} \tag{3.1}$$

where $Q(m^3)$ is the volume of the debris, $K(-)$ is a wear coefficient indicating the severity of the wear, $W(N)$ is the normal load, $L(m)$ is the sliding distance and $H(Pa)$ is the hardness of the softest surface in contact. In our case the hardness would always be that of the wall, as the wear of particles is not taken into account. Measuring the hardness should be done with, for example, the Brinell or Vickers hardness test, as they have the correct units (Pa). However, as long as one is consistent in its use, it probably does not matter much which hardness scale is used. The wear usually evolves very slow, therefore the volume of the debris is multiplied by a wear acceleration constant, in order to speed up the simulations. This constant and the constants K and $1/H$ are multiplied together, so they could be seen as a single constant. Often when talking about the wear coefficient, this umbrella term is meant instead.

3.2 Calculating and processing wear

After MercuryDPM has calculated all forces on a wall, a virtual function computeWear is called, which can be overridden to implement a certain wear model and handle the evolution of the wall. A wear model returns the volume that has to be removed at a certain point. Depending on the implementation, this can be a point on an underlying grid, or a random point that then has to be snapped to that grid. The latter can be done by picking the closest grid point or by taking a weighted average over multiple grid points. Once all wear has been calculated and assigned to the grid, it is used to evolve the wall to match the worn down debris.

In this work the wall interactions are used to calculate the debris caused by each particle. For each interaction the wear is calculated, where interactions of 'ghost' particles are ignored, as well as old interactions, since the computeWear method is called at a time where these have not yet been removed. W is the absolute normal force of the interaction. Using the relative velocity and the normal relative velocity, the tangential relative velocity is calculated and after multiplying by the time step gives the sliding distance L. All other variables are constants which values have been specified beforehand and thus Q can be calculated, after which it is assigned to one or more points in the grid.

3.3 Geometries

Geometries can be described precisely by a mathematical equation, or can be approximated by a mesh of triangles. MercuryDPM has many mathematically described geometries, however most of them are targeting a specific use case and are not very suitable for (local) changes is shape, for example a screw geometry for screw conveyors. Only one geometry has general use cases and that is the NurbsWall, made with NURBS. It is able to (locally) evolve and can be perfectly smooth.

This is in contrast to triangle meshes, which can only approximate a certain smoothness by using more and more triangles, slowing the code down. The non-smoothness is generally not a problem, as the effect on the overall particle flow is insignificant. However, the focus now is on the wall and the forces and evolution it experiences, so a smooth surface might be beneficial. There are ways used in computer graphics to smooth out triangle meshes, such as Gouraud shading and Phong shading. The latter calculates the normal at a point on the triangle by linearly interpolating between the vertex normals, which themselves are the average of the face normals of all triangles sharing that vertex. This might have potential use cases for DEM as well, but this has not been further researched.

In addition to triangle meshes not being smooth, a particle in contact with the shared edge of two triangles will not experience the correct resulting forces. This is due to the fact that MercuryDPM does not handle overlapping contact areas very well and will handle the same (partial) overlap multiple times, see Figure 3.1. This can be compared to a particle in contact with two coinciding walls and the overlap and reaction force being computed for each wall, instead of just one. Again, for the overall particle flow this is usually not an issue, but for the wear it might be. It has later been fixed with a so-called 'triangle wall correction', however at the time of making the decision on which geometry to use, this was not the case.

A benefit of triangle meshes is that they can be read directly from STL files, which are used a lot in industry. NURBS can also be read from different file formats, but MercuryDPM does not support that yet. Another difference in the MercuryDPM implementation is that for the NurbsWall less changes are expected to be needed, since for example moving a control point should be relatively easy to do. The TriangleWall, however, exist on its own for each triangle and not as a mesh. So updates to the mesh, e.g. moving a vertex, require the implementation of a new approach for handling triangles. The difference in the amount of additional work seem significant.

Initially, the choice was made to use NURBS as the basis for the wearable geometry. However, during the project we switched back to triangle meshes, as NURBS turned out to be more problematic than anticipated.

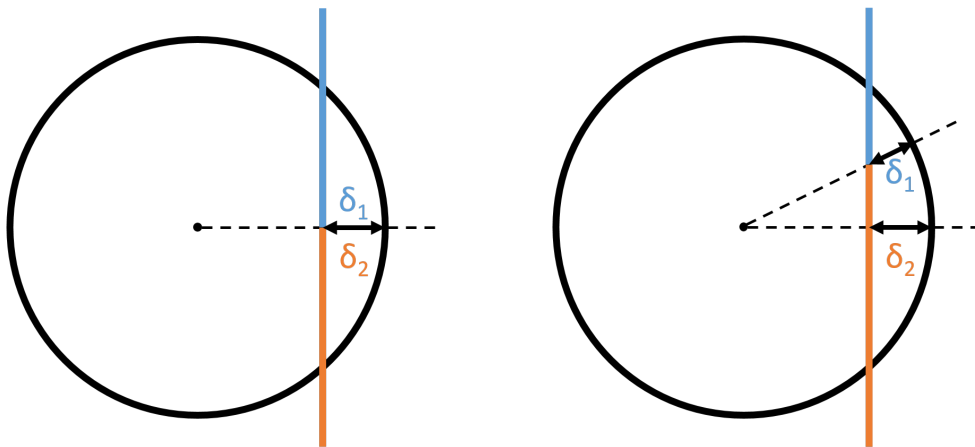


Figure 3.1: Two examples of a particle in contact with two triangles at the same time. Instead of a single overlap δ with the wall, there are two overlaps δ_1 and δ_2.

Chapter 4

NURBS

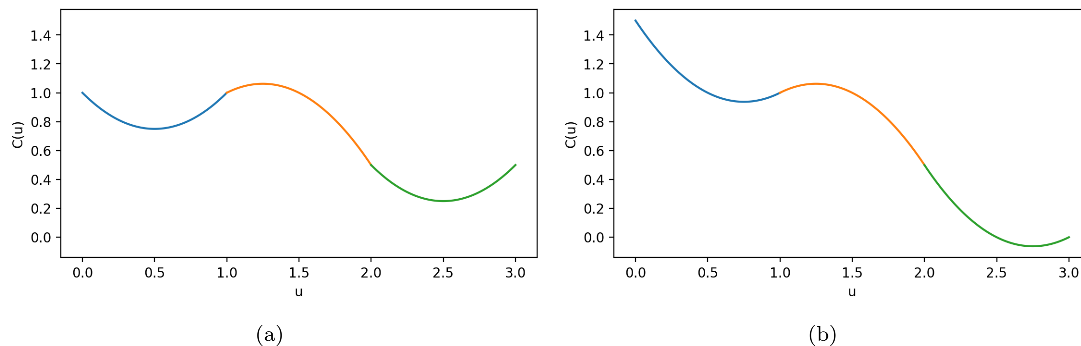
4.1 Introduction to NURBS

This is a brief introduction meant to give an intuitive feel about Non-Uniform Rational B-Splines. The focus is on curves only (one parameter u), with surfaces (two parameters u and v) being more mathematically involved but in principal the same. To go more in depth we refer the reader to The NURBS Book [6], the NURBS Wikipedia page [12], and the online NURBS calculator [3]. The latter especially is useful to play around with different properties and see how that affects the resulting curve. Furthermore, the python scripts to create all plots in this section can be found in Appendices B.1 to B.5. The following sections will explain each term of the NURBS acronym.

4.1.1 Splines

Piecewise polynomial functions are a way to define curves by piecing multiple short segments of different polynomials together. A simple example can be seen in Figure 4.1(a), where the curve is defined as

$$C(u) = \begin{cases} u^2 - u + 1, & 0 \le u < 1 \\ -u^2 + 2.5u - 0.5, & 1 \le u < 2 \\ u^2 - 5u + 6.5, & 2 \le u < 3 \end{cases} \tag{4.1}$$



(a) (b)

Figure 4.1: Piecewise polynomial functions with C^0 (a) and C^1 (b) continuity. Generated with script in Appendix B.1.

The extremes of the segments are called breakpoints, here at $u = 0$, $u = 1$, $u = 2$, and $u = 3$. Continuity at the breakpoints is often an important consideration. This curve is continuous at the (internal) breakpoints, since $C_0(1) = C_1(1)$ and $C_1(2) = C_2(2)$, or generally $C_i(u_{i+1}) = C_{i+1}(u_{i+1})$, where u_i and $C_i(u)$ are the ith breakpoint and polynomial segment, respectively. This

is known as C^0 continuity (not to be confused with the curve $C(u)$), since the zeroth derivative, i.e. the position, is equal. In general, C^k continuity is when $C_i^{(j)}(u_{i+1}) = C_{i+1}^{(j)}(u_{i+1})$ for all $0 \leq j \leq k$, where $C_i^{(j)}(u)$ denotes the jth derivative of $C_i(u)$. A slightly different curve with C^1 continuity, i.e. both position and tangent are equal, is shown in Figure 4.1(b) and is defined as

$$C(u) = \begin{cases} u^2 - 1.5u + 1.5, & 0 \leq u < 1 \\ -u^2 + 2.5u - 0.5, & 1 \leq u < 2 \\ u^2 - 5.5u + 7.5, & 2 \leq u < 3 \end{cases} \tag{4.2}$$

The degree of a curve is equal to the highest degree of the polynomials. Even if one of the polynomials in this example would not have be quadratic, the degree would still be 2. The highest possible continuity cannot exceed the degree.

A **spline** defines how to calculate the piecewise polynomial functions using control points and other criteria. The other criteria differ for different types of splines and have to do with, for example, the continuity, whether or not the curve should pass through the control points, and computation speed for the evaluation of the curve. The polynomial segments are parametric functions, usually normalised so the domain over all segments is $[0, 1]$. For splines, the breakpoints are usually called knots and are stored in a knot vector in non-decreasing order. Knots in a knot vector can be repeating, but when talking about actual breakpoints, only the distinct knot values are meant. The polynomial segments are defined the by the knot spans of non-zero length.

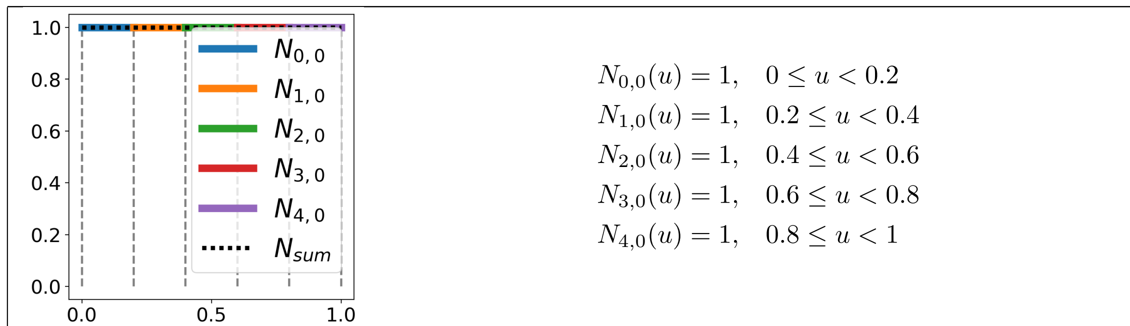
4.1.2 B-splines

A **B-spline**, or basis spline, is a spline based on basis functions. Given a knot vector and a degree, the basis functions can be computed using the recursive formula from Mansfield, de Boor, and Cox [6]

$$\begin{aligned} N_{i,0}(u) &= \begin{cases} 1, & u_i \leq u < u_{i+1} \\ 0, & \text{otherwise} \end{cases} \\ N_{i,p}(u) &= \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u) \end{aligned} \tag{4.3}$$

where $N_{i,p}(u)$ is the ith basis function of degree p. Each basis function of degree p is a linear combination of two basis functions of degree $p - 1$. The recursion ends at degree 0, for which the basis function is a step function. For every incrementing degree, one less basis function exist and each basis function is defined over one more knot span.

As an example, take the knot vector $U = [0, 1, 2, 3, 4, 5]$ and normalise it so it is on the domain $[0, 1]$: $U = [0, 0.2, 0.4, 0.6, 0.8, 1]$. There are five knot spans and therefore five basis functions for degree 0, four for degree 1, three for degree 2, two for degree 3, and one for degree 4. Table 4.1 shows an overview of the basis functions per degree. A B-spline should only be evaluated on knot spans for which the basis functions add up to 1, i.e. they are fully supported. In fact, the first and last p knot spans are never fully supported, while the other 'internal' knot spans are.



$$N_{0,1}(u) = \begin{cases} 5u, & 0 \leq u < 0.2 \\ -5u + 2, & 0.2 \leq u < 0.4 \end{cases}$$

$$N_{1,1}(u) = \begin{cases} 5u - 1, & 0.2 \leq u < 0.4 \\ -5u + 3, & 0.4 \leq u < 0.6 \end{cases}$$

$$N_{2,1}(u) = \begin{cases} 5u - 2, & 0.4 \leq u < 0.6 \\ -5u + 4, & 0.6 \leq u < 0.8 \end{cases}$$

$$N_{3,1}(u) = \begin{cases} 5u - 3, & 0.6 \leq u < 0.8 \\ -5u + 5, & 0.8 \leq u < 1 \end{cases}$$

$$N_{0,2}(u) = \begin{cases} 12.5u^2, & 0 \leq u < 0.2 \\ -25u^2 + 15u - 1.5, & 0.2 \leq u < 0.4 \\ 12.5u^2 - 15u + 4.5, & 0.4 \leq u < 0.6 \end{cases}$$

$$N_{1,2}(u) = \begin{cases} 12.5u^2 - 5u + 0.5, & 0.2 \leq u < 0.4 \\ -25u^2 + 25u - 5.5, & 0.4 \leq u < 0.6 \\ 12.5u^2 - 20u + 8, & 0.6 \leq u < 0.8 \end{cases}$$

$$N_{2,2}(u) = \begin{cases} 12.5u^2 - 10u + 2, & 0.4 \leq u < 0.6 \\ -25u^2 + 35u - 11.5, & 0.6 \leq u < 0.8 \\ 12.5u^2 - 25u + 12.5, & 0.8 \leq u < 1 \end{cases}$$

$$N_{0,3} = \begin{cases} 20\frac{5}{6}u^3, & 0 \leq u < 0.2 \\ -62.5u^3 + 50u^2 - 10u + \frac{2}{3}, & 0.2 \leq u < 0.4 \\ 62.5u^3 - 100u^2 + 50u + 7\frac{1}{3}, & 0.4 \leq u < 0.6 \\ -20\frac{5}{6}u^3 + 50u^2 - 40u - 10\frac{2}{3}, & 0.6 \leq u < 0.8 \end{cases}$$

$$N_{1,3} = \begin{cases} 20\frac{5}{6}u^3, & 0.2 \leq u < 0.4 \\ -62.5u^3 + 87.5u^2 - 37.5u + 5\frac{1}{6}, & 0.4 \leq u < 0.6 \\ 62.5u^3 - 137.5u^2 + 97.5u - 21\frac{5}{6}, & 0.6 \leq u < 0.8 \\ -20\frac{5}{6}u^3 + 62.5u^2 - 62.5u + 20\frac{5}{6}, & 0.8 \leq u < 1 \end{cases}$$

$$N_{0,4} = \begin{cases} 26\frac{5}{12}u^4, & 0 \leq u < 0.2 \\ -104\frac{1}{6}u^4 + 104\frac{1}{6}u^3 - 31.25u^2 + 4\frac{1}{6}u - \frac{5}{24}, & 0.2 \leq u < 0.4 \\ 156.25u^4 - 312.5u^3 + 218.75u^2 - 62.5u + 6\frac{11}{24}, & 0.4 \leq u < 0.6 \\ -104\frac{1}{6}u^4 + 312.5u^3 - 343.75u^2 + 162.5u - 27\frac{7}{24}, & 0.6 \leq u < 0.8 \\ 26\frac{1}{24}u^4 - 104\frac{1}{6}u^3 + 156.25u^2 - 104\frac{1}{6}u + 26\frac{1}{24}, & 0.8 \leq u < 1 \end{cases}$$

Table 4.1: Basis functions for the knot vector $U = [0, 0.2, 0.4, 0.6, 0.8, 1]$. From top to bottom, the degrees from 0 to 4. For clarity, the equations and the plots are only shown for where they are non-zero. Also plotted is the sum of the basis functions, which add up to 1 for knot spans which are fully supported. Generated with script in Appendix B.2.

A B-spline curve is defined as

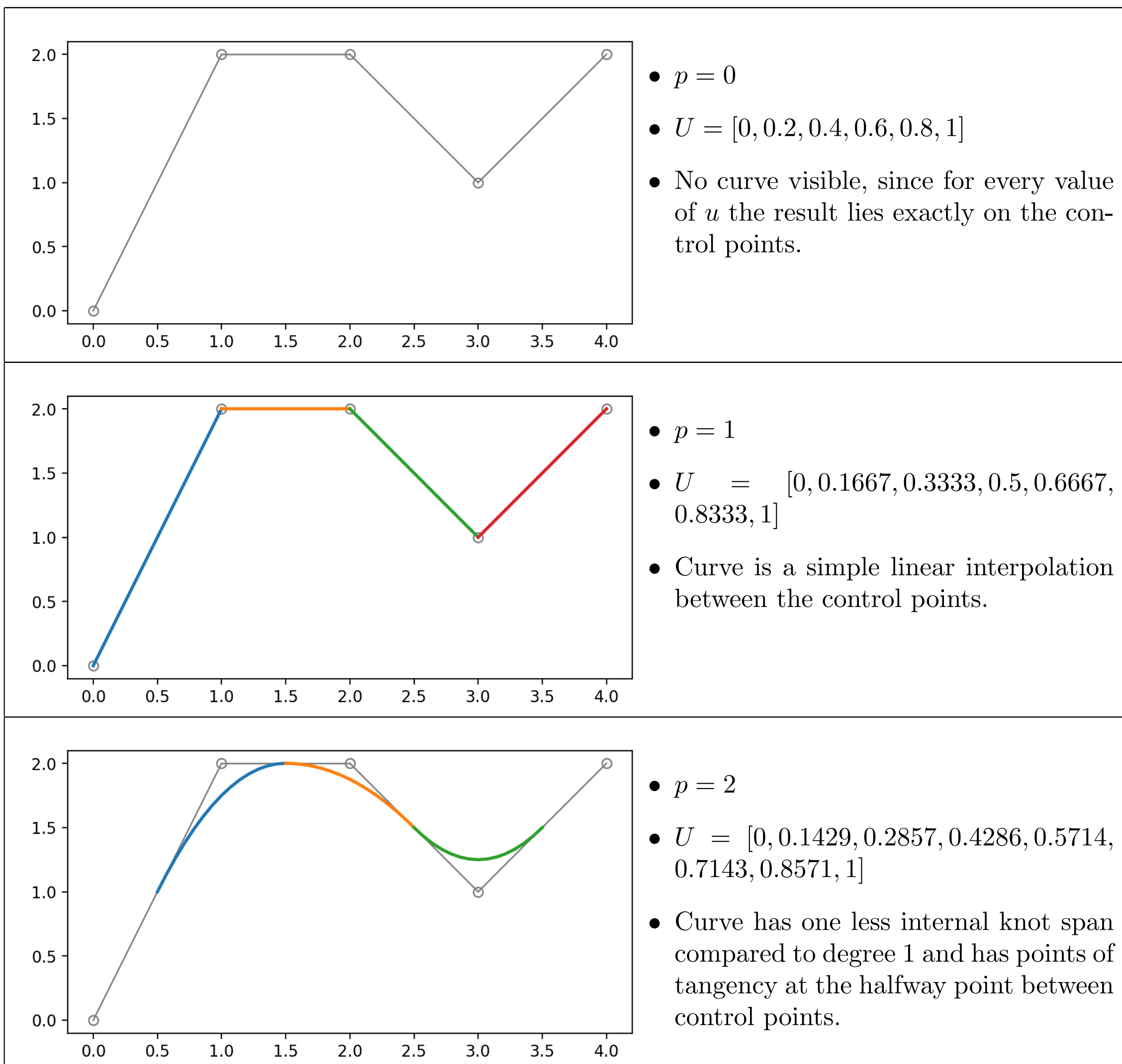
$$C(u) = \sum_{i=0}^{n} N_{i,p}(u)\boldsymbol{P}_i \tag{4.4}$$

where $N_{i,p}(u)$ is the ith basis function of degree p and $\boldsymbol{P}_i$ is the ith control point. Clearly, the number of basis functions must be the same as the number of control points. This means that

when a new control point is added, the knot vector must also get an additional value. The same is true for when the degree is incremented, since otherwise the number of basis functions would be reduced by one, as previously explained. In fact, the relation is: number of knots = number of control points + degree + 1. Usually the control points and the degree are given inputs, and the knot vector is created accordingly, e.g. uniformly spaced between 0 and 1. It is not the knot values themselves, but rather the ratios between the knot span lengths, that affect the shape of the curve. More about this in Section 4.1.4.

For $p > 1$ the curve will not go through any control points. In some cases it might happen, but it is not strictly mathematically defined. The exception being when a curve is 'clamped', in which case the first and last control point will be the start and end point of the curve. Clamping occurs when the not fully supported knot spans have zero length and can be done separately at the start and the end. Clamping at the start is done by setting the first $p + 1$ knots equal to the first knot, and vice versa for clamping at the end.

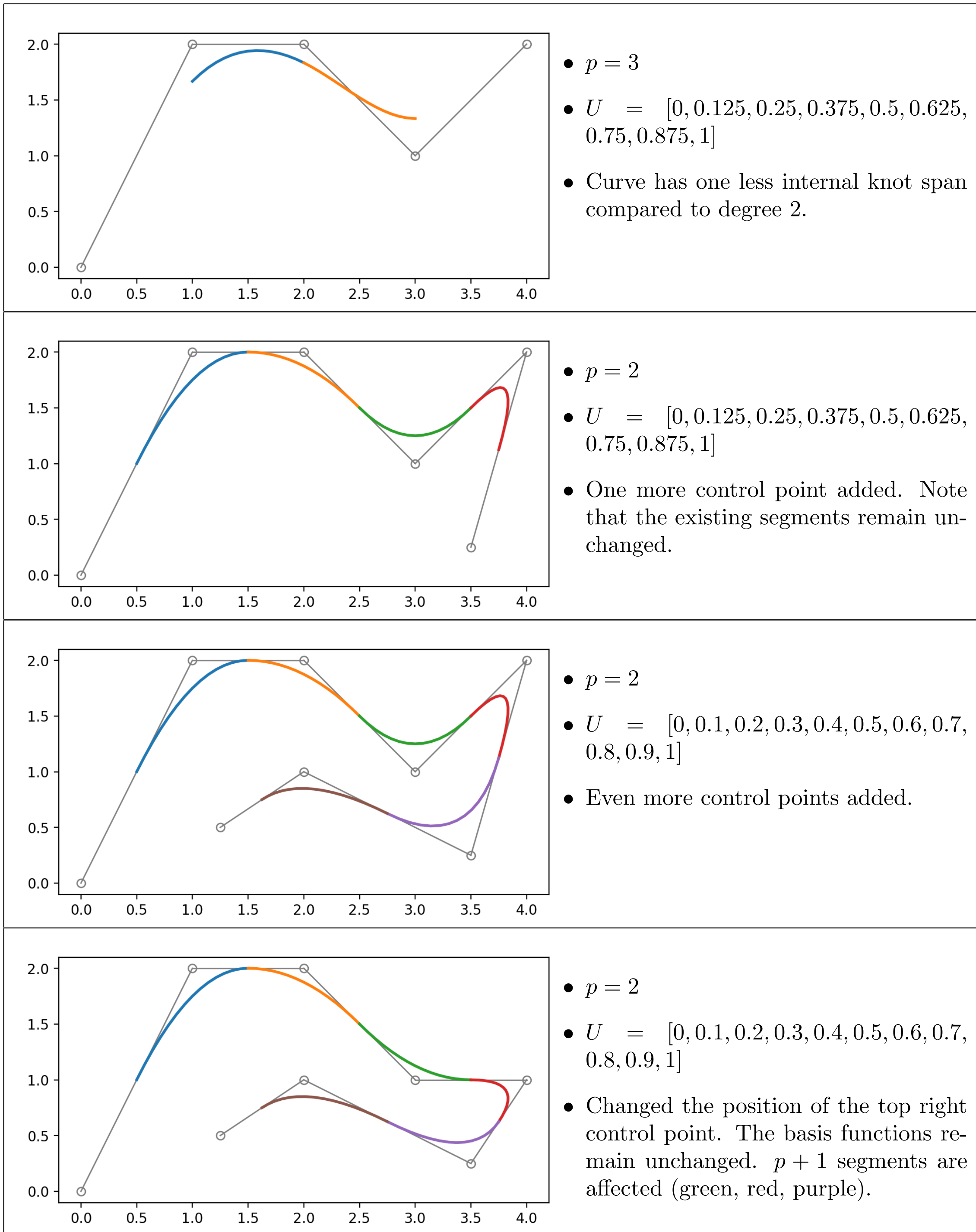
Table 4.2 shows a step by step approach for creating different B-splines and how different properties affect the shape of the curve. The change compared to the preceding B-spline is usually minimal, but the curves can differ a lot. All knot vectors are uniformly created from 0 to 1, with the exception of a few that are clamped afterwards.



- $p = 0$
- $U = [0, 0.2, 0.4, 0.6, 0.8, 1]$
- No curve visible, since for every value of u the result lies exactly on the control points.

- $p = 1$
- $U = [0, 0.1667, 0.3333, 0.5, 0.6667, 0.8333, 1]$
- Curve is a simple linear interpolation between the control points.

- $p = 2$
- $U = [0, 0.1429, 0.2857, 0.4286, 0.5714, 0.7143, 0.8571, 1]$
- Curve has one less internal knot span compared to degree 1 and has points of tangency at the halfway point between control points.

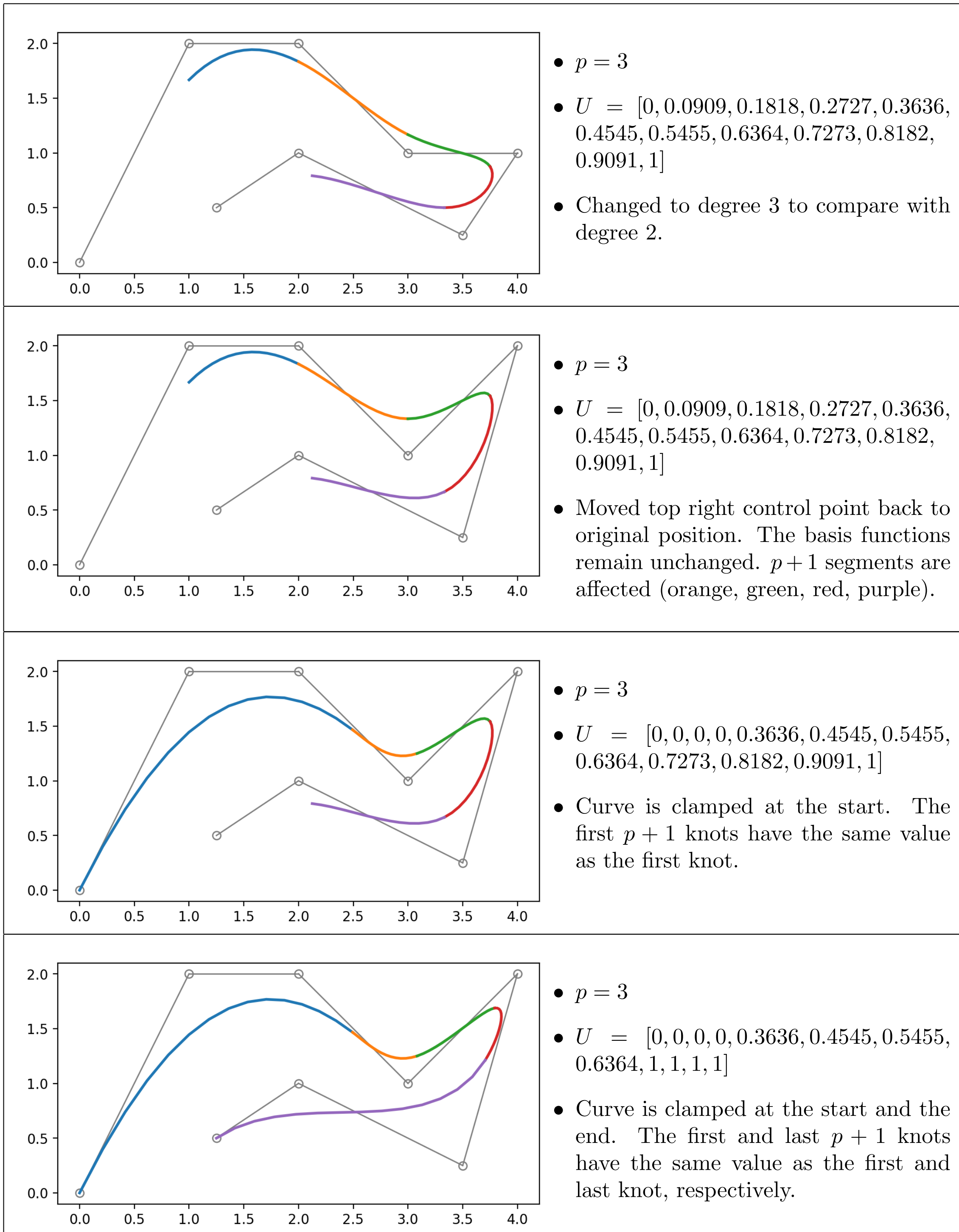


- $p = 3$
- $U = [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1]$
- Curve has one less internal knot span compared to degree 2.

- $p = 2$
- $U = [0, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875, 1]$
- One more control point added. Note that the existing segments remain unchanged.

- $p = 2$
- $U = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
- Even more control points added.

- $p = 2$
- $U = [0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1]$
- Changed the position of the top right control point. The basis functions remain unchanged. $p + 1$ segments are affected (green, red, purple).

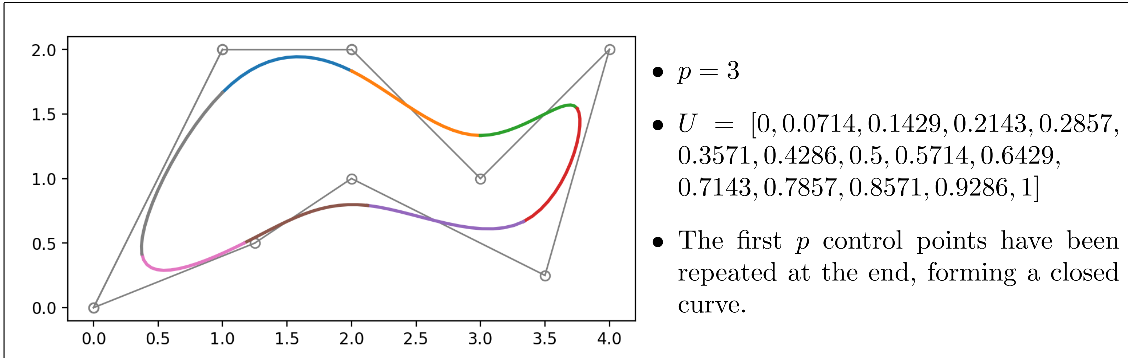


- $p = 3$
- $U = [0, 0.0909, 0.1818, 0.2727, 0.3636, 0.4545, 0.5455, 0.6364, 0.7273, 0.8182, 0.9091, 1]$
- Changed to degree 3 to compare with degree 2.

- $p = 3$
- $U = [0, 0.0909, 0.1818, 0.2727, 0.3636, 0.4545, 0.5455, 0.6364, 0.7273, 0.8182, 0.9091, 1]$
- Moved top right control point back to original position. The basis functions remain unchanged. $p+1$ segments are affected (orange, green, red, purple).

- $p = 3$
- $U = [0, 0, 0, 0, 0.3636, 0.4545, 0.5455, 0.6364, 0.7273, 0.8182, 0.9091, 1]$
- Curve is clamped at the start. The first $p + 1$ knots have the same value as the first knot.

- $p = 3$
- $U = [0, 0, 0, 0, 0.3636, 0.4545, 0.5455, 0.6364, 1, 1, 1, 1]$
- Curve is clamped at the start and the end. The first and last $p + 1$ knots have the same value as the first and last knot, respectively.



- $p = 3$
- $U = [0, 0.0714, 0.1429, 0.2143, 0.2857, 0.3571, 0.4286, 0.5, 0.5714, 0.6429, 0.7143, 0.7857, 0.8571, 0.9286, 1]$
- The first p control points have been repeated at the end, forming a closed curve.

Table 4.2: Step by step creation of different B-splines. The change in input compared to preceding B-splines is minimal, to clearly show the influence of different properties. Inputs are the degree and a list of control points. The control points polygon is shown in light gray. The different colours of the curve indicate the knot spans. The resulting knot vector, uniformly created between 0 and 1, is given as well for clarity. Generated with script in Appendix B.3.

4.1.3 Rational B-splines

Adding weights to the control points makes the B-splines **rational**. Equation 4.4 changes to

$$C(u) = \frac{\sum_{i=0}^{n} N_{i,p}(u) w_i \boldsymbol{P}_i}{\sum_{i=0}^{n} N_{i,p}(u) w_i} \tag{4.5}$$

where w_i is the weight for control point $\boldsymbol{P}_i$. Increasing a weight pulls the curve more towards that control point, and vice versa for decreasing the weight. The shape of the curve is affected by the ratio between weights, not the values themselves. When all weights are equal, a regular B-spline is formed. Usually a value of 1 is used to indicate a neutral state.

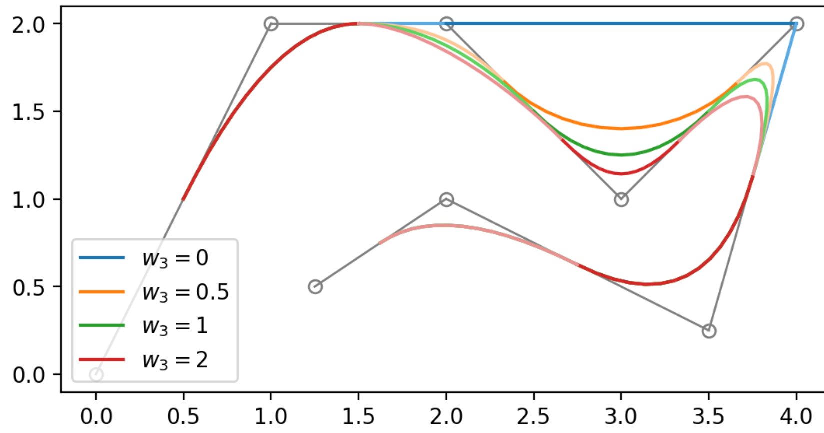
Figure 4.2 shows a few rational B-splines, with all weights set to 1 and different values for the fourth weight. The weights can have an effect on the continuity of the curve, as is clear from the case where $p = 2$ and $w_3 = 0$. Similar to moving a control point, the number of knot spans affected by a change in weight is $p+1$. When increasing a single weight, the evaluated point $C(u)$ for fixed u moves in a straight line towards the control point.

4.1.4 Non-uniform rational B-splines

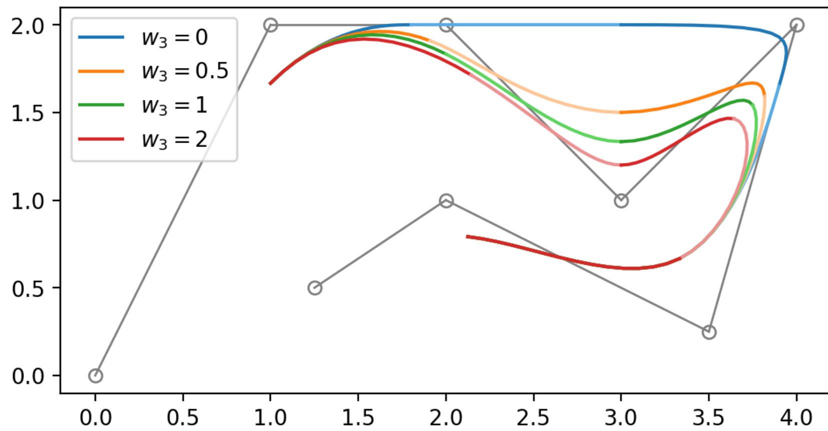
So far we have only seen B-splines with uniform knot vectors, with the exception of the ones that were clamped, although the internal knot spans were still uniform. Knot vectors can be **non-uniform**, however, and the B-spline definition of Equation 4.4 is already perfectly capable of handling those. Here, a few examples will be given on how changing the knot vector can affect the shape of the curve.

The ratios between knot span lengths define the shape of the curve, not the knot values themselves. For example, $U = [0, 1, 2, 3]$ will give the same curve as $U = [0, 2, 4, 6]$ and $U = [2, 3, 4, 5]$. Figure 4.3 shows how changing a single knot value within a uniform knot vector affects the basis functions and the shape of the curve for degree 2 and 3. The number of basis functions affected is $p + 2$, however the number of knot spans over which the basis functions have been affected is $2p$. Indeed, the curve is affected for $2p$ knot spans.

NURBS are very powerful and can be used to create many shapes. It is also possible to create the same shape in multiple different ways. For example, Figure 4.4 shows the same circle using a different number of control points. The values for the knots and weights can be mathematically derived, which is beyond the scope of this work. In a similar way it is possible to add a control



(a)



(b)

Figure 4.2: Rational B-splines for degree 2 (top) and 3 (bottom), with $w_i = 1$ and different values for w_3. Light and dark tones of the line colour indicate the knot spans, of which $p + 1$ are affected by the change in weight. The control points polygon is shown in light gray. Generated with script in Appendix B.4.

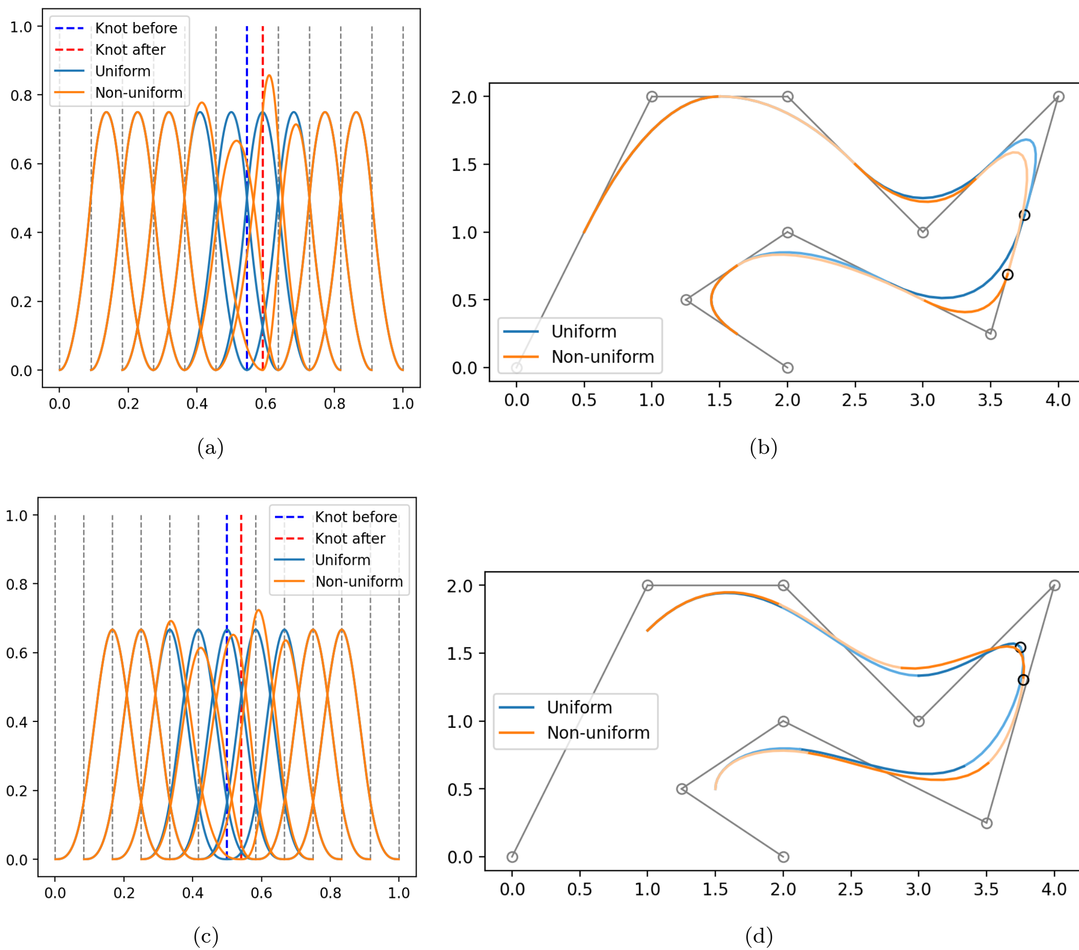


Figure 4.3: Non-uniform and uniform B-splines (right) and their basis functions (left) for degree 2 (top) and 3 (bottom). A single knot value (k_6) in the initially uniform knot vector is updated to show its effect on the basis functions and the B-spline curve. $p+2$ basis functions have been affected over $2p$ knot spans. On the right, the knot position is indicated by a black circle. Light and dark tones of the line colour indicate the knot spans and the control points polygon is shown in light gray. Generated with script in Appendix B.5.

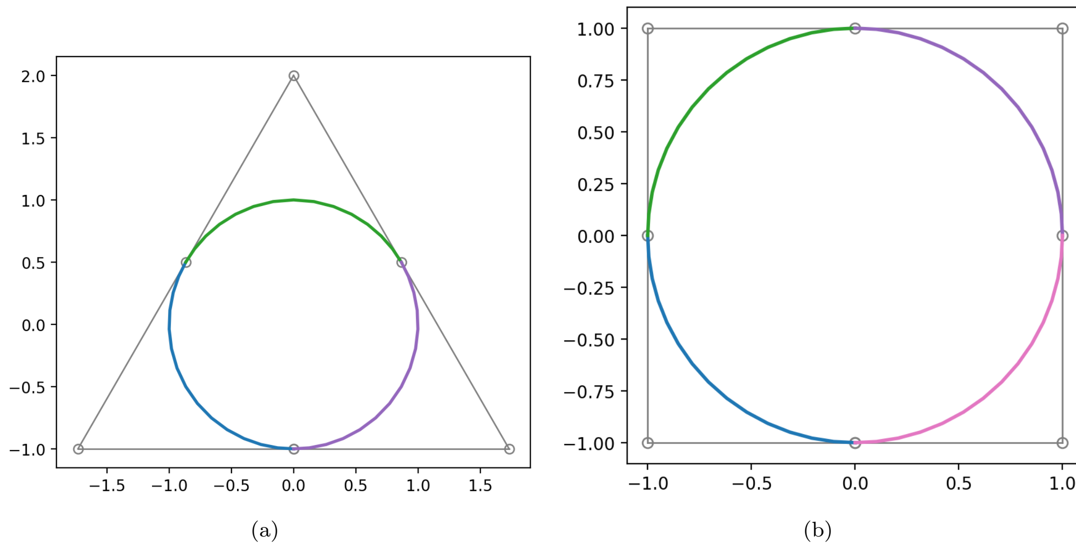


Figure 4.4: The same circle created using NURBS with 7 (a) and 9 (b) control points. On the left $U = [0, 0, 0, \frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{2}{3}, 1, 1, 1]$ and $w = [1, \frac{1}{2}, 1, \frac{1}{2}, 1, \frac{1}{2}, 1]$. On the right $U = [0, 0, 0, \frac{1}{4}, \frac{1}{4}, \frac{2}{4}, \frac{2}{4}, \frac{3}{4}, \frac{3}{4}, 1, 1, 1]$ and $w = [1, \frac{1}{2}\sqrt{2}, 1, \frac{1}{2}\sqrt{2}, 1, \frac{1}{2}\sqrt{2}, 1, \frac{1}{2}\sqrt{2}, 1]$. The degree of both is 2. The different colours of the curve indicate the knot spans and the control points polygon is shown in light gray. Generated with script in Appendix B.5.

point in between others, without affecting the shape of the curve. This control point can then be used for more local control of the shape of the curve.

Changing the knot vector is not a very intuitive approach for changing the shape of the curve. In many 3D modeling software it is not even an option to edit the knot values, for this exact reason. Generally, moving control points or changing the weights is the most direct and intuitive approach to manipulate a curve.

4.1.5 NURBS surface

For completeness sake the equation for NURBS surfaces is shown here, although not much will be said about it as it is 'just' adding a second direction v to NURBS curves. The control points $\boldsymbol{P}_{i,j}$ now form a 2D 'grid' with corresponding weights $w_{i,j}$, where i and j are the iterators in u and v direction, respectively. The degree can be different in each direction and is denoted by p for direction u and q for direction v. The definition of the basis function in Eq. 4.4 remains unchanged, but is now calculated separately for each direction: $N_{i,p}(u)$ and $N_{j,q}(v)$. The equation for a NURBS surface then becomes

$$S(u, v) = \frac{\displaystyle\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u) N_{j,q}(v) w_{i,j} \boldsymbol{P}_{i,j}}{\displaystyle\sum_{i=0}^{n}\sum_{j=0}^{m} N_{i,p}(u) N_{j,q}(v) w_{i,j}} \tag{4.6}$$

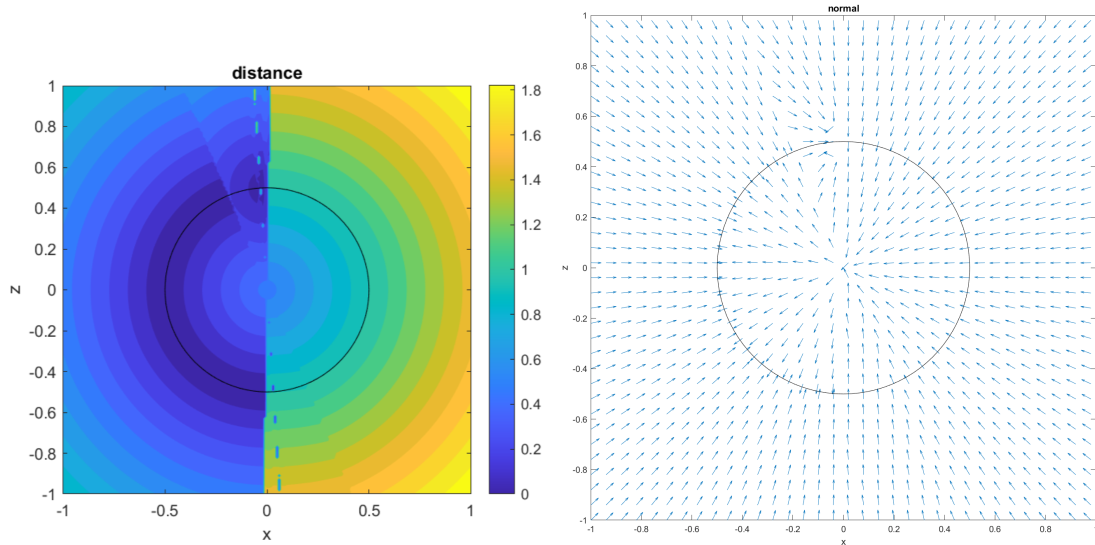


Figure 4.5: The distance (left) and normal direction (right) for a grid of points around a 2D slice of a NURBS cylinder, before any improvements to the distance calculation had been implemented.

4.2 Improving MercuryDPM implementation

The NURBS implementation in MercuryDPM was expected to work out of the box. It was thought that only a few simple additional things might have to be added, such as for example a method for moving control points, but not much more. Unfortunately it turned out that the implementation was lacking in a lot of aspects. It apparently was not tested very well and must not have been used by anybody, since many things were clearly wrong.

One such thing was the contact detection algorithm. It uses the point projection algorithm from Piegl and Tiller [6] (chapter 6.1, pages 229-234), however the implementation simply did not match the one in the book. It was more than simple typos; certain steps were completely wrong and also in the wrong order. The algorithm at its core is a Newton-Raphson iteration trying to find the shortest distance to the surface. The distance and normal direction are therefore not analytically computed, as opposed to written in Weinhart et al. [9].

To test the distance calculation, a grid of points was placed around a NURBS circle (2D slice of a cylinder) and for each point the distance and normal direction was calculated and plotted, see Figure 4.5. Clearly, the points on the right side of the figure have the wrong distance. This is because the distance is calculated from the opposite side of the circle, i.e. a local minimum distance was found, instead of the global one. The normal directions also show that this is the case, because outside the circle they all point in the right direction, but inside the circle they point in the exact opposite direction.

In order to find a global minimum distance, instead of an accidental local minimum, it is very important to have a good starting point for the iteration. The starting point was originally based on the closest control point and with its index the starting u and v were obtained from the corresponding knot vectors. This did not make much sense and only worked for very simple shapes for which the iteration would have succeeded independent of the starting point anyway. To improve this, for a certain number of u and v values the point on the surface is calculated and the closest of these points is then used as a starting point for the iteration. Initially it was tried to simply use the values from the knot vectors for this, however this proved to be insufficient. Taking three times the number of knots and uniformly spacing them between the start and end knot seemed to give a reasonable improvement. Of course, taking as many starting points as possible would further improve this, but it slows the code down drastically and undermines the whole reason for using Newton-Raphson iteration. An improvement is shown in Figure 4.6, which shows that the distance and normal directions are correct almost everywhere.

At the top of the circle the distance calculation is still wrong. This is the place where the two

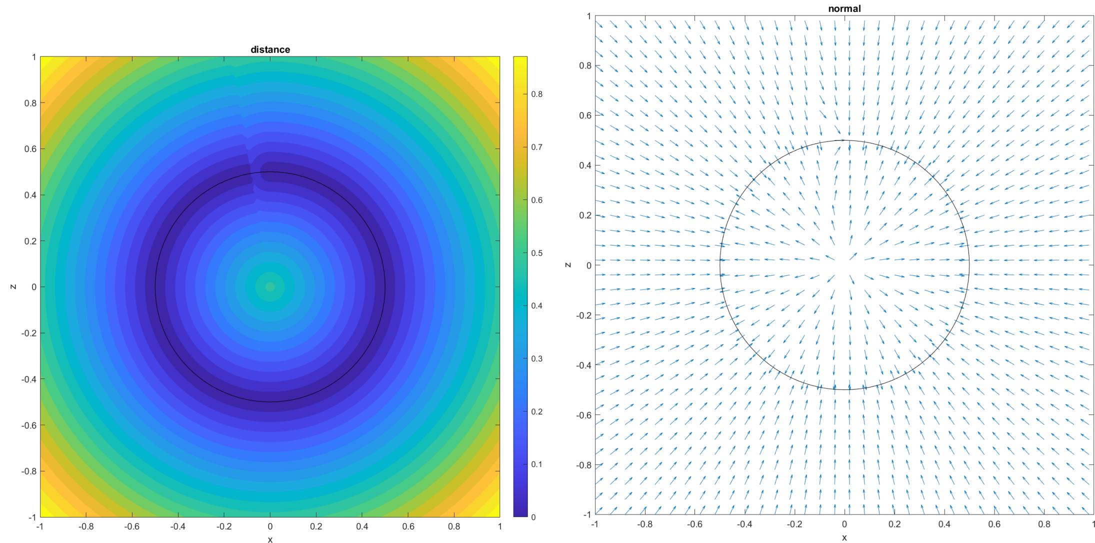


Figure 4.6: The distance (left) and normal direction (right) for a grid of points around a 2D slice of a NURBS cylinder, after implementing some improvements to the distance calculation.

ends of the surface meet to form a closed cylinder. The problem here is that during the iterations the point on the surface was never wrapped around to the other end of the surface. It would simply find the edge of the surface instead, depending on which side the closest starting point happened to be. The error is less pronounced than that of the previous plot, simply because a higher number of starting points were used, which caused slightly more points to start on the correct side of the surface. A proper implementation of the point projection algorithm does make sure the iterations wrap around to the other side of the surface and naturally solves this issue.

The last major problem with the contact detection algorithm is that the Newton-Raphson iteration often fails to converge. A certain tolerance is used to tell if the minimum distance found is within an acceptable range. Increasing the tolerance will increase the convergence rate; the question is to what extend this can be done. A few different scenarios were noticed that caused non-convergence. For example, when the iterations kept switching back and forth between two points, neither of which satisfied the tolerance. It could be argued that the halfway point between them should be the final closest point, however experimenting with this did not lead to satisfying results. Another example is some iterations coming quite close to being within the tolerance, but never quite reaching it. The result would still be slightly wrong, so it was not an issue of simply increasing the tolerance. After a certain maximum fail rate, the last distance found is returned. This sometimes happened to be an almost correct result, but other times it would be complete rubbish. One improvement that could still be implemented here is to remember the best result for all iterations and return that one instead.

A few NURBS properties have been noticed to influence the convergence rate, such as the degree, repeated knots, and coinciding control points. Unfortunately, no actual relation or logic has been found. It is suspected that the continuity of a surface is important, for example C^0 continuity is not enough, because for a rectangular shape the corners had problems converging. But to what extend the continuity matters is unclear. It is not a case of the smoother the better, because increasing the degree sometimes caused the convergence to fail more often.

Figure 4.7 shows a 2D slice of an almost square surface, unclamped on both sides. There are still many spots where the distance calculation is wrong, sometimes forming interesting patterns, in this case a pretty butterfly. Although very pretty, it just goes to show how the contact detection is not yet where it should be. So far no solution was found and at this point it was decided to leave this for now, to hopefully come across a solution while working more with NURBS.

Particles not in the vicinity of the NURBS surface can quickly be eliminated by having some sort of convex hull argument. This would be a lot quicker than running the full contact detection algorithm. It has not been implemented, however, so that during development and usage of the

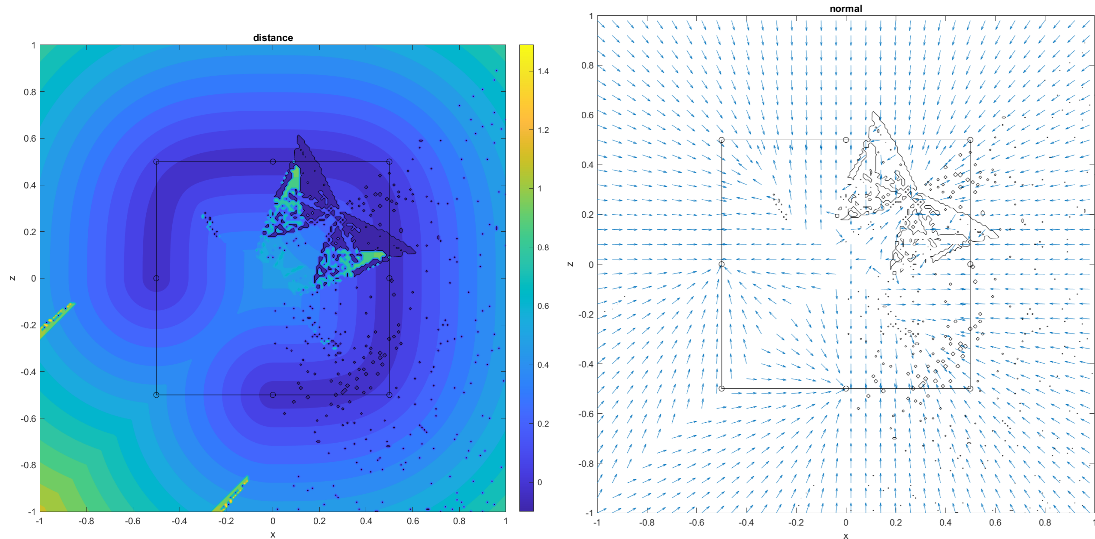


Figure 4.7: The distance (left) and normal direction (right) for a grid of points around a 2D slice of an unclamped square-like NURBS surface.

NurbsWall hopefully a better picture could be obtained for how often and for what scenarios the contact detection fails. It would also not be a solution for the contact detection failing itself, because any particle, no matter how far away from the surface, can theoretically have a radius big enough to still be in contact with the surface.

Many more small improvements have been made. For example, making the code work for unclamped surfaces, by not assuming the knots range from 0 to 1, but instead use the degree to obtain the correct start and end knot values like $U[p]$ and $U[len(U) - p - 1]$, respectively. Or, some corrections to the vtu file writing to fix a bug in the triangulation of the surface. Also, a new MercuryDPM tool has been written, which allows us to visualise the control points in addition to the surface itself. It is written in a very generalised way, so it can be used for many different use cases. Detailed information about this can be found in Appendix C.

4.3 Wearing

When particles flow over a NURBS surface and it wears down, the shape of the surface must be updated to account for that wear. To do so, the control points could be moved or their weights could be altered. Changing the weights, however, has its limits, since for example in a flat plane created by a flat grid of control points, updating a weight would not do anything to the shape of the surface. Therefore, moving the control points is the logical choice.

The question then is which control point or points to move. Depending on the degree and the number of control points, moving a control point can change the surface over a very wide range. To have more local control over the shape of the surface, control points could be added near where the wear is occurring. However, to what limit should more and more control points be added? A threshold could be used to try to snap to the closest existing control point first, and only add a new one when none exists. The same result and an easier implementation, however, is to start with a certain resolution in control point spacing to begin with and always snap to the closest one. For a general surface in 3D space this would mean the need of automatic control point refinement. This has not been implemented, rather the choice was made to start of simple and to get something working first.

One could question whether a single or multiple control points should be moved. A knot span is affected by $p + 1$ control points, so why not update them all? The reverse is also true, i.e. a single control point affects $p + 1$ knot spans. So updating them all would only widen the effect on the surface, which is the opposite of the local control that we are after. Furthermore, should we snap to the closest control point in physical space, or calculate which one has the most influence on

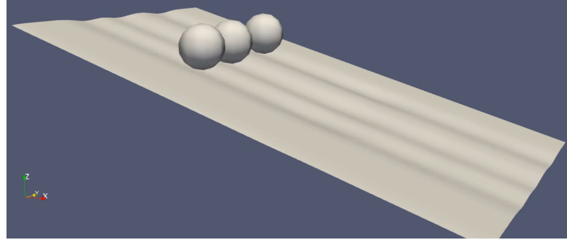


Figure 4.8: Three particles flowing over a NURBS surface and wearing it down.

that position? Assuming the control point spacing has been refined to a sufficient degree, snapping to a single closest control point should be good enough for an initial trial. The more refinement, the closer the control points will be to the actual surface.

A new WearableNurbsWall class is created in MercuryDPM, which inherits from NurbsWall and overrides the computeWear method. The control points are initialised in a 2D flat grid with u in the x-direction and v in the y-direction. The x- and y-positions are calculated from an input length and resolution in u and v, respectively, while the z-positions are set to 0. Again for simplicity's sake, the control points only move down in z-direction. This quasi 2D plane is used as a starting point to keep the number of variables as low as possible.

For each interaction with the wall, the resulting debris is added to the control point closest to the interaction contact point. After all interactions have been handled, the surface is evolved accordingly. Since this is potentially computationally expensive and the evolution might be so little that it increases the relative error, it is only done after a certain number of time steps. The debris from the interactions is accumulated over those time steps and reset to zero after the surface has evolved. To evolve the surface, first a guess is made by simply subtracting the debris from the z-positions. The volume under the surface before and after is calculated, which gives a change in volume. Together with the total debris this gives a ratio by how much the initial guess should be corrected. So instead of subtracting the debris directly, it is multiplied by the ratio first. This assumes that the evolution is sufficiently small so that the relation can be assumed linear.

To the author's knowledge no method exists for calculating the volume under a NURBS surface. As a first step an attempt was made to derive an equation for the area under a NURBS curve, but this was to no avail. To not waste too much time on such details, but rather to get something working first, the volume is calculated by triangulating the surface. The total volume is then simply the sum of the volumes under each triangle. For each triangle that is the area of its projection onto the plane $z = 0$ multiplied by the average z-position of the three vertices. The projection is found simply by setting $z = 0$ and the area is then $A = \frac{1}{2}|\boldsymbol{A} \times \boldsymbol{B} + \boldsymbol{B} \times \boldsymbol{C} + \boldsymbol{C} \times \boldsymbol{A}|$, where $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are the vertices of the triangle.

To ensure the particles flow smoothly when passing a periodic boundary, the surface should line up properly at both ends. This is similar to a closed surface, without the surface actually being closed. To make a open surface closed, the first $p + 1$ control points have to be repeated at the end. In this case this has to be done in both directions, i.e. the first and last $p + 1$ control points have to be added respectively at the end and the start. The first and last control points themselves can be skipped, as they are assumed to already corresponds to each other and lie directly on the periodic boundaries. The copied control points should follow the movement of the original control points. For this to work it is important that the first and last $p + 1$ knots are uniformly spaced.

Figure 4.8 shows a simulation of a flat plane with three particles flowing on top of it and periodic boundaries on the sides for repeated flow. Slowly the surface evolves due to the wear of the particles acting upon it. The grooves form directly over a line of control points initially closest to the particles. Indeed, the particles moved ever so slightly sideways to align with the grooves. For a similar reason the final velocity of the particles differs slightly. This dependency on the resolution will always exist on a particle level, however for bulk flow it should not be a problem.

4.4 Limitations

The simple approach of a quasi 2D plane, made with grid of control points, does give somewhat reasonable looking results. Extending this to 3D, however, poses a bit of a problem. As mentioned, no known method exists for calculating the volume under a NURBS surface, let alone the volume between two NURBS surfaces. The reason for starting with a simple quasi 2D plane was so we could calculate the volume under the surface by triangulating it. At the time a similar approach for calculating the volume between two separately triangulated surfaces was deemed not doable or at least very hard. Section 5.2.2 describes how a single triangle mesh is moved and how its volume change can be calculated and in hindsight a similar approach could have been possible here.

Triangulating the NURBS surface in and of itself seems a bit counter productive, because why not use a triangle mesh to begin with. One could argue that at least for contact detection the surface will be smoother, but as discussed before, that might not really be the case. The contact detection still regularly fails to converge and it is unclear how much of a problem this will be. At the very least, it does not feed confidence.

For these reasons, triangle meshes are starting to look more like the better option. For the volume change calculation nothing would change, in fact it would improve, because for a triangle mesh all triangle information is always known, instead of having a newly triangulated surface each time. The volume change for evolving the triangle mesh would simply be the sum of the volume changes of each triangle. And the volume change for each triangle is easy to calculate. Or so we thought, Section 5.2.2 will explain more about why this turned out harder than expected.

In the meantime, the problem of a particle in contact with multiple triangle edges or vertices had been fixed. So this is not a reason anymore to not use triangle meshes. The non-smoothness is still not great, but it is simply something we have to deal with. Besides that, the TriangleWall has existed for a long time in MercuryDPM and is well tested and used a lot. Because of this, building a mesh from TriangleWalls seems less prone to error. Thus the choice was made to abandon NURBS surfaces and switch to triangle meshes instead. The goal being to get something working in 3D within a reasonable time span, so the effect of evolution itself can be studied.

We conclude this chapter by saying that the WearableNurbsWall class is an unfinished product and should be used with care and only with the goal of learning or improving. To a lesser extend, the same can be said about the NurbsWall class. Make sure you understand NURBS and be critical of the results.

Chapter 5

Triangles

5.1 TriangleMeshWall

5.1.1 Mesh

To create a mesh of triangles use is made of the already existing TriangleWall, as it is well tested and speeds up the implementation. It has 3 vertices, with indices 0-2 in clockwise direction. For contact detection the call to the getInteractionWith method can then simply be passed through to each TriangleWall in the mesh, as explained in more detail in Section 5.1.3.

The easiest way to store all TriangleWalls is to add all instances to a vector, which for a non-evolvable mesh would be all that is needed. However, the mesh must be able to evolve, which means that updating a vertex requires updating all triangles sharing that vertex. Therefore, all vertices in the mesh are stored in a vector as well, where a vertex is the struct shown in Listing 5.1. It stores the position and a vector with pairs of indices, with the first value being the index to the triangles vector and the second value being the local vertex index of the TriangleWall. For the triangles it is also useful to know from which vertices it is made of, therefore a triangle in the triangles vector is not a TriangleWall, but the struct shown in Listing 5.2. It stores the TriangleWall and an array of 3 indices to the vertices vector. Note that a Vertex can have many triangles attached to it, while a Triangle always has 3 vertices.

```
1 struct Vertex
2 {
3     Vec3D position;
4     std::vector<std::pair<unsigned, unsigned>> triangleIndices;
5 };
```

Listing 5.1: The Vertex struct.

```
1 struct Triangle
2 {
3     TriangleWall wall;
4     std::array<unsigned, 3> vertexIndices{};
5 };
```

Listing 5.2: The Triangle struct.

A set function, part of it shown in Listing 5.3, takes a vector of points and a vector of cells, where each point is a Vec3D and each cell is an array with three indices to the vector of points, indicating which points form a triangle. First the points are added as Vertex to the vector of vertices, which are always considered to be in local frame. Then for each cell a TriangleWall is

created, with vertices from the points vector. Note that the position of the TriangleWall is always the origin of the local coordinate system. The TriangleWall and cell, i.e. vertex indices array, are then added as Triangle to the triangles vector. Finally, the index of the last added Triangle is added to the triangle indices vector of the corresponding vertices.

```
// Add all the points as Vertex to the vector.
for (auto& p : points)
    vertices_.emplace_back(p);

for (auto& c : cells)
{
    TriangleWall tw;
    // The position is the origin of the local coordinate system,
    // which is always (0, 0, 0) (not getPosition()!).
    // This is needed in case the mesh has an angular velocity.
    tw.setVertices(points[c[0]], points[c[1]], points[c[2]], Vec3D(0.0, 0.0, 0.0));
    if (species != nullptr)
        tw.setSpecies(species);

    // Add as Triangle with TriangleWall and vertexIndices to the vector.
    triangles_.emplace_back(tw, c);

    // Add the Triangle index and the local vertex index to each of the Vertices.
    const unsigned tIndex = triangles_.size() - 1;
    vertices_[c[0]].triangleIndices.emplace_back(tIndex, 0);
    vertices_[c[1]].triangleIndices.emplace_back(tIndex, 1);
    vertices_[c[2]].triangleIndices.emplace_back(tIndex, 2);
}
```

Listing 5.3: Processing the points and cells in the set function and adding them as vertices and triangles to their corresponding vectors.

5.1.2 Periodic boundaries

Mesh evolution at periodic boundaries should be applied to both boundaries. For example, if for a simple square mesh the periodic boundaries lie exactly on the left and right column of vertices, then each of these vertices has a so-called ‘periodic companion’ on the other side of the mesh. When moving one, the other should move the same amount. The vertex index of each companion pair is stored in a vector of pairs. A vertex can have multiple periodic companions, for example when the mesh has periodic boundaries on the left and right and at the front and back, then the bottom left vertex has the top left, top right, and bottom right vertices as periodic companions. These can all be stored in the same vector and to ensure no vertices get handled multiple times, each pair should be unique, irregardless of the order of the first and second value.

To find the periodic companions of a vertex, its vertex index is compared to each pair value and, in case of a match, the other pair value is stored, see Listing 5.4. The resulting vector holds the vertex indices of the periodic companions and is returned from the method.

It should be noted that this way of handling periodic boundaries is only valid when the mesh near the periodic boundaries is a mirror opposite of itself. For unstructured or irregular shaped meshes this is less — or not at all — suitable.

```
for (auto& p : periodicCompanions_)
{
    if (p.first == index)
        indices.push_back(p.second);
    else if (p.second == index)
        indices.push_back(p.first);
}
```

Listing 5.4: Get vector with indices of periodic companions.

Listing 5.5 shows how to move a single vertex. First the position of the vertex itself is updated, then each triangle with that vertex updates the corresponding local vertex of the TriangleWall. For each of the periodic companions the exact same is done. In the end a call is made to update the bounding box, which is used to speed up contact detection.

```
void TriangleMeshWall::moveVertex(const unsigned index, const Vec3D& dP)
{
    // Update position of Vertex in vector.
    vertices_[index].position += dP;

    // Update each Triangle wall of the triangles who share this Vertex.
    for (auto& ti : vertices_[index].triangleIndices)
        triangles_[ti.first].wall.moveVertex(ti.second, dP);

    // Do the same for each of the possible periodic companions of this Vertex.
    for (unsigned idx : getPeriodicCompanions(index))
    {
        // Update position of Vertex in vector.
        vertices_[idx].position += dP;

        // Update each Triangle wall of the triangles who share this Vertex.
        for (auto& ti : vertices_[idx].triangleIndices)
            triangles_[ti.first].wall.moveVertex(ti.second, dP);
    }

    updateBoundingBox();
}
```

Listing 5.5: Moving a vertex.

5.1.3 Contact detection

The getInteractionWith method of the inherited BaseWall is overridden to calculate the possible contact between a particle and the mesh. Contact with multiple triangles is possible, however MercuryDPM does not handle it very well when the contact is on the edge of two neighbouring triangles. This has to do with overlapping contact areas and can be solved by setting a group id for use in the wall handler. The triangles themselves are not added to the wall handler, however, so this cannot be used. The algorithm handling the group id had a comment saying it is not very efficient, therefore instead of trying to copy and refactor it for use in this case, the choice was made — for now — to keep things simple and only consider the contact with the highest overlap. The tiny error compared to the actual reaction force has no significant effect on the particle flow. The effect it might have on the wearing down of a surface is assumed to also be insignificant, but this has not been researched.

The contact detection starts by rotating and translating the particle position from global to local coordinates, using the mesh orientation and position in global frame, as shown in Listing 5.6[1]. Then we loop over all triangles and call the getInteractionWith method for each triangle and store the highest overlap and a reference to the TriangleWall. Interactions are automatically added to the interactionHandler, so when an interaction does not have the highest overlap (anymore) it should be removed from the handler. Listing 5.7 shows the code for this loop.

```
const Vec3D posOriginal = p->getPosition();
p->setPosition(getOrientation().rotateBack(posOriginal - getPosition()));
```

Listing 5.6: Translating and rotating a particle from global to local frame.

[1]The rotate/rotateBack methods of the Quaternion class have been updated to allow returning of the resulting Vec3D as well, instead of only updating the one passed by reference. This allows for easy one-liners instead of having multiple lines of code. Benchmarking showed no difference in speed, rather it showed a possible slight improvement.

```
for (auto& t : triangles_)
{
    // Checks if the particle is interacting with this triangle
    BaseInteraction* i = t.wall.getInteractionWith(p, timeStamp, interactionHandler);

    // When an interaction exists
    if (i != nullptr)
    {
        // When the overlap is greater than that of any previous interactions
        if (i->getOverlap() > maxOverlap)
        {
            // Remove the previous maximum overlap interaction from the handler
            if (maxWall != nullptr)
            {
                BaseInteraction* prevMaxI =
                    interactionHandler->getInteraction(p, maxWall, timeStamp);
                interactionHandler->removeObject(prevMaxI->getIndex());
            }

            // Update the maximum overlap and store the interacting wall
            maxOverlap = i->getOverlap();
            maxWall = &t.wall;
        }
        else
        {
            // Remove the last added interaction, as it is not the one with
            // maximum overlap and should be discarded.
            interactionHandler->removeObject(i->getIndex());
        }
    }
}
```

Listing 5.7: Looping through the triangles and storing the interaction with maximum overlap.

Before doing any force calculations the particle is set back to its original position in global frame. The interaction was obtained in local frame, so we have to update its normal and contact point to be in global frame. Similarly, but only temporarily, we have to change the TriangleWall position and (angular) velocity to be in global frame, so that the correct values are used in the force calculations. After doing so, the actual force/torque calculations are handled as usual, for which the code is copied line by line from DPMBase::computeForcesDueToWall. All possible interactions with the triangles are now handled, so for the TriangleMeshWall itself a null pointer is returned, i.e. no interaction.

5.1.4 Bounding box

The contact detection algorithm is a bit computational heavy since it has to loop through all triangles, therefore two layers of defence are added to prevent a particle from being subjected to these calculations. Both layers use a bounding box, which in this context is the smallest axis-aligned box that fits around the mesh. The first layer is the HGrid, which efficiently checks if the particle is within the bounding box in global frame and ignores it otherwise. The second layer is at the start of the getInteractionWith method, where a check is done if the particle is within the bounding box in local frame. Since the local bounding box can have a much tighter fit around the mesh, depending on the orientation of the mesh in global frame, it can still eliminate lots of potential particles the HGrid has let through. The local bounding box thus has the most influence on speed improvement, however the global bounding box is checked at an earlier stage and therefore still useful to also have included.

Updating the local bounding box is done by looping over all vertices and storing the minimum and maximum x-, y- and z-coordinates. For the global bounding box we could rotate and translate each vertex to global frame and then do the same thing. This, however, is computationally quite involved and it is much simpler to fit a box around the rotated and translated local bounding box. This means that the global bounding box can be slightly bigger than strictly needed, but that is a well worth trade-off, assuming the mesh has been set up in such a way that the local bounding

box is minimized.

Updating the local bounding box is only needed when a vertex in the mesh has changed. The global bounding box needs updating when the local bounding box has been updated and when the position or orientation of the mesh itself has changed. This is also a big reason why updating the global bounding box should not be too costly, because giving the mesh a prescribed motion or orientation results in many changes to its position and orientation, respectively.

For a few simple test cases, using both bounding boxes resulted in $5.77 - 7.38$ times faster execution time. As expected the local bounding box has the most influence, especially when certain mesh orientations in global frame resulted in a large global bounding box. Still, leaving the global bounding box in there as well did seem to hint towards a slightly bigger speed improvement, although the difference was small.

5.2 Wearing

5.2.1 Computing the wear

The computeWear method loops over the triangles and for each triangle loops over their interactions, for which it then calculates the debris according to the wear model used, see Listing 5.8. The debris is multiplied by the interaction normal before being assigned to multiple vertices. This gives the direction in which to move the vertices, as well as an initial value, since evolving the mesh is a iterative process. Debris from different triangles assigned to the same vertex are added together, so the final direction will be a weighted average. The total debris is also remembered, as it is the target volume change for evolving the mesh, which is done once all the wear has been computed.

Using the contact point on the wall, the triangle is subdivided into three smaller triangles. The ratio of the debris assigned to a vertex is equal to the area of the opposite triangle divided by the total area. This is known as barycentric interpolation and it means that a vertex will be assigned a larger portion of the debris, the closer the contact point is to it.

It is important to use the contact point on the wall and not the contact point itself, as for barycentric interpolation the point used should lie exactly on the triangle, otherwise the ratios will get skewed. To get the contact point on the wall, simply add half the overlap multiplied by the interaction normal to the contact point. Even though the overlaps are usually very small and the overall effect will be minimal, it is still prefered to correct for this. Furthermore, the point and the debris are translated and rotated from global to local frame first.

5.2.2 Evolving the mesh

Evolving the mesh to match the worn down debris requires us to know the change in volume between the updated and original mesh. For a single triangle this is not analytically possible, since two moved vertices and their original positions don't necessarily lie in plane. However, the neighbouring triangle solves this problem, since the division boundary between them does not have any influence on the total change in volume, no matter if it is in plane or not. It is therefore perfectly valid to simply pick one of the diagonals of the division boundary and use it to calculate the change in volume for both triangles. At the edges of the mesh, where there are no neighbouring triangles, this is a less valid approach. However, the overall influence is minimal, especially since wearing at the edges is ill-defined, see Section 5.3, and meshes should preferably be build in such a way that the edges remain wear-free.

The volume change for a single triangle can now be calculated by considering the tetrahedrons formed by moving the vertices one by one, see Figure 5.1. Each tetrahedron has a volume of $V = \frac{1}{6}|(\boldsymbol{a} \times \boldsymbol{b}) \cdot \boldsymbol{c}|$, where $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{c}$ are the directional vectors defining the tetrahedron. Within a mesh, moving a vertex will form one tetrahedron for every triangle connected to it. The order in which the vertices are moved does not matter, so a simple loop suffices.

Initially the vertices are moved by the debris assigned to them. The ratio between the target volume change and the total volume change of the mesh should be 1, within a certain tolerance. When this is not the case, the assigned debris are multiplied by the ratio and a recursive call is made. Once the ratio is close enough to 1 or when the maximum number of recursive calls have

```
// Initialise debris vector with all zero Vec3Ds.
std::vector<Vec3D> debrisContainer(vertices_.size(), Vec3D(0.0, 0.0, 0.0));
Mdouble totalDebris = 0.0;

for (auto& t : triangles_)
{
    for (BaseInteraction* interaction : t.wall.getInteractions())
    {
        // Ignore old interactions and interactions from periodic particles.
        if (interaction->getTimeStamp() <= getHandler()->getDPMBase()->getNumberOfTimeSteps() ||
            static_cast<BaseParticle*>(interaction->getP())->getPeriodicFromParticle() != nullptr)
            continue;

        const Mdouble W = interaction->getAbsoluteNormalForce();
        // Pythagoras to get tangential magnitude from the normal magnitude and
        // the relative velocity vector.
        const Mdouble tangentialRelativeVelocity = std::sqrt(
            interaction->getRelativeVelocity().getLengthSquared() -
            interaction->getNormalRelativeVelocity() * interaction->getNormalRelativeVelocity());
        const Mdouble L = tangentialRelativeVelocity * getHandler()->getDPMBase()->getTimeStep();
        const Mdouble Q = wearAcceleration_ * wearCoefficient_ * W * L / hardness_;

        totalDebris += Q;
        // The contact point in normal direction is halfway the overlap. However,
        // to properly store the debris the position exactly on the wall should be used.
        Vec3D contactPointOnWall = interaction->getContactPoint() +
            0.5 * interaction->getOverlap() * interaction->getNormal();
        // Debris is removed in the direction of the interaction normal
        // (rotated back to lab frame).
        storeDebris(t, getOrientation().rotateBack(contactPointOnWall - getPosition()),
            getOrientation().rotateBack(interaction->getNormal()) * -Q, debrisContainer);
    }
}

moveVerticesToMatchVolume(debrisContainer, totalDebris);
```

Listing 5.8: The computeWear method.

been made, the recursion stops and the vertices have reached their final position. For more details on moving the vertices to match a change in volume, see Appendix D.2.

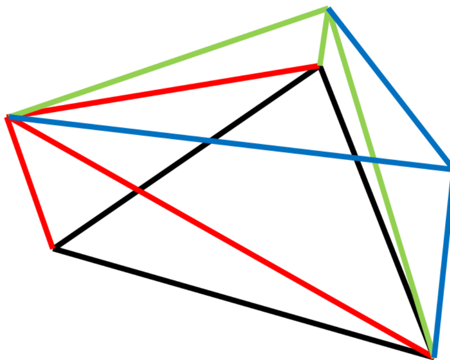


Figure 5.1: Three tetrahedrons formed by moving the vertices of a triangle one by one, in order of red, green, blue.

5.3 Limitations

In the direction of the wear the mesh is considered infinitely thick, i.e. it can keep wearing down indefinitely. The mesh itself, however, is infinitely thin. Wear happening from both sides can therefore effectively cancel each other out, since vertices might move up and down at different

moments in time, but overall remain at roughly the same position. Furthermore, the change in volume for a single triangle is ill-defined when its new orientation intersects its original orientation. The calculations work fine, but are a bit off, since technically the part below and above the original orientation should be calculated separately, which is not done.

As explained in Section 5.2.2, the change in volume for a triangle can be calculated by considering the three tetrahedrons formed by moving the vertices one by one. This assumes that there is a neighbouring triangle that will account for the fact that two moved vertices do not necessarily lie in place with their original positions. However, at the edges of the mesh there are no neighbouring triangles, so this assumption is unjustified. Furthermore, a triangle will have a volume change of zero when all its vertices move in plane. For a single triangle within the mesh this is only possible when it does not lie in plane with a neighbouring triangle and the latter has assigned certain debris to the shared vertices. The triangle itself having a volume change of zero is then perfectly fine, since the neighbouring triangle does have an actual volume change. Again, at the edges of the mesh this is not the case and can therefore cause issues.

With the current implementation this can all be solved by giving the wall a thickness, by adding multiple meshes to form an enclosed volume, similar to how the geometry is created in Chapter 6. When there is a lot of wear and the enclosed volume is quite thin, it is still possible for both surfaces to meet and the same problem happening once more. At this point the wall should have gotten a hole in it, which is not something that is implemented to happen automatically, It could, however, be implemented in the drivers code, e.g. by overriding the actionsAfterTimeStep method, but this has not been tried.

5.4 Possible improvements

Initially the TriangleWall was used because it was well tested and did what was needed. However, the vertices of a TriangleWall are created and stored in the object itself and cannot be shared with other TriangleWalls, e.g. by using pointers. This means that more Vec3D instances are created than strictly needed and it is also the reason for the slightly complex way of having the indices to the vertices vector stored. This can all be solved by disregarding the TriangleWall and fully embracing the Triangle struct/class, where things like the contact detection can simply be copied over. Then all vertices in the Triangle would be pointers pointing to vertices in the vertices vector, so updating a vertex only has to be done in one place. It must be noted that in the current form some indices to the vertices or triangles vectors could probably already have been replaced by pointers, although it could also lead to unexpected bugs, for example, when one forgets to also update the TriangleWall vertices.

Some limitations of the current implementation can be solved by using tetrahedrons instead of triangles. The walls will always have a certain thickness, removing the weird infinitely thick wall issue. There are two ways this can be done. The first is to keep track of the total debris for each tetrahedron and delete it once it exceeds its own volume. The downside is that the wall probably is not very smooth once it starts wearing down, meaning the resolution has a lot of influence. Although there might be applications were this is actually preferred. Another way is by moving the vertices and only delete a tetrahedron once it collapses into itself. The upside is that the wear will be smoother, but the downside is that the same problems with moving the vertices to approximate the volume change exist, which is also computationally a lot more involved. For both methods, holes can be formed naturally and also wear at (thin) edges does not require a second thought and just works.

The contact detection rotates and translates the particle position to local frame. It does this by actually updating the particle position, since the particle itself is passed to the getInteractionWith method of each TriangleWall. It then later is set back to its original position. It is bad practice to temporarily update the particle position like that; instead, only the position should be passed to, for example, the getDistanceAndNormal method. The getInteractionWith method was used because it handled all kinds of things that are needed for the interaction. In hindsight, there is most likely a way this could have been done differently, but at that time it seemed like the best fitted solution.

A tight fitting local bounding box has a huge influence on the speed of the contact detection.

Since the bounding box is axis-aligned, a good fit happens only when the mesh itself is also axis-aligned. The local vertices positions could be updated to make sure this is always the case. This would be done only once, when initializing the mesh. This is especially useful when reading in geometries from input files, e.g. stl files, since in that case there is no control over what the vertices positions are going to be.

Chapter 6

Simulations

The left panel in Figure 6.1 depicts a vibrating screen used for filtering granular material. Particles flow from the back to the front and fall through the gaps in between the bars. The bars get narrower at the tips, therefore widening the gap between them and allowing larger particles to fall through. When the bars wear down and the gap between them increases, the filtering process is of course negatively impacted.

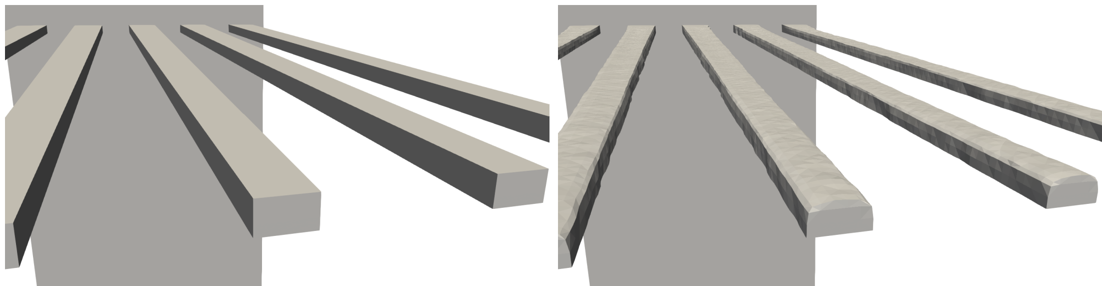


Figure 6.1: Vibrating screen before (left) and after (right) wearing down.

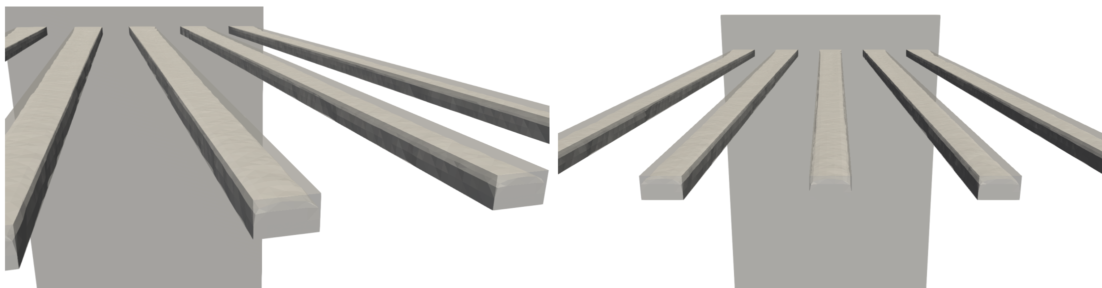


Figure 6.2: Overlay of initial and worn down profile of vibrating screen.

Only five bars are modelled here, with periodic boundaries on the left and right side, mimicking an infinitely wide screen. The sides of a bar are creating using the createFourPointMesh function and they are added together to form a single mesh using the addToMesh function. These helper functions are explained in more detail in Appendices D.4 and D.5.

Multiple simulations with different wear coefficients — all other parameters kept the same — were run. Since the wear acceleration, hardness and wear coefficient are all multiplied together, keeping the former two at 1 means we can later assign partial values to either of these terms when we see fit. The right panel in Figure 6.1 shows one of the results. It is the result that is visually the most pleasing, as others showed very little or unreasonable much wear. In the later case, the bars collapsed onto themselves and the wear was clearly not physical anymore. Figure 6.2 shows

the worn down profile with the initial profile overlain on top. Clearly, the top of the bars are worn down more than the sides. This makes sense and compares nicely to real life, because the particles flow longer and press harder (higher normal force) on top of the bars.

There are two methods that can be used to compare the wear to that in the real world. The first is by visually comparing the two profiles, optionally in more detail by 3D scanning the profile. The second is by analysing the influence on the process, in this case the filtering of the particles, and find out if similar changes over time are noticed. Unfortunately, both types of real world data are unavailable to us at the time of writing. However, there is still much to say about the influence of the wear coefficient.

6.1 Number of recursive calls

As explained before, moving the vertices to match a certain change in volume is a iterative process, as there is no analytical solution. Trying to solve it results in higher order equations, which are hard — if not impossible — to solve by hand. When the change in volume is very small the higher order terms have less influence, causing the solution to be more linear and therefore converge quicker. This is indeed observed in a simple test where a randomised mesh is given random displacements and a target volume change, where both are multiplied by the same factor. For lower values of that factor the solution converged quicker, as seen in Figure 6.3, where the number of (recursive) calls needed is plotted against the factor used. From a factor of around 0.1 the number of calls start to go to infinity. Higher values were tried, but for clarity not shown in the plot, as from around 0.86 onward the number of calls reached a 10001 limit.

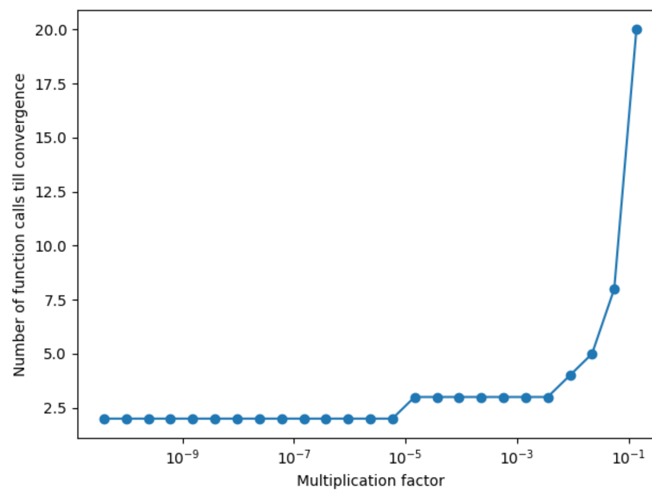


Figure 6.3: The number of (recursive) function calls needed to match the moving vertices of a mesh to the required change in volume, plotted against a multiplication factor. The multiplication factor indicates the severity of the initial displacements and the required volume change.

These results show that higher wear coefficients need more iterations to converge to the target volume change. It also shows that there is a limit on how high the wear coefficient can be before the simulation breaks, although this most likely also depend on other factors, such as the mesh density. A limit is imposed on the number of recursive calls that can be made, both for safety and to prevent excessive run times. When the limit is reached, the solution at that point is still processed, which of course is less accurate. It might therefore be the case that higher wear coefficients are more likely to have less accurate results.

6.2 Number of warning messages

If this is true, a higher number of recursion-limit-reached warnings are expected for higher wear coefficients. Figure 6.4 shows this for the 8 simulations that were run. The three highest wear coefficients can be ignored, as the simulations were stopped well before they were finished due to the wear being unreasonably high. Clearly, the number of warnings increase as the wear coefficient increases. Interestingly though, the lowest three wear coefficients all have a similar number of warnings, after which it jumps up with an order of magnitude. This might be an indication that the wear coefficient should not be increased passed this stage, as then the result starts to be less accurate and eventually falls apart. Another reason could be that the average number of recursions needed is closer to the limit and exceeding the limit therefore happens more often. This is, however, not likely the case, as the lower wear coefficients do not show an upwards trend and from Figure 6.3 we do expect a huge growth once a critical point is reached.

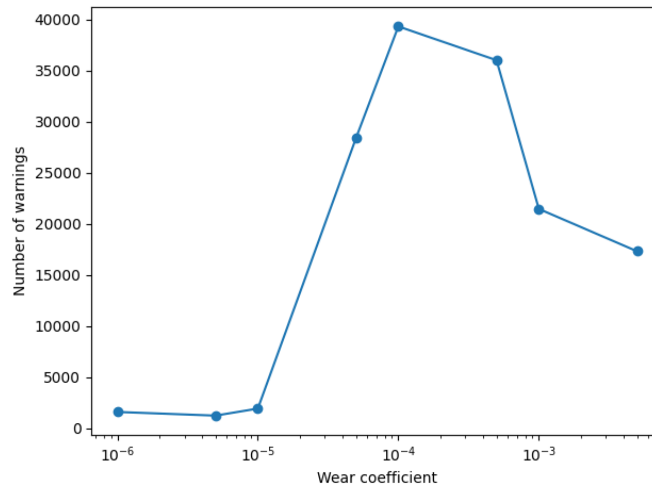


Figure 6.4: The total number of recursion-limit-reached warnings for different wear coefficients. Note: the simulations with the highest three wear coefficients were cut short due to unreasonably much wear.

It must be noted that although 40 000 warnings sounds like a lot, the function has been called 50 000 000 times in total, so it is just 0.08% of the time. Furthermore, the solution processed does not necessarily have to be completely wrong, as it might just mean that effectively a slightly higher tolerance is used. The number of warnings also do not seem to be affected by the wear of previous time steps, as they do not get worse over time. This is shown in Figure 6.5, where the number of warnings are grouped in bins of 3333 time steps.

6.3 The wear over time

The wall vtu files can be used to see how the wear evolves over time. After correcting the vertices position for the vibrating motion, they can be compared to their initial position. There are two ways to quantify the wear. The first being the average distance travelled by each vertex and the second being the actual change in volume. The first method is quicker and easier to implement, while the second method should be a bit more precise. The results for each are shown in Figures 6.6(a) and 6.6(c), respectively.

From the wear equation it is clear that the wear is a linear function of the wear coefficient. This is true on a time step level, but it does not have to be true on a larger timescale. If it is true, dividing the wear by the wear coefficient should result in similar values, as is done for Figures 6.6(b) and 6.6(d). Although they do follow a very similar trend, they do not quite line up perfectly. Indeed, the ratio of the resulting wears is always a bit lower than the ratio of the wear coefficients

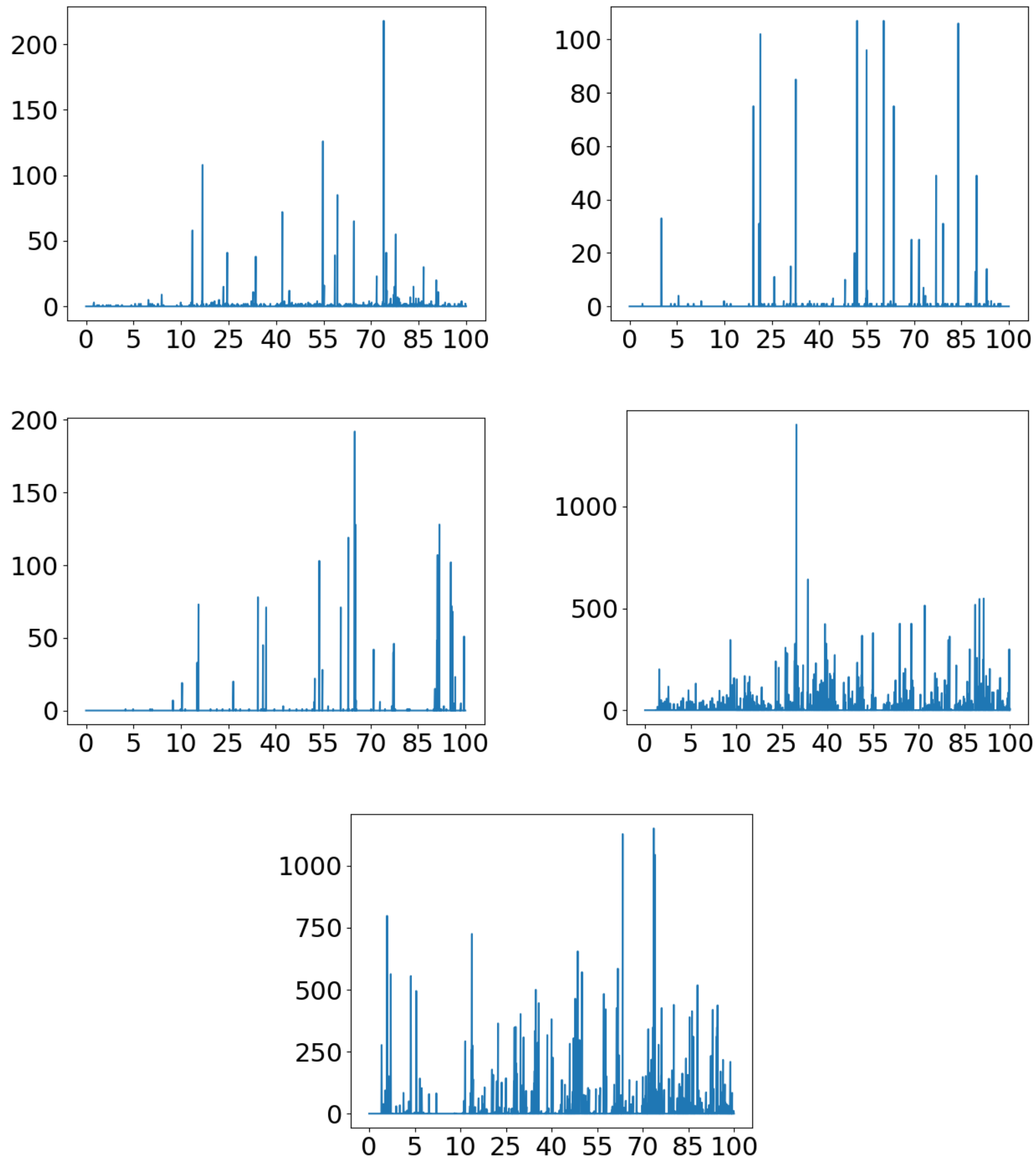


Figure 6.5: The number of recursion-limit-reached warnings, grouped in bins of 3333 time steps, plotted against time (s). The wear coefficients are, from left to right, top to bottom: 1e-6, 5e-6, 1e-5, 5e-5, 1e-4.

used. This is also clearly shown in Figure 6.7, where for every $10s$ the normalised volume change is plotted against the wear coefficient. Judging from the logarithmic version of the plot, it is possible that there is a exponential relationship, however there is too little data to say with any certainty. In other words, the wear is relatively lower, the higher the wear coefficient is.

One reason for the lower relative wear could be the load of the particles on the wall. When the mesh has evolved, the overlap of the particles and the wall will be less, and they might even come loose. This results in a lower normal load, resulting in turn in less wear to occur for the next time step. However, when the wear coefficient is small, the wall will not evolve as much and the load of the particles will remain more constant throughout the time steps. This then results in relatively more wear to occur.

Could it be that if often the same number of iterations are needed, this would lead to a positive

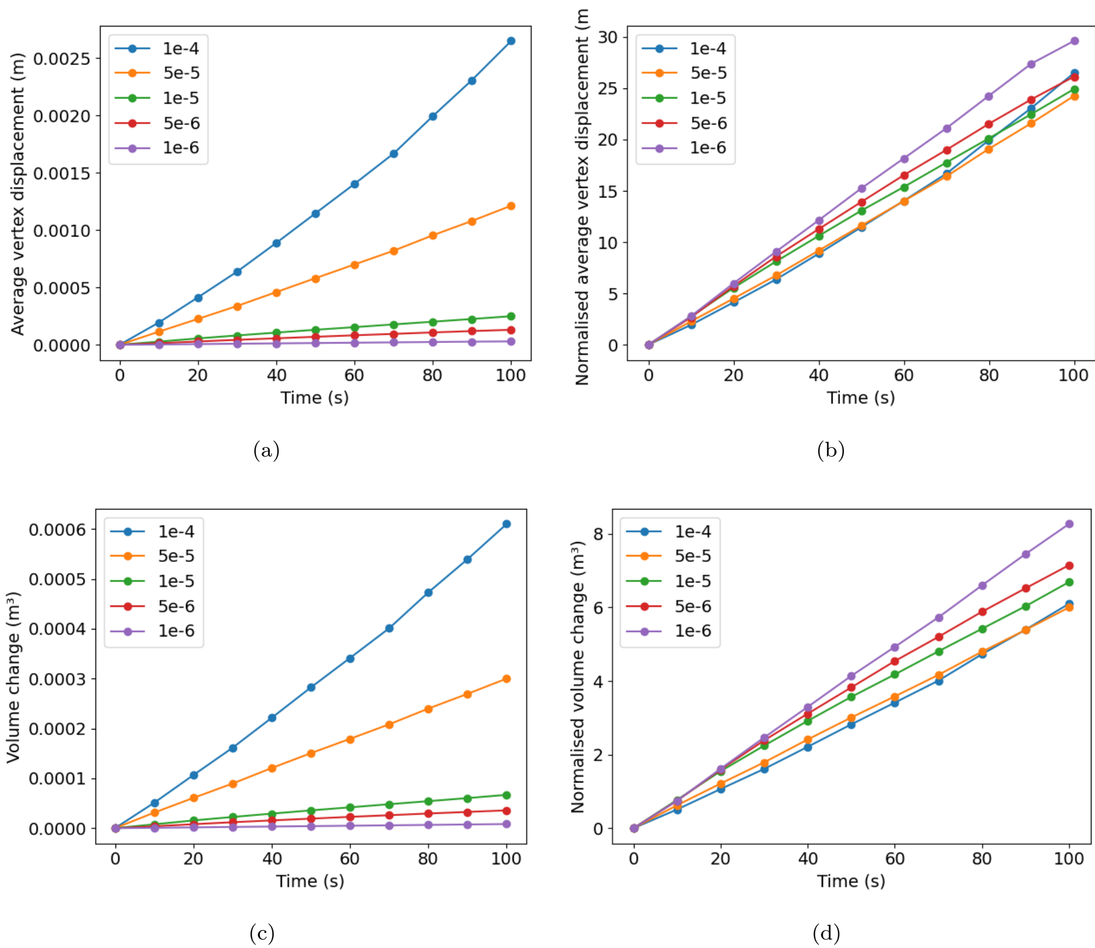


Figure 6.6: The average vertex displacement (top) and volume change (bottom) over time. On the right, the left plot normalised by the wear coefficient.

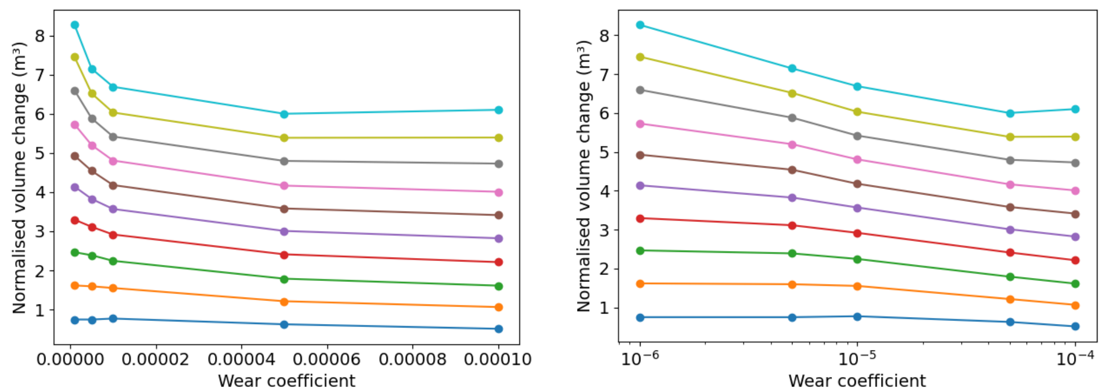


Figure 6.7: The volume change — normalised by the wear coefficient — plotted against the wear coefficient, for different times with a time interval of $10s$. Both linear (left) and logarithmic (right) x-axis are shown.

or negative bias for the absolute error? In this case a negative bias towards too little wear. The absolute error is, of course, bigger for bigger volume changes, since only the relative error is used for the tolerance. Between the different simulations the only variable indicating how much the volume

change should be is the wear coefficient. This means the absolute error is directly proportional to the wear coefficient, meaning that normalising it by the wear coefficient would again lead to the same relative error. So no, this cannot be a reason. Furthermore, the absolute errors are incredibly small, so they cannot account for the big differences that we see here.

6.4 Conclusion

From this brief example implementation the following can be concluded. There is a limit to how high the wear coefficient can be, because at some point too many iterations are needed to evolve the mesh. This will slow down the code and make it less accurate in case the recursion limit is reached. Secondly, the higher the wear coefficient is, the lower the relative wear is. This does not have to mean that the result is less accurate, but it is something that should be considered when comparing to the real world. All simulations need some wear acceleration anyway, as it is impossible to mimic the wear exactly and also expect the simulation to finish within a reasonable time span. Although proper verification and validation are still needed, for now we can conclude that the implementation looks promising.

6.5 Recommendation

Besides doing more verification and validation, many more things can be researched, which include, but are not limited to:

1. How does the simulated wear compare to that in the real world?
2. What is the exact relationship between the wear and the wear coefficient on a larger timescale?
3. What causes the iterative process of moving the vertices to sometimes fail and how can this be improved?
4. How strict should the tolerance for matching the change in volume be?
5. How does the mesh density affect wear?
6. What are the differences between using an unstructured or a structured mesh?

Chapter 7

Conclusions and recommendations

7.1 Conclusions

This work has shown the potential of using DEM to simulate abrasive wear with active evolving surfaces. The simulated wear visually compares nicely to real life, as areas with more particle-wall interactions and a higher load are worn down more. Increasing the wear coefficient results in more wear to occur, of course, however the wear is relatively less when comparing it to the wear using a lower wear coefficient. This can be explained by the normal force of the particles reducing, the further the surface moves away from them. The wear coefficient here is the umbrella term of the wear coefficient, the surface hardness, and the numerical wear acceleration factor, which are all multiplied together to give a single constant.

Evolvable surfaces were made in two different ways, namely using NURBS and using triangle meshes. For the former some quasi-2D results were obtained. The latter extended this to full 3D surfaces which were able to wear down in all directions. It did have a limit in how much the wear could be accelerated, because at some point the iterative process of evolving the surface failed to converge.

7.2 Recommendations

After some quasi-2D results, NURBS were abandoned because the evolving part was hard to extend to 3D. No known method for calculating the volume difference between two NURBS surfaces exists, nor were we able to derive one. It could still potentially be interesting to look into for someone with a mathematical background.

For anyone else we recommend to continue with the triangle mesh implementation, but to give the implementation a complete overhaul to be more in line with proper programming concepts, e.g. use pointers to update all vertices at once, instead of having all kinds of index references. A potential better approach, however, could be to use tetrahedrons instead. This eliminates some of the problems and edge cases the triangle mesh has. In any case, a lot of verification and validation is still needed, so do not assume the current implementation is perfect.

Bibliography

[1] D. Boemer and J. P. Ponthot. "A generic wear prediction procedure based on the discrete element method for ball mill liners in the cement industry". In: *Minerals Engineering* 109 (2017), pp. 55–79.

[2] D. Forsström and P. Jonsén. "Calibration and validation of a large scale abrasive wear model by coupling DEM-FEM; Local failure prediction from abrasive wear of tipper bodies during unloading of granular material". In: *Engineering Failure Analysis* 66 (2016), pp. 274–283.

[3] Pawan Gami. *NURBS Demo - Evaluator for Non Uniform Rational B-Splines*. `http://nurbscalculator.in/`.

[4] A. Jafari and R. Abbasi Hattani. "Investigation of parameters influencing erosive wear using DEM". In: *Friction* 8 (2020), pp. 136–150.

[5] J. T. Kalala, M. Bwalya, and M. H. Moys. "Discrete element method (DEM) modelling of evolving mill liner profiles due to wear. Part II. Industrial case study". In: *Minerals Engineering* 18 (2005), pp. 1392–1397.

[6] Les Piegl and Wayne Tiller. *The NURBS Book*. Monographs in Visual Communications. Springer-Verlag, 1997. ISBN: 9783540615453.

[7] G. W. Stachowiak and A. W. Batchelor. "Abrasive, Erosive and Cavitation Wear". In: *Tribology Series* 24 (1993), pp. 557–612.

[8] *VTK User's Guide*. 11th ed. Kitware, 2010.

[9] T. Weinhart et al. "Fast, flexible particle simulations - An introduction to MercuryDPM". In: *Computer Physics Communications* 249 (2020), pp. 1353–1360.

[10] T. Weinhart et al. "MercuryDPM: A Fast and Flexible Particle Solver Part A: Technical Advances". In: *Springer Proceedings in Physics* 188 (2016), pp. 1353–1360.

[11] Wikipedia. *Archard equation*. `https://en.wikipedia.org/wiki/Archard_equation`.

[12] Wikipedia. *Non-uniform rational B-spline*. `https://en.wikipedia.org/wiki/Non-uniform_rational_B-spline`.

[13] R. Xia et al. "Discrete Element Method- (DEM-) Based Study on the Wear Mechanism and Wear Regularity in Scraper Conveyor Chutes". In: *Mathematical Problems in Engineering* 2019 (2019).

Appendix A

Source code

During this project, MercuryDPM was still using SVN as its version control system, which later changed to git. Within the SVN repository a new branch was created for this project where all the new codes, changes to existing codes, bug fixes, etc. were committed. Many of these things were not directly related to the project per se, but were very useful to MercuryDPM as a whole. Giving access to this branch would be the easiest way to share all the coding work, however due to MercuryDPM changing its version control system to git, the SVN repository is not accessible anymore. Many branches in SVN were moved across to git by simultaneously merging it into the master and this branch was no exception. The single merge commit for this can be found by the hash `aa3ab0c5` (long hash `aa3ab0c5339f171d9a8bbdc7d71ae13fdc1951a6`).

The MercuryDPM source code is publicly available at `https://bitbucket.org/mercurydpm/mercurydpm/src/master/`, which can be viewed via a web browser without the need for full installation. To get a closer look at the exact changes due to this merge commit, you would have to `git clone` the repository, i.e.

```
git clone https://bitbucket.org/mercurydpm/mercurydpm.git
```

This allows you to, for example, list a summary of all changes by running

```
git diff --shortstat aa3ab0c5^..aa3ab0c5
```

which gives

```
42 files changed, 5317 insertions(+), 341 deletions(-)
```

Other useful flags besides `--shortstat` are `--name-only`, `--name-status`, and `--numstat`. The result of the latter is shown in Table A.1. Some important bug fix in the restart files was already merged to the master at an earlier date, so does not show up in this list. It involved about 30 restart files and a few lines of code in the read and write functions in DPMBase.cc.

To compile and run the code, a full installation is required for which the instructions can be found at `https://www.mercurydpm.org/downloads/developers-version`. Once fully installed, navigate to `Drivers/USER/JanWillem/` in the build directory to compile (`make Screen`) and run (`./Screen`) the vibrating screen simulation from Chapter 6.

(+)	(-)	**file**
1	1	Drivers/SelfTests/Boundaries/SelfTestData/PolydisperseInsertionBoundarySelfTest.restart
1	1	Drivers/Sinter/SelfTestData/SinterBed0.restart
1	1	Drivers/Sinter/SelfTestData/SinterBed1.restart.5
23	0	Drivers/USER/JanWillem/CMakeLists.txt
82	0	Drivers/USER/JanWillem/NurbsWallEdgeCaseSelfTest.cpp
324	0	Drivers/USER/JanWillem/Screen.cpp
504	0	Drivers/USER/JanWillem/SelfTestData/NurbsWallEdgeCaseSelfTest.data
38	0	Drivers/USER/JanWillem/SelfTestData/NurbsWallEdgeCaseSelfTest.restart
2	2	Drivers/UnitTests/SelfTestData/MovingWallsInfiniteWallInteraction$_1$39.vtu
3	3	Kernel/BaseInteractable.h
228	108	Kernel/DPMBase.cc
33	27	Kernel/DPMBase.h
5	0	Kernel/InteractionHandler.cc
2	0	Kernel/InteractionHandler.h
36	0	Kernel/Math/Quaternion.cc
10	0	Kernel/Math/Quaternion.h
397	74	Kernel/Nurbs/NurbsSurface.cc
146	32	Kernel/Nurbs/NurbsSurface.h
150	0	Kernel/Nurbs/NurbsUtils.cc
38	0	Kernel/Nurbs/NurbsUtils.h
2	2	Kernel/VTKWriter/InteractionVTKWriter.cc
222	0	Kernel/VTKWriter/VTKData.cc
137	0	Kernel/VTKWriter/VTKData.h
104	0	Kernel/VTKWriter/WallDetailsVTKWriter.cc
66	0	Kernel/VTKWriter/WallDetailsVTKWriter.h
82	50	Kernel/WallHandler.cc
60	5	Kernel/WallHandler.h
5	0	Kernel/Walls/BaseWall.h
53	8	Kernel/Walls/NurbsWall.cc
7	7	Kernel/Walls/NurbsWall.h
946	0	Kernel/Walls/TriangleMeshWall.cc
314	0	Kernel/Walls/TriangleMeshWall.h
32	6	Kernel/Walls/TriangleWall.cc
17	0	Kernel/Walls/TriangleWall.h
56	13	Kernel/Walls/TriangulatedWall.cc
11	1	Kernel/Walls/TriangulatedWall.h
404	0	Kernel/Walls/WearableNurbsWall.cc
105	0	Kernel/Walls/WearableNurbsWall.h
185	0	Kernel/Walls/WearableTriangleMeshWall.cc
136	0	Kernel/Walls/WearableTriangleMeshWall.h
254	0	Kernel/Walls/WearableTriangulatedWall.cc
95	0	Kernel/Walls/WearableTriangulatedWall.h

Table A.1: Number of insertions (+) and deletions (-) per changed file for the merge commit `aa3ab0d5`.

Appendix B

NURBS

All codes in this chapter can be copied to a `script_name.py` file and run as `python3 script_name.py`. For some scripts it is necessary that some of the other scripts also exists in the same folder.

B.1 piecewise_polynomials.py

```
import numpy as np
from matplotlib import pyplot as plt


# Create two figures and axes for C0 and C1 continuity plots.
fig_c0, ax_c0 = plt.subplots()
fig_c1, ax_c1 = plt.subplots()
ax_c0.set_aspect('equal')
ax_c1.set_aspect('equal')
ax_c0.set_xlabel('u')
ax_c0.set_ylabel('C(u)')
ax_c1.set_xlabel('u')
ax_c1.set_ylabel('C(u)')


# First segment.
u = np.linspace(0, 1, 100)
C0 = u**2 - u + 1
C1 = u**2 - 1.5*u + 1.5
ax_c0.plot(u, C0)
ax_c1.plot(u, C1)

# Second segment.
u = np.linspace(1, 2, 100)
C0 = -u**2 + 2.5*u - 0.5
C1 = -u**2 + 2.5*u - 0.5
ax_c0.plot(u, C0)
```

```
ax_c1.plot(u, C1)

# Third segment.
u = np.linspace(2, 3, 100)
C0 = u**2 - 5*u + 6.5
C1 = u**2 - 5.5*u + 7.5
ax_c0.plot(u, C0)
ax_c1.plot(u, C1)


# For both figures, the x limits are the same, but the y limits differ.
# Get both y limits, find the extremes, and set both to be equal.
ylim_c0 = ax_c0.get_ylim()
ylim_c1 = ax_c1.get_ylim()
ylim_min = min(ylim_c0[0], ylim_c1[0])
ylim_max = max(ylim_c0[1], ylim_c1[1])
ax_c0.set_ylim([ylim_min, ylim_max])
ax_c1.set_ylim([ylim_min, ylim_max])


# Save both figures.
filename_c0 = 'piecewise_polynomial_c0.png'
filename_c1 = 'piecewise_polynomial_c1.png'
fig_c0.savefig(filename_c0, dpi=200, bbox_inches='tight')
print(f'Plot saved: {filename_c0}')
fig_c1.savefig(filename_c1, dpi=200, bbox_inches='tight')
print(f'Plot saved: {filename_c1}')
```

B.2 basis_functions.py

```
import numpy as np
from matplotlib import pyplot as plt
from sympy import symbols, lambdify


def get_basis_function(basis_functions, p, knots, u, i):
  """
  Returns a list of formulas per knot span.
  Assumes for all p > 0 that the lower degree basis functions already exist in the 3D list of basis
↪   functions.

  :param basis_functions: 3D list of basis functions, with all lower degree basis functions already
↪   present when p > 0.
  :param p: degree
  :param knots: knot vector
  :param u: parametric variable
  :param i: basis function iterator
```

```
    :return: list of formulas per knot span
    """

    if p == 0:
        N = np.zeros(len(knots) - 1)
        N[i] = 1
        return N
    else:
        left = basis_functions[p-1][i]
        right = basis_functions[p-1][i+1]
        f = (u - knots[i]) / (knots[i+p] - knots[i]) if knots[i+p] - knots[i] != 0 else 0
        g = (knots[i+p+1] - u) / (knots[i+p+1] - knots[i+1]) if knots[i+p+1] - knots[i+1] != 0 else 0
        return f * left + g * right


def get_basis_functions(degree_max, knots, u):
    """
    Returns a 3D list, with for each degree a list of basis functions, and for each basis function a list
↪   of formulas per knot span.
    1st dimension: iterates over the degree; from 0 to degree_max.
    2nd dimension: iterates over the basis functions; from 0 to the number of knot spans minus the degree.
    3rd dimension: iterates over the formulas per knot span; from 0 to the number of knot spans.

    For example, for a max degree of 2 and in case of 4 knot spans, the resulting 3D matrix looks like:
    [ [ [ f000, f001, f002, f003 ], [ f010, f011, f012, f013 ], [ f020, f021, f022, f023 ], [ f030, f031,
↪   f032, f033 ] ],
      [ [ f100, f101, f102, f103 ], [ f110, f111, f112, f113 ], [ f120, f121, f122, f123 ] ],
      [ [ f200, f201, f202, f203 ], [ f210, f211, f212, f213 ] ] ]
    E.g. f012 is the formula for the third (index 2) knot span, of the second (index 1) basis function, for
↪   degree 0 (index 0).

    :param degree_max: maximum degree to evaluate
    :param knots: knot vector
    :param u: parametric variable
    :return: 3D list, with for each degree a list of basis functions, and for each basis function a list of
↪   formulas per knot span.
    """

    basis_functions = []

    # For each degree.
    for p in range(degree_max + 1):
        # Append new empty list.
        basis_functions.append([])
        # For each basis function.
        for i in range(len(knots) - 1 - p):
            # Append list of formulas.
            basis_functions[p].append(get_basis_function(basis_functions, p, knots, u, i))
```

```
  return basis_functions


def evaluate_lambda(func, x):
  """
  Evaluates the given lambda function by the given input value(s) and returns the result with the same
↪   shape as the input shape.
  Regular evaluation of constant functions would not yield the input shape, but return a single constant
↪   instead.
  In such a case, this function will, for example, return an array of constants instead.

  :param func: single variable lambda function to evaluate
  :param x: value(s) to evaluate lambda function with
  :return: result of evaluation
  """

  return func(x) + 0*x


def get_basis_function_x_y(basis_function, knots, u):
  """
  Evaluates the formulas of the basis function for each knot span.

  :param basis_function: list of formulas per knot span defining the basis function
  :param knots: knot vector
  :param u: parametric variable
  :return: tuple with 2 arrays for horizontal and vertical axis, respectively
  """

  xs = np.array([])
  ys = np.array([])

  # For each knot span.
  for i in range(len(knots) - 1):
    # Generate uniformly spaced values over the knot span and evaluate the formula.
    x = np.linspace(knots[i], knots[i+1], 100)
    y = evaluate_lambda(lambdify(u, basis_function[i]), x)
    xs = np.append(xs, x)
    ys = np.append(ys, y)

  return (xs, ys)


def plot_basis_functions(basis_functions, degree_max, knots, u):
  """
  Plots the basis functions for each degree in a separate plot, together with the sum of the basis
↪   functions.

  :param basis_functions: 3D list of basis functions
```

```
  :param degree_max: maximum degree to plot
  :param knots: knot vector
  :param u: parametric variable
  """

  # Increase a few font sizes.
  plt.rc('font', size=24)
  plt.rc('xtick', labelsize=18)
  plt.rc('ytick', labelsize=18)

  for p in range(degree_max + 1):
    # Clear figure for next iteration. Set equal aspect ratio and domain [0, 1] (plus small margin).
    plt.clf()
    plt.gca().set_aspect('equal')
    plt.gca().set_xlim([-0.05, 1.05])
    plt.gca().set_ylim([-0.05, 1.05])

    # Plot vertical lines to indicate the knot spans.
    for i in knots:
      plt.plot([i, i], [0, 1], '--', color='gray', linewidth=2)

    # Plot the individual basis functions.
    for i in range(len(basis_functions[p])):
      (x, y) = get_basis_function_x_y(basis_functions[p][i], knots, u)
      # Ignore the parts where y is 0.
      x = x[y != 0]
      y = y[y != 0]
      plt.plot(x, y, label=f'$N_{{{i},{p}}}$', linewidth=7)

    # Plot the summed basis functions.
    (x, y) = get_basis_function_x_y(sum(basis_functions[p][:]), knots, u)
    plt.plot(x, y, 'k', dashes=[1, 1], label='$N_{sum}$', linewidth=4)

    plt.legend(loc='upper right')
    filename = f'basis_functions_degree_{p}.png'
    plt.savefig(filename, dpi=200, bbox_inches='tight')
    print(f'Plot saved: {filename}')


def main():
  u = symbols('u')
  degree = 4
  knots = np.array([0.0, 0.2, 0.4, 0.6, 0.8, 1.0])
  basis_functions = get_basis_functions(degree, knots, u)
  plot_basis_functions(basis_functions, degree, knots, u)


if __name__ == '__main__':
  main()
```

B.3 basis_splines.py

```
import numpy as np
from matplotlib import pyplot as plt
from sympy import symbols, lambdify
from types import SimpleNamespace
from basis_functions import get_basis_functions, evaluate_lambda


def plot_control_points_polygon(control_points, ax=None):
  """
  Plots the polygon formed by the control points.

  :param: List of control points.
  :param ax: Axes to plot on. If None, the current axes is used.
  """

  if ax is None:
    ax = plt.gca()

  for i in range(len(control_points)):
    plt.plot(control_points[i][0], control_points[i][1], 'o', color='gray', markerfacecolor='none')
    if i > 0:
      plt.plot([control_points[i-1][0], control_points[i][0]], [control_points[i-1][1],
      ↪  control_points[i][1]], color='gray', linewidth=1)


def evaluate_basis_spline_at_knot_span(AZ, knot_span_index, num_evaluate):
  """
  Evaluates the B-spline on the given knot span.

  :param AZ: SimpleNamespace with all B-spline properties.
  :param knot_span_index: Index of the knot span to evaluate.
  :param num_evaluate: The number of points to evaluate.
  :return: List of x-, y-, and z-values.
  """

  # Uniformly spaced list of parametric values to evaluate.
  t = np.linspace(AZ.knots[knot_span_index], AZ.knots[knot_span_index + 1], num_evaluate)
  # Change to column vector.
  t = t[:, None]

  # Initialize list of zeros.
  xyz = np.zeros((num_evaluate, 3))

  # Loop over control points / basis functions.
```

```python
  for i in range(len(AZ.control_points)):
    # Get the part of the basis function for the knot span of interest and evaluate it.
    f = AZ.basis_functions[AZ.degree][i][knot_span_index]
    g = evaluate_lambda(lambdify(AZ.u, f), t)
    # Multiply by the control point and add to sum.
    xyz += g * AZ.control_points[i]

  return xyz


def get_basis_spline_x_y_z(AZ):
  """
  Gets a list of x-, y-, and z-values per internal knot span of the B-spline.
  1st dimension: number of evaluated points per knot span.
  2nd dimension: x-, y-, and z-values.
  3rd dimension: the internal knot spans.

  :param AZ: SimpleNamespace with all B-spline properties.
  :return: List with x-, y-, and z-values.
  """

  num_evaluate = int(15)
  xyz = np.empty((num_evaluate, 3, len(AZ.knots) - 2*AZ.degree - 1))

  # Loop over internal knot spans.
  for i in range(AZ.degree, len(AZ.knots) - AZ.degree - 1):
    xyz[:, :, i - AZ.degree] = evaluate_basis_spline_at_knot_span(AZ, i, num_evaluate)

  return xyz


def plot_basis_spline(AZ, filename):
  """
  Plots the B-spline in xy-plane.

  :param AZ: SimpleNamespace with all B-spline properties.
  """

  plt.clf()
  plt.gca().set_aspect('equal')

  plot_control_points_polygon(AZ.control_points)

  xyz = get_basis_spline_x_y_z(AZ)
  plt.plot(xyz[:, 0, :], xyz[:, 1, :], linewidth=2)

  plt.savefig(filename, dpi=200, bbox_inches='tight')
  print(f'Plot saved: {filename}')
```

```python
def get_uniform_knot_vector(AZ):
    """
    Creates a uniformly spaced knot vector on the range [0, 1].

    :param AZ: SimpleNamespace with all B-spline properties.
    :return: The knot vector.
    """

    num_knots = len(AZ.control_points) + AZ.degree + 1
    return np.linspace(0.0, 1.0, num_knots)


def main():
    # Simple way to store all properties from a to z. Easy to extend and use between the different scripts
    ↪ and their functions.
    AZ = SimpleNamespace()
    AZ.u = symbols('u')

    # Start with a few control points.
    AZ.degree = 0
    AZ.control_points = np.array([[0.0, 0.0, 0.0], [1.0, 2.0, 0.0], [2.0, 2.0, 0.0], [3.0, 1.0, 0.0], [4.0,
    ↪ 2.0, 0.0]])
    AZ.knots = get_uniform_knot_vector(AZ)
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    plot_basis_spline(AZ, 'basis_spline_0.png')

    # Increment degree.
    AZ.degree = 1
    AZ.knots = get_uniform_knot_vector(AZ)
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    plot_basis_spline(AZ, 'basis_spline_1.png')

    # Increment degree.
    AZ.degree = 2
    AZ.knots = get_uniform_knot_vector(AZ)
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    plot_basis_spline(AZ, 'basis_spline_2.png')

    # Increment degree.
    AZ.degree = 3
    AZ.knots = get_uniform_knot_vector(AZ)
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    plot_basis_spline(AZ, 'basis_spline_3.png')

    # Back to degree 2. Add another control point.
    AZ.degree = 2
    AZ.control_points = np.vstack([AZ.control_points, [3.5, 0.25, 0.0]])
    AZ.knots = get_uniform_knot_vector(AZ)
```

```
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    plot_basis_spline(AZ, 'basis_spline_4.png')

    # Add a few more control points.
    AZ.control_points = np.vstack([AZ.control_points, [2.0, 1.0, 0.0], [1.25, 0.5, 0.0]])
    AZ.knots = get_uniform_knot_vector(AZ)
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    plot_basis_spline(AZ, 'basis_spline_5.png')

    # Update the position of a control point. The knot vector and degree remain unchanged, so no need to
    ↪  recalculate the basis functions.
    AZ.control_points[4][1] = 1.0
    plot_basis_spline(AZ, 'basis_spline_6.png')

    # Increase the degree.
    AZ.degree = 3
    AZ.knots = get_uniform_knot_vector(AZ)
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    plot_basis_spline(AZ, 'basis_spline_7.png')

    # Move control point back to original position.
    AZ.control_points[4][1] = 2
    plot_basis_spline(AZ, 'basis_spline_8.png')

    # Clamp the knot vector at the start.
    AZ.knots[:AZ.degree+1] = AZ.knots[0]
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    plot_basis_spline(AZ, 'basis_spline_9.png')

    # Clamp the knot vector at the end.
    AZ.knots[-AZ.degree-1:] = AZ.knots[-1]
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    plot_basis_spline(AZ, 'basis_spline_10.png')

    # Repeat the first degree control points at the end, to make it a closed curve.
    AZ.control_points = np.vstack([AZ.control_points, AZ.control_points[:AZ.degree]])
    AZ.knots = get_uniform_knot_vector(AZ)
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    plot_basis_spline(AZ, 'basis_spline_11.png')


if __name__ == '__main__':
    main()
```

B.4 rational_basis_splines.py

```
import numpy as np
from matplotlib import pyplot as plt, colors
from sympy import symbols, lambdify
from types import SimpleNamespace
from basis_functions import get_basis_functions, evaluate_lambda
import basis_splines as bs
import colorsys


def evaluate_basis_spline_at_knot_span(AZ, knot_span_index, num_evaluate):
  """
  Evaluates the rational B-spline on the given knot span.

  :param AZ: SimpleNamespace with all rational B-spline properties.
  :param knot_span_index: Index of the knot span to evaluate.
  :param num_evaluate: The number of points to evaluate.
  :return: List of x-, y-, and z-values.
  """

  # Uniformly spaced list of parametric values to evaluate.
  t = np.linspace(AZ.knots[knot_span_index], AZ.knots[knot_span_index + 1], num_evaluate)
  # Change to column vector.
  t = t[:, None]

  # Initialize list of zeros.
  xyz = np.zeros((num_evaluate, 3))
  w = np.zeros((num_evaluate, 1))

  # Loop over control points / basis functions.
  for i in range(len(AZ.control_points)):
    # Get the part of the basis function for the knot span of interest and evaluate it.
    f = AZ.basis_functions[AZ.degree][i][knot_span_index]
    g = evaluate_lambda(lambdify(AZ.u, f), t)
    # Multiply by the control point and weight and add to sum.
    xyz += g * AZ.control_points[i] * AZ.weights[i]
    w += g * AZ.weights[i]

  # Normalise by the sum of the weights.
  xyz /= w

  return xyz


def scale_colour_lightness(rgb, scale):
  """
  Scales the lightness of the given colour and returns it.
```

```
  :param rgb: Colour to be scaled.
  :param scale: How much to scale. < 1: darker; > 1: lighter.
  :return: The scaled colour.
  """

  # Convert rgb to hls.
  h, l, s = colorsys.rgb_to_hls(*rgb)
  # Manipulate h, l, s values and return as rgb.
  return colorsys.hls_to_rgb(h, min(1, l * scale), s = s)


def plot_rational_basis_spline(AZ, label, ax=None):
  """
  Plots the rational B-spline onto the provided axes.

  :param AZ: SimpleNamespace with all rational B-spline properties.
  :param label: Legend label for this curve.
  :param ax: Axes to plot on. If None, the current axes are used.
  """

  if ax is None:
    ax = plt.gca()

  xyz = bs.get_basis_spline_x_y_z(AZ)

  # Plot each knot span separately.
  for i in range(len(xyz[0, 0, :])):
    if i == 0:
      # For the first knot span: auto generate the next line colour and remember it; label the curve.
      line, = ax.plot(xyz[:, 0, i], xyz[:, 1, i], label=label)
      line_color = line.get_color()
    elif i % 2 == 0:
      # For every even knot span: use the previous line colour; no label.
      ax.plot(xyz[:, 0, i], xyz[:, 1, i], color=line_color)
    else:
      # For every odd knot span: use the lightened previous line colour; no label.
      ax.plot(xyz[:, 0, i], xyz[:, 1, i],
       ↪  color=scale_colour_lightness(colors.ColorConverter.to_rgb(line_color), 1.5))


def calculate_and_plot(AZ, degree, weight_index):
  """
  Keeps all weights equal to 1, except for one.
  Calculates the necessary steps and plots the different rational B-splines in a single plot.

  :param AZ: SimpleNamespace with all NURBS properties.
  :param degree: The degree to use.
  :param weight_index: Index of the weight to update.
```

```
    """

    # Assign the degree and calculate the uniform knot vector and the basis functions. (Re-)initialize all
    ↪  weights to 1.
    AZ.degree = degree
    AZ.knots = bs.get_uniform_knot_vector(AZ)
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    AZ.weights = np.ones(len(AZ.control_points))

    plt.clf()
    bs.plot_control_points_polygon(AZ.control_points)

    # Plot the curve for different values of the weight.
    AZ.weights[weight_index] = 0.0
    plot_rational_basis_spline(AZ, f'$w_{{{weight_index}}}=0$')

    AZ.weights[weight_index] = 0.5
    plot_rational_basis_spline(AZ, f'$w_{{{weight_index}}}=0.5$')

    AZ.weights[weight_index] = 1.0
    plot_rational_basis_spline(AZ, f'$w_{{{weight_index}}}=1$')

    AZ.weights[weight_index] = 2.0
    plot_rational_basis_spline(AZ, f'$w_{{{weight_index}}}=2$')

    plt.gca().set_aspect('equal')
    plt.legend()
    filename = f'rational_basis_splines_degree_{AZ.degree}.png'
    plt.savefig(filename, dpi=200, bbox_inches='tight')
    print(f'Plot saved: {filename}')


def main():
    # Override function implementation.
    bs.evaluate_basis_spline_at_knot_span = evaluate_basis_spline_at_knot_span

    # Simple way to store all properties from a to z. Easy to extend and use between the different scripts
    ↪  and their functions.
    AZ = SimpleNamespace()
    AZ.u = symbols('u')

    AZ.control_points = np.array([[0.0, 0.0, 0.0], [1.0, 2.0, 0.0], [2.0, 2.0, 0.0], [3.0, 1.0, 0.0], [4.0,
    ↪  2.0, 0.0], [3.5, 0.25, 0.0], [2.0, 1.0, 0.0], [1.25, 0.5, 0.0]])

    # Change a weight value and make some plots to see the effect. Do this for different degrees.
    calculate_and_plot(AZ, 2, 3)
    calculate_and_plot(AZ, 3, 3)
```

```
if __name__ == '__main__':
  main()
```

B.5 non_uniform_rational_basis_splines.py

```
import numpy as np
from matplotlib import pyplot as plt
from sympy import symbols, lambdify
from types import SimpleNamespace
from basis_functions import get_basis_functions, evaluate_lambda, get_basis_function_x_y
import basis_splines as bs
import rational_basis_splines as rbs


def plot_basis_functions(AZ, label, ax):
  """
  Plots all basis functions onto the provided axes.

  :param AZ: SimpleNamespace with all NURBS properties.
  :param label: Legend label for these basis functions.
  :param ax: Axes to plot on.
  """

  # Plot the individual basis functions.
  for i in range(len(AZ.basis_functions[AZ.degree])):
    (x, y) = get_basis_function_x_y(AZ.basis_functions[AZ.degree][i], AZ.knots, AZ.u)
    # Ignore the parts where y is 0.
    x = x[y != 0]
    y = y[y != 0]
    if i == 0:
      # For the first knot span: auto generate the next line colour and remember it; label the curve.
      line, = ax.plot(x, y, label=label)
      line_color = line.get_color()
    else:
      # For the other knot spans: use the previous line colour; no label.
      ax.plot(x, y, color=line_color)


def calculate_and_plot(AZ, degree, knot_index):
  """
  Starts with a uniform knot vector and updates a single knot.
  Calculates the necessary steps and plots the basis functions and the resulting curves in separate
↪  plots.
  For each plot, shows the uniform and non-uniform result together.

  :param AZ: SimpleNamespace with all NURBS properties.
  :param degree: The degree to use.
```

```
:param knot_index: Index of the knot to update.
"""

# Create two figures and axes for plotting the basis functions and the NURBS curves separately.
fig_bf, ax_bf = plt.subplots()
fig_bs, ax_bs = plt.subplots()

# Assign the degree and calculate the initial uniform knot vector and the basis functions.
AZ.degree = degree
AZ.knots = bs.get_uniform_knot_vector(AZ)
AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)

# The knot will be updated as a fraction over the knot span length to the next knot.
fraction = 0.5
knot_updated = AZ.knots[knot_index] + fraction * (AZ.knots[knot_index+1] - AZ.knots[knot_index])

# Plot vertical lines to indicate the knot spans. Ignore the knot that will be updated.
for k in AZ.knots:
  if k != AZ.knots[knot_index]:
    ax_bf.plot([k, k], [0, 1], '--', color='gray', linewidth=1)

# Plot vertical line for the knot before and after being updated.
ax_bf.plot([AZ.knots[knot_index], AZ.knots[knot_index]], [0, 1], '--', color='blue', label='Knot
↪  before')
ax_bf.plot([knot_updated, knot_updated], [0, 1], '--', color='red', label='Knot after')

bs.plot_control_points_polygon(AZ.control_points, ax_bs)

# Plot the basis functions, the curve, and a marker for the knot.
plot_basis_functions(AZ, 'Uniform', ax_bf)
rbs.plot_rational_basis_spline(AZ, 'Uniform', ax_bs)
xyz = rbs.evaluate_basis_spline_at_knot_span(AZ, knot_index, 1)
ax_bs.plot(xyz[:, 0], xyz[:, 1], 'ok', markerfacecolor='none')

# Update the knot.
AZ.knots[knot_index] = knot_updated
AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)

# Plot the basis functions, the curve, and a marker for the knot.
plot_basis_functions(AZ, 'Non-uniform', ax_bf)
rbs.plot_rational_basis_spline(AZ, 'Non-uniform', ax_bs)
xyz = rbs.evaluate_basis_spline_at_knot_span(AZ, knot_index, 1)
ax_bs.plot(xyz[:, 0], xyz[:, 1], 'ok', markerfacecolor='none')

ax_bf.set_aspect('equal')
ax_bs.set_aspect('equal')
ax_bf.legend()
ax_bs.legend()
filename_bf = f'non_uniform_rational_basis_functions_degree_{AZ.degree}.png'
```

```
    filename_bs = f'non_uniform_rational_basis_splines_degree_{AZ.degree}.png'
    fig_bf.savefig(filename_bf, dpi=200, bbox_inches='tight')
    print(f'Plot saved: {filename_bf}')
    fig_bs.savefig(filename_bs, dpi=200, bbox_inches='tight')
    print(f'Plot saved: {filename_bs}')


def main():
    # Override function implementation.
    bs.evaluate_basis_spline_at_knot_span = rbs.evaluate_basis_spline_at_knot_span

    # Simple way to store all properties from a to z. Easy to extend and use between the different scripts
    ↪  and their functions.
    AZ = SimpleNamespace()
    AZ.u = symbols('u')

    # Create a simple curve with all weights equal to 1.
    AZ.control_points = np.array([[0.0, 0.0, 0.0], [1.0, 2.0, 0.0], [2.0, 2.0, 0.0], [3.0, 1.0, 0.0], [4.0,
    ↪  2.0, 0.0], [3.5, 0.25, 0.0], [2.0, 1.0, 0.0], [1.25, 0.5, 0.0], [2.0, 0.0, 0.0]])
    AZ.weights = np.ones(len(AZ.control_points))

    # Change a knot value and make some plots to see the effect. Do this for different degrees.
    knot_index = 6
    calculate_and_plot(AZ, 2, knot_index)
    calculate_and_plot(AZ, 3, knot_index)


    # Create a circle, using a triangle shape.
    AZ.control_points = np.array([[0.0, -1.0, 0.0], [-np.sqrt(3), -1.0, 0.0], [-np.sqrt(3)/2, 0.5, 0.0],
    ↪  [0.0, 2.0, 0.0], [np.sqrt(3)/2, 0.5, 0.0], [np.sqrt(3), -1.0, 0.0], [0.0, -1.0, 0.0]])
    AZ.weights = np.array([1.0, 1/2, 1.0, 1/2, 1.0, 1/2, 1.0])
    AZ.knots = np.array([0.0, 0.0, 0.0, 1/3, 1/3, 2/3, 2/3, 1, 1, 1])
    AZ.degree = len(AZ.knots) - len(AZ.control_points) - 1
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    bs.plot_basis_spline(AZ, 'non_uniform_rational_basis_spline_circle_from_triangle.png')

    # Create a circle, using a square shape.
    AZ.control_points = np.array([[0.0, -1.0, 0.0], [-1.0, -1.0, 0.0], [-1.0, 0.0, 0.0], [-1.0, 1.0, 0.0],
    ↪  [0.0, 1.0, 0.0], [1.0, 1.0, 0.0], [1.0, 0.0, 0.0], [1.0, -1.0, 0.0], [0.0, -1.0, 0.0]])
    AZ.weights = np.array([1.0, np.sqrt(2)/2, 1.0, np.sqrt(2)/2, 1.0, np.sqrt(2)/2, 1.0, np.sqrt(2)/2,
    ↪  1.0])
    AZ.knots = np.array([0.0, 0.0, 0.0, 1/4, 1/4, 1/2, 1/2, 3/4, 3/4, 1, 1, 1])
    AZ.degree = len(AZ.knots) - len(AZ.control_points) - 1
    AZ.basis_functions = get_basis_functions(AZ.degree, AZ.knots, AZ.u)
    bs.plot_basis_spline(AZ, 'non_uniform_rational_basis_spline_circle_from_square.png')


if __name__ == '__main__':
    main()
```

Appendix C

Wall details VTK

Every wall can be visualised in ParaView by triangulating it and writing the triangles to vtu files. Sometimes it is useful to view additional details of a wall, for example the NURBS control points, but this is not possible within the existing format. Therefore we need to write this to a separate vtu file, which was first done in a quick and dirty way in the NurbsWall class itself, however if more walls were to do this it would get messy and unmanageable very quickly. To that end, in line with particles, walls, interactions, etc. which each have their own VTKWriter class, a WallDetailsVTKWriter class is created. This is more involved to implement, but it is also more expandable and it fits better in the overall architecture of MercuryDPM.

C.1 Vtu file structure

The vtu files use unstructured grids in XML file format, for which detailed information can be found in the VTK User's Guide [8] Chapter 19.3. There are four main elements in the file, namely Points, Cells, PointData and CellData. The Points hold all positions in 3D space and the Cells specify which points are related to form a single cell. Figure 19-20 of the VTK User's Guide [8] show the different cell types. The PointData and CellData are data directly related to each point/cell, i.e. each row corresponds to the same row of the Points/Cells. The Cells consist of connectivity, offsets and types. The connectivity stores the indices of the points forming a cell and the offsets store the number of indices in each cell. The types store the type of cell as an integer, for example VTK_TRIANGLE=5 and VTK_PIXEL=8. Listing C.1 shows what an vtu file might look like, in this case forming a simple 4 point cell with each point a different radius, see Figure C.1.

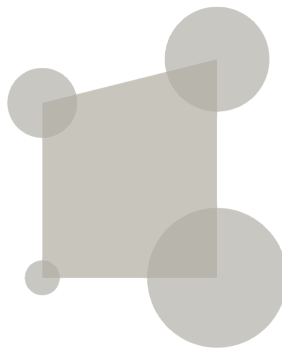


Figure C.1: Rendering of the example vtu file.

```xml
<?xml version="1.0"?>
<VTKFile type="UnstructuredGrid" version="0.1" byte_order="LittleEndian">
  <UnstructuredGrid>
    <Piece NumberOfPoints="4" NumberOfCells="1">
      <Points>
        <DataArray type="Float32" Name="Position" NumberOfComponents="3" format="ascii">
          -0.5 -0.5 0.0
          -0.5 0.5 0.0
          0.5 0.75 0.0
          0.5 -0.5 0.0
        </DataArray>
      </Points>
      <PointData Vectors="vector">
        <DataArray type="Float32" Name="Radius" format="ascii">
          0.1
          0.2
          0.3
          0.4
        </DataArray>
      </PointData>
      <Cells>
        <DataArray type="Int32" Name="connectivity" format="ascii">
          0 1 3 2
        </DataArray>
        <DataArray type="Int32" Name="offsets" format="ascii">
          4
        </DataArray>
        <DataArray type="UInt8"  Name="types" format="ascii">
          8
        </DataArray>
      </Cells>
    </Piece>
  </UnstructuredGrid>
</VTKFile>
```

Listing C.1: Example vtu file.

C.2 VTKData class

To make our lives easier a VTKData class is created, which stores all points, point data and cells (we are not interested in cell data at the moment). Each point is stored as Vec3D in a standard vector. Similarly the types are stored in a vector of integers and the connectivity is stored as a 2D vector of integers, see Listing C.2. The odd one out is the point data, as there can be a variable amount of them. They are stored in a unordered map with the key being the name of the point data. Since the number of components within each point data is fixed, there is no need to store the values in a 2D vector. Instead the values are stored in a single vector of doubles and the number of components is implied by the length of this vector and the number of points. This has the benefit of not having unnecessary overhead from many short vectors, which might only have to store a single value.

```cpp
std::vector<Vec3D> points_;
std::unordered_map<std::string, std::vector<Mdouble>> pointData_;
std::vector<std::vector<size_t>> connectivity_;
std::vector<int> types_;
```

Listing C.2: Data storage in the VTKData class.

With a few additional helper methods it is now really easy to create a vtu file, as shown in Listing C.3. Here the example vtu file from Listing C.1 is created, assuming the four points and corresponding radii are both stored in a vector.

```
VTKData data;
for (int i = 0; i < 4; i++)
{
    data.addToPoints(points[i]);
    data.addToPointData("Radius", radii[i]);
}
data.addToCells({0, 1, 3, 2});
data.addToTypes(8);
data.writeVTKData("Example.vtu");
```

Listing C.3: Code to create the example vtu file from Listing C.1.

C.3 General VTK writers structure and usage

The usual implementation and usage of a vtk writer class is best explained by looking at the one for walls. The BaseVTKWriter is a class template, which takes as type a handler, e.g. WallHandler. The class is also abstract, as it contains the pure virtual function writeVTK. All vtk writers inherit from the base class, providing the correct handler, and have to implement the writeVTK function. The WallVTKWriter does that by looping over the walls in the handler and calling the virtual method renderWall from the BaseWall class. It provides a VTKContainer instance, which is an already existing struct used to store points and connectivity indices and could be seen as a simpler form of the VTKData class, where by default the type is triangle strips, since all walls are triangulated for visualisation.

The vtu files can be written never, once or every save time, which is indicated by the file types NO_FILE, ONE_FILE and MULTIPLE_FILES, respectively. By default no files are written, but users can change this in their driver code via the WallHandler setWriteVTK method. The DPMBase class holds an instance of the WallVTKWriter class and every save time will call the writeVTK method, when the WallHandler getWriteVTK method indicates to do so.

C.4 Wall details

The WallDetailsVTKWriter also inherits from the BaseVTKWriter with the WallHandler as handler. The writeVTK method also loops through the walls in the handler and calls the virtual method writeWallDetailsVTK of the BaseWall class. The difference is that for each type of wall a separate vtu file is preferred, so each type of wall gets its own VTKData class instance. Therefore we first check if a file must be written for this type of wall and then do a dynamic cast to check if the type of wall match. Listing C.4 shows an example for the NurbsWall.

```
VTKData dataNurbsWall;
for (const auto& w : handler_)
{
    if (shouldWrite(WallHandler::DetailsVTKOptions::NURBSWALL)
        && dynamic_cast<NurbsWall*>(w))
    {
        w->writeWallDetailsVTK(dataNurbsWall);
    }
}
if (shouldWrite(WallHandler::DetailsVTKOptions::NURBSWALL))
    dataNurbsWall.writeVTKData(generateFileName("NurbsWall"));
```

Listing C.4: How the NurbsWall wall detail vtu file is generated in the writeVTK method of the WallDetailsVTKWriter class.

The generateFileName method is just for ease of use to have consistent filenames. The shouldWrite method checks whether or not a file should be written and is shown in Listing C.5. MULTIPLE_FILES should of course always be written, while ONE_FILE should only be written once when the file counter is zero.

```cpp
bool WallDetailsVTKWriter::shouldWrite(WallHandler::DetailsVTKOptions type) const
{
    FileType fileType = handler_.getWriteDetailsVTK(type);
    return (fileType == FileType::ONE_FILE && fileCounter == 0)
        || fileType == FileType::MULTIPLE_FILES
        || fileType == FileType::MULTIPLE_FILES_PADDED;
}
```

Listing C.5: Method to check if a file should be written.

To indicate for which type of wall vtu files should be written an enum class called DetailsVTKOptions is added to the WallHandler. Each type which has details vtu writing implemented must be added here as well, with a unique fixed integer value, since these are added to the restart file. An unordered_map, with as key-value the option and file type, stores which options are set. By default it remains empty, only the options which are set are added. This is done in the setWriteDetailsVTK method, which takes an option and a file type as arguments and adds them to the unordered_map. The getWriteDetailsVTK method takes an option as argument and tries to find the key in the unordered_map. On success the corresponding file type is returned, otherwise the default file type NO_FILE is returned. Sometimes it is useful to know if there are any files at all to be written, which the getWriteDetailsVTKAny method indicates. It returns true when there is at least one file in the unordered_map that has a file type not equal to NO_FILE. Listing C.6 shows the set, get and get any method.

```cpp
void WallHandler::setWriteDetailsVTK(DetailsVTKOptions option, FileType fileType)
{
    // Automatically adds option when it doesn't exist in the map yet
    writeDetailsVTK_[option] = fileType;
}

FileType WallHandler::getWriteDetailsVTK(DetailsVTKOptions option) const
{
    // Try to find the option to see if it's already added to the map or not
    auto it = writeDetailsVTK_.find(option);
    // When the iterator does not equal the end, the key has been found
    if (it != writeDetailsVTK_.end())
        return it->second;
    else
        return FileType::NO_FILE; // Else return a default file type
}

bool WallHandler::getWriteDetailsVTKAny() const
{
    // Simple lambda expression to check if any of the file types are not of type NO_FILE
    return std::any_of(writeDetailsVTK_.begin(), writeDetailsVTK_.end(),
                       [](const auto& p) { return p.second != FileType::NO_FILE; });
}
```

Listing C.6: The set, get and get any method for writing wall details to vtk.

When implementing the wall details for a new type of wall four things need to be done. First, the wall details option must be added to the WallHandler. Second, the should write and dynamic cast must be added to the writeVTK method in the WallDetailsVTKWriter. Third, the writeDetailsVTK method of the BaseWall must be overwritten for that wall to implement the actual vtu structure. Fourth, optional but highly recommended, the writePythonFileForVTKVisualisation method in DPMBase can be edited to also include automatic opening and visualising of the wall details.

Listing C.7 shows how the wall detail file types are added to the restart file. They are only added when there are any to be written, so that the flag "writeWallDetailsVTK" is not unnecessarily added causing the existing selftests with restart files to fail. After adding the flag the number of options to be expected is added. Then for each option the enum value and the file type is written.

Finally, the file counter is written, which is the same for all options.

Reading the options from the restart file is shown in Listing C.8. First the number of options is read and then a loop reads and sets each enum value and file type. At last the file counter is read and set.

```
if (wallHandler.getWriteDetailsVTKAny())
{
    std::unordered_map<WallHandler::DetailsVTKOptions, FileType>
        writeWallDetailsVTK = wallHandler.getWriteWallDetailsVTKAll();
    os << " writeWallDetailsVTK " << writeWallDetailsVTK.size();
    for (const auto& p : writeWallDetailsVTK)
        os << " " << static_cast<int>(p.first) << " " << p.second;
    os << " " << wallDetailsVTKWriter_.getFileCounter();
}
```

Listing C.7: Adding wall details file types to the restart file.

```
if (!dummy.compare("writeWallDetailsVTK"))
{
    unsigned numberOfOptions, fileCounter, enumType;
    FileType fileType;
    line >> numberOfOptions;
    for (unsigned  i = 0; i < numberOfOptions; i++)
    {
        line >> enumType >> fileType;
        wallHandler.setWriteDetailsVTK(
            static_cast<WallHandler::DetailsVTKOptions>(enumType), fileType);
    }
    line >> fileCounter;
    wallDetailsVTKWriter_.setFileCounter(fileCounter);
    line.clear();
    line >> dummy;
}
```

Listing C.8: Reading wall details file types from the restart file.

C.5 NURBS control points wall details

The writeWallDetailsVTK method takes as argument a reference to a VTKData instance, so that multiple NurbsWall instances all write to the same file. The control points positions will be the points in the vtu file and will be visualised as spheres. The control points weights will be added as point data so that they can be used as radius of the spheres. The points will be connected with a rectangle like shape, i.e. a type of VTK_PIXEL = 8. Furthermore, an ID unique to each NurbsWall instance is added as point data, to easily distinguish between the control points of different instances, e.g. by colouring them by ID.

Listing C.9 shows the writeWallDetailsVTK method for the NurbsWall. For a bit of speed optimization we first reserve memory for the new points, point data and cells. Then we loop through the control points and rotate and translate each point to global frame. The point, weight and ID are all added to the VTKData instance. The connectivity is then added with four indices and a type 8. An example VTK image is shown in Figure C.2.

```
const std::vector<std::vector<Vec3D>>& controlPoints = nurbsSurface_.getControlPoints();
const std::vector<std::vector<double>>& weights = nurbsSurface_.getWeights();

// Reserve memory for number of points and cells about to be added
const unsigned int np = controlPoints.size() * controlPoints[0].size();
const unsigned int nc = (controlPoints.size() - 1) * (controlPoints[0].size() - 1);
data.reservePoints(np, { "Weight", "ID" });
data.reserveCells(nc);

// Number of points in v-direction
size_t nv = controlPoints[0].size();
// Point index offset, the point indices have to be offset by the number of points
// already present at the start (from previously added data)
size_t pio = data.getPoints().size();
// Get last id only when possible and increment it, otherwise start with 0
double id = data.getPointData()["ID"].empty() ? 0 : data.getPointData()["ID"].back() + 1;

for (int i = 0; i < controlPoints.size(); i++)
{
    for (int j = 0; j < controlPoints[i].size(); j++)
    {
        Vec3D p = controlPoints[i][j];
        getOrientation().rotate(p);
        p += getPosition();
        data.addToPoints(p);
        data.addToPointData("Weight", weights[i][j]);
        data.addToPointData("ID", id);

        if (i > 0 && j > 0)
        {
            // Basic 2D/1D mapping for indexing: nv * i + j (+ point index offset)
            // 4 points to form a rectangle: point, point to the left,
            // point down, point to the left and down
            data.addToConnectivity({ nv*i+j+pio, nv*(i-1)+j+pio,
                nv*i+j-1+pio, nv*(i-1)+j-1+pio });
            data.addToTypes(8);
        }
    }
}
```

Listing C.9: The writeWallDetailsVTK method of the NurbsWall class.

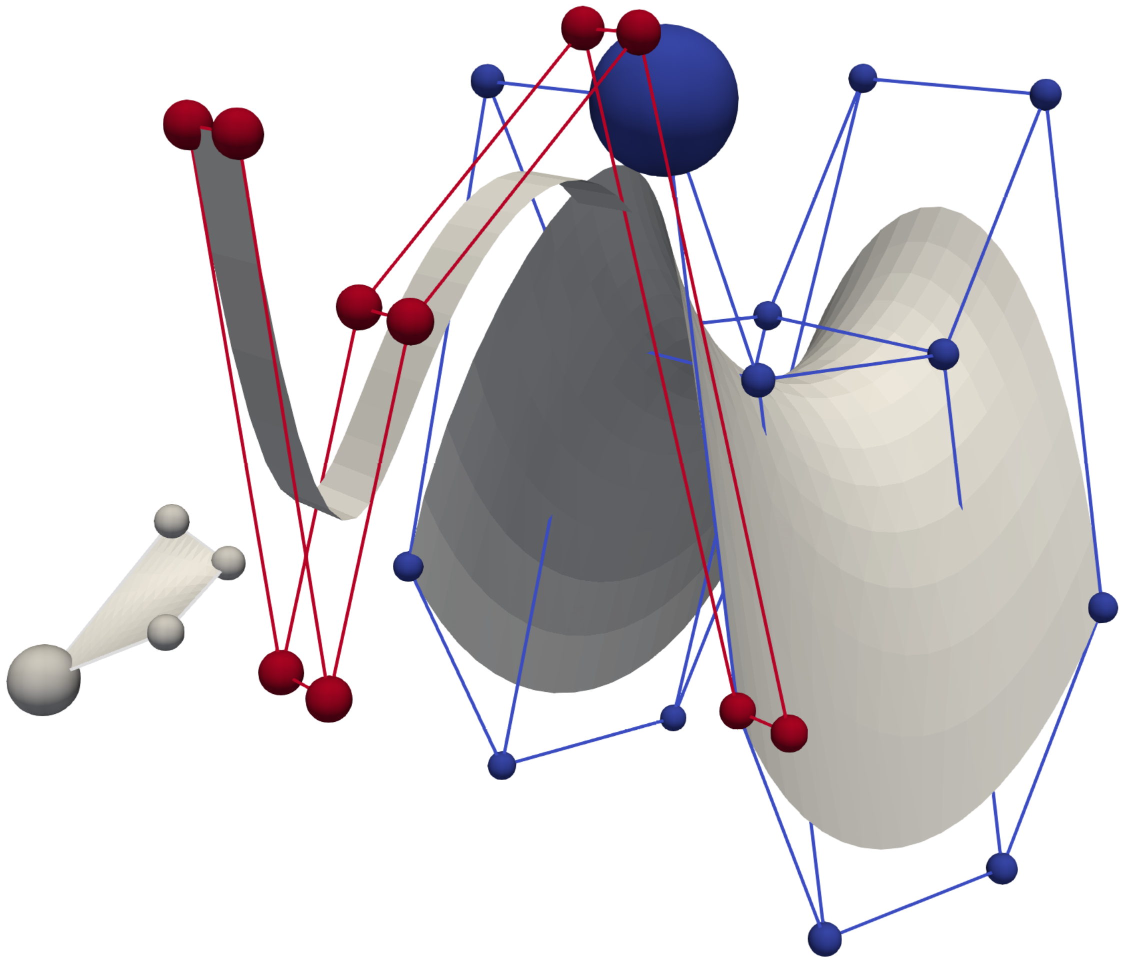


Figure C.2: Three different NURBS shown with their control points, coloured by ID. The radius of the control points are proportional to their weight.

Appendix D

TriangleMeshWall

D.1 Writing to VTK

The writeVTK function is overridden and takes in a VTKContainer as argument, which stores points and triangle strips, i.e. cells. Each triangle already stores the vertex indices of the vertices of which it is made, which is exactly what a cell is. It is therefore quite straightforward to add the points and cells to the VTKContainer, as seen in Listing D.1. The only details are that the vertices are rotated and translated from local to global frame and that the triangle cells have the vertex indices offset by the number of points originally in the VTKContainer.

```
const unsigned long s = vtk.points.size();

for (auto& v : vertices_)
{
    vtk.points.push_back(getOrientation().rotate(v.position) + getPosition());
}

for (auto& t : triangles_)
{
    vtk.triangleStrips.push_back({ static_cast<double>(s + t.vertexIndices[0]),
                                   static_cast<double>(s + t.vertexIndices[1]),
                                   static_cast<double>(s + t.vertexIndices[2]) });
}
```

Listing D.1: The writeVTK function.

D.2 Moving vertices to match a change in volume

The goal of this function is to move all vertices in such a way that the volume created between the new and original mesh, i.e. the volume change, matches a certain target volume. The displacements are given, of which the direction remains unchanged, but the magnitude is updated to get to the correct volume change. This is done recursively, as it is not possible to calculate analytically. The ratio between the target volume and the volume change will get closer and closer to 1, until it is within a certain tolerance and the recursion will end. When the conversion to the target volume is too slow, a safety limit for the number of recursive calls will stop the recursion.

Moving the vertices one by one creates three tetrahedrons for each triangle, see Figure 5.1. For each vertex we calculate the volume of the tetrahedrons of each triangle connected to it. For each of the triangles that is the tetrahedron formed by its vertices and the current vertex after updating it. After the tetrahedral volumes have been calculated the vertex is updated, so that when it is needed by other vertices the updated position is used, ensuring the proper tetrahedrons are formed. The implementation of this can be seen in Listing D.2, where getVolumeTetrahedron returns the volume of the tetrahedron formed by the given four points. It must be noted that the

vertices vector here is a copy of the real vertices, so that the original vertices remain unchanged. Only when the recursion has finished are the real vertices updated.

```
Mdouble totalVolume = 0.0;
for (int i = 0; i < vertices.size(); i++)
{
    for (auto& p : vertices[i].triangleIndices)
    {
        Triangle& t = triangles_[p.first];
        totalVolume += getVolumeTetrahedron(vertices[t.vertexIndices[0]].position,
                                            vertices[t.vertexIndices[1]].position,
                                            vertices[t.vertexIndices[2]].position,
                                            vertices[i].position + displacements[i]);
    }
    vertices[i].position += displacements[i];
}
```

Listing D.2: Calculating the tetrahedral volumes.

The ratio between the target volume and the volume change is then calculated and compared to 1, with a tolerance of $1 \cdot 10^{-6}$. The value of this tolerance and the maximum number of recursive calls, defaulted to 15, are chosen to give results in a reasonable time, but have not been studied to find out their exact influence. Once the recursion should stop, the real vertices are updated, as well as the real triangle walls, as shown in Listing D.3. When the recursion should continue, the displacements are multiplied by the volume ratio, the maximum number of recursive calls is decremented and the recursive call is made.

```
int num = vertices_.size();
for (int i = 0; i < num; i++)
    vertices_[i].position += displacements[i];

for (Triangle& t : triangles_)
    t.wall.moveVertices({ displacements[t.vertexIndices[0]],
        displacements[t.vertexIndices[1]],
        displacements[t.vertexIndices[2]] });

updateBoundingBox();
```

Listing D.3: Updating the real vertices and triangles.

In case of periodic boundaries, vertices which are periodic companions should move in the same way. Therefore their displacements are added to each other, as shown in Listing D.4. This should, however, only be done for the first call to the function, not for any recursive calls after that. To that end, a static boolean indicating it is the first call is initially set to true, on the first call set to false and when the recursion finishes set back to true.

D.3 Creating a parallelogram mesh

The createParallelogramMesh function does exactly what its name implies. The parallelogram is defined by three points, namely by the bottom left (origin), top left (v-direction) and bottom right (u-direction) corners. The three points are taken as arguments in clockwise order as $\boldsymbol{p}_0$, $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$, together with resolutions du and dv. The two direction vectors from $\boldsymbol{p}_0$ to $\boldsymbol{p}_1$ and $\boldsymbol{p}_2$ are unit vectors, calculated as $\boldsymbol{q}_u = (\boldsymbol{p}_2 - \boldsymbol{p}_0)/(|\boldsymbol{p}_2 - \boldsymbol{p}_0|)$ and $\boldsymbol{q}_v = (\boldsymbol{p}_1 - \boldsymbol{p}_0)/(|\boldsymbol{p}_1 - \boldsymbol{p}_0|)$. With two nested loops looping over the number of points needed in u- and v-direction, with iterators i and j respectively, each new point $\boldsymbol{p}$ is calculated as

$$\boldsymbol{p} = \boldsymbol{p}_0 + \boldsymbol{q}_u \cdot i \cdot du + \boldsymbol{q}_v \cdot j \cdot dv. \tag{D.1}$$

The number of points needed in each direction is one more than the number of segments, which is obtained by dividing the side length by the resolution and rounding this value up to get a whole

```
static bool firstRecursiveCall = true;
if (firstRecursiveCall)
{
    firstRecursiveCall = false;

    if (!periodicCompanions_.empty())
    {
        std::vector<Vec3D> disps = displacements;
        for (auto& p : periodicCompanions_)
        {
            displacements[p.first] += disps[p.second];
            displacements[p.second] += disps[p.first];
        }
    }
}
```

Listing D.4: Handling the periodic boundaries on the first function call.

number. The resolution is then updated for this number of segments, meaning the final resolution is smaller or equal to the resolution specified.

The set function is used to create the actual mesh and takes as arguments a vector of points and a vector of cells, where each point is a Vec3D and each cell is an array with three indices to the points vector. Within the loop two cells are created between the current point, the point above, the point to the right and the point diagonally to the right and above. The diagonal is chosen in such a way that a zigzag pattern forms in the mesh. This is achieved by picking one diagonal when both iterators are odd or even and picking the other diagonal otherwise.

Finally, two more arguments indicate if the mesh is periodic in u and/or v. For the u-direction this means the points in the first and last column will be periodic companions and for the v-direction the same is true for the first and last row. When both periodic in u and v also the opposite corner points of the mesh are also periodic companions.

D.4 Creating a four point mesh

The createFourPointMesh function creates a mesh that is defined by four corner points ($\boldsymbol{p}_{0-3}$ in clockwise order) and a number of segments in u- and v-direction (n_u and n_v). It is similar to the createParallelogramMesh function as described in Appendix D.4, however the resulting mesh is not necessarily flat nor does it have fixed resolutions, i.e. the triangles can be different in size. The direction vectors in this context are not normalised to be unit vectors and are

$$\begin{aligned} \boldsymbol{q}_u &= \boldsymbol{p}_3 - \boldsymbol{p}_0, \\ \boldsymbol{q}_{v0} &= \boldsymbol{p}_1 - \boldsymbol{p}_0, \\ \boldsymbol{q}_{v1} &= \boldsymbol{p}_2 - \boldsymbol{p}_3, \\ \mathrm{d}\boldsymbol{q}_v &= (\boldsymbol{q}_{v1} - \boldsymbol{q}_{v0})/n_u, \end{aligned} \tag{D.2}$$

where $\mathrm{d}\boldsymbol{q}_v$ is the three dimensional step size for linear interpolation between the left and right direction vectors in v. With two nested loops each new point is calculated as

$$\begin{aligned} \boldsymbol{q}_v &= \boldsymbol{q}_{v0} + i \cdot \mathrm{d}\boldsymbol{q}_v, \\ \boldsymbol{p} &= \boldsymbol{p}_0 + \boldsymbol{q}_u \cdot i/n_u + \boldsymbol{q}_v \cdot j/n_v, \end{aligned} \tag{D.3}$$

where i and j are the iterators for the loops in u and v, respectively. So each point is found by taking i steps along the bottom direction vector in u and then taking j steps along the corresponding direction vector in v. The cells are, again, added in a zigzag pattern.

D.5 Adding a mesh to a mesh

The addToMesh function takes another TriangleMeshWall and adds it to itself to form a single mesh. It is very useful for creating more complex geometries, especially when they consist of

multiple simple flat meshes, for example created by the createParallelogramMesh or createFourPointMesh functions.

Vertices that already exists in the mesh are not added again, but any references made are updated to work with the existing vertex. For triangles this check is not done, since it is not a situation that should ever occur when using this function properly.

New vertices and triangles are simply appended to the end of the existing vertices and triangles vectors. Each vertex stores the triangle indices of the triangles it is attached to, therefore for all new vertices these triangle indices have to be updated by adding the number of triangles in the current mesh to it. For the new triangles the same is true for their stored vertex indices, however since the exact number of new vertices added can be lower than the original size, they cannot be updated as easy. Instead, each new vertex has its updated index stored in a separate vector. Their original index can then be used to access their updated index.

Listing D.5 shows the function in its completeness. First the new vertices are added, by looping over them and for each vertex updating their triangle indices. To check whether or not the vertex already exists in the mesh, a loop over the existing vertices checks if their positions match and if so, the existing vertex index is stored. The index was initialized as -1, which is used to check if there was a match. If this is not the case, the index is overwritten by the index of the now newly added vertex. In case there was a match, the existing vertex gets the triangle vertices of the new vertex added to it, since these triangles are now attached to it. The now updated index of this vertex is added to the updated indices vector.

The new triangles are added by looping over them and appending them to the triangles vector. For each triangle its vertices indices are updated from the updated indices vector. Finally, the bounding box is updated, since the mesh has changed.

```
// Remember the number of vertices and triangles originally in this mesh.
const int numVertices = vertices_.size();
const int numTriangles = triangles_.size();
// For each of the added vertices, their updated vertex is temporarily stored in a vector.
// This is the cleanest/easiest way to later update the vertex indices of each added
// triangle.
std::vector<unsigned> updatedVertexIndices;
updatedVertexIndices.reserve(mesh.vertices_.size());

// Loop through the vertices to be added.
for (Vertex& v : mesh.vertices_)
{
    // The triangle indices can simply be updated.
    for (auto& p : v.triangleIndices)
        p.first += numTriangles;

    // Initialize updated vertex index as -1, to later check if it has been set or not.
    int index = -1;
    // Loop through original vertices and compare vertex positions.
    for (int i = 0; i < numVertices; i++)
    {
        if (v.position.isEqualTo(vertices_[i].position,
                std::numeric_limits<Mdouble>::epsilon()))
        {
            // When vertices share position, update vertex index.
            index = i;
            break;
        }
    }

    // When the updated index is not set, the vertex is not in the mesh yet
    // and should be added.
    if (index == -1)
    {
        index = vertices_.size();
        vertices_.push_back(v);
    }
    else
    {
        // The vertex is already in the mesh, add the triangle indices
        // of the added mesh to it.
        for (auto& p : v.triangleIndices)
            vertices_[index].triangleIndices.push_back(p);
    }

    // Store the updated index.
    updatedVertexIndices.push_back(index);
}

// Loop through the triangles to be added.
for (Triangle& t : mesh.triangles_)
{
    // Update the vertex indices.
    for (unsigned& idx : t.vertexIndices)
        idx = updatedVertexIndices[idx];
    triangles_.push_back(t);
}

updateBoundingBox();
```

Listing D.5: The addToMesh function.