

## MASTER

### GPGPU interpolation of volumetric data for optimal motion planning

van de Schoot, Bas A.C.

*Award date:*  
2018

[Link to publication](#)

#### **Disclaimer**

This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

#### **General rights**

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

# GPGPU interpolation of volumetric data for optimal motion planning

B.A.C. van de Schoot

D&C 2018.016

Master's Thesis: Preperation Phase

Coach(es):      dr. A Saccon,  
                     dr. G. Dubbelman

Supervisor:     prof. dr. H. Nijmeijer

Eindhoven University of Technology  
Department of Mechanical Engineering  
Dynamics & Control

Eindhoven, January, 2018



---

## Abstract

There are multiple different methods of motion planning. Two common methods are sampling-based motion planning and optimal motion planning. This thesis is solely concerned with the optimal motion planning and not with the sampling-based motion planning. These optimal motion planning algorithms solve an optimal control problem in order to calculate the optimal trajectory. These optimal control problems can be solved using different algorithms. This thesis wants to use a method which minimize the optimal control problem using Newton-like minimization methods. Therefore, the optimal control problem has to be a twice continuously differentiable function. This thesis researches a fast interpolation method which results in a twice continuously differentiable description of the environment from a 3D scalar space.

This research focuses on a method of describing the environment by interpolating a discrete implicit representation. Therefore, this research investigates fast interpolation methods which can interpolate a 3D scalar space. The literature proposes to use a digital filter method as the interpolation method for optimal motion planning. However, no mathematical proof was given in the literature. Furthermore, different interpolation methods and implementations are compared based on their calculation time. The comparison has shown that a GPGPU (General-Purpose computing on Graphics Processing Units) implementation of the digital filter method is the fastest interpolation for optimal motion planning.

The proposed GPGPU interpolation method is implemented into a discrete PProject Operator-based Newton method for Trajectory Optimization (dPRONTO) algorithm. The interpolation method is used to interpolate synthetic and experimental data which describe the environment. The experimental descriptions are obtained using the KinectFusion framework with a Kinect camera v1. Unfortunately, the default open-source implementation of KinectFusion (KinFu) does not result in a discrete implicit representation of the environment. Therefore, some parameters of the KinFu algorithm are tuned to provide a proper discrete implicit environment description. The interpolation method is able to create a twice continuously differentiable function from these discrete implicit representations. Moreover, the dPRONTO algorithm is able to calculate a locally optimal trajectory from the interpolated synthetic and experimental environment descriptions.

The proposed interpolation method has proven to be faster than other methods. The proposed interpolation method can be used in an optimal motion planning algorithm. However, the proposed interpolation is useless when the interpolated description of the environment does not accurately approximate the exact environment. Especially, the surfaces of the objects has to be approximated accurately. The optimal motion planning algorithm calculates the trajectory based on the interpolated description of the environment. Hence, a collision between the real environment and the robot is plausible when the interpolated surface is not the same as the exact surface. Therefore, the interpolation method is validated based on its accuracy of approximating the surfaces of the obstacles. The accuracy of the approximation of the surface is validated as a function of the obstacle radius to grid size ratio. Therefore, the validation gives the grid size for scanning a certain environment with a desired accuracy. Moreover, the error of the surface approximation can be compensated in the optimal control problem.



# Contents

1	Introduction	1
1.1	Robot motion planning . . . . .	1
1.2	Environment descriptions . . . . .	1
1.3	Research challenges . . . . .	2
1.4	Research objective and methodology . . . . .	3
1.5	Contribution . . . . .	4
1.6	Report outline . . . . .	4
2	Mathematical preliminaries and background material	6
2.1	Implicit representation . . . . .	6
2.2	KinectFusion . . . . .	7
2.3	Background of interpolation methods . . . . .	9
2.3.1	One Dimensional interpolation . . . . .	9
2.3.2	Comparing basic multivariate interpolations . . . . .	12
2.3.3	Multivariate cubic b-spline . . . . .	13
2.3.4	Fast multivariate interpolation . . . . .	14
2.3.5	Summary . . . . .	15
2.4	Newton's method . . . . .	16
2.4.1	Convergence rate . . . . .	16
2.4.2	Convergence rate of an approximated function . . . . .	17
2.4.3	Terminal condition . . . . .	17
2.4.4	Backtracking line search . . . . .	18
2.5	Summary . . . . .	18
3	Proposing a fast interpolation method for optimal motion planning	20
3.1	Digital filter vs tridiagonal interpolation . . . . .	20
3.2	The implementation of the proposed interpolation method . . . . .	21
3.3	Verification of the proposed interpolation method . . . . .	26
3.3.1	One dimensional CUDA interpolation . . . . .	26
3.3.2	Multivariate CUDA interpolation . . . . .	29
3.4	Zero level set interpolation error as function of the radius to grid size ratio . . . . .	35
3.5	Summary . . . . .	41
4	Numerical experiments	42
4.1	Calculation time of the different interpolation methods . . . . .	42
4.2	Optimal motion planning with collision constraints from point cloud data . . . . .	45
4.2.1	OMP based on interpolated TSDF . . . . .	45
4.2.2	OMP based on KinectFusion data . . . . .	51
4.3	Summary . . . . .	55
5	Conclusions and recommendations	57
5.1	Conclusions . . . . .	57
5.2	Recommendations . . . . .	58
A	B-splines is a special case of piecewise polynomial	64
B	Convergence rate Newton's Method	65

C	CUDA source code	66
C.1	1D interpolation . . . . .	66
C.2	Interpolate 3D grid in x-direction . . . . .	67
C.3	Interpolate 3D grid in y-direction . . . . .	67
C.4	Interpolate 3D grid in z-direction . . . . .	68
C.5	Function which calculates the number of threads and blocks . . . . .	69
D	The addition images of the multivariate interpolation validation	71
E	Signed Distance Field of an edge	76
F	Parameters of dPRONTO algorithm	78

# 1 Introduction

## 1.1 Robot motion planning

A motion planning algorithm calculates a feasible path from a starting point to a desired endpoint. Where a feasible path is defined as a path that does not collide with obstacles and can be accomplished by the robot in terms of hardware restrictions. There are different methodologies for motion planning.

A common method of motion planning is sampling-based motion planning. sampling-based motion planning chooses random positions to check whether these are free. All the free points are combined, resulting in a free space. Next, the sampling-based motion planning algorithm searches for a feasible path. This feasible path is found when the discrete positions of the path and connecting paths are entirely in the free space. The main disadvantage of sampling-based motion planning is that the final trajectories often contain unnecessary and jerky motions [1].

Another common method for motion planning is solving an optimal control problem. The algorithms which solve motion planning using an optimal control problem are referred to as optimal motion planning algorithms. These methods describe the robot and the surrounding in a constraint optimal control problem. These optimal control problems have an objective functional which should be minimized while satisfying certain dynamics and constraints. Due to the fact that the dynamics are part of the optimal control problem, these optimal motion planning algorithms result in much less jerky trajectories. The optimal motion planning algorithms are able to avoid obstacles when the obstacles are defined via an inequality constraint. The inequality constraint that describes the environment will be referred to as the obstacle avoidance constraint.

Ideally, the optimal motion planning algorithm is able to plan the optimal trajectory while it is exploring and scanning the environment. Therefore, the obstacle avoidance constraint should be calculated as fast as possible.

## 1.2 Environment descriptions

There are multiple different optimal motion planning algorithm available (CHOMP [2], STOMP [3], TrajOpt [4]). These optimal motion planning algorithms are already able to calculate a desired path while describing a static environment. These methods use different solutions for describing the environment. The simplest solution for describing the environment is to create an occupancy map. However, the optimal motion planning algorithm requires an inequality constraint to describe the environment. Therefore, a common method is to use a Signed Distance Function (SDF) to describe the environment. An SDF at point  $x$  is defined as the distance between point  $x$  with respect to the surface of the closest obstacle. Moreover, the SDF of point  $x$  is negative when the point is inside of an obstacle. When point  $x$  is on the outside of the obstacle it has a positive SDF. When the point is on the surface an obstacle it has a value of zero in the SDF. All the points with value zero are referred to as the zero level set of the SDF. The structure of the SDF results in an implicit representation of the environment. Concluding, an SDF can be used as an inequality constraint which describes the environment.

Mainprice et al. [5] propose to use a different method to describe the environment, namely to use an Electric Potential Field instead of an SDF. The computation of the electric potential is an expensive calculation. Consequently, [5] does not use the exact function to describe the environment but an interpolated version of a discrete Electric Potential Field (EPF), which seems to be a proper candidate to describe the environment.



### 1.3 Research challenges

The static environment should be implemented using inequality constraints in optimal motion planning. Therefore, the environment has to be represented with a description which has a clear difference between the inside and outside of an obstacle. First, the desired environment description is discussed. Second, the two different surrounding descriptions which have already been discussed are discussed with respect to the desired description.

Mainprice et al. [5] compare three different optimal motion planning algorithms: STOMP, CHOMP and Gauss-Newton. All three optimal motion planning algorithms have a different method of minimizing the objective function. The STOMP algorithm uses solely the objective function to calculate the local derivative. The CHOMP algorithm uses a first derivative of the objective function. The Gauss-Newton uses the first and second derivative of the objective function. The results of the comparison of the three optimizers show that using a first and second derivative of the objective function results in a faster minimization over time. Therefore, the optimal motion planning should use a Newton-like minimization method. The downside of using a Newton-like minimization method is that continuous first and second derivatives of the environment are required. Therefore, the obstacle avoidance constraint should satisfy this requirement.

As shown in case of the EPF, calculating the exact description can be expensive in terms of computation. Moreover, an environment commonly is obtained using scanning methods. Therefore, an environment commonly is represented discrete instead of a continuous. Therefore, an interpolation method is needed in order to use the environment description in optimal motion planning. The simplest solution of transforming the discrete representation into a continuous representation would be using a linear interpolation method. Moreover, the continuous first and second derivatives can be obtained using a central difference approximation with a linear interpolation. However, the book of Kelly [6] states that a central approximation in a Newton-like minimization might not converge. Therefore, an interpolation method is required which has a continuous and an exact first and second derivative. This requirement will be referred to as the smoothness requirement. A function which satisfies this requirement is commonly referred to as a twice continuously differentiable function or a  $C^2$  function.

Previously, it was stated that a common method of obtaining a description of an environment is using a sensor. This research does not focus on dynamic environments which change fast over time but focusses on static environments. However, most environments have not been scanned when a robot arrives for the first time. Moreover, an environment might have changed when a robot revisits an environment. Therefore, a robot should be able to scan and plan its environment simultaneously. As a result, the interpolation method should be able to get a fast  $C^2$  description of the environment.

In Section 1.2, several optimal motion planning algorithms have been mentioned like CHOMP and STOMP. Both methods use an SDF to describe the environment. However, these optimal motion planning algorithms do not require that the environment has to be described with a  $C^2$  function. Hence, these methods of describing the environment cannot be used in a Gauss-Newton method. Mainprice et al. [5] have investigated the possibility of using a Gauss-Newton method in optimal motion planning. The Gauss-Newton method should satisfy the smoothness requirement. However, Mainprice et al. use a default cubic interpolation method to interpolate a discrete EPF. This default cubic interpolation method will not be fast enough. Therefore, research is desired into a fast interpolation method which results in a  $C^2$  function.

## 1.4 Research objective and methodology

The main goal of this thesis is to implement and validate a fast interpolation method of 3D scalar fields for optimal motion planning. Therefore, this thesis will research the fastest interpolation method currently available in the literature. The fastest interpolation method will be implemented. The implemented interpolation method will be validated to ensure it is correct and it is faster than other interpolation methods. Moreover, the fast interpolation algorithm will be used to interpolate discrete Signed Distance Fields to get a  $C^2$  function which describes the environment. The interpolated Signed Distance Function will be used in an optimal motion planning algorithm. First, the discrete Signed Distance Fields will be created using synthetic data. Second, a 3D scanning technique will be used to create a sensor/experimental Signed Distance Field. Therefore, the interpolation method will be validated for optimal motion planning for synthetic and experimental data. However, interpolation of the discrete SDF (interpolated SDF) will never be exactly the same as the exact SDF. Consequently, the interpolated SDF might give a false representation of the obstacle in the optimal motion planning. This might result in a feasible trajectory based on the interpolated SDF which is not feasible for the exact SDF. Hence, the optimal motion planning algorithm finds a trajectory which results in a collision with the actual obstacle. In order to ensure no collisions due to the interpolation method, the zero level set of the interpolated SDF is compared with the exact SDF. The difference between the zero level sets will be evaluated for different obstacle radius to grid size ratios.

As discussed before, the ideal optimal motion planning is able to calculate the trajectory in real time. Hence, the interpolation method should be as fast as possible. This research investigates which interpolation method is the fastest. A possible solution to reduce the calculation time is using GPGPU (General-Purpose computing on Graphics Processing Units) calculation. The advantage of GPU (Graphics Processing Unit) calculations with respect to CPU (Central Processing Unit) calculations is the number of cores available in the system. A GPU normally has a couple of hundred cores available while a CPU commonly has 4 cores available. However, a GPU core is not as fast as a CPU core. Therefore, GPGPU calculation is solely beneficial when the interpolation method is able to run many calculations in parallel. GPGPU calculations have been made easily accessible due to the introduction of NVIDIA's CUDA (Compute Unified Device Architecture). CUDA enables a programmer to execute *c*-code on an NVIDIA GPU.

The interpolation method will be implemented in an optimal motion planning algorithm using the interpolated SDF as an environment inequality constraint. The optimal motion planning algorithm that is used in this research is a discrete Project Operator-based Newton method for Trajectory Optimization (dPRONTO). The dPRONTO algorithm is an in-house optimal motion planning method. The dPRONTO algorithm is able to calculate the optimal trajectory for a point mass. Moreover, it is assumed that the entire static environment is known before the optimization.

This thesis investigates the interpolation of sensor data for optimal motion planning. In this research, the experimental SDF is chosen to be obtained using a 3D scanning method. This research chooses to use a Kinect camera version 1 together with KinectFusion to get a discrete 3D representation of the environment. The Kinect camera version 1 is a depth camera of the game console x-box and KinectFusion is an algorithm which is able to combine the depth images into a 3D representation of the environment. This 3D scanning technique has been chosen because of its low cost and the technique is easily accessible. 3D scanning will result in a discrete representation of the environment. In other words, 3D scanning will not result in an obstacle avoidance constraint which satisfies the smoothness requirement. Therefore, the 3D scanning output has to be interpolated with an interpolation method which creates a  $C^2$  function.

Sensor data might contain noise resulting in an imperfect representation of the environment. The presence of noise can be reduced by using smoothing splines [7]. Schoenberg and Reisch are two examples of researchers which have proposed to use smoothing spline to approximate the gridded data instead of passing through the points exactly. The smoothing splines should result in a smoother function than exact interpolation methods. However, the smoothing splines do not pass through the grid points but approximate the gridded data. The KinectFusion algorithm combines multiple different depth images into a single global representation of the environment using volumetric integration. The volumetric integration [8] combines multiple depth images by weighting depth measurements for the same position from different images. Due to the weighting of different depth images, the volumetric integration might already filter most of the noise out. The output of the KinectFusion algorithm is assumed to be accurate and no smoothing splines will be applied. The requirement that the interpolation method should be exactly equal to the discrete representation at the grid points is referred to as the pass-through requirement.

## 1.5 Contribution

The contribution of this research to the literature is the following:

1. This research proposes a fast interpolation method for optimal motion planning. In section 2.3, a literature study provides a fast interpolation method. The fastest interpolation method of the literature study is implemented into an algorithm which can be used for optimal motion planning in Chapter 3. The proposed interpolation method is validated that it satisfies the smoothness and pass-through requirements. Moreover, the calculation time of the proposed interpolation method is compared with the calculation time of two different interpolation methods. The comparison of the calculation times can be found in section 4.1.
2. The accuracy of the proposed interpolation method is validated by investigating the zero level set error of the interpolated space with respect to the exact zero level set. Moreover, this validation investigates the accuracy of the zero level set approximation over different obstacle radius to grid size ratios. This validation can be found in section 3.4.
3. The proposed interpolation method is used to calculate an obstacle avoidance constraint for an OMP algorithm. The proposed interpolation method is used to interpolate simulated obstacles for an in-house optimal motion planning algorithm. Moreover, the proposed interpolation method is combined with KinectFusion. The interpolated KinectFusion data is used as an obstacle avoidance constraint in the same in-house optimal motion planning algorithm. The results of the implementation of the proposed interpolation method in OMP algorithm can be found in section 4.2.

## 1.6 Report outline

How obstacle surfaces can be described using implicit representation is explained in Chapter 2. Moreover, Chapter 2 discusses how KinectFusion framework can be used to get a discrete implicit of an obstacle. As discussed in this introduction, the discrete representation of an obstacle has to be interpolated fast. Chapter 2 concludes which interpolated method is the fastest method according to the literature which satisfies the requirements. Chapter 2 discusses exact Newton-like methods and Newton-like methods based on difference approximation derivatives.

Chapter 2 concludes which interpolation method is the fastest, according to the literature. However, the literature does not provide proof that the interpolation method is always faster. Therefore, Chapter 3 shows that the chosen method is faster than other interpolation methods. Moreover, Chapter 3 validates that the interpolation method is accurate and satisfies the pass-through and smoothness requirement. Chapter 3 discusses the implementation of the fast interpolation method. Chapter 3 ends

with the validation of the zero level set approximation as a function of the obstacle radius to grid size ratio.

Although Chapter 2 shows that the chosen method is faster than other interpolation methods, Chapter 4 shows how much the proposed interpolation method faster is than other interpolation methods. The main goal of this thesis is to propose a fast interpolation method for optimal motion planning. Therefore, Chapter 4 shows results of an optimal motion planning algorithm based interpolated synthetic data. Moreover, the KinectFusion framework is used as input for the obstacle avoidance constraint in Chapter 4.

Chapter 5 summarises the conclusions of this research. Moreover, Chapter 5 discuss some challenges which still need to be tackled in order to use this method in real-time optimal motion planning. Moreover, some recommendations for future research are made.



## 2 Mathematical preliminaries and background material

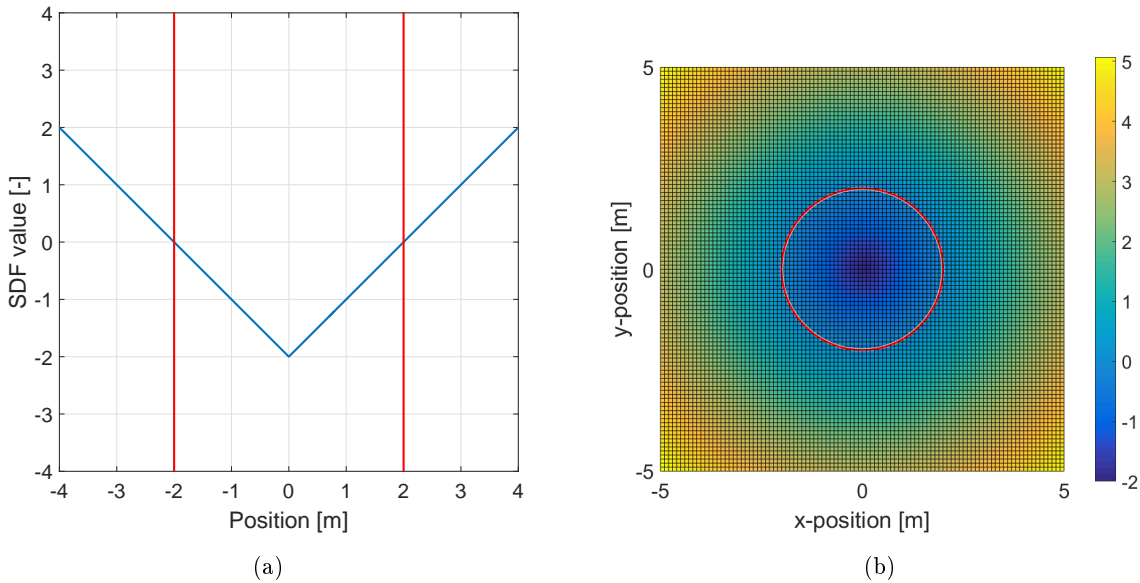
This chapter will discuss the background of several different topics which are used in this research. First, this section explains how an implicit representation can be used to describe the surface of an obstacle. Second, the KinectFusion framework is explained and how it can result in a discrete implicit representation of an environment. Third, a literature study on interpolation methods discusses the fastest interpolation method for optimal motion planning. Finally, this section discusses exact and inexact Newton's method and the convergence rates of both methods.

### 2.1 Implicit representation

Chapter 1 states that the obstacle avoidance constraint is chosen to be an implicit representation of the environment. The implicit representation is defined using three different subspaces. The first subspace is the set of points which define the obstacle surface. This subspace is represented by a zero in the implicit representation. This set of points is referred to as the zero-level set. The second subspace is the set of points which are inside the obstacle. The implicit representation represents these set points with a positive value. The third subspace defines the set of points outside the obstacles. These points are shown with a negative value in the implicit representation. This implicit representation of the obstacle results in the following equation

$$c(x) : \mathbb{R}^3 \rightarrow \mathbb{R} \begin{cases} = 0 & , \ x \text{ is on the surface of an object,} \\ < 0 & , \ x \text{ is inside an object,} \\ > 0 & , \ x \text{ is outside an object,} \end{cases} \quad (2.1)$$

where  $c$  is the obstacle avoidance constraint and  $x$  is a position. When the output of the obstacle avoidance constraint represents the distance from the given position to the closest surface than the obstacle avoidance constraint has a physical meaning. A Signed Distance Function (SDF) describes the distance between position  $x$  and the closest surface to this position. Figures 2.1 shows a 1D and 2D example of an SDF.

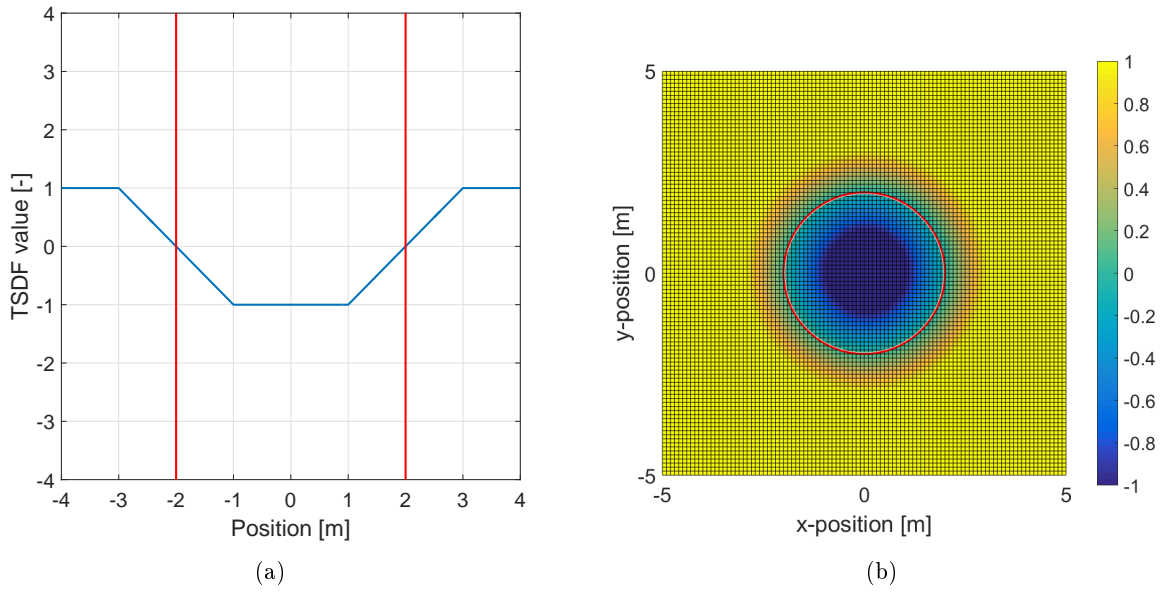


**Figure 2.1:** Two examples of an SDF where the surface of the obstacles are shown in red. (a) shows a 1D obstacle of a width of 4 [m]. (b) shows a 2D obstacle with a radius of 2[m].

The distance to an obstacle is not interesting when all the obstacles are not close to the point. When this is the case, the SDF can be truncated using a truncation threshold resulting in a Truncated Signed Distance Function (TSDF). The TSDF is a truncated and normalized version of the SDF. The truncation thresholds transform the SDF to a TSDF using the following equation

$$TSDF(x) = \begin{cases} 1 & , SDF(x) > \bar{D} \\ SDF(x)\bar{D}^{-1} & , |SDF(x)| < \bar{D} \\ -1 & , SDF(x) < -\bar{D} \end{cases} \quad (2.2)$$

here,  $\bar{D}$  is the truncation threshold. The TSDFs of the SDFs of Figure 2.1 are shown in Figure 2.2. In Figure 2.2, the truncation threshold is set to 1[m].

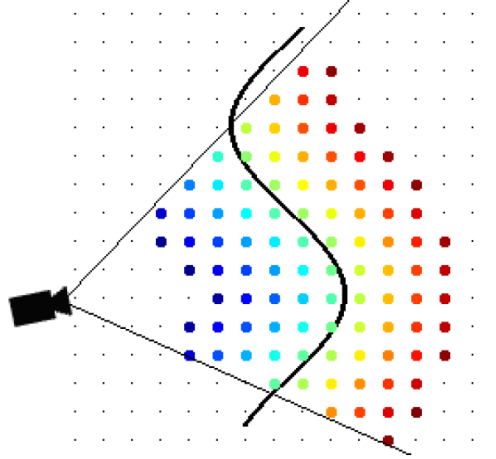


**Figure 2.2:** The TSDF of the two examples of Figure 2.1. Also here are the surface of the obstacles shown in red. (a) shows a 1D obstacle of a width of 4 [m]. (b) shows a 2D obstacle with a radius of 2[m]. The truncation threshold is 1[m] in both examples.

The TSDF of the examples used in Figure 2.2 are relatively simple. However, an autonomous robot should also be able to cope with complex environments and should represent the environment in 3D. A common method of obtaining the surrounding is scanning.

## 2.2 KinectFusion

A well-known 3D scanning method is KinectFusion with a moving depth camera [9]. This section explains how KinectFusion determines the TSDF of an environment based on depth images. A depth camera measures the distance between a pixel of an image and the closest surface with respect to that pixel. A single image of the depth camera results in a Signed Distance from one point of view. Figure 2.3 shows a side view of a single depth image.



**Figure 2.3:** A two-dimensional example of a Signed Distance (SD), where positive distances ( $d$ ) are shown by blue to green ( $d \in [0 \infty)$ ), negative distances are shown by green to red ( $d \in (-\infty 0]$ ) and the black line represent the obstacle surface( $d = 0$ ). [10]

As can be seen in Figure 2.3, a pixel in front of the closest surface has a positive value and a pixel behind the closest surface has a negative value. The closest surface of the obstacle to the camera is represented by the value zero in the SDF. However, a single image does not give an accurate representation of the obstacle. Volumetric integration can merge multiple depth images into a single 3D grid [8]. The images are merged by using the pose and position of the camera to translate the relative positions of a depth image to a global position within a predefined 3D grid. These global positions are used to combine multiple depth images from different points of view. The value of a global point is calculated by weighting the measurements of different depth images. The global Signed Distance is updated using the following equations

$$D(x) = \frac{\sum w_i(x)d_i(x)}{\sum w_i(x)}, \quad (2.3)$$

$$W(x) = \sum w_i(x). \quad (2.4)$$

However, volumetric integration solely works when the global position of the camera is known. In other words, the global 3D space is relative to the starting pose of the camera. Moreover, the camera movement is tracked in KinectFusion using the Iterative Closest Point algorithm (ICP) of Besl and McKay [11]. A global SD can be computed based on the camera tracking of the ICP and the volumetric integration. A grid point in the global 3D space is called as a voxel as a 2D image consists out of multiple pixels.

A single depth image stores all the distances between a pixel and the closest obstacle. All these distances have to be converted into the 3D grid. This operation will take a long time and the most important information is which pixels are close to an obstacle. Therefore, the SDF is Truncated (TSDF) in order to reduce the information. KinectFusion calculates the discrete TSDF using the following equation

$$TSDF(x) = \begin{cases} 1 & , D(x) > \bar{D} \\ D(x)\bar{D}^{-1} & , |D(x)| < \bar{D} \\ -1 & , D(x) < -\bar{D} \end{cases}, \quad (2.5)$$

where  $\bar{D}$  is the truncation threshold in  $[m]$ . The benefit of the truncation is that a voxel far from an obstacle will never be updated. This TSDF will result in a 3D space where voxels have a value

- 1 when the voxel is farther from a surface than the truncation threshold and the voxel is outside an obstacle.



- between  $(0 \dots 1)$  when the voxel is less than the TSDF value times the truncation distance away from a surface and the voxel is outside an obstacle.
- 0 when the voxel is on a surface of an obstacle.
- between  $(-1 \dots 0)$  when the voxel is less than the TSDF value times the truncation distance away from a surface and the voxel is inside an obstacle.
- $-1$  when the voxel is farther from a surface than the truncation threshold and the voxel is inside an obstacle.

Newton-Like minimization requires a  $C^2$  function, as discussed in Chapter 1. However, the output of the KinectFusion will be a discrete TSDF. Therefore, the discrete TSDF of KinectFusion has to be interpolated in order to use the measurement in optimal motion planning.

## 2.3 Background of interpolation methods

As discussed in Chapter 1, the interpolation method has to satisfy two requirements. The interpolation method has to result in a  $C^2$  function and has to pass through the grid points. Moreover, the interpolation method should be as fast as possible. The literature study on interpolation methods starts with discussing the basics of piecewise polynomial interpolation and the B-spline interpolation in 1D. Thereafter, this section discusses the multidimensional interpolation of these two interpolation methods. A multidimensional interpolation might be referred to as multivariate interpolation. Furthermore, a fast multivariate interpolation method is discussed. Finally, a brief summary will discuss the best interpolation method based on the literature.

### 2.3.1 One Dimensional interpolation

The basics of piecewise polynomial interpolation and B-spline interpolation are first explained 1D. Starting with piecewise polynomial interpolation. Interpolating a discrete signal with a single polynomial will result in a complex function in order to satisfy the pass-through requirement. In order to reduce the complexity, a discrete signal will be interpolated using multiple polynomials which each describe the interpolation between two grid points. Each polynomial has to be of order three to satisfy the smoothness requirement. Therefore, a cubic piecewise polynomial will ensure that the interpolation output is a  $C^2$  function. Each cubic piecewise polynomial can be represented by the following equation

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3, \quad (2.6)$$

where  $a_i$ ,  $b_i$ ,  $c_i$  and  $d_i$  are the coefficients of a polynomial.  $S_i(x)$  is the cubic spline which interpolates the function between grid points  $x_i$  and  $x_{i+1}$ . The pass-through and smoothness requirements result in the following requirements for the piecewise polynomial interpolation

$$S_i(x_i) = f(x_i) = f_i, \quad (2.7)$$

$$S_i(x_{i+1}) = S_{i+1}(x_{i+1}), \quad (2.8)$$

$$S'_i(x_{i+1}) = S'_{i+1}(x_{i+1}), \quad (2.9)$$

$$S''_i(x_{i+1}) = S''_{i+1}(x_{i+1}), \quad (2.10)$$

where  $f_i$  is the value at the grid point at position  $x_i$ . A piecewise cubic spline interpolation method is proposed by Mathews [12]. Mathews uses these requirements to calculate the coefficients of (2.6). Mathews combines (2.6), (2.7), (2.8), (2.9) and (2.10) which is a problem with four unknowns per polynomial. The combination of the four equations result in a problem with solely one unknown per polynomial.

The resulting problem of combining (2.6), (2.7), (2.8), (2.9) and (2.10) is represented with the following equation with the new unknowns  $D_i$

$$\begin{bmatrix} h_1 & 2(h_1 - h_2) & h_2 & 0 & \dots & \dots & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & \dots & 0 & h_{n-1} & 2(h_{n-1} - h_n) & h_n \end{bmatrix} \begin{bmatrix} D_2 \\ \vdots \\ \vdots \\ D_{n-1} \end{bmatrix} = \begin{bmatrix} u_1 \\ \vdots \\ \vdots \\ u_{n-1} \end{bmatrix}, \quad (2.11)$$

where

$$u_i = 6 \left( \frac{f_{i+1} - f_i}{h_i} - \frac{f_i - f_{i-1}}{h_{i-1}} \right), \quad i \in [1 \ n] \quad (2.12)$$

where  $h_i$  is the grid size. These  $D_i$ 's can be used to calculate the unknown of (2.6) by

$$\begin{aligned} a_i &= f_i, \quad i \in [1n] \\ b_i &= \frac{f_{i+1} - f_i}{h_i} - \frac{h_i(2D_i + D_{i+1})}{6}, \quad i \in [1 \ n] \\ c_i &= \frac{D_i}{2}, \quad i \in [1 \ n] \\ d_i &= \frac{D_{i+1} - D_i}{6h_i}, \quad i \in [1 \ n] \end{aligned}$$

These coefficients together with (2.6) can be used to calculate an interpolation function which satisfies all the requirements of the introduction. The tridiagonal linear system (2.11) is not able to calculate the first  $D_1$  and last  $D_n$  coefficients. These two last coefficients are calculated with endpoint constraints. There are multiple different boundary conditions which can be used to add the two needed equations. De Boor [7] gives a few examples of endpoints constraints. The first boundary condition is the complete interpolation, complete interpolation assumes that the first or second derivative at the endpoints is known. Another boundary condition is the free-end condition, free-end conditions assume that the second derivative at the endpoints is zero. The free-end conditions are also known as natural spline interpolation. Third boundary condition discussed by De Boor is the “not-a-knot” condition. The “not-a-knot” condition chooses to approximate the coefficients at the boundary using

$$c_{-1} = c_0,$$

$$c_{n-1} = c_{n-2},$$

here  $n$  is the number of coefficients. According to De Boor, the “not-a-knot” condition is the best choice when no information is available on the derivatives at the endpoints. All unknown  $D_i$ 's can be calculated when the boundary conditions are added to the tridiagonal linear system (2.11).

Thévenaz et al. [13] propose a different interpolation method. Thévenaz et al. call their interpolation method a generalized interpolation method. The difference between the traditional interpolation and the proposed generalized interpolation is the difference in coefficients. The formula of the generalized interpolation is

$$f(x) = \sum_i c_i \psi_i(x - x_i) \quad , \quad (2.13)$$

where  $x_i$  is a position on the grid,  $x$  can be any position in space,  $\psi_i$  is a chosen basis function around grid point  $x_i$  and  $c_i$  is the coefficients corresponding to the basis function at grid point  $x_i$ . The output of the interpolation method will be a  $C^2$  function when the basis functions are chosen to be second order

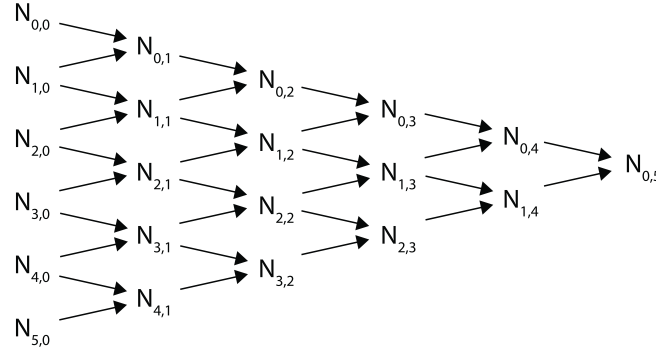
continuous. The paper [13] provides multiple basis functions, which satisfy the smoothness requirement. A commonly used basis function is the b-spline. A b-spline function can be calculated using the De Boor-Cox algorithm [7]. The De Boor-Cox algorithm uses a recursion to calculate a b-spline of a certain order. The De Boor-Cox algorithm starts with a non-continuous function

$$N_{i,0}(u) = \begin{cases} 1, & \text{if } u_i \leq u < u_{i+1}, \\ 0, & \text{if otherwise,} \end{cases} \quad (2.14)$$

where  $u_i$  are the grid positions. The De Boor-Cox algorithm uses the following equation to determine a b-spline of one order higher

$$N_{i,p}(u) = \frac{u - u_i}{u_{i+p} - u_i} N_{i,p-1}(u) + \frac{u_{i+p+1} - u}{u_{i+p+1} - u_{i+1}} N_{i+1,p-1}(u), \quad (2.15)$$

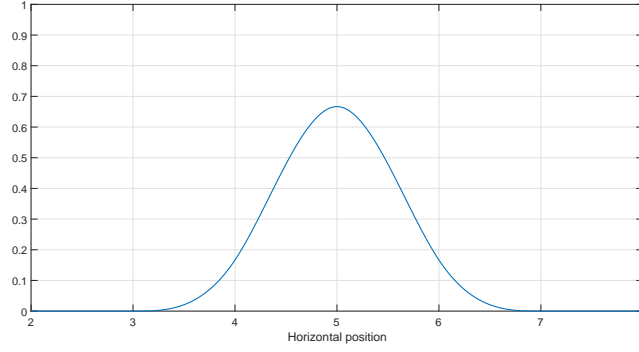
where  $N_{i,p}$  is the desired b-spline of order  $p$ . For example, to create a fifth order b-spline, one needs to have six input points, as shown in Figure 2.4.



**Figure 2.4:** An example of a recursion tree which is used by the De Boor-cox algorithm to calculate the shape of a certain b-spline. This example is the recursion tree of a fifth order b-spline.

To satisfy the smoothness requirement of Chapter 1, the interpolated function  $f(x)$  needs to be at least of an order three. According to De Boor [14], a combination of  $N_{i,p-1}$  and  $N_{i+1,p-1}$  produces a function, which, in general, has one more continuous derivative than either one of them.  $N_{i,0}$  is not a continuous function and  $N_{i,1}$  is a continuous function which has no continuous derivatives. Moreover,  $N_{i,2}$  is a continuous function and has one continuous derivative. In other words, when (2.15) is used, the function  $N_{i,p}$  has  $p - 1$  continuous derivatives. Therefore, a third order b-spline ( $p = 3$ ) is needed to interpolate the discrete grid in order to satisfy the smoothness requirement. To determine the third order b-spline, the De Boor-Cox algorithm needs four input points. It is assumed the number of b-splines is equal to the number of grid points. Moreover, the  $i$ -th b-spline is chosen to be symmetric on the  $i$ -th grid point. Figure 2.5 shows an example of the cubic b-spline around point  $i = 5$  with a grid size of one. Due to the symmetry of the b-spline, the equation of the  $i$ -th b-spline can be written as a function of the distance ( $|u_i|$ ) between the desired point  $x$  and the center of the b-spline  $x_i$  divided by the grid size. The equation of a third order b-spline, or cubic b-spline, is

$$\phi_i(u_i) = \begin{cases} \frac{2}{3} - \frac{1}{2}|u_i|^2(2 - |u_i|), & 0 \leq |u_i| < h, \\ \frac{1}{6}(2 - |u_i|)^3, & h \leq |u_i| < 2h, \\ 0, & 2h \leq |u_i|, \end{cases} \quad (2.16)$$



**Figure 2.5:** Example of a centred b-spline around point 5.

where  $h$  is equal to the grid size and  $u_i = (x - x_i)/h$ . The b-splines need accurate coefficients in order to satisfy the pass-through requirement. The proper coefficient can be calculated using

$$\begin{bmatrix} f_2 \\ \vdots \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} \phi_{1,3}(x_2) & \dots & \phi_{n,3}(x_2) \\ \vdots & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ \phi_{1,3}(x_{n-1}) & \dots & \phi_{n,3}(x_{n-1}) \end{bmatrix} \begin{bmatrix} c_2 \\ \vdots \\ c_{n-1} \end{bmatrix} = \frac{1}{6} \begin{bmatrix} 1 & 4 & 1 & 0 & \dots & \dots & 0 \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \dots & 0 & 1 & 4 & 1 \end{bmatrix} \begin{bmatrix} c_2 \\ \vdots \\ \vdots \\ \vdots \\ c_{n-1} \end{bmatrix}, \quad (2.17)$$

where  $\phi_{i,3}$  is the cubic b-spline around the position  $x_i$ . Also here, the coefficients can be calculated by solving a tridiagonal linear system. Like the piecewise polynomial interpolation, the tridiagonal system can not determine the coefficients at the boundary. Therefore, the B-spline also need to applied endpoint constraints. Barsky [15] describes the end constraints for b-spline interpolation. Barsky uses phantom points to interpolate the endpoints properly. These phantom points are used determine the boundary conditions with respect to the derivatives at the boundary. However, the gridded data does not have derivatives boundaries. Barsky provides another method called double vertices. Here, the boundary coefficients  $c_1$  and  $c_n$  are also used for coefficients outside the scope ( $c_0$  and  $c_{n+1}$ ), resulting in the following equation

$$\begin{bmatrix} f_1 \\ \vdots \\ f_{n-1} \end{bmatrix} = \frac{1}{6} \begin{bmatrix} 5 & 1 & 0 & \dots & \dots & \dots & 0 \\ 1 & 4 & 1 & 0 & \ddots & \ddots & \vdots \\ 0 & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & 0 & 1 & 4 & 1 \\ 0 & \dots & \dots & \dots & 0 & 1 & 5 \end{bmatrix} \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ \vdots \\ \vdots \\ c_{n-1} \\ c_n \end{bmatrix}. \quad (2.18)$$

### 2.3.2 Comparing basic multivariate interpolations

The basics of two different 1D interpolation method have been discussed in the previous section. This section discusses how these interpolation methods can be used to interpolate a 3D grid. First, the 3D interpolation of the piecewise polynomial interpolation is discussed. The tricubic piecewise polynomial interpolation method is proposed by Lekien et al. [16], which uses the following equation to calculate the

piecewise polynomials

$$f(x, y, z) = \sum_{i,j,k}^3 a_{ijk} x^i y^j z^k, \quad (2.19)$$

where  $a_{ijk}$  the coefficients of the tricubic interpolation. Therefore, each polynomial which describes the interpolation between 8 voxels has 64 coefficients. Lekien et al. chose to simplify the interpolation because of computational complexity. However, the simplified method of Lekien et al. does not provide a  $C^2$  function as the output of the interpolation. The total number of unknown coefficients in tricubic interpolation is given by the following formula

$$N_{coef} = 64(n_x - 1)(n_y - 1)(n_z - 1), \quad (2.20)$$

where  $N_{coef}$  are the number of coefficients,  $n_x$ ,  $n_y$  and  $n_z$  are the number of grid points in the  $x$ ,  $y$  and  $z$  direction. Where b-spline interpolation uses the following equation to interpolate a three-dimensional space

$$f(x, y, z) = \sum_{i=1}^{n_1} \sum_{j=1}^{n_2} \sum_{k=1}^{n_3} c_{i,j,k} B(x - x_i, y - y_j, z - z_k), \quad (2.21)$$

where  $B(x, y, z)$  is the three dimensional b-spline. According to Schumaker [17], (2.21) can be reduced when the point cloud is an uniform grid. The reduced equation of (2.21) is the following equation

$$f(x_l, y_m, z_n) = f_{lmn} = \underbrace{\sum_{i=1}^{N_x} \left[ \sum_{j=1}^{N_y} \underbrace{\left[ \sum_{k=1}^{N_z} c_{i,j,k} \gamma_k(z_n) \right]}_{W_{i,j,n}} \psi_j(y_m) \right]}_{V_{i,m,n}} \phi_i(x_l), \quad (2.22)$$

where  $f(x_l, y_m, z_n)$  is the interpolated function at points  $x_l$  and  $y_m$  and  $z_n$ . The basis function  $\phi_i(x)$  is a 1D b-spline in  $x$ -direction around point  $x_i$ .  $\psi_j(y)$  is a basis function in  $y$ -direction around point  $y_j$ .  $\gamma_k(z)$  is a basis function  $z$ -direction around point  $z_k$ .  $N_x$ ,  $N_y$  and  $N_z$  are the number of grid points and b-splines in respectively  $x$ -,  $y$ - and  $z$  direction, resulting in the following equation for the number of unknown coefficients

$$N_{coef,b} = N_x N_y N_z. \quad (2.23)$$

As can be seen from (2.20) and (2.23), the total number of unknowns in cubic b-spline interpolation is less than the total number of unknowns in cubic piecewise polynomial interpolation. Therefore, cubic b-spline interpolation should have a lower calculation time than cubic piecewise polynomial interpolation.

### 2.3.3 Multivariate cubic b-spline

The coefficients of cubic b-spline interpolation have to be calculated correctly in order to satisfy the pass-through requirement. Schumaker [17], De Boor [7] and Bartels [18], all use (2.22) as the function to interpolate a three-dimensional space. (2.22) can be rewritten into three equations

$$f_{lmn} = \sum_{i=1}^{N_x} V_{i,m,n} \phi_i(x_l), \quad (2.24)$$

$$V_{i,m,n} = \sum_{j=1}^{N_y} W_{i,j,n} \psi_j(y_m), \quad (2.25)$$

$$W_{i,j,n} = \sum_{k=1}^{N_z} c_{i,j,k} \gamma_k(z_n). \quad (2.26)$$

These equations show that a three-dimensional cubic b-spline interpolation can be seen as multiple one-dimensional interpolations. The above equation can be solved, by solving the following equations

$$\begin{bmatrix} \phi_1(x_1) & \dots & \phi_{N_x}(x_1) \\ \vdots & \ddots & \vdots \\ \phi_1(x_{N_x}) & \dots & \phi_{N_x}(x_{N_x}) \end{bmatrix} \begin{bmatrix} V_{1,m,n} \\ \vdots \\ V_{N_x,m,n} \end{bmatrix} = \begin{bmatrix} f_{1,m,n} \\ \vdots \\ f_{N_x,m,n} \end{bmatrix}, \quad \forall m \in [1 \dots N_y], \quad \forall n \in [1 \dots N_z], \quad (2.27)$$

$$\begin{bmatrix} \psi_1(y_1) & \dots & \psi_{N_y}(y_1) \\ \vdots & \ddots & \vdots \\ \psi_1(y_{N_y}) & \dots & \psi_{N_y}(y_{N_y}) \end{bmatrix} \begin{bmatrix} W_{i,1,n} \\ \vdots \\ W_{i,N_y,n} \end{bmatrix} = \begin{bmatrix} V_{i,1,n} \\ \vdots \\ V_{i,N_y,n} \end{bmatrix}, \quad \forall i \in [1 \dots N_x], \quad \forall n \in [1 \dots N_z], \quad (2.28)$$

$$\begin{bmatrix} \gamma_1(z_1) & \dots & \gamma_{N_z}(z_1) \\ \vdots & \ddots & \vdots \\ \gamma_1(z_{N_z}) & \dots & \gamma_{N_z}(z_{N_z}) \end{bmatrix} \begin{bmatrix} c_{i,j,1} \\ \vdots \\ c_{i,j,N_z} \end{bmatrix} = \begin{bmatrix} W_{i,j,1} \\ \vdots \\ W_{i,j,N_z} \end{bmatrix}, \quad \forall i \in [1 \dots N_x], \quad \forall j \in [1 \dots N_y]. \quad (2.29)$$

These equation shows that a 3D space can be interpolated with  $(N_x \times N_y)$  1D interpolations in  $z$ -direction,  $(N_x \times N_z)$  1D interpolations in  $y$ -direction and  $(N_z \times N_y)$  1D interpolations in  $x$ -direction. In other words, multivariate interpolation can be accomplished by multiple 1D interpolations. The coefficients of the multivariate interpolation can also be calculated by multiple 1D interpolation (2.18).

#### 2.3.4 Fast multivariate interpolation

Chapter 1 discusses that a fast interpolation method is desired. Therefore, this section discusses a fast multivariate interpolation method. The multivariate b-spline interpolation consists out of two parts. First, calculating the correct coefficients to interpolate the voxel grid. Second, the calculation of a value in the space based on the coefficients. The calculation of the correct value based on the coefficients will be referred to as the value calculation part. Therefore, the calculation time of the algorithm can be improved in two parts. Both parts are evaluated on improvements.

Unser et al. [19] [20] propose a fast b-spline interpolation method. According to [20], the digital filter method interpolates faster than the tridiagonal matrix method (2.18). However, [20] does not provide proof that the digital filter method is always faster than other methods. [20] proposes to use a digital filter to calculate the coefficients of the b-spline interpolation. Like in the multivariate interpolation of the previous section, a multi-dimensional digital filter can be seen as multiple one-dimensional digital filters. Therefore, the proposed method of [20] is explained solely in one-dimension. The digital filter interpolation rewrites (2.13) using discrete b-splines

$$f_i = (\phi * c)(x_i), \quad (2.30)$$

where  $f_i$  is the value of the gridded data at point  $x_i$ ,  $\phi$  are the b-splines and  $c$  are the coefficients around point  $x_i$ . Using the convolution theorem [21], the equation above can be rewritten as

$$F(z) = \Phi(z)C(z), \quad (2.31)$$

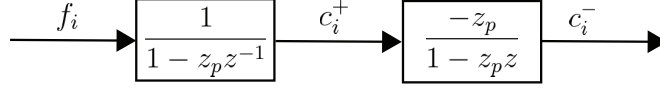
where  $F(z)$ ,  $\Phi(z)$  and  $C(z)$  are the  $z$ -transforms of  $f_i$ ,  $\phi_i$  and  $c_i$ . Therefore, the coefficients can be calculated using the following equation

$$C(z) = \Phi^{-1}(z)F(z), \quad (2.32)$$

where,

$$\Phi^{-1}(z) = \frac{\lambda}{z + 4 + z^{-1}} = \lambda \left( \frac{1}{1 - z_p z^{-1}} \right) \left( \frac{-z_p}{1 - z_p z} \right), \quad (2.33)$$

where  $z_p$  is the smallest root of the polynomial  $z^2 + 4z + 1$  and  $\lambda$  is a gain. In case of the cubic b-splines,  $z_p$  is equal to  $\sqrt{3} - 2$  and  $\lambda$  is equal to 6. As can be seen in (2.33), the digital filter consist of a causal and an anti-causal filter. These causal and anti-causal filters are visualized in Figure 2.6.



**Figure 2.6:** Graphic representation of the causal and anti-causal filter. [20]

Here  $c_i^+$  is the output of the causal filter and  $c_i^-$  is the output of the anti-causal filter. Usually, anti-causal functions cause problems because the next point of the signal is not known. However, the anti-causal filter means that the current coefficient depends on the next pixel, in this case. Therefore, the digital filter results in the following equations

$$c_i^+ = f_i + z_p c_{i-1}^+, \quad (2.34)$$

$$c_i^- = z_p (c_{i+1}^- - c_i^+), \quad (2.35)$$

$$c_i = \lambda c_i^-. \quad (2.36)$$

As in the previous interpolation methods, the interpolation needs a boundary condition. Paper [20] imposes a zero derivative at the endpoints. Resulting in the following equation for boundary conditions

$$c_0^+ = f_0 + \frac{1}{1 - z_p^{2N}} \sum_{k=0}^{N-1} (z_p^{k+1} + z_p^{2N-k}) f_k, \quad (2.37)$$

and

$$c_{N-1}^- = -\frac{z_p}{1 - z_p} c^+(N-1), \quad (2.38)$$

where  $N$  are the number of b-splines. These coefficients can then be used to calculate the value at any point in space using (2.22). Ruijters et al. [22] show that the digital filter method of [20] can be implemented into CUDA. Therefore, the coefficients of the b-splines can be calculated fast using the digital filter method. Calculating the desired output based on these coefficients will be the same as in (2.22).

### 2.3.5 Summary

A cubic b-spline interpolation has significant fewer unknowns than a tricubic piecewise polynomial interpolation. Hence, it is assumed that b-spline interpolation is faster than a tricubic piecewise polynomial interpolation. Moreover, a tricubic interpolation method is a much more complex interpolation method. The coefficients of the b-splines still need to be calculated while satisfying the pass-through requirement. Unser et al. [19] state that their digital filter method is much faster in calculating the b-spline coefficients than using (2.18). However, the paper of Unser et al. does not provide mathematical proof that their method is faster than other methods. Consequently, the method of Unser et al. has to be verified whether the digital filter is indeed faster than solving (2.18).

## 2.4 Newton's method

Newton's method or Newton-Raphson method is a root-finding algorithm. Newton's method approximates the actual solution ( $x^*$ ) by the two-term Taylor expansion [6]. Therefore, Newton's method results in the following equation to find the root

$$x_{i+1} = x_i - f'(x_i)^{-1}f(x_i), \quad (2.39)$$

with  $x_i$  is the point at iteration  $i$  and  $f(x_i)$  is the function value of point  $x_i$ . Newton's method iterates over (2.39) resulting in an algorithm which approximates the actual root with

$$x_\infty = x^*. \quad (2.40)$$

The proof that Newton's method converges is given later in this thesis. Newton's method can solely be used when the function satisfies three assumptions. First, the function  $f(x)$  has to have a solution which results in

$$f(x^*) = 0. \quad (2.41)$$

Second, the first derivative  $f'$  is Lipschitz continuous with Lipschitz constant  $\gamma$ . Third, the first derivative  $f'$  is nonsingular. These three assumptions are referred to as the standard assumptions.

Newton's method can also be used to find a local minimum of a function  $g(x)$ . Instead of solving  $g(x) = 0$ , Newton's method is used to find

$$g'(x) = 0, \quad (2.42)$$

where  $g'(x)$  is the first derivative of the function  $g(x)$ . The minimizer is defined as  $x^*$ . (2.42) is not sufficient to find a minimum or minimizer. Point  $x^*$  is solely a minimum when it satisfies (2.42) and the following equation

$$g''(x^*) > 0, \quad (2.43)$$

where  $g''(x)$  is the second derivative of function  $g(x)$ . The minimum finding algorithm will iterate over the following equation

$$x_{i+1} = x_i - g''(x_i)^{-1}g'(x_i). \quad (2.44)$$

### 2.4.1 Convergence rate

The benefit of Newton's method is its convergence rate. Simpler root finding algorithms, like gradient descent method, have a linear convergence rate. However, Newton's method has a local quadratic convergence rate [6]. In other words, Newton's method has locally the following convergence rate

$$\|e_{i+1}\| \leq K\|e_i\|^2, \quad (2.45)$$

where  $K$  is a constant which is  $> 0$  and

$$e_i = x_i - x^*. \quad (2.46)$$

The proof that Newton's method satisfies the quadratic convergence rate locally is given in [6] and can also be found in Appendix B. Newton's method has locally a quadratic convergence when the standard assumptions hold. Newton's method is within the quadratic convergence rate when  $x_i$  is close enough to the actual solution.



### 2.4.2 Convergence rate of an approximated function

However, linear interpolation of the voxel grid is not reliable in Newton-Like minimization according to Chapter 1. The Newton's method which has to calculate with a linear interpolation will be referred to as inexact Newton's method. Due to the difference approximation for the derivatives, the difference approximation is an approximation of the exact derivative of  $g(x)$ . The inexact Newton's method does not calculate with  $g'(x)$  but with

$$g'(x) + \epsilon(x), \quad (2.47)$$

where  $\epsilon(x)$  is the error of the approximated derivative with respect to the exact derivative. Moreover, the second derivative is also approximated with a central difference. The forward difference approximation of  $g''(x)$  is

$$\frac{g'(x+h) + \epsilon(x+h) - g'(x) - \epsilon(x)}{h}, \quad (2.48)$$

where  $h$  is the grid size. Let  $\bar{\epsilon}$  be defined as  $\|\epsilon(x)\| \leq \bar{\epsilon}, \forall x$  then

$$g''(x) - \frac{g'(x+h) + \epsilon(x+h) - g'(x) - \epsilon(x)}{h} = O(h + \bar{\epsilon}/h). \quad (2.49)$$

This upper bound is minimized when  $h = \sqrt{\bar{\epsilon}}$ . Based on this information, the convergence rate of Newton's method can be calculated, resulting in the following convergence rate

$$\|e_{i+1}\| \leq K_d(\bar{\epsilon} + (\|e_i\| + h)\|e_i\|), \quad (2.50)$$

where  $K_d$  is some positive constant. (2.50) shows that the inexact Newton's method might not converge. Hence, the convergence will stagnate and cease to decrease when  $\|e_\infty\| \approx \bar{\epsilon}$ .

Therefore, the linear interpolation and difference approximation will not converge to the minimum. Moreover, (2.50) solely holds in the case that the grid size satisfies

$$h = \sqrt{\bar{\epsilon}}. \quad (2.51)$$

In case of an autonomous interpolation of an unknown environment, the linear interpolation might have a significant error for a certain grid size. Therefore, the combination of linear interpolation and difference approximations is not desired in Newton-Like minimization.

### 2.4.3 Terminal condition

Because the actual minimizer is not known in most cases, Newton's method needs a terminal condition which triggers when Newton's method is close to the exact solution. In the exact Newton's method has local quadric convergence rate, as in (2.45). The Newton's method iterates over (2.39) resulting in

$$\begin{aligned} s &= -f'(x_i)^{-1}f(x_i), \\ &= x_{i+1} - x_i, \\ &= x_{i+1} - x^* + x^* - x_i, \\ &= e_{i+1} + e_i. \end{aligned}$$

Combining the results above with the quadratic convergence rate of (B.7), the size of the Newton step  $s$  is equal to

$$\|s\| \leq \|e_{i+1}\| + \|e_i\| \leq O(\|e_i\|^2) + \|e_i\|. \quad (2.52)$$

As a result, the size of the Newton step gives a good indication of the size of the error.

However, (2.52) does not hold in the inexact Newton's method. In case of the inexact Newton's method, the Newton's step is equal to

$$\|s\| = \|e_{i+1}\| + \|e_i\| \quad (2.53)$$

$$\leq \|e_i\| + K_d(\bar{\epsilon} + (\|e_i\| + h)\|e_i\|) \quad (2.54)$$

$$\leq K_d\bar{\epsilon} + (K_d(\|e_i\| + h) + 1)\|e_i\|. \quad (2.55)$$

These equations imply the same problem as the convergence rate of the inexact Newton's method. When  $\|e_i\| \gg \sqrt{\bar{\epsilon}}$  then the Newton's step will have the same property as the exact Newton's method. However, the Newton's step will be  $O(\bar{\epsilon})$  when  $\|e_i\| \leq \sqrt{\bar{\epsilon}}$ . Therefore, Newton's step is not as reliable in the inexact Newton's method as in the exact Newton's method. Therefore, Newton's step should still give an idea of the error.

#### 2.4.4 Backtracking line search

This section has briefly described the literature on Newton's method. However, this thesis uses Newton's method which differs as it has been discussed in this section. When this thesis refers to Newton's method, it actually refers to Newton's method together with an Armijo-Goldstein backtracking. This research uses the following equation instead of (2.44)

$$x_{i+1} = x_i - \alpha g''(x_i)^{-1} g'(x_i), \quad (2.56)$$

where  $\alpha$  is the step size. Moreover, the step size is assumed to be appropriate when the following condition is satisfied

$$g(x_i + \alpha_i s_i) \leq g(x_i) + c\alpha \nabla g'(x_i)^T s_i, \quad (2.57)$$

where

$$s_i = g''(x_i)^{-1} g'(x_i), \quad (2.58)$$

and  $c \in [0, 1]$  is a constant. When the minimization does not satisfy (2.57) then the step size is updated

$$\alpha_{i+1} = \alpha_i \beta, \quad (2.59)$$

where  $\beta$  is the backtracking constant. The proof that this method still converges to a local minimum, is given in the book of Nocedal and Wright [23].

## 2.5 Summary

This chapter has discussed the mathematical preliminaries and background information for this thesis. The background of implicit representation are explained in Section 2.1. This thesis will use a Truncated Signed Distance Field (TSDF) as an implicit representation for describing the obstacle avoidance constraint. An environment can be scanned using a depth camera and KinectFusion Framework to create a 3D TSDF of it. The basic information on KinectFusion Framework has been explained in Section 2.2. The output the KinectFusion Framework is a discrete representation of the environment. However, the obstacle avoidance constraint has to satisfy two requirements as discussed in Chapter 1. In order to satisfy these requirements, the discrete TSDF has to be interpolated. In Section 2.3, several interpolation methods have been research in order to find the fastest interpolation method which satisfies the requirements of Chapter 1. The literature study does not provide which interpolation method is the fastest. Unser et al. [20] claim that their digital filter method is the fastest method. However, the paper [20] does not give a mathematical proof of being faster. Concluding, the method of Unser et al. has to be compared with other interpolation methods. In Chapter 1, the interpolation method is stated to be a  $C^2$  function else it can not be used in Newton-Like minimization methods. The idea behind Newton-Like

minimization methods has been discussed in Section 2.4. The Newtons minimization method is discussed for a function which is a  $C^2$  function and for a function which does not satisfy the smoothness requirement. The conclusion of Section 2.4 is that Newtons minimization methods with  $C^2$  function will always convergence when certain assumptions are met. However, Newtons minimization might not convergence when the function which is minimized is not  $C^2$ .

### 3 Proposing a fast interpolation method for optimal motion planning

This chapter discusses the interpolation method, which is proposed to interpolate the discrete TSDF. The interpolation method has to satisfy two different requirements as introduced in the introduction. The first requirement is the smoothness requirement which states that the interpolation method provide a  $C^2$  function. The second requirement is the pass-through requirement which states that the interpolation method should pass-through the discrete TSDF points. A third desired property of the interpolation method was introduced in Section 2.3.2. Several interpolation methods require a uniform grid in order to interpolate the point cloud. Last, the interpolation method should be as fast as possible. Unser et al. [20] states their method is the fastest interpolation method. this chapter will start with a comparison of Unser's digital filter with the tridiagonal method of calculating the coefficients. Secondly, this chapter will discuss the implementation of the chosen interpolation method. Thirdly, the proposed interpolation method is verified by means of numerical simulations that the proposed interpolation method satisfies the above-mentioned requirements. Lastly, the interpolated TSDF can not be exactly the same as the exact TSDF. Moreover, the optimal motion planning will be done on the interpolated TSDF. Therefore, the optimal motion planning algorithm might find a feasible solution which is not feasible with respect to the exact TSDF. The last section discusses this problem more in-depth. Furthermore, this section validates the error of the zero level set of the interpolated TSDF with respect to the zero level set of the exact TSDF. In this validation, the accuracy of the zero level set approximation is validated with respect surrounding and the used grid size. The surrounding or obstacle is represented by the smallest radius which is present.

#### 3.1 Digital filter vs tridiagonal interpolation

In Section 2.3.5, two different methods of obtaining the coefficients for cubic b-spline interpolation have been discussed. According to Unser et al. [19], their digital filter method is the fastest method to calculate the coefficients. This section compares the digital filter method with a second interpolation method. The second method, discussed in the background, is the tridiagonal matrix method for calculating the coefficients. The calculation time of the digital filter method is compared with the calculation time of the tridiagonal matrix method.

The digital filter method is optimized for low calculation time. However, the tridiagonal interpolation method is not optimized for calculation time. Hence, the comparison will solely be fair when the tridiagonal method is optimized for calculation time as well. According to Conte and De Boor [24], the Thomas algorithm is a fast solution to solve tridiagonal matrix systems. Thomas algorithm is used to solve the tridiagonal method as fast as possible.

As discussed in the background, multivariate interpolation is the same as multiple 1D interpolation methods when the point cloud is a uniform grid. As discussed in the introduction of this chapter, the grid which is interpolated is required to be uniform. Therefore, the calculation times of the multivariate method can be compared based on the calculation time of their 1D interpolations. The computation time of an algorithm depends on which computer hardware and software is used. Table 3.1 shows which computer hardware has been used in this thesis. The software which has been used can be seen in Table 3.2.

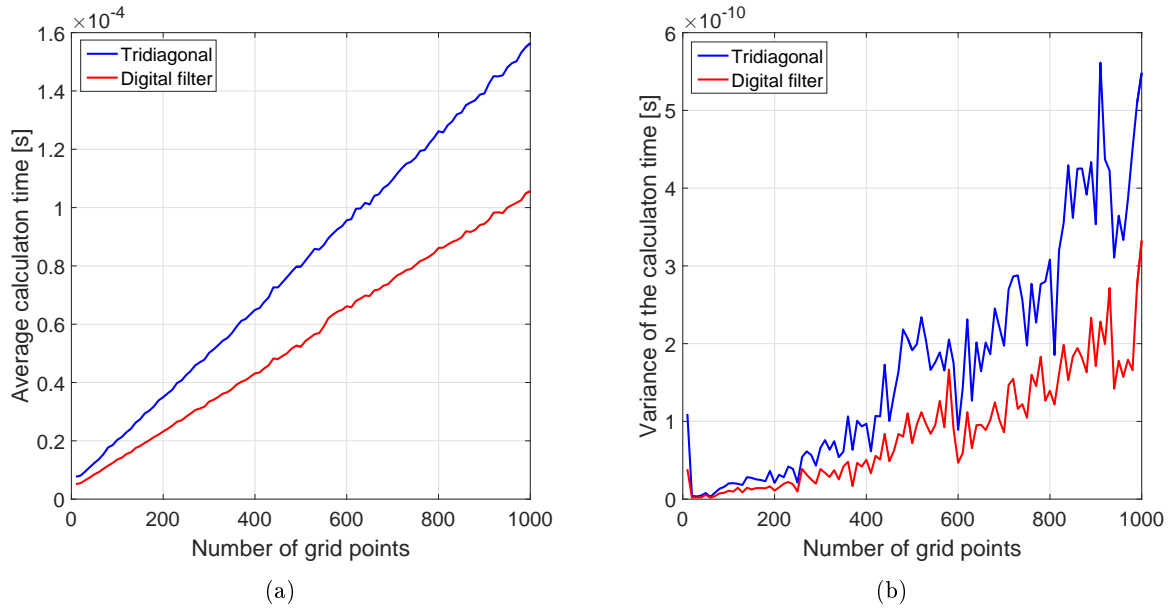
Device	Type
CPU	Intel Core i7-4702MQ CPU @ 2.2GHZ 2.20
GPU	Nvidia GeForce 940M, 2GB
RAM	single 8GB DDR3 1600 GHz

**Table 3.1:** *Computer hardware of this thesis.*

Program	Version
Ubuntu	14.04
CUDA	8.0
Matlab	2015b

**Table 3.2:** *Software which have been used in this research.*

The calculation time of both interpolation methods can be seen in Figure 3.1. Figure 3.1(a) shows that average calculation time of the digital filter method is lower than the tridiagonal method. Figure 3.1(b) shows that the variance of both interpolation methods is negligible over the entire test. The interpolation methods have interpolated 1000 different white noise signal of a certain length. The reason for multiple interpolations is to ensure brief delays in the processor do not influence the results. Moreover, the results of this test give an idea of the consistency of the algorithms. Based on Figure 3.1, the digital filter method is significantly faster in calculating the coefficients than the tridiagonal method.



**Figure 3.1:** *Comparing Unser's interpolation method with solving tridiagonal interpolation with Thomas algorithm. Figure (a) shows the average calculation time of both interpolation methods. Figure (b) shows the variance in calculation time of the interpolation methods.*

### 3.2 The implementation of the proposed interpolation method

As shown in Section 2.3.5 and previous section, Unser's digital filter method is the desired interpolation method for optimal motion planning. The Section 2.3.4 discusses the formulas which can be used to calculate the coefficients of a cubic b-spline interpolation. The implementation of Unser's digital filter

method to calculate the coefficients is discussed in this section. Moreover, the equations which describe the interpolated version of the discrete TSDF are discussed. The interpolated version of the discrete TSDF will be referred to as the interpolated TSDF.

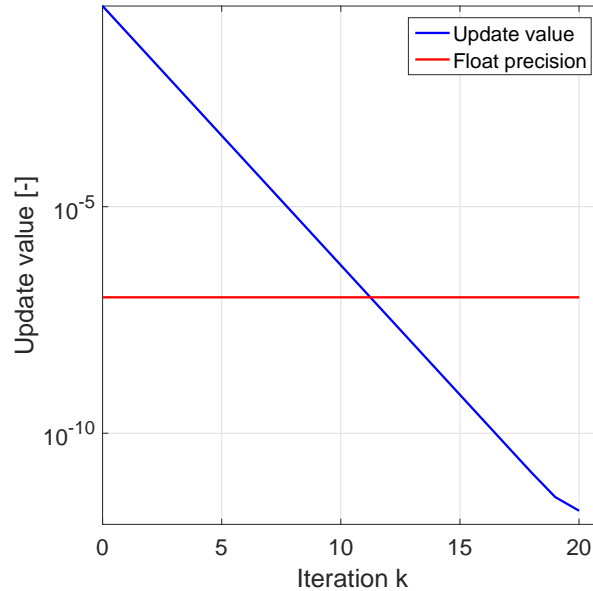
Chapter 2 shows that multivariate interpolation can be accomplished by interpolation an N-dimensional space with multiple 1-dimensional interpolations. Therefore, the 1D digital filter is explained before discussing the multivariate interpolation. Unser et al. [20] propose to use a zero first derivative at the boundary as a boundary condition. Therefore, the first coefficients of the one-dimensional digital filter interpolation is calculated using the following formula

$$c_0^+ = f_0 + \frac{1}{1 - z_p^{2N}} \sum_{k=0}^{N-1} (z_p^{k+1} + z_p^{2N-k}) f_k, \quad (3.1)$$

where  $f_0$  is the value of the first grid point,  $N$  is the number of grid size,  $z_p$  is the pole of the cubic b-spline and  $c_0^+$  is the first coefficient of the causal filter. However, the machine precision of a computer is not taken into account. A standard for floating-point arithmetic is IEEE Std 754-2008 [25]. With this method, a float stores a value with 16-bits precision. Resulting in an error relative to the absolute value. For example, a float with the value of 1 has a floating point precision of  $\pm 1.2e - 7$  while a float value of 1000 has a precision of  $\pm 6.1e - 5$ . Moreover, the signal  $f_k$  has to be around the same order of magnitude. In our case of a TSDF, the signal  $f_k$  should always satisfy this requirement. Therefore, the updated value is solely influential when

$$\beta_k = |z_p^{k+1} + z_p^{2N-k}| > 1.1921e^{-7}, \quad (3.2)$$

where  $\beta_k$  is the updated value at iteration  $k$ .



**Figure 3.2:** The absolute updated value against  $k$ -th iteration.

Figure 3.2 shows the updated value with respect to the value of  $k$ . As can be seen, the updated value is lower than the float precision when  $k > 12$ . Based on the same precision of the float, (3.1) can be

simplified. When the number of grid points is bigger than 12, the following equations hold

$$z_p^{2N} < 1.1921e^{-7}, \quad \forall N > 12, \quad (3.3)$$

$$z_p^{2N-k} < 1.1921e^{-7}, \quad \forall N > 12, \quad (3.4)$$

$$z_p^{k+1} < 1.1921e^{-7}, \quad \forall k > 12. \quad (3.5)$$

If the equations above are used together with the float point precision, (3.1) can be reduced to

$$c_0^+ = f_0 + \sum_{k=0}^{N-1} z_p^{k+1} f_k. \quad (3.6)$$

This first coefficient can be used to calculate the other coefficient

$$c_i^+ = f_i + z_p c_{i-1}^+, \quad (3.7)$$

where  $c_i^+$  is the causal coefficient of the  $i$ th grid point and  $f_i$  is the value of the discrete function at the  $i$ th grid point. If the algorithm has looped over all grid points, the causal filter has finished. The anti-causal filter starts with the following boundary condition

$$c_{N-1}^- = -\frac{z_p}{1-z_p} c_{N-1}^+, \quad (3.8)$$

where  $c_{N-1}^-$  is the anti-causal coefficient of the last grid point. The other anti-causal coefficients are calculated using

$$c_i^- = z_p(c_{i+1}^- - c_i^+). \quad (3.9)$$

Last the anti-causal coefficients are multiplied by a gain resulting in the interpolation coefficients

$$c_i = \lambda c_i^-, \quad (3.10)$$

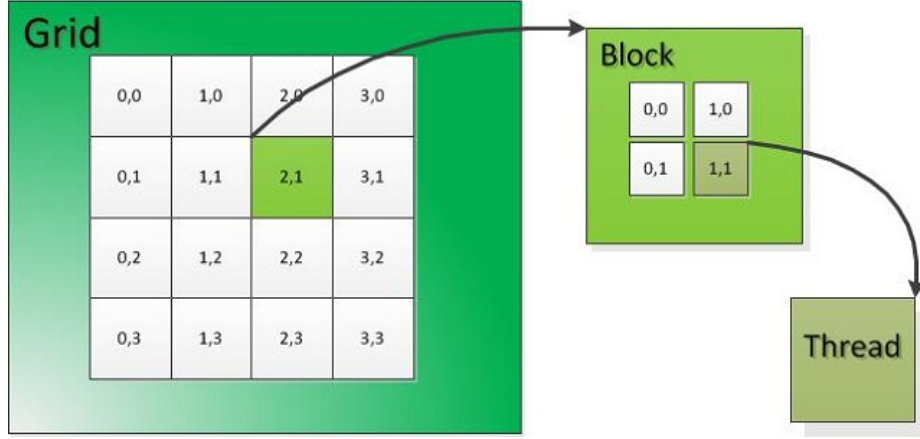
where  $c_i$  is the coefficient corresponding to the  $i$ -th coefficient and  $\lambda$  is the gain. In the cubic b-spline interpolation, this gain is calculated using

$$\lambda = (1 - z_p) \left( 1 - \frac{1}{z_p} \right). \quad (3.11)$$

This 1D interpolation method is the proposed interpolation method for interpolating a point cloud. This interpolation method will be referred to as the proposed 1D interpolation. The proposed 1D interpolation is implemented in CUDA. The proof that the proposed method benefits from a GPU implementation is given later in this report. However, a 1D interpolation does not influence another 1D interpolation. Therefore, the interpolation could benefit from a GPU implementation. A GPU with  $n_{gpu}$  CUDA cores can be faster than a CPU with  $n_{cpu}$  cores when the GPU is able to run  $n_{gpu}$  1D interpolation faster than a single CPU core can run  $n_{gpu}/n_{cpu}$  1D interpolation. The CUDA code of the proposed 1D interpolation can be found in Appendix C.1.

As discussed in the introduction, CUDA and MATLAB are used to calculate on the GPU. However, before multivariate interpolation can be explained, a few CUDA definitions have to be introduced. A CUDA core is a GPU core which can execute some CUDA code. The amount of CUDA cores available on a GPU depends on the GPU version. In this thesis, Nvidia GeForce 940m is used and this GPU has 384 CUDA cores. These CUDA cores or threads are stored in a so-called grid on the GPU. In order to run the CUDA code, this grid has to be divided into multiple blocks with each the same size. With each block having its own block identification. And each thread/CUDA core has its own identification per

block. Figure 3.3 shows the CUDA thread hierarchy in a 2D example. This thesis will not investigate which number of threads and block result in the fastest interpolation. The reason for not investigating this choice is the problem is complex and the influence of the choice is assumed to be minimal. For all CUDA calculations, the number of threads per block and blocks is calculated with a function. This function is given in Appendix C.5.



**Figure 3.3:** A visual representation of the CUDA thread hierarchy. [26]

A multivariate interpolation is accomplished by multiple 1D interpolation methods, as discussed in the Section 2.3.5. To explain how a three-dimensional point cloud is interpolated, a discrete TSDF of size  $(n_x \times n_y \times n_z)$  is used. This point cloud is not stored with a three-dimensional tensor but with a vector. This vector will be referred to as the coefficient vector. The coefficient vector is initialized with the discrete TSDF. This coefficient vector is first interpolated in the x-direction. The discrete TSDF can be divided in  $(n_y \times n_z)$  lines in x-direction. Each line consists out of  $n_x$  number of points. Each line is interpolated using the proposed 1D interpolation. Therefore, the discrete TSDF is interpolated in x-direction by interpolating  $(n_y \times n_z)$  a lines of  $n_x$  points. As discussed before, the GPU can run these  $(n_y \times n_z)$  in parallel. Each CUDA core on the GPU will do a 1D interpolation. However, each CUDA core needs to know where it can find the first point on the line, where it can find the next point on the line and the last point on its line. In case of the interpolation in the x-direction, the next point on the line can be found at the next point in the coefficient vector. The 1D interpolation has found its last point when it has visited  $n_x$  points. These values are the same for each interpolation in the x-direction. The sole difference between the interpolation is the starting point of the interpolation. A CUDA core knows where to start because its starting position is determined based on its id. The index starting position in the coefficient vector calculated with the following equation

$$id_x = (zn_y + y)n_x, \quad (3.12)$$

where

$$y = bx \times bdx + tx, \quad (3.13)$$

$$z = by \times bdy + ty, \quad (3.14)$$

where  $bx$  and  $by$  are the index of the block in  $x$ - and  $y$ -direction,  $bdx$  and  $bdy$  are the size of block in threads and  $tx$  and  $ty$  are the index of a thread in  $x$ - and  $y$ -direction. The source code of the interpolation of the discrete TSDF in  $x$ -direction can be found in appendix C.2. The output of the  $x$ -direction interpolation is used as an input for the interpolation in  $y$ -direction. The idea of the interpolation is the same, however, this time the interpolation is in  $y$ -direction. Consequently, the 3D grid has to be interpolated  $(n_x \times n_z)$  times using a 1D interpolation in  $y$ -direction. The length of each 1D interpolation in  $y$ -direction is  $n_y$ .



Moreover, if the point of the line is found at index  $i$  the next point of the line will be found at index  $i + n_x$  in the coefficient vector. The first point on the line is calculated using the following formulas

$$id_y = zn_y n_x + x, \quad (3.15)$$

where

$$x = bx \ bdx + tx, \quad (3.16)$$

$$z = by \ bdy + ty. \quad (3.17)$$

The implementation of the interpolation of a three-dimensional grid can be found in Appendix C.3. The interpolation in  $z$ -direction is the last interpolation. Here, the number of 1D interpolations is  $(n_x \times n_y)$  and the length of each interpolation is  $n_z$ . If the point on the line has index  $j$ , the next point on the line can be found at index  $j + n_x \ n_y$  of the coefficient vector. The starting point of the interpolation is calculated using

$$id_z = yn_y + x, \quad (3.18)$$

where

$$x = bx \ bdx + tx, \quad (3.19)$$

$$y = by \ bdy + ty. \quad (3.20)$$

The source code of the  $z$ -direction interpolation can be found in Appendix C.4. If all these interpolation methods are computed, the cubic b-spline coefficients of the discrete TSDF has been calculated.

If the cubic b-spline coefficients are known, the value of the interpolated TSDF can be calculated for each point within the space. The desired position is defined as  $[x_{des}, y_{des}, z_{des}]$  and this corresponds with the following index

$$[l, m, n] \equiv \left[ \left\lfloor \frac{x_{des}}{h} \right\rfloor, \left\lfloor \frac{y_{des}}{h} \right\rfloor, \left\lfloor \frac{z_{des}}{h} \right\rfloor \right], \quad (3.21)$$

where  $h$  is the grid size of the uniform grid,  $\lfloor \cdot \rfloor$  is the floor operation. Based on (3.21), (2.22) can be reduced to

$$f(x_{des}, y_{des}, z_{des}) = \sum_{i=l-1}^{l+2} \sum_{j=m-1}^{m+2} \sum_{k=n-1}^{n+2} c_{i,j,k} \gamma_k(z_{des}) \psi_j(y_{des}) \phi_i(x_{des}). \quad (3.22)$$

This function does not take the boundary conditions into account. When  $k, l$  or  $m$  are zero or respectively  $n_x - 1$ ,  $n_y - 1$  or  $n_z - 1$  then (3.22) does not hold. (3.22) can not solve these cases because  $c_{i,j,k}$  does not exist for the cases

$$i = -1 \wedge n_x,$$

$$j = -1 \wedge n_y,$$

$$k = -1 \wedge n_z.$$

As discussed in Section 2.3.4, the digital filter method implies a zero first derivatives at the boundary. The digital filter mirrors the signal at the boundaries to imply a zero first derivative at the boundary. Therefore, the following equations are used to calculate the unknown coefficients

$$c_{-1,j,k} = c_{1,j,k},$$

$$c_{i,-1,k} = c_{i,1,k},$$

$$c_{i,j,-1} = c_{i,j,1},$$

$$c_{n_x,j,k} = c_{n_x-2,j,k},$$

$$c_{i,n_y,k} = c_{i,n_y-2,k},$$

$$c_{i,j,n_z} = c_{i,j,n_z-2}.$$

(3.22) can be used to calculate the interpolated TSDF value at a desired position. However, the optimal motion planning has to know the derivatives at a desired position as well. The derivative of (3.22) solely depend on the derivatives of the b-splines. Therefore, the Jacobian of (3.22) is given by

$$\nabla f(x_{des}, y_{des}, z_{des}) = \sum_{i=l-1}^{l+2} \sum_{j=m-1}^{m+2} \sum_{k=n-1}^{n+2} c_{i,j,k} \nabla (\gamma_k(z_{des}) \psi_j(y_{des}) \phi_i(x_{des})). \quad (3.23)$$

The Hessian of (3.22) is given by the following formula

$$\nabla^2 f(x_{des}, y_{des}, z_{des}) = \sum_{i=l-1}^{l+2} \sum_{j=m-1}^{m+2} \sum_{k=n-1}^{n+2} c_{i,j,k} \nabla^2 (\gamma_k(z_{des}) \psi_j(y_{des}) \phi_i(x_{des})), \quad (3.24)$$

### 3.3 Verification of the proposed interpolation method

In the previous chapter, an interpolation method is proposed to interpolate the discrete TSDF. As discussed in the introduction, the interpolation method should satisfy two requirements in order to create an interpolated TSDF which can be used in optimization. The first requirement is the smoothness requirement which requires the interpolated TSDF to be a  $C^2$  function. The proposed interpolation method uses cubic b-spline, therefore, the basis function has an exact and continuous first and second derivative. Hence, the smoothness requirement is satisfied by definition and does not need a verification. The second requirement is the pass-through requirement which states that the interpolated TSDF should be exactly the same at the grid points as the discrete TSDF at the grid points. The pass-through requirement is satisfied according to Unser et al. [20]. The method of Unser et al. is called a digital filter, which implies that the signal is altered. However, the pass-through requirement states the signal should not be altered. This chapter verifies whether the pass-through requirement is satisfied. Moreover, the correctness of the proposed interpolation is validated by comparing the proposed interpolation with a default MATLAB interpolation method.

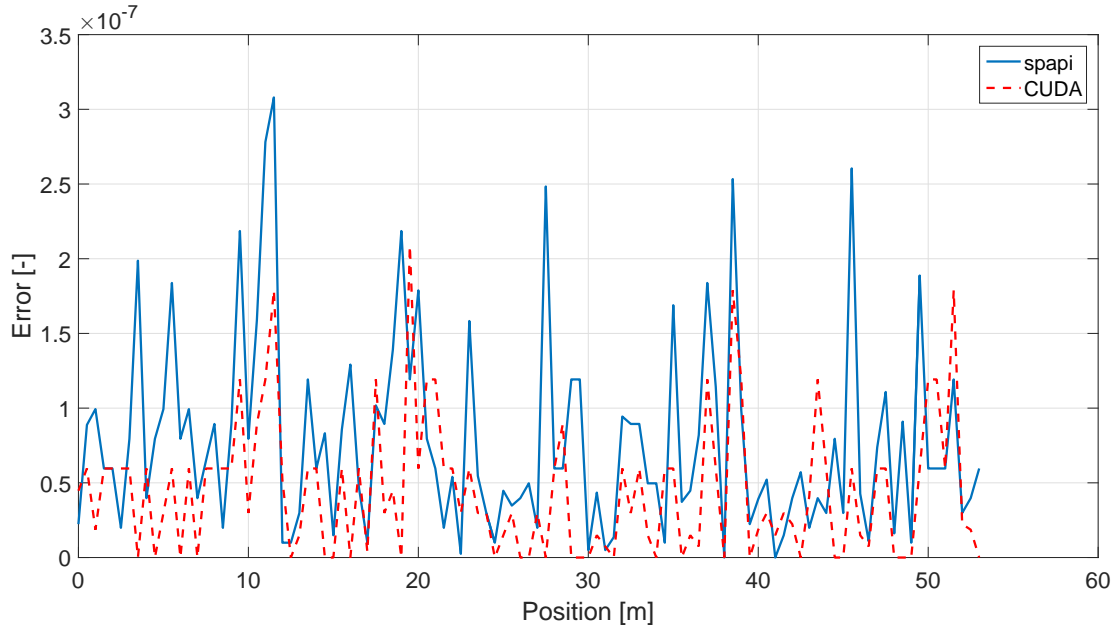
The proposed interpolation method is compared with a default MATLAB interpolation algorithm ('spapi') [14]. Unser [19] states that the digital filter indirectly imposes a zero first derivative at the endpoints. To get a proper comparison, the MATLAB function should have exactly the same boundary condition. The function 'spapi' does not contain the desired boundary condition. However, the function can force the derivative at the endpoints to have a desired value. MATLAB has a double precision by default and the CUDA implementation has single precision. Therefore, the MATLAB interpolation is forced to use single precision instead of double precision.

As discussed in the background, multivariate interpolation can be accomplished by multiple one-dimensional interpolations. Therefore, the verification and validation of the proposed interpolation are first done on proposed 1D interpolation. Second, the proposed multivariate interpolation is validated.

#### 3.3.1 One dimensional CUDA interpolation

As discussed in the introduction of this section, the proposed interpolation method is first verified and validated in 1D. The pass-through requirement of the proposed 1D interpolation is verified by comparing the recreation of the discrete TSDF with the interpolated TSDF at the grid positions. Second, the interpolation method is validated with respect to a MATLAB function. The MATLAB function should also satisfy the pass-through requirement. Therefore, the MATLAB function will be verified together with the proposed 1D interpolation.

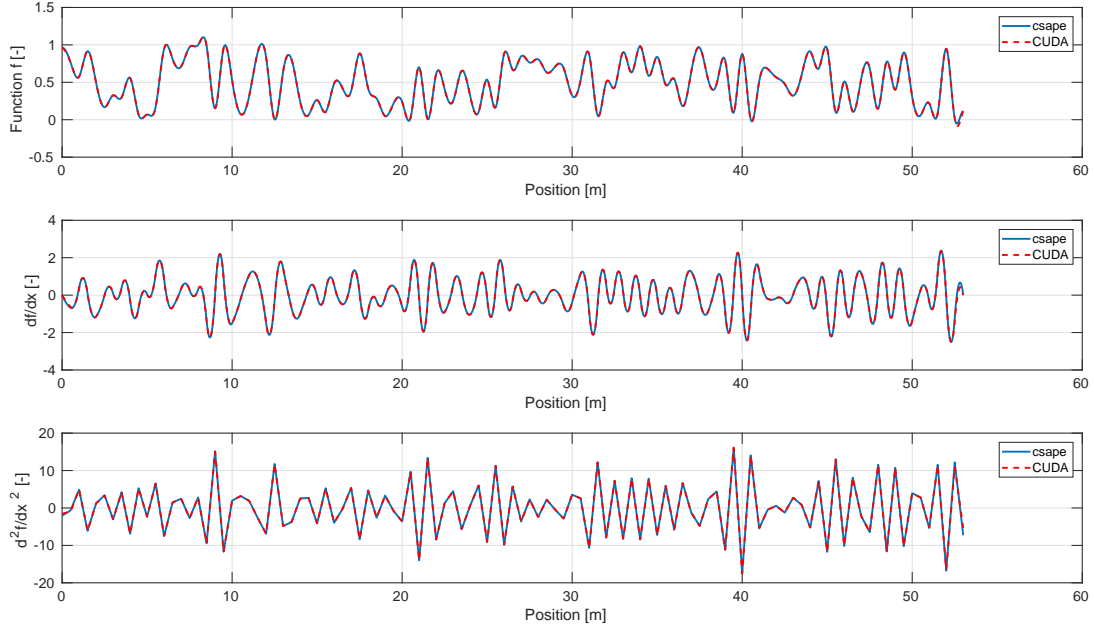
The pass-through requirement states that the interpolated TSDF at the position of the grid points should be exactly the same as the discrete TSDF. To verify whether the proposed and the MATLAB interpolation satisfy this requirement, both interpolation methods have to calculate the value of the interpolated TSDF at the grid points. The discrete TSDF is created by a 1D white noise signal. The interpolation methods are used to interpolate the discrete TSDF and calculate the interpolated TSDF values at the grid positions. The output of the interpolated TSDF at the grid positions determined by the proposed interpolation will be referred to as the discrete proposed interpolation. The results of the recreation of the discrete TSDF by the MATLAB function will be referred to as the discrete MATLAB interpolation. Figure 3.4 shows the absolute error of the discrete proposed interpolation and the discrete MATLAB interpolation with respect to the original discrete TSDF. As can be seen in Figure 3.4, the absolute error of both interpolation methods is  $\pm 1e^{-7}$ . A float has a precision (single precision) of  $1.1921e^{-7}$ , as discussed in Chapter 3.2. Therefore, both interpolation methods satisfy the pass-through requirement, based on Figure 3.4.



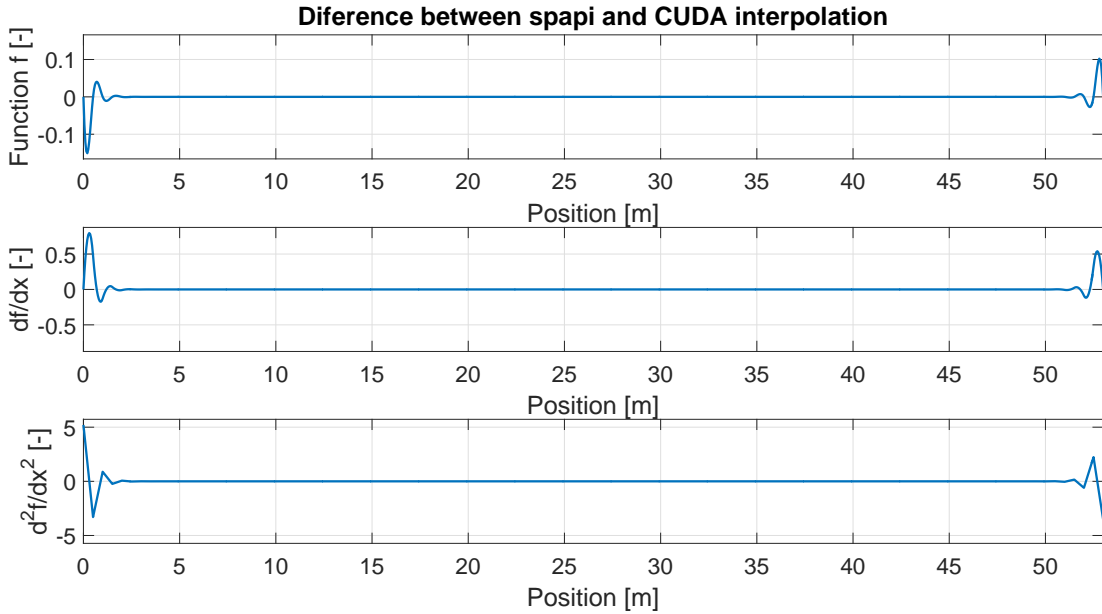
**Figure 3.4:** *The absolute error of both interpolation methods with respect to original discrete TSDF.*

It is proven that the proposed 1D interpolation and the MATLAB interpolation satisfy the pass-through requirement, the proposed 1D interpolation can be compared with the MATLAB 1D interpolation. The interpolation methods will be validated based three different functions. First, the interpolated TSDFs of both interpolation methods are compared whether they are the same interpolated TSDFs. Second, the first derivative of the interpolated TSDF of both method should be identical. Third, the second derivative in space should result in the same function.

The discrete TSDF in this test is again created by a 1D white noise signal. However, the interpolation methods are used to create a discrete interpolated TSDF with a finer grid than the grid of the discrete TSDF. The refined grid is a factor 10 smaller than the grid size of the discrete TSDF. The refined discrete 1D interpolation should be exactly the same as the refined discrete MATLAB interpolation.



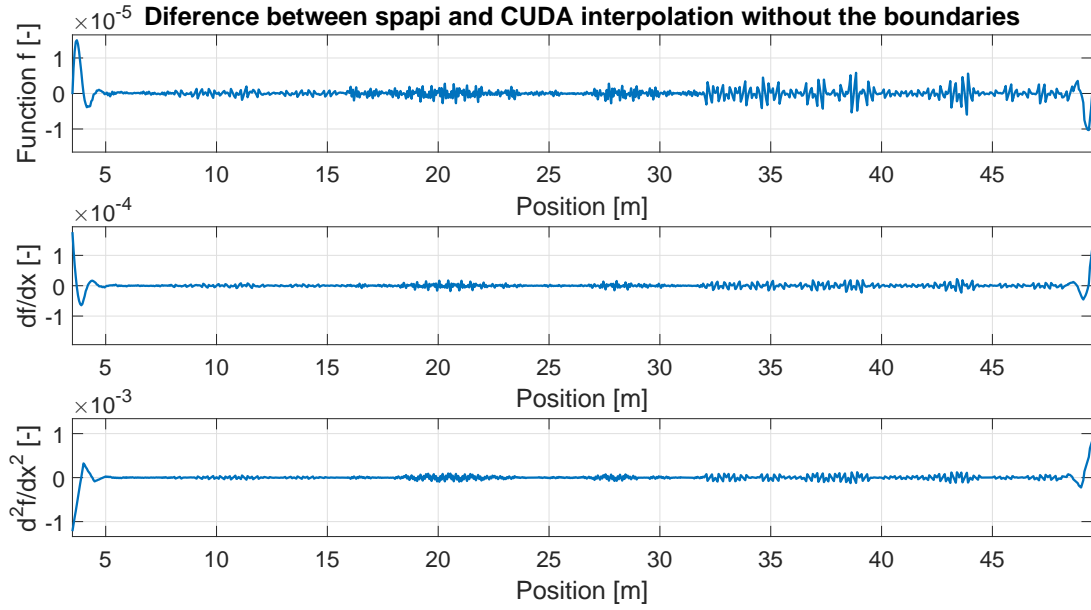
**Figure 3.5:** The refined interpolation of MATLAB interpolation and the proposed interpolation.



**Figure 3.6:** The absolute difference between MATLAB interpolation and the proposed interpolation.

Figure 3.5 shows the refined discrete MATLAB interpolation and the refined discrete proposed 1D interpolation and their first and second derivatives. As can be seen, the refined discrete MATLAB interpolation and the refined discrete proposed 1D interpolation seem to provide exactly the same interpolation. Figure 3.6 shows a significant difference between the MATLAB interpolation and the proposed interpolation method. The difference between the MATLAB and proposed interpolation is significantly larger than the single precision error. The difference between the interpolation method at the boundaries could be

caused by a slight difference in imposing the zero first derivative at the boundaries. Figure 3.5 also shows that both interpolation methods satisfy the endpoint condition of a zero first derivative at the boundaries. Moreover, there is no proof that the boundary condition of the MATLAB interpolation method is a better approximation than the boundary condition of the proposed interpolation method. Hence, the difference between the interpolation methods will be compared without the points close to the boundaries. Figure 3.7 shows the absolute difference between the interpolation methods without the first and last 70 refined points. As can be seen in this Figure, the difference between the methods is negligible when their difference is compared away from the boundaries. Therefore, the proposed 1D interpolation algorithm is concluded to be correct.

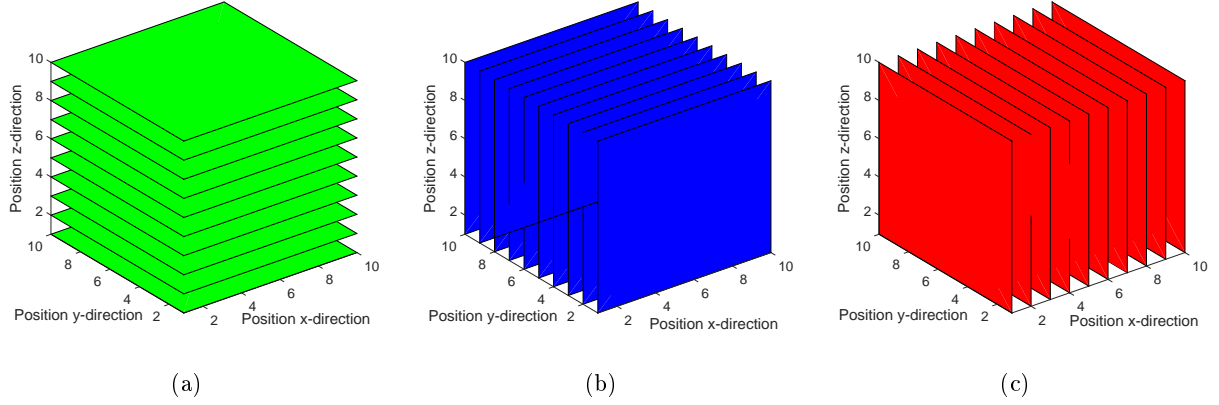


**Figure 3.7:** The absolute difference between ‘spapi’ and CUDA interpolation without the first and last 70 values.

### 3.3.2 Multivariate CUDA interpolation

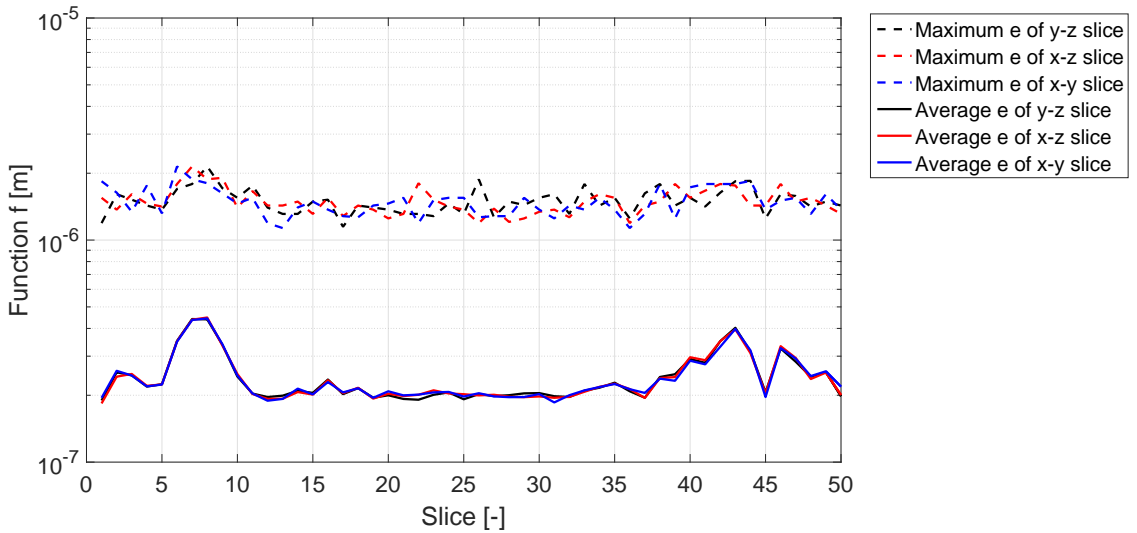
In the previous section, the proposed interpolation method is compared with a MATLAB interpolation method in 1D. However, the optimal motion planning desires to have multi-dimensional obstacle avoidance constraint. This section compares the same interpolation method, however, a three-dimensional space will be interpolated instead of a 1D signal. The three-dimensional spaces or discrete TSDFs are created using a 3D white noise signal. The size of these spaces should be set as close as possible to the KinFu space but not too large for the MATLAB function. The MATLAB function is not able to cope with large spaces due to memory overload. Therefore, the discrete TSDF has a size of  $50 \times 50 \times 50$  voxels. This is the largest discrete TSDF volume which can be refined with a factor 2 by the MATLAB interpolation method.

The visualization of the absolute difference ( $e$ ) between the MATLAB interpolation and proposed interpolation can not be made with default plotting function of MATLAB. Therefore, a new plotting method is introduced: the slice plotting. In slice plotting, the three dimensional voxel grid is replaced by various slices:  $n_z$  slices in  $x - y$  direction,  $n_y$  slices in  $x - z$  direction and  $n_x$  slices  $y - z$  direction. Figure 3.8 shows the slices which represent a 3D tensor in the slice plotting.

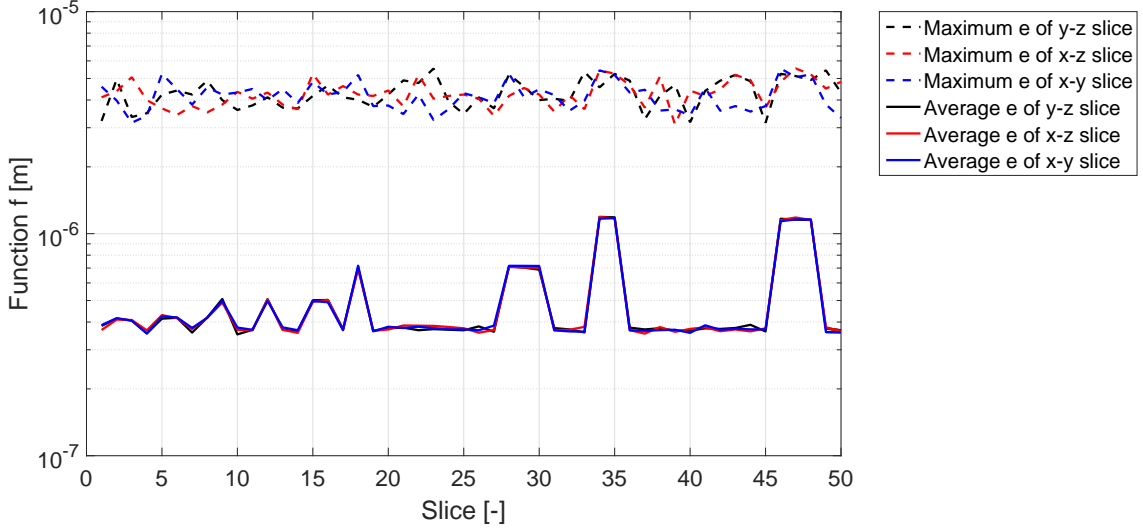


**Figure 3.8:** An  $10 \times 10 \times 10$  tensor divided into three different slice plots. The tensor is represented with 10  $x - y$  slices in (a). The tensor is represented with 10  $x - z$  slices in (b). The tensor is represented with 10  $y - z$  slices in (c).

In this slice plotting, the x-axis represents the index of the slice. Therefore, the figure represents the voxel grid in three directions. The slice plotting is used to plot the average and the maximum absolute difference between the multivariate Matlab and proposed multivariate interpolation which can be found in each slice. Figure 3.9 shows the error of the MATLAB interpolation at the grid points with respect to the original data. As can be seen, the error of the MATLAB interpolation is slightly larger than the single precision error. However, each voxel has been visited by three different 1D interpolations. Therefore, the error in multivariate interpolation might be slightly higher than in a 1D interpolation. The error in Figure 3.9 can still be assumed to be caused by the single precision error. Figure 3.10 shows the error of the proposed interpolation method when recalculating the discrete TSDF. Also here, the error is bigger than in the 1D interpolation. However, the error of the proposed multivariate interpolation is still assumed to be the result of the single precision error.



**Figure 3.9:** The absolute difference ( $e$ ) between MATLAB interpolation and the original discrete TSDF visualized by a mean error and maximum error of a slice.

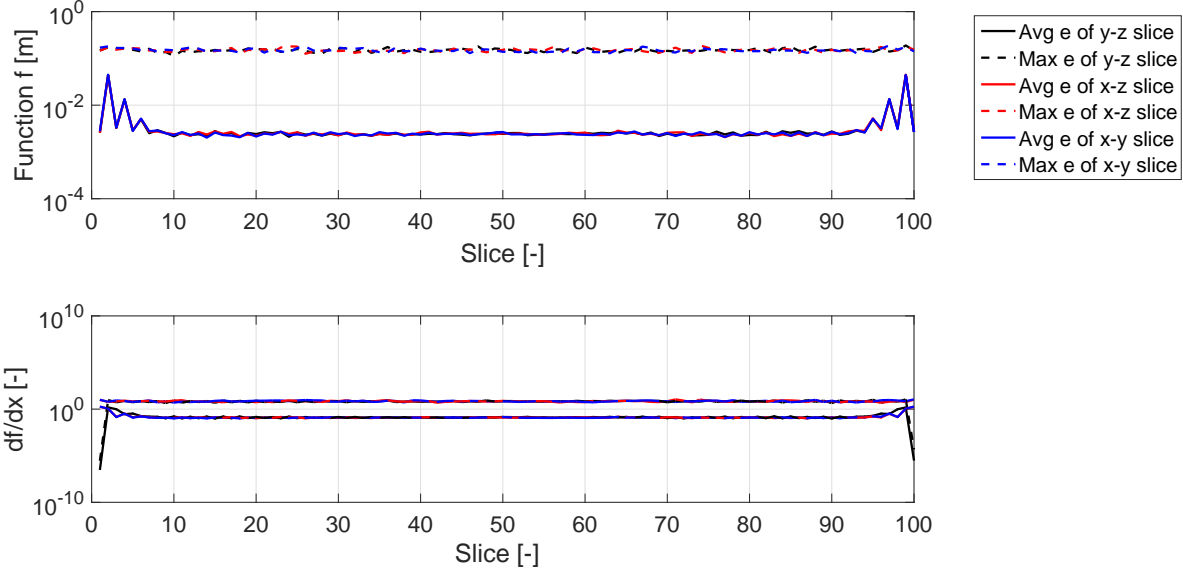


**Figure 3.10:** *The absolute difference ( $e$ ) between proposed interpolation and the original discrete TSDF visualized by a mean error and maximum error of a slice.*

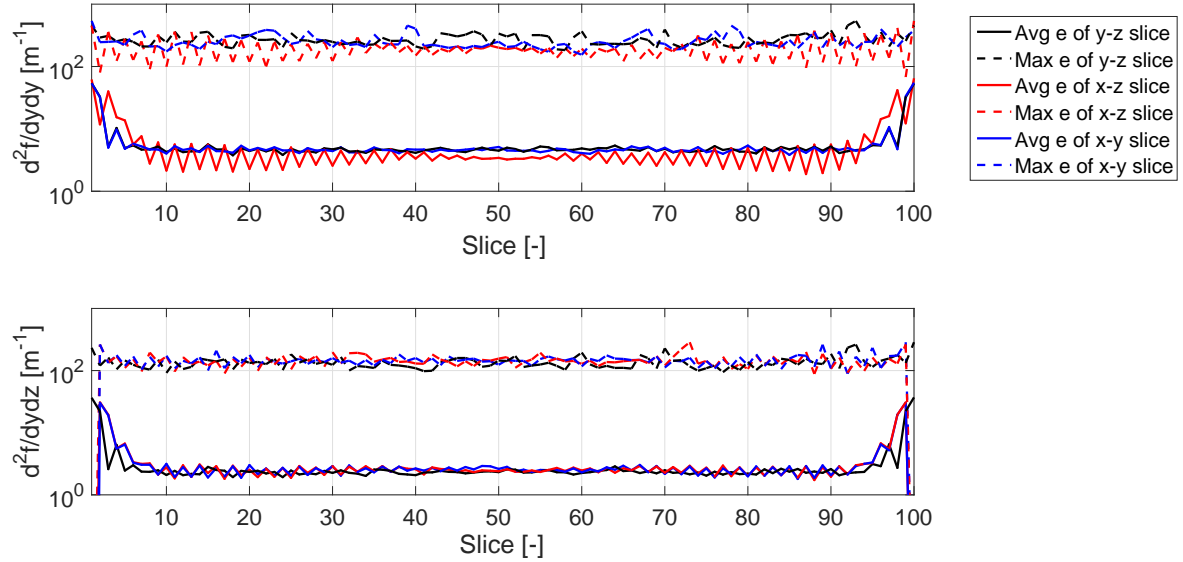
The figures above probe that both interpolation methods satisfy the pass-through requirement in multivariate interpolation. Therefore, the accuracy of the proposed interpolation method can be validated using the MATLAB interpolation. The discrete TSDF is again of the size  $50 \times 50 \times 50$ . Due to the small memory available in the MATLAB interpolation, the refined grid size is half the size of the original grid size. The MATLAB interpolation is not able to run this test with a smaller grid size.

Even though, the Hessian will be visualized by the slice plotting method. The Hessian will be too large to visualize properly. In the three-dimensional voxel space, the Hessian will be a three by three matrix. However, the Hessian has to satisfy the smoothness requirement and due to Schwarz's theorem [27], the Hessian is symmetric. Hence, the visualisation of the Hessian will solely show:  $d^2 f/dx^2$ ,  $d^2 f/dy^2$ ,  $d^2 f/dz^2$ ,  $d^2 f/dxdy$ ,  $d^2 f/dydz$  and  $d^2 f/dzdx^2$ . This section will solely show the figures of  $f$ ,  $df/dx$ ,  $d^2 f/dydz$  and  $d^2 f/dy^2$  because all other derivatives show the same behaviour. The other figures are shown in Appendix D.

Figure 3.11 show the absolute difference in the proposed interpolation and the MATLAB interpolation. The absolute difference between the two function is much larger than expected. Moreover, Figures 3.11 and 3.12 show a severe difference between derivatives of the two interpolation methods as well. These differences indicate there is a significant difference between the two function. However, as shown in the one-dimensional case, the difference between the two interpolation methods are mostly around the endpoints.

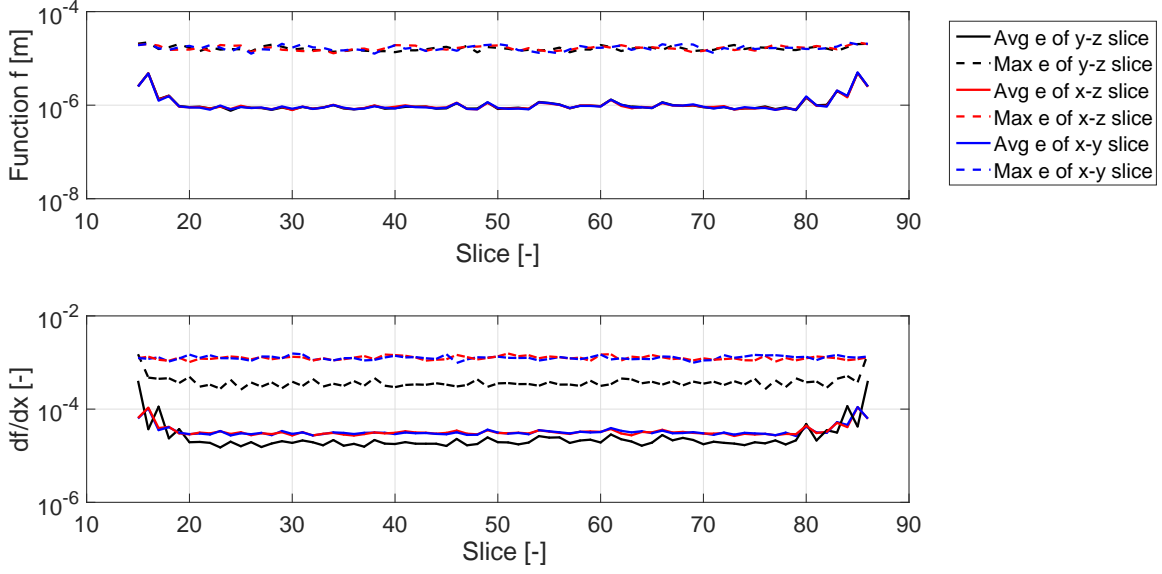


**Figure 3.11:** The absolute difference between MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice. The maximum value of function  $f$  is  $\pm 1$  which results in a floating point precision of  $1.191e-7$ . The maximum value of the derivative  $df/dx$  is  $\pm 20$  which has a floating point precision of  $1.9e-6$ .

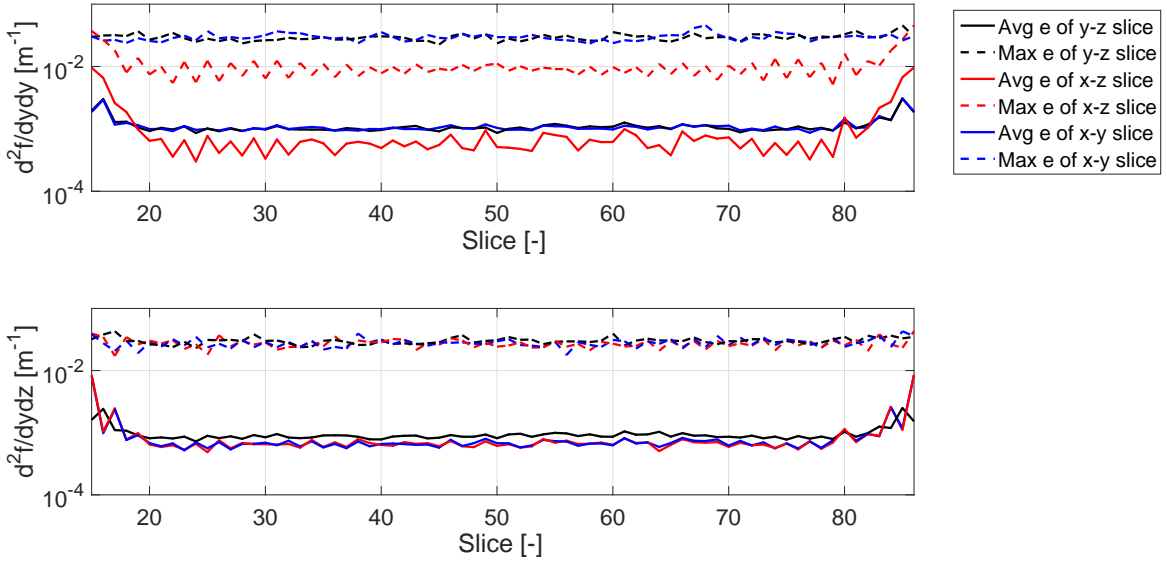


**Figure 3.12:** The absolute difference between the second derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice. The maximum value of  $d^2f/dy^2$  is  $\pm 900$  which results in a floating point precision of  $6.1e-5$ . The maximum value of the derivative  $d^2f/dydz$  is  $\pm 500$  which has a floating point precision of  $6.1e-5$ .





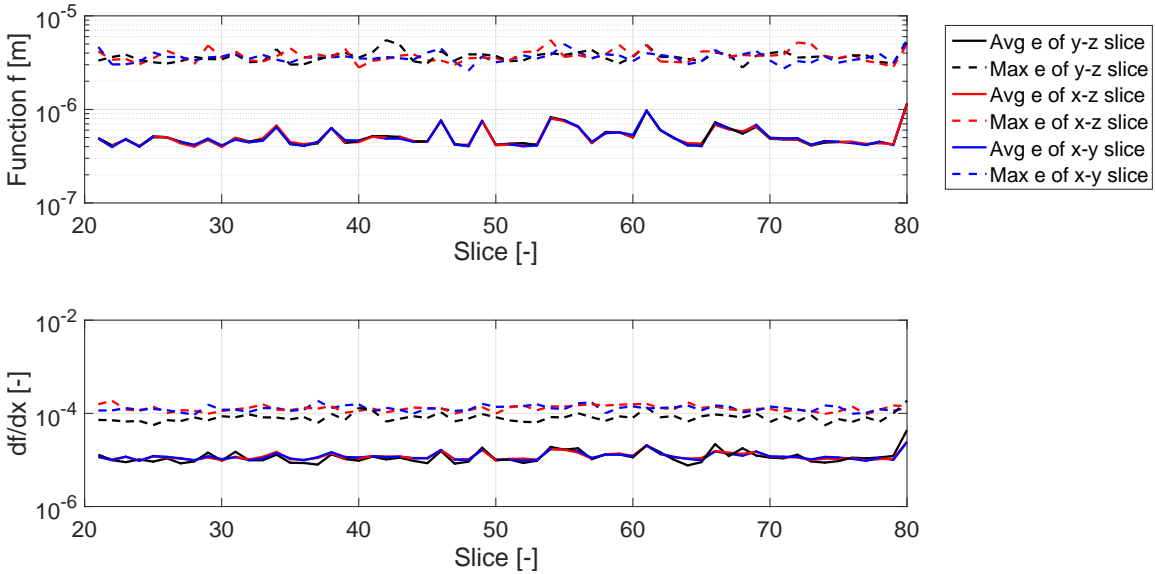
**Figure 3.13:** The absolute difference between MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice when the 14 refined points close to the boundary are neglected. The maximum value of function  $f$  is  $\pm 1$  which results in a floating point precision of  $1.191e-7$ . The maximum value of the derivative  $df/dx$  is  $\pm 20$  which has a floating point precision of  $1.9e-6$ .



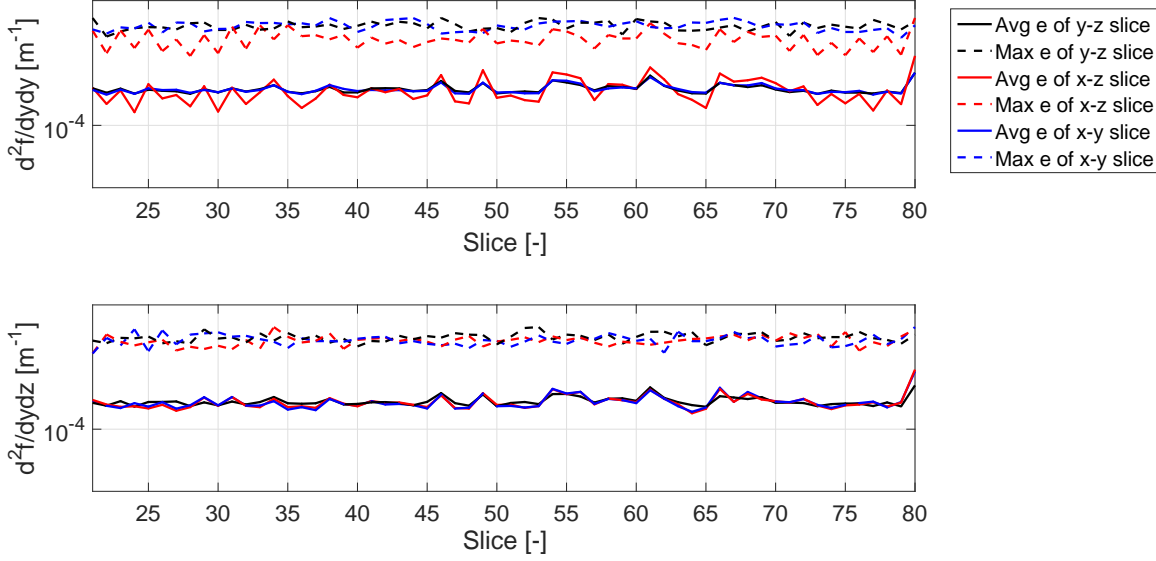
**Figure 3.14:** The absolute difference between the second derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice when the 14 refined points close to the boundary are neglected. The maximum value of  $d^2f/dy^2$  is  $\pm 900$  which results in a floating point precision of  $6.1e-5$ . The maximum value of the derivative  $d^2f/dydz$  is  $\pm 500$  which has a floating point precision of  $6.1e-5$ .

As discussed in the one-dimensional case, the interpolation methods do not approximate the boundaries exactly the same. Therefore, the multivariate case is validated by neglecting the first and last 14 refined grid points in each direction. Resulting in comparing a smaller matrix of the original matrix. As can be seen in Figures 3.13 and 3.14, the absolute differences between the two interpolation methods is smaller when the outside voxels are neglected. The difference between the two methods is still larger than the floating point error. Changing one single coefficient early on in the interpolation will result in a slightly different interpolation at the end. Hence, the boundary conditions are applied different resulting in a slightly different interpolation. The proposed multivariate interpolation is concluded to be accurate based in Figures 3.13 and 3.14.

As can be seen in the  $df/dx$  plot of Figure 3.13, the maximum difference in the  $y - z$  slice is significantly smaller than the maximum of the other slices and the average difference is increases at the boundaries. Which implies that the boundary conditions in x-direction still have a significant influence on the maximum error. The same trends can be seen in all the other plots of Figures 3.13 and 3.14. As discussed before, the error caused by the boundary conditions is neglected in this comparison. The Figures 3.13 and 3.14 show that the boundary conditions might still have influence. In Figure 3.15 and 3.16 show the same comparison of the same data while losing more points around the boundaries of the scope. As can be seen in these figures, there is no difference between the directions. Even though, the difference is negligible when the first and last 14 points at the boundary are neglected. The difference is minimal when the first and last 13 points are neglected.



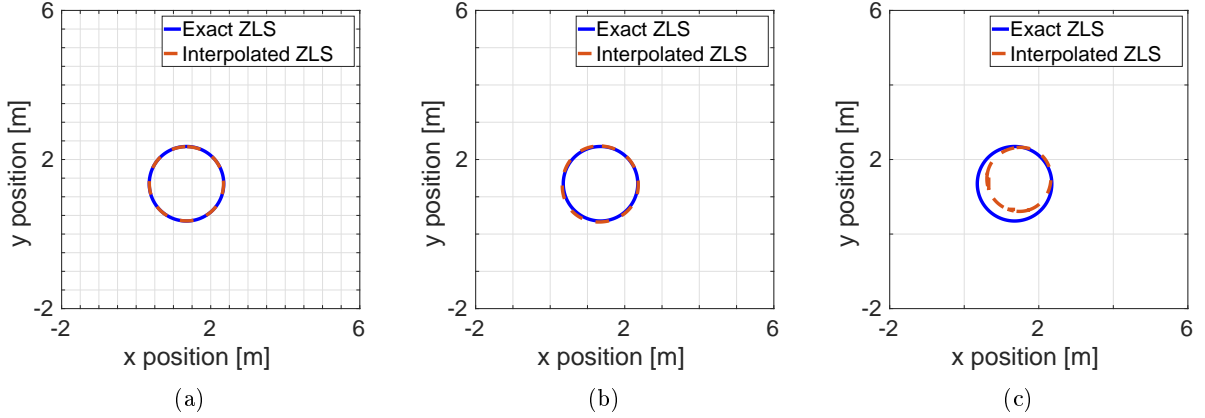
**Figure 3.15:** The absolute difference between MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice when the 20 refined points close to the boundary are neglected. The maximum value of function  $f$  is  $\pm 1$  which results in a floating point precision of  $1.191e - 7$ . The maximum value of the derivative  $df/dx$  is  $\pm 20$  which has a floating point precision of  $1.9e - 6$ .



**Figure 3.16:** The absolute difference between the second derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice when the 20 refined points close to the boundary are neglected. The maximum value of  $d^2f/dy^2$  is  $\pm 900$  which results in a floating point precision of  $6.1e-5$ . The maximum value of the derivative  $d^2f/dydz$  is  $\pm 500$  which has a floating point precision of  $6.1e-5$ .

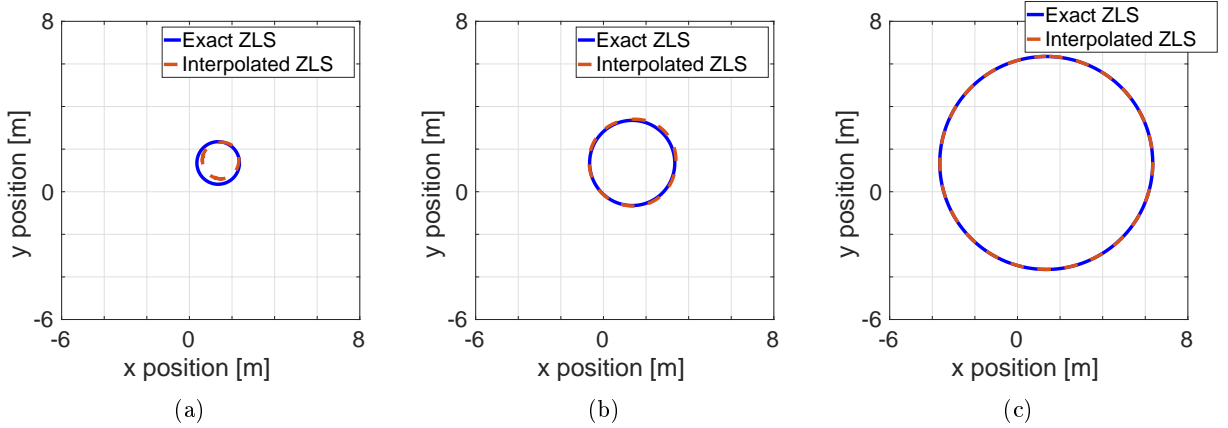
### 3.4 Zero level set interpolation error as function of the radius to grid size ratio

The proposed interpolation of Section 3.2 has been verified and validated. However, this validation does not provide any information on the performance of the proposed interpolation method in optimal motion planning. The performance of the interpolation method has to be validated in optimal motion planning. As discussed in the introduction, the collision avoidance constraint is used to find a feasible solution. The collision avoidance constraint is created by interpolating a point cloud. However, the interpolated TSDF is never exactly the same as the exact TSDF. Therefore, the zero level set of the interpolated TSDF is not exactly the same as the zero level set of the exact TSDF. This might result in a feasible trajectory based on the interpolate TSDF, however, this trajectory might collide with the real obstacle. This problem can easily be explained with a 2D obstacle avoidance constraint. In this example, the obstacle which has to be avoided is a circle with a certain radius. Figure 3.17 shows the same exact zero level set of circle obstacle and the interpolated zero level set of this obstacle. In Figure 3.17a, the obstacle avoidance constraint is sampled with a dense grid size. Figure 3.17c shows the same obstacle sampled with a very coarse grid size. Moreover, Figure 3.17b shows again the same obstacle with a grid size in between the grid sizes of the other figures. As can be seen in these figures, the smallest grid size results in the most accurate zero level set in the interpolated avoidance constraint. If the obstacle avoidance constraint of Figure 3.17c is used then it is possible that the feasible trajectory of OMP might collide with the real obstacle. Therefore, the grid size severely influences the accuracy the zero level set approximation when using the proposed interpolation method.



**Figure 3.17:** *The same obstacle sampled with different grid sizes.*

Figure 3.17 shows the accuracy of the interpolated zero level related to the grid size. However, the accuracy of the interpolation method depends on the ratio of the radius to the grid size. Figure 3.18 shows three circle obstacle with each a different radius but sampled with the same grid size. As can be seen in the figure, the zero level set of the interpolated TSDF is closer to the exact zero level set when the radius is larger. Combining the results of Figure 3.17 and Figure 3.18, the interpolated zero level set will be a more accurate approximation of the exact zero level set when the radius to grid size ratio is larger. Therefore, the accuracy of the zero level set approximation is investigated with respect to the grid size to radius ratio.



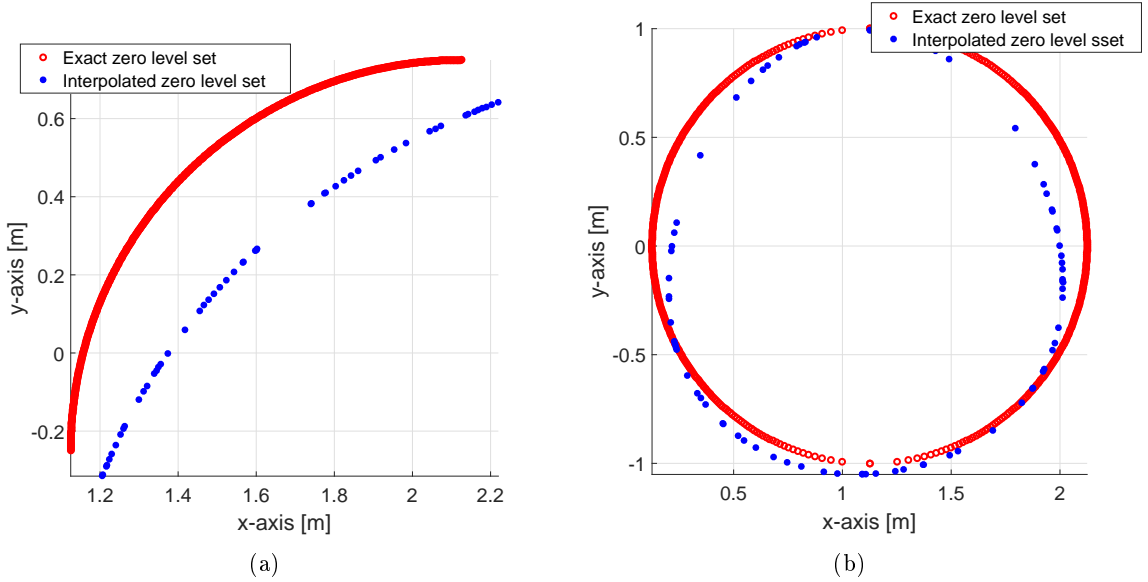
**Figure 3.18:** *The same obstacle sampled with different grid sizes.*

The error of the interpolated zero level set has to be defined in order to validate the accuracy of the interpolated zero level set. The error of the zero level set approximation is defined as

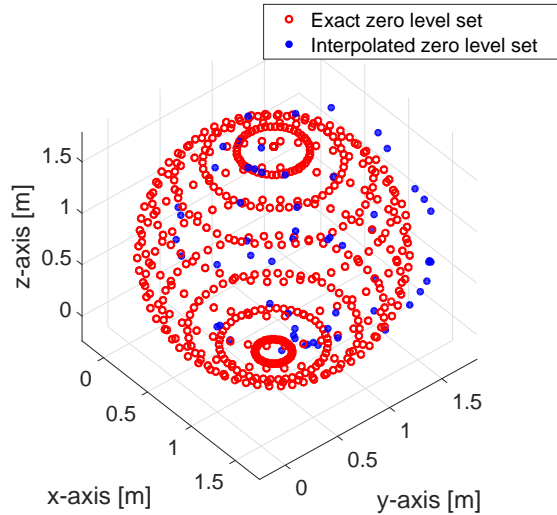
$$e = \frac{\max(d(S_e, S_i))}{h}, \quad (3.25)$$

where  $S_e$  and  $S_i$  are, respectively, the zero level sets of the exact and the interpolated TSDF,  $d(S_e, S_i)$  is the distance between the two zero level sets and  $h$  is the grid size. The zero level set of the exact TSDF can be calculated because it is created using an user-defined function. However, the zero level set of the interpolated TSDF has to be calculated because there is no given formula for the zero level set. Therefore,

the zero level set of the interpolated TSDF has to be determined. Here is chosen to use a root finding algorithm. In Section 2.4, Newton's method is discussed as a root finding algorithm. The interpolated TSDF does not satisfy the requirements for Newton's method. However, Newton's method might still be able to find enough roots to determine zero level set of the interpolated TSDF. The absolute tolerance of when a root has been found is set to  $1e^{-4}h$ . The absolute tolerance is dependent on the grid size to ensure that error caused by the absolute tolerance has a constant influence in (3.25).



**Figure 3.19:** A visualisation of the worst cases of zero-level set of (a) the edge and (b) the circle. The worst case is defined as the case with the least amount of points found on the interpolated zero level set.



**Figure 3.20:** A visualisation of the worst cases of the zero-level approximation of the sphere case. The worst case is defined as the case with the least amount of points found on the interpolated zero level set.

Figures 3.19 and 3.20 show the worst cases of the interpolated zero level sets determined by Newton's method for three different obstacles. The worst cases are defined as the case in which the least amount

of points have been found by Newton's method. As can be seen, the interpolated zero level sets have enough points to approximate the zero level set. Moreover, the roots are spread out over the entire zero level set domain. Therefore, Newton's method can be assumed to be a practical solution for calculating the interpolated zero level set.

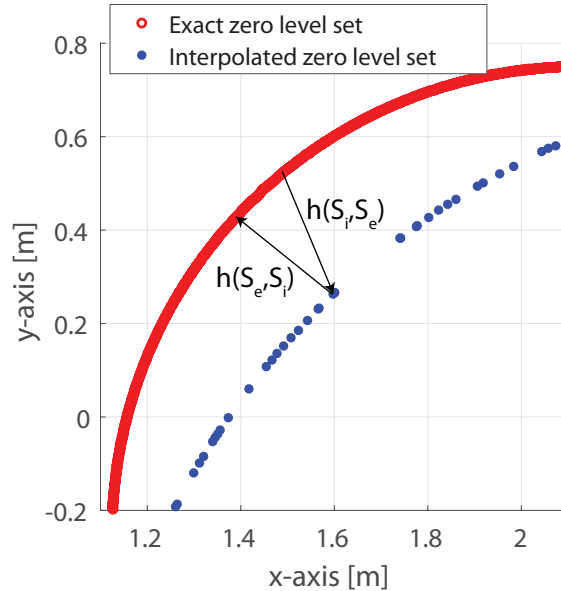
Newton's method is able to determine a discrete zero level set approximation of the interpolated zero level set. (3.25) states that the error of the zero level set approximation is calculated by the distance between the exact and the interpolated zero level set. A common method for determining the distance between two sets is the Hausdorff distance. The Hausdorff distance is defined as

$$d_h(X, Y) = \max(h(X, Y), h(Y, X)), \quad (3.26)$$

where

$$h(X, Y) = \max_{x \in X} \min_{y \in Y} \|x - y\|, \quad (3.27)$$

where  $X$  is a finite point set  $X = \{x_1, \dots, x_n\}$ ,  $Y$  is a finite point set  $Y = \{y_1, \dots, y_m\}$ ,  $\|\cdot\|$  is the Euclidean norm and  $h(X, Y)$  is the directed Hausdorff distance [28] from  $X$  to  $Y$ . The directed Hausdorff distance calculates the closest point  $y \in Y$  for a certain point  $x$  and this is done for all points  $x \in X$ . Resulting in the distance between the two sets. The maximum distance between the two sets is assumed to be the directed Hausdorff distance. The Hausdorff distance is the maximum of the directed Hausdorff distance in both directions. Figure 3.21 shows both directed Hausdorff distances for a subproblem of Figure 3.19b. As can be seen, the directed Hausdorff distance  $h(S_i, S_e)$  does not give an idea of the error of the zero level set approximation but the error caused by Newton's method. The gap between the interpolated zero level set has more influence on the directed Hausdorff distance ( $h(S_i, S_e)$ ) than the error of the zero level set approximation. Therefore, the undirected Hausdorff does not represent the distance between the exact and interpolated zero level set when the interpolated zero level set is calculated with Newton's method.



**Figure 3.21:** An example of the two directed Hausdorff distances between the exact and interpolated zero level set.

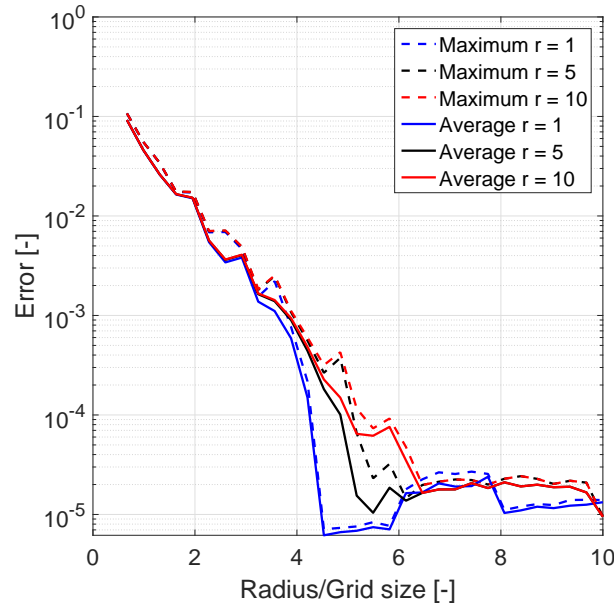
The directional Hausdorff distance  $h(S_e, S_i)$  can be used to approximate the distance between the exact and interpolated zero level set. The exact zero level set can be represented by a formula. If this continuous

representation of the exact zero level set is used then (3.27) can be written as

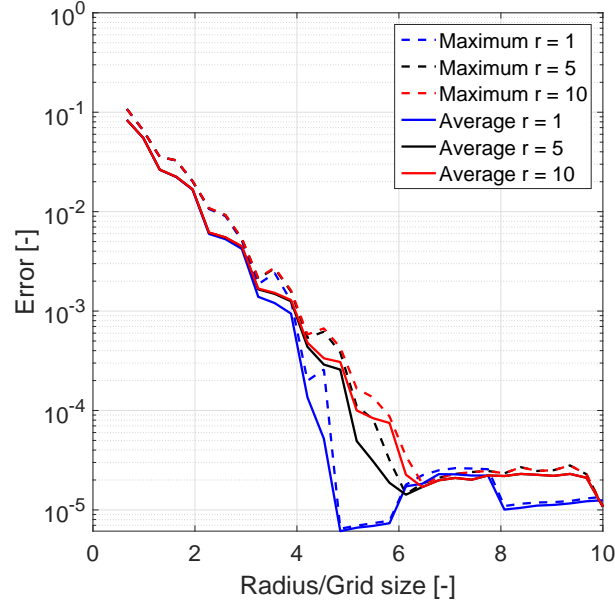
$$h(S_e, S_i) = \|r_e - r_i\|, \quad (3.28)$$

where  $r_e$  is the radius of the exact obstacle,  $r_i$  is the distance between the position on the interpolated zero level set and the center of the exact obstacle and  $\|\cdot\|$  is again the Euclidean norm.

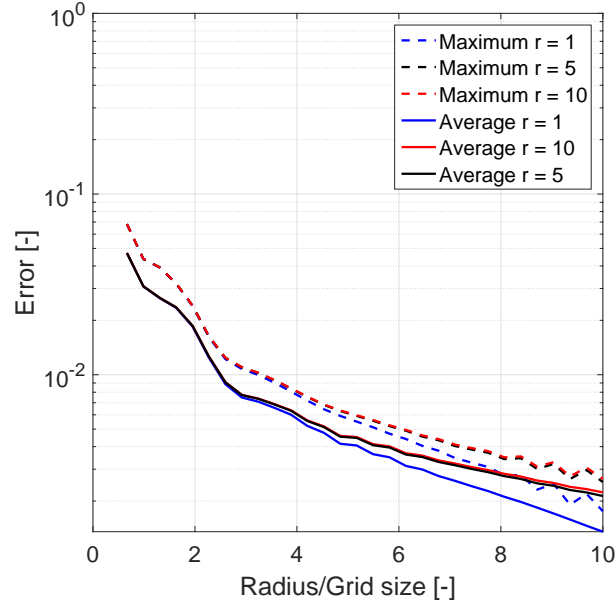
This validation is done on the three different obstacles. Figure 3.24 shows the error of interpolating an edge with respect to the radius to grid size ratio. Figure 3.23 shows the error of interpolating a cylinder with respect to the radius to grid size ratio. Figure 3.22 shows the error of interpolating a sphere with respect to the radius to grid size ratio. As be seen in Figures 3.23 and 3.22, these two cases show a similar error. The error is smaller than the error of Newton's method when the radius to grid size ratio is larger than 5. However, the edge case has a significant larger zero level set approximation error than the cylinder or sphere case. The main difference between the edge case and the other two is the fact that the SDF of the edge is not a  $C^2$  function. This lack of smoothness results in an interpolated function that overshoots the exact SDF. The overshoot is probably the cause of the higher error in the edge case. This thesis proposed to use a 3D scanner to obtain the discrete obstacle avoidance constraint. Hence, there is no guarantee that the exact SDF of the environment is  $C^2$  function. Nonetheless, the error in the edge case is less than 1% of the grid size when the radius to grid size ratio is larger than 4. The error of a 3D scanner is probably significantly larger than 1% of the grid size. Therefore, the error of the edge case is already fairly accurate when the grid size to radius ratio is larger than 4. Figures 3.24, 3.23 and 3.22 show that the maximum zero level set error is not significantly larger than the average error. However, the difference is a large enough to state that the interpolation method has not the same accuracy over the entire domain. In other words, the distance between the zero level set and the closest grid point influences the accuracy of the interpolation method.



**Figure 3.22:** The error of the interpolation method for the sphere case.



**Figure 3.23:** *The error of the interpolation method for the cylindrical case.*



**Figure 3.24:** *The error of the interpolation method for the edge case.*

With this method, the error of the interpolated zero level set with respect to the exact zero level set can be validated. The error as described above can be compensated in optimal motion planning in order to ensure no collision. When the surrounding is implemented using an inequality constraint as discussed in the book of Dreyfus [29], the error of the zero level set approximation can be implemented in the obstacle avoidance constraint in order to avoid collisions. The inequality of the obstacle avoidance constraint has to be changed to

$$c_{surrounding} - c_{safety} < 0, \quad (3.29)$$

where  $c_{surrounding}$  is the original obstacle avoidance constraint and  $c_{safety}$  is the safety margin. This



safety margin has to be chosen with respect to the smallest radius in the surrounding and the chosen grid size. The Figures 3.23, 3.22 and 3.24 give an indication of how large one should chose their safety margin.

### 3.5 Summary

This chapter discusses the proposed interpolation method for optimal motion planning. First, this chapter compares the calculation time between the digital filter method and another interpolation method. This comparison shows that the digital filter method is indeed the fastest interpolation method in the given cases. Second, the CUDA implementation of proposed interpolation method has been discussed. Third, the proposed method has shown that it verifies the pass-through and smoothness requirement in the given scenarios, as it should according to the literature. Fourth, the interpolation method has been validated by comparing the proposed interpolation method with a default MATLAB interpolation. The validation showed that the proposed interpolation method and the MATLAB interpolation did not result in the exactly the same interpolation. The difference between the interpolation method was solely significant around the boundary points. Therefore, it is assumed that the boundary conditions are applied differently, resulting in a different interpolation. Fifth, the accuracy of the zero level set approximation of the proposed interpolation method has been validated for a given radius to grid size ratio. This validation is done for three different obstacles: an edge, a cylinder and a sphere. The validation has shown that the edge case had the largest error over the entire domain. However, this error is already smaller than 1% of the grid size when the radius to grid size ratio is larger than 4.

## 4 Numerical experiments

Chapter 3 proposes an interpolation method for creating an obstacle avoidance constraint from a discrete TSDF. This interpolation method has been validated and verified on the pass-through and smooth requirement. However, the interpolation method should be as fast as possible. This requirement has not been verified yet. This section starts verifying the calculation time of the proposed method with respect to two other interpolation methods. The first method which is used to validate the calculation time of the proposed method is a CPU version of the proposed method. The second method is the MATLAB interpolation which has been used to validate the correctness of the proposed interpolation method in Chapter 3. Thereafter, the proposed interpolation method is used to interpolate discrete TSDF in order to use the discrete TSDF representation of an obstacle in optimal motion planning. First, an artificial space is created as a discrete TSDF. Second, the Kinect camera and KinectFusion is used to create the discrete TSDF.

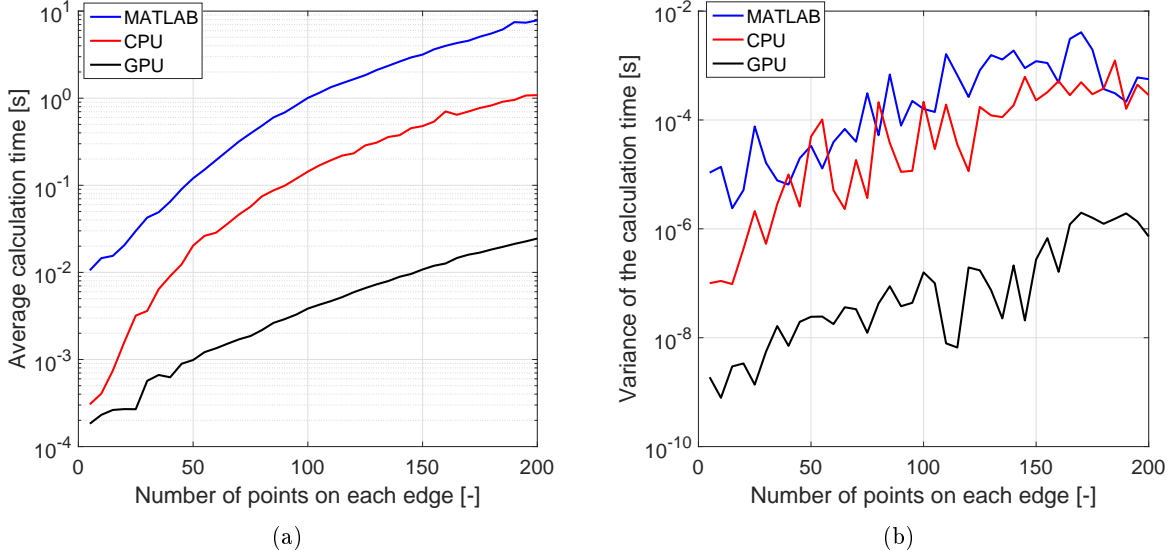
### 4.1 Calculation time of the different interpolation methods

A cubic b-spline interpolation consists of two different parts. The first part of b-spline interpolation is calculating the coefficients. This part will be referred to as the coefficients calculation. The second part of cubic b-spline interpolation is determining the values of function and its first and second derivatives based on the coefficients for a number of desired positions. This part is referred to as the output calculation. The calculation times of both aspects of the cubic b-spline interpolations will be compared.

This section compares the calculation times of the proposed GPGPU interpolation method of Section 3.2 with two different methods: a CPU/MATLAB implementation of the proposed interpolation method and a standard MATLAB interpolation. The standard MATLAB interpolation has also been used in Section 3.3 to validate the accuracy of the proposed interpolation method. The proposed GPGPU interpolation method of Chapter 3 will be referred to as the GPU interpolation in this section. The CPU version of the proposed interpolation method will be referred to as the CPU interpolation. The CPU interpolation uses solely one core instead of all four cores available. Therefore, the CPU interpolation could be at most four times as fast. The calculation time of these three interpolation methods will be compared in this section. The computer hardware and software which have been used in these time comparison is given in Tables 3.1 and 3.2.

The calculation time of the coefficients calculation is compared based on interpolating a discrete TSDF which the same number of point in each direction and the same grid size in each direction. A TSDF which has the same number of points and the same grid size in each direction will be referred to as a box TSDF. This box discrete TSDF is created using a 3D white noise signal. To validate the consistency of all three interpolation method, the interpolation methods have to interpolate ten different discrete TSDF of the same size. Figures 4.1 show the calculation time of the three different interpolation methods with respect to the number of point on each edge of the box TSDF. As can be seen in Figure 4.1(a), the GPU interpolation is significantly faster than the other two interpolation methods. Moreover, the difference between the GPU interpolation and the other interpolation methods increases when the number of points on the axis increase. Figure 4.1(a) also shows that the CPU interpolation is also faster than the MATLAB interpolation. Figure 4.1(b) shows that the variance of the calculation times is significantly smaller than the average calculation times for all three interpolation methods. Therefore, the calculation time of the interpolation is assumed to be constant for a given number of points on the edges. The results of Figure 4.1(b) are not as expected with respect to Figure 4.1(a). The curvature of the GPU implementation is smooth when the number of points on each edge is larger than 50. When the number of points on each edge is lower than 50, the curvature is significantly less smooth than above 50. Consequently, a higher

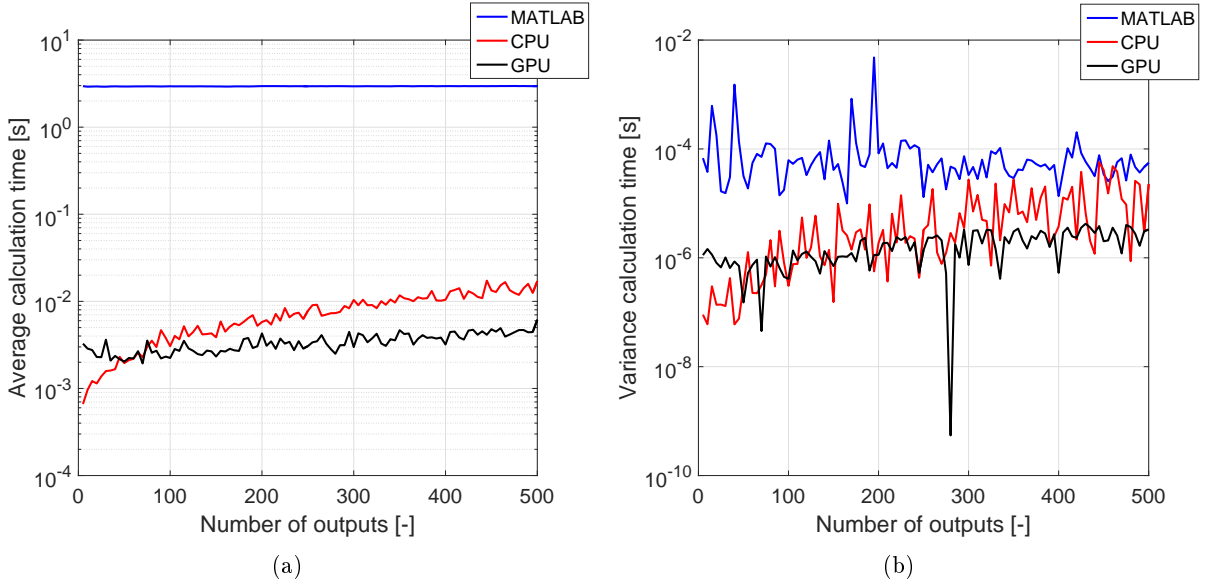
variance in the GPU implementation is expected in Figure 4.1(b) but is not available. The curvature of the GPU implementation might be less smooth due to the chosen number of threads and blocks. The influence of this choice is assumed to be not significant for this thesis. Though, the influence of this choice might be relevant in case of less number of points on each edge. In case of this thesis, the voxel spaces has more points on each edge than 50.



**Figure 4.1:** Comparing the calculation times of the coefficients of the GPU interpolation, the CPU interpolation and the MATLAB interpolation. In part (a) shows the average calculation time of the three interpolation methods. In part (b) shows the variance in calculation time of the three interpolation methods.

The second part of a cubic b-spline interpolation is to calculate the interpolated TSDF values and the derivatives at desired positions. The discrete TSDF is pre-calculated and has a fixed size of  $50 \times 50 \times 50$ . In this comparison, the calculation time of the coefficients is not taken into account. Therefore, this validation focusses solely on the calculation time from the coefficients to the desired output values.

The desired positions are chosen randomly within the space described by the discrete TSDF. The interpolation method has to calculate the outputs for 10 different discrete TSDFs in order to check the variance of the interpolation methods. Figures 4.2 show the calculation times of the output calculation of all three interpolation method. As can be seen, the GPU interpolation is faster than the other methods when the number of outputs is set higher than 50. The CPU interpolation is faster in calculating the outputs when the number of outputs is lower than 50.



**Figure 4.2:** Comparing the calculation times of the coefficients of the GPU interpolation, the CPU interpolation and the MATLAB interpolation. In part (a) shows the average calculation time of the three interpolation methods. In part (b) shows the variance in calculation time of the three interpolation methods.

As can be seen in Figures 4.1, the GPU interpolation method calculates the 3D b-spline coefficients significantly faster than the CPU interpolation and the MATLAB interpolation. Even when the CPU interpolation uses four cores in parallel then the GPU interpolation still would be faster. Moreover, the proposed GPU interpolation is also faster in calculating the output when the number of outputs is larger than 60. Moreover, the calculation time increases significantly less than the CPU interpolation when the number of outputs increases.

Optimal motion planning describes the trajectory with a discrete version of the trajectory. Therefore, the number of outputs is desired to be higher than 50 in most cases. However, the CPU interpolation could run in parallel. Therefore, the CPU interpolation could be up to four times faster than the current algorithm. As a result of multi-core CPU implementations, the CPU interpolation could be faster when the number of outputs is less than 200.

Concluding, the coefficients should always be calculated using the proposed interpolation method when calculation time should be as low as possible. Moreover, the output of the interpolation should be calculated with the proposed interpolation method when the number of outputs is higher than 50 and when only one CPU core is used. In case of the current CPU, the parallel CPU interpolation could be up to four times faster than the current version. Therefore, it might be better to use a CPU version for the second part of the cubic b-spline interpolation. However, if the second part of the interpolation is done on the GPU, the coefficients have to be stored in the CPU memory from the GPU memory. Storing the coefficients in the CPU memory will result in some time loss. The combination of calculating the coefficients with the GPU and calculating the output using a CPU is not really interesting. Therefore, the GPGPU version of the interpolation method is proposed to be used in an optimal motion planning algorithm.

## 4.2 Optimal motion planning with collision constraints from point cloud data

The goal of this thesis is to propose an interpolation method which is able to interpolate a discrete function for optimal motion planning. Section 3.2 proposes an interpolation method which satisfies all the requirements. This section validates whether this interpolation method is indeed able to create an obstacle avoidance constraint for optimal motion planning. This section firstly validates whether the proposed interpolation method can be used on synthetic data. Secondly, the proposed interpolation method is combined with the KinectFusion Framework in order to calculate an obstacle avoidance constraint from an environment.

### 4.2.1 OMP based on interpolated TSDF

This section evaluates whether the proposed interpolation method can be used for creating an obstacle avoidance constraints. The obstacle avoidance constraint will be created artificially and interpolated using the proposed interpolation method. However, an optimal motion planning problem is needed to validate the obstacle avoidance constraint. This section starts with explaining the optimal motion planning algorithm which will be used to validate the interpolated TSDF. In Chapter 1, optimal motion planning is described calculating a trajectory while minimizing a certain cost. The optimal control problem which is to be solved in this thesis is given by

$$\begin{aligned} \min_{x(\cdot), u(\cdot)} \quad & \int_0^T l(x(\tau), u(\tau), \tau) d\tau + m(x(T)), \\ \text{s.t.} \quad & \dot{x} = g(x(t), u(t), t), \quad x(0) = x_0, \\ & c_j(x(t), u(t), t) \leq 0, \quad t \in [0, T], \quad j \in [1, \dots, k], \end{aligned} \quad (4.1)$$

where  $x(t)$  is the state of the system,  $u(t)$  is the control input,  $l(x(\tau), u(\tau), \tau)$  is the cost,  $m(x(T))$  is the terminal cost and  $c_j(x(t), u(t), t)$  describe the constraints of the system. Häusler et al. propose to approximate the optimized trajectory functional with an approximated log-barrier function [30]. The log-barrier approximation of (4.1) results in the following optimized trajectory functional

$$\begin{aligned} \min_{x(\cdot), u(\cdot)} \quad & \int_0^T l(x(\tau), u(\tau), \tau) - \epsilon \sum_j \log(-c_j(x(\tau), u(\tau), \tau)) d\tau + m(x(T)), \\ \text{s.t.} \quad & \dot{x} = g(x(t), u(t), t), \quad x(0) = x_0, \end{aligned} \quad (4.2)$$

where  $\epsilon$  is a constant which is bigger than zero. When  $\epsilon$  goes to zero, the minimum of the approximation will go to the minimum of the optimized trajectory functional. A small size of  $\epsilon$  results in an optimal control problem which is difficult in term of computation. Consequently, the optimal control problem is first solved for a large  $\epsilon$ . Thereafter, the  $\epsilon$  is decreased and this new optimal control problem uses the minimizer of the optimal control with the previous  $\epsilon$  as an initial guess. Interior point optimization is commonly referred to as continuation. 4.2 is difficult to minimize due to the continuous representation of all the functions. In practice, the optimal control problem of (4.2) is solved using a numerical solver. A approach to solve (4.2) is to discretize (4.2) in order to use a numerical solver. A discretized version of (4.2) is given by

$$\begin{aligned} \min_{x(\cdot), u(\cdot)} \quad & \sum_{k=0}^{N-1} \left[ l_k(x_k, u_k) - \epsilon \sum_j \log(-c_j(x_k, u_k)) \right] + m_N(x_N), \\ \text{s.t.} \quad & x_{k+1} = g_k(x_k, u_k), \quad x(0) = x_0. \end{aligned} \quad (4.3)$$

Damen et al. [31] have compared three different solvers for the discrete optimal control problem given in (4.3). Therefore, three solvers have an implementation available at Eindhoven University of Technology (TU/e). For this thesis, the discrete PROjection Operator-based Newton method for Trajectory Optimization (dPRONTO) will be used to solve the optimal control problem. The dPRONTO algorithm has been chosen because it is an in-house solution. Hence, the knowledge and an implementation of the dPRONTO algorithm are available at the TU/e. A more in-depth explanation of the PRONTO algorithm can be found in [32].

The dPRONTO implementation of Damen et al. is able to handle continuous dynamics as an input while solving the discrete Functional of equation (4.3). The dPRONTO algorithm rewrites the optimal control problem of (4.3) into a cost function

$$\min_{\xi \in U} g(\xi), \quad (4.4)$$

where  $\xi$  is a trajectory which is determined based on  $x$  and  $u$  and  $g(\xi)$  the projected cost function. As discussed in the Chapter 1, it is preferred to solve the minimum using Newton-Like minimization methods. This refers to how the projected cost function is minimized in the dPRONTO algorithm. According to Section 2.4, linear interpolation might not be able to solve the optimal control problem.

To demonstrate the possibility of using the GPGPU interpolation method of Chapter 3 in an optimal control problem like (4.2), the GPGPU interpolation method is implemented in the dPRONTO algorithm of the TU/e. This thesis focusses on optimal motion planning for a point mass. The dynamics of the optimal motion problem are defined by Newton's second law. The state vector is given by

$$x = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \\ x_6 \end{bmatrix}, \quad (4.5)$$

with

$$\dot{x}_1 = x_2, \quad \dot{x}_3 = x_4, \quad \dot{x}_5 = x_6, \quad (4.6)$$

where  $(x_1, x_3, x_5)$  is the position in a Cartesian coordinate system and  $(x_2, x_4, x_6)$  are the velocities of the point mass. These dynamics can be given in the following matrix notation

$$\dot{x} = Ax + Bu, \quad (4.7)$$

with,

$$A = \begin{bmatrix} 0, 1, 0, 0, 0, 0 \\ 0, 0, 0, 0, 0, 0 \\ 0, 0, 1, 0, 0, 0 \\ 0, 0, 0, 0, 0, 0 \\ 0, 0, 0, 0, 1, 0 \\ 0, 0, 0, 0, 0, 0 \end{bmatrix}, \quad (4.8)$$

$$B = \begin{bmatrix} 0, 0, 0 \\ \frac{1}{M}, 0, 0 \\ 0, 0, 0 \\ 0, \frac{1}{M}, 0 \\ 0, 0, 0 \\ 0, 0, \frac{1}{M} \end{bmatrix}, \quad (4.9)$$

where  $M$  is the mass of the object which is moved,  $x$  is the position in space and speed in x, y, z-direction and  $u$  is the force in x, y, z-direction. The dPRONTO algorithm needs certain inputs before calculation.  $Q_n$  penalizes the error between the desired end position and the end point of the proposed trajectory. In other words, the variable  $Q$  is used as a soft constraint for function  $m_N(x_N)$  of equation (4.3). In this case, the terminal constraint is given by

$$m_N(x_N) = \frac{1}{2}(x_N - x_{des})^T R(x_N - x_{des}), \quad (4.10)$$

where,  $x_{des}$  is the desired end position and  $x_{des}^T$  is the transposed of the input  $x_{des}$ . The function  $l_k$  defines the cost of the trajectory. The cost of the trajectory is defined by the input of the system, resulting in the following stage cost

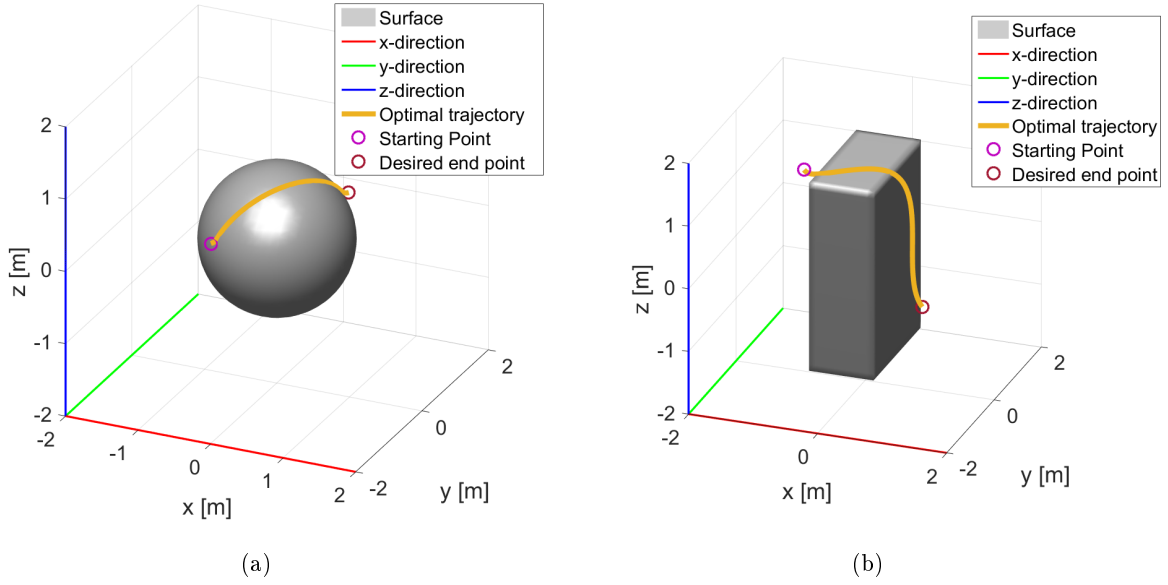
$$l_k = u_k^T R u_k, \quad k \in [1 \ N] \quad (4.11)$$

where  $u_k^T$  is the transposed of the input  $u_k$ . This stage cost should result in a trajectory where the input is minimized. The minimum is assumed to be found when the absolute descent is smaller than a certain threshold ( $AbsTol$ ). The trajectory is initialized by the starting position ( $x_0$ ) to the desired end position ( $x_{des}$ ) with a constant speed. Moreover, the trajectory consists out of  $N + 1$  number of points. In this thesis, the trajectory has to move from point a to b in a fixed time ( $t_f$ ).  $\alpha$  is the Armijo's rule constant and  $\beta$  is the backtracking constant, as discussed in Section 2.4. The inequality constraints are added in (4.3) using a barrier function. However, this barrier function has another parameter besides  $\epsilon$  and that parameter is referred to as the relaxation parameter ( $\delta$ ). The influence of the relaxation parameter can not be seen in (4.3) or (4.2). The relaxation parameter is added because (4.3) and (4.2) can not be computed when a constraint  $c_i$  is positive. Therefore, the log-barrier function is replaced with a quadratic polynomial when the constraint is smaller than  $-\delta$ . Resulting in the following barrier function

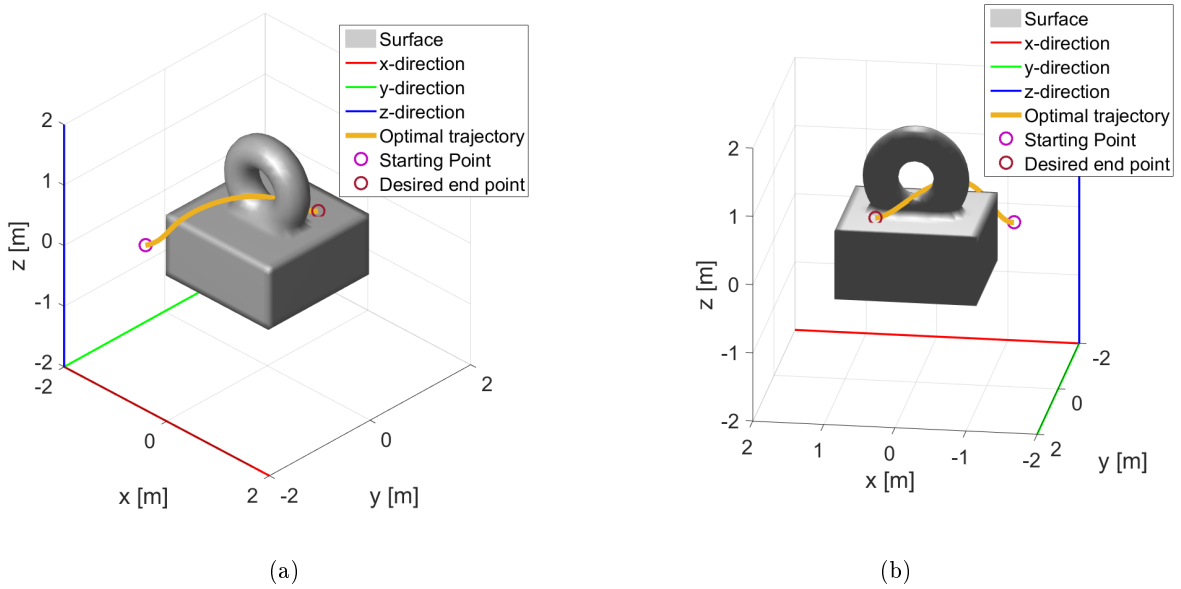
$$\beta(z) = \begin{cases} -\log(z) & , z > \delta \\ -\log\left(\delta + \frac{1}{2}\left[\left(\frac{z-2\delta}{\delta}\right)^2 - 1\right]\right) & , z \leq \delta. \end{cases} \quad (4.12)$$

The quadratic polynomial does not interfere with the smoothness requirement because the function  $\beta$  is a  $C^2$  function. The values of these dPRONTO parameters are given in the Appendix F.

The obstacle avoidance constraint should be added to the optimal motion problem. In this section, the obstacle avoidance constraint is calculated using the proposed interpolation method of Section 3.2 and an artificially created discrete TSDF. The discrete TSDF of the simulation obstacles are created using the SDF given by Quilez [33]. Moreover, the SDFs are truncated with a truncation threshold of  $1[m]$ . The optimal trajectories around three different obstacles are presented in this section. The first obstacle is a sphere because the SDF of a sphere is a smooth function with the exception of the center of the sphere. The second obstacle is a box because the SDF of a box is not a  $C^2$  function itself. A box and a torus shape are combined based on the smooth transition mentioned by Quilez [33]. The third obstacle is chosen to be a bit more complex shape than the other two. First, the proposed interpolation method of Section 3.2 will be used to determine the obstacle avoidance constraint. Later, a linear interpolation combined with central approximation will be used to compare the proposed interpolation method with the least computational method. Figure 4.3 shows the optimal trajectory around the sphere and the box calculated using the proposed interpolation method. Figure 4.4 shows the optimal trajectory around the box and torus combination calculated with the proposed interpolation method.



**Figure 4.3:** The optimal trajectory around a spherical obstacle (a) and around a box obstacle (b).

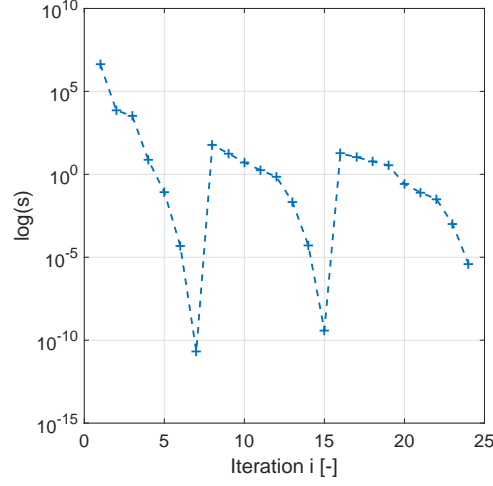


**Figure 4.4:** The optimal trajectory around an obstacle which is a box and a torus combined. Two different views of the same trajectory are shown.

As can be seen in Figures 4.3 and 4.4, the optimal motion planning algorithm finds a feasible solution. However, these Figures do not show that the trajectory is a local minimizer of (4.2). The size Newtons step gives a proper indication of the size of the error, as discussed in Section 2.4. Therefore, the trend of the size of the Newtons step gives an indication of the trend of the size of the error. Figure 4.5 shows an example of the size of Newton's step of the dPRONTO algorithm. As can be seen in Figure 4.5, the descent of the dPRONTO algorithm shows quadratic convergence. The steps at iteration 6 and 15 are the consequence of the continuation. Decreasing  $\epsilon$  creates a different optimal control problem. Therefore,

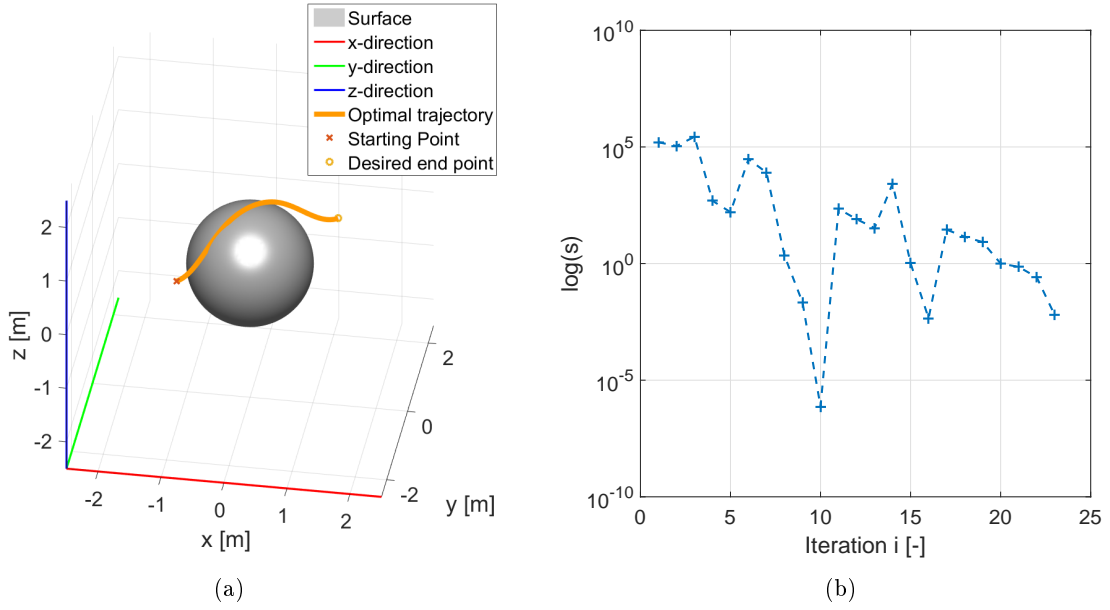


the error increases when  $\epsilon$  is decreased. Therefore, it can be concluded that the trajectories found in optimal motion planning are local minimizers of the optimal control problem (4.2).



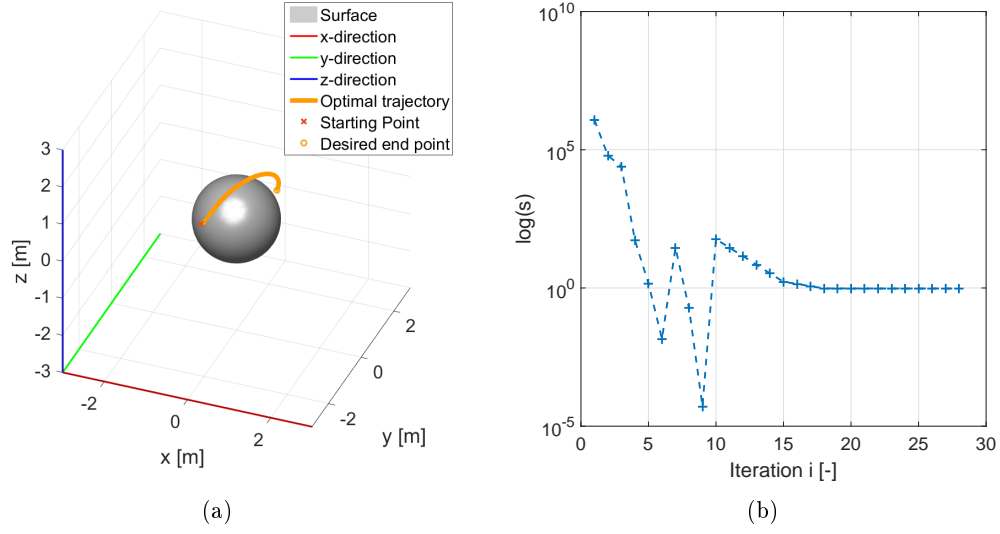
**Figure 4.5:** An example of the descent convergence of the dPRONTO algorithm.

In Section 2.4, optimization using linear interpolation is discussed as well. From literature, it was concluded that linear interpolation with a central approximation for the derivatives might work. The maximum error caused by the central approximation has to be significantly smaller than the error of Newton's method. The obstacle avoidance constraint which is determined with linear interpolation with central approximation will be referred to as the linear interpolated obstacle avoidance constraint. The linear interpolated obstacle avoidance constraint of the optimal control problem is transformed into the projected cost function. Hence, the influence of the error of the linear interpolated obstacle avoidance constraint in the minimization is difficult to derive. Therefore, the possibility of using linear interpolated obstacle avoidance constraint in optimal motion planning will be validated using simulations. Figure 4.6 shows the trajectory of an optimal motion planning based on a linear interpolated obstacle avoidance constraint. As can be seen from Figures 4.6 and 4.3, the linear interpolation method uses almost the same amount of iterations to minimize the optimal control problem. Figure 4.6 is a bit misleading because the grid size is twice as dense in the linear interpolated case in order to solve the optimal control problem. Moreover, the terminal condition in the cubic case is  $1e^{-5}$  and in the linear interpolated case the terminal condition is set to  $1e^{-2}$ . The terminal condition is set higher in the linear interpolated case because it was not able to find a trajectory with a lower Newton's step. The trajectory of Figure 4.6 is feasible for the linear interpolated obstacle avoidance constraint, however, the trajectory collides with the actual obstacle. This solely indicates that linear interpolation is not as accurate as the cubic interpolation. Figure 4.6 shows a few increases in the Newton's step at iteration 10 and 16. These increases in Newton's step are the result of continuation. Concluding, Figure 4.6 does not prove linear interpolation with central difference approximation is slower or less reliable than cubic interpolation.

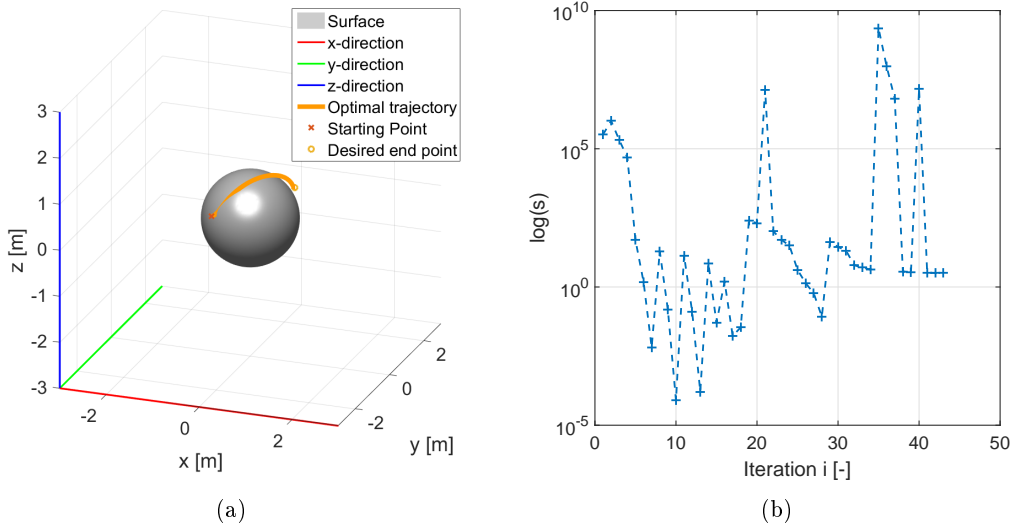


**Figure 4.6:** An example of a local minimizer of the optimal motion planning based on a linear interpolated obstacle avoidance constraint. In part (a) the trajectory which is the minimizer. In part (b) the Newton's step versus the iteration number.

Figure 4.7 shows the results of a different optimal motion planning. In Figure 4.7, the same obstacle is used as in Figure 4.6, however, the trajectory has a different start and end position. As can be seen, the size of the Newton's step does not show a local quadratic convergence rate. Moreover, the Newton's step seems to stagnate after 17 iterations. According to the theory of Section 2.4, this could be the result of the error in the central difference approximation and linear interpolation. The optimal motion planning is not able to determine an optimal trajectory for this particular start and end position when the grid size is set to be twice as dense. Figure 4.8 shows the results of the optimal motion planning when the grid size is twice as dense as in Figure 4.7. As can be seen in Figure 4.8, the trajectory is much shorter than the trajectory of Figure 4.7. Figure 4.8(b) shows that it has found multiple local minima. However, the final trajectory is not a local minimum because the size of the Newton's step is larger than the absolute tolerance. Moreover, the trajectory of Figure 4.8 collides with the obstacle. Furthermore, optimal motion planning based on a linear interpolated obstacle avoidance constraint of a box is not possible. Shown in this Section, an optimal motion planning algorithm can already fail when a simple obstacle is represented by a linear interpolated obstacle avoidance constraint. Hence, linear interpolated obstacle avoidance constraints are not desired in autonomous optimal motion planning.



**Figure 4.7:** An example of a local minimizer of the optimal motion planning based on a linear interpolated obstacle avoidance constraint. In part (a) the trajectory which is the minimizer. In part (b) the Newton's step versus the iteration number.

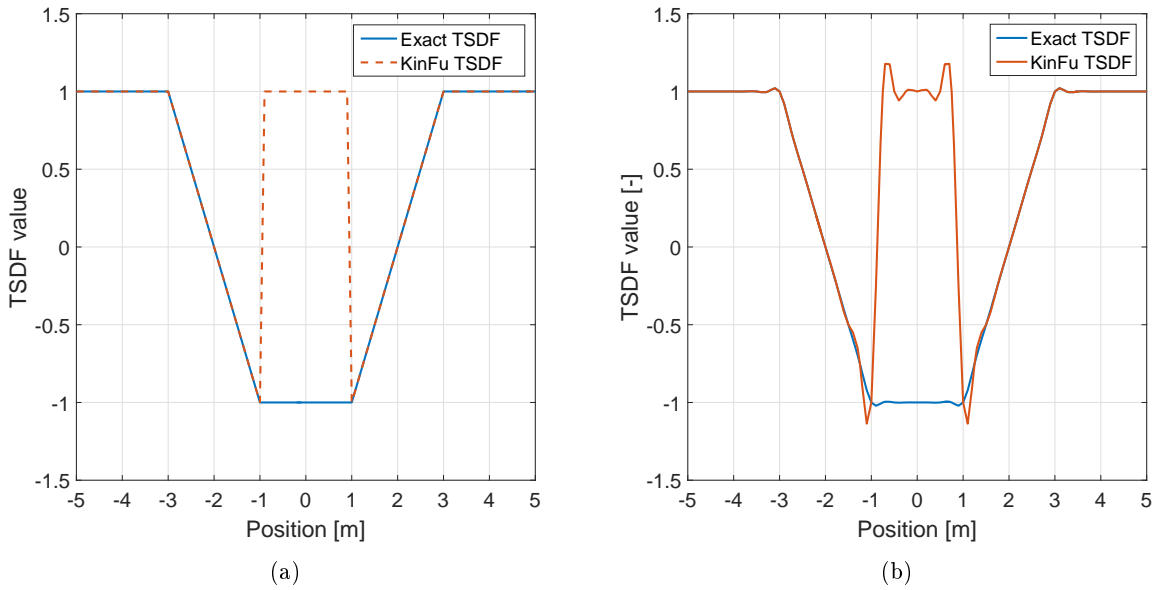


**Figure 4.8:** An example of a local minimizer of the optimal motion planning based on a linear interpolated obstacle avoidance constraint. In part (a) the trajectory which is the minimizer. In part (b) the Newton's step versus the iteration number.

#### 4.2.2 OMP based on KinectFusion data

In Section 4.2.1, the GPGPU accelerated interpolation method is used to interpolate the discrete TSDF to create a proper obstacle avoidance constraint. Discussed in Section 4.2.1, a linear interpolated obstacle avoidance constraint can cause a problem in an optimal motion planning algorithm. One of the goals of this thesis was to demonstrate that KinectFusion and a moving depth camera can be used to create the discrete TSDF. Whether the KinectFusion data can be used to create a discrete TSDF which can be used in optimal motion planning will be discussed in this section.

The KinectFusion framework is available as open-source software under the name KinFu [9]. However, the implementation of the KinectFusion framework is not exactly the same as the KinectFusion framework. In KinFu, each voxel is initialized as a voxel which is far from a surface. In other words, each voxel is initialized with the value 1. KinFu only stores the TSDF values in a voxel when the TSDF value is between  $]-1, 1[$  or when the global voxel already has a value which is not 1. Therefore, the 3D grid has only voxels with the value 1 unless a measurement has shown different. Figure 4.9 shows the exact TSDF of (2.5) and the reduced TSDF which is used in KinFu of a 1D obstacle. As can be seen in Figure 4.9, the reduced TSDF is not the same as the exact TSDF inside the obstacle. This error occurs when the truncation threshold is smaller than twice the thickness of an obstacle. In order to ensure no hollow obstacles, the truncation threshold should be larger than twice the thickness of the thickest obstacle in the environment.

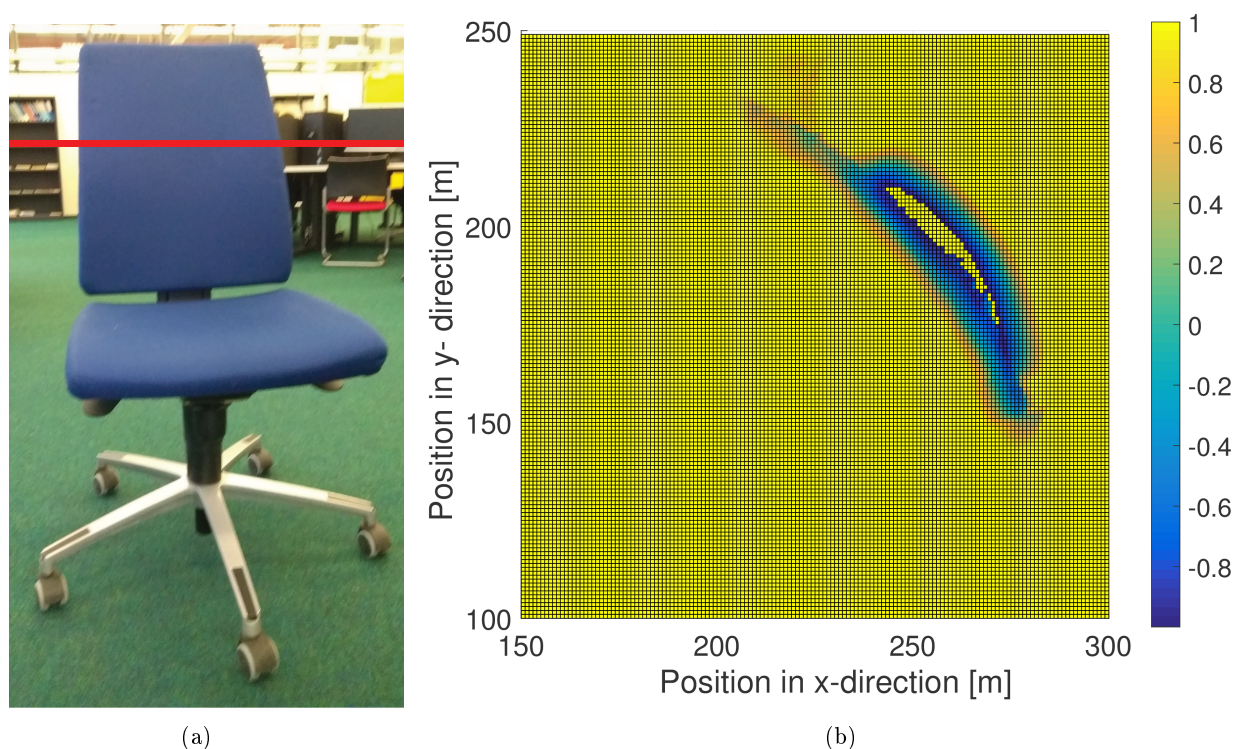


**Figure 4.9:** A 1D example of the exact TSDF and the KinFu TSDF with a truncation threshold of 1[m]. Part (a) shows the TSDF functions. Part (b) shows a interpolated version of the same TSDF functions.

The KinFu algorithm has a default truncation threshold of 0.03 [m]. As illustrated in Figure 4.9, this truncation threshold will result in an obstacle which is hollow inside when the obstacle is thicker than the truncation threshold. The hollowness of the obstacles might cause issues in the optimal motion planning. When the optimal motion planning is initialized through an obstacle which is hollow then the optimal motion planning might not be able to solve the optimal control problem. The points which are inside an obstacle are assumed to be negative. However, the points inside a hollow obstacle are positive. Therefore, the optimal motion planning algorithm cannot know that these points are inside an obstacle. Hence, the optimal motion planning algorithm will not push these points to outside the obstacle. Moreover, the hollowness is an issue for the smoothness requirement of Chapter 3. To satisfy the smoothness requirement, the discrete TSDF is interpolated in order to get a  $C^2$  function. The discontinuity of the discrete TSDF results in significant overshoot in the interpolation as shown in Figure 4.9(b).

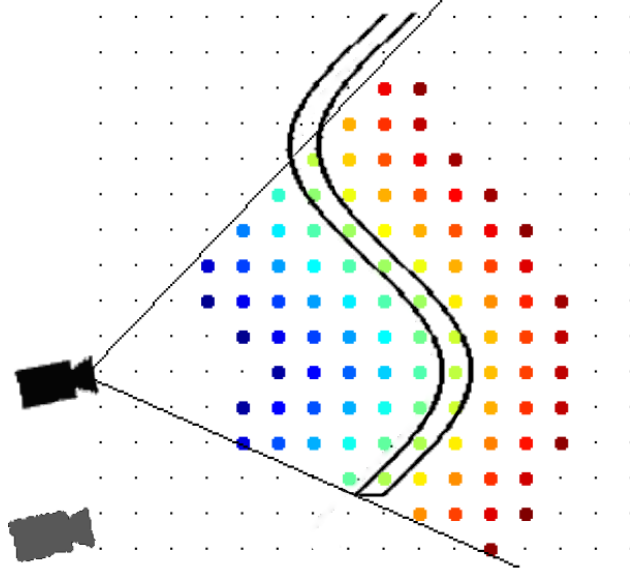
To illustrate the hollowness issue occurs in the KinFu data, an office chair has been scanned using the default KinFu algorithm. Figure 4.10(a) shows an office chair and a red line which approximates the location of the plane of Figure 4.10(b). Figure 4.10(b) shows plane of the KinFu TSDF which intersects

with the backrest of the chair. As can be seen in Figure 4.10(b), the backrest of the chair is hollow as expected from Figure 4.9. This hollowness of obstacles results in different values within the obstacles than expected. A position within an obstacle might be evaluated as outside an obstacle when the obstacle is hollow. Therefore, the KinFu cannot be used with the default settings.



**Figure 4.10:** Part (a) shows the office chair and a red line. The red line indicates approximate location of the plane of part (b). Part (b) shows KinFu data of the plane. As can be seen in Part (b), the chair is scanned as an hollow object.

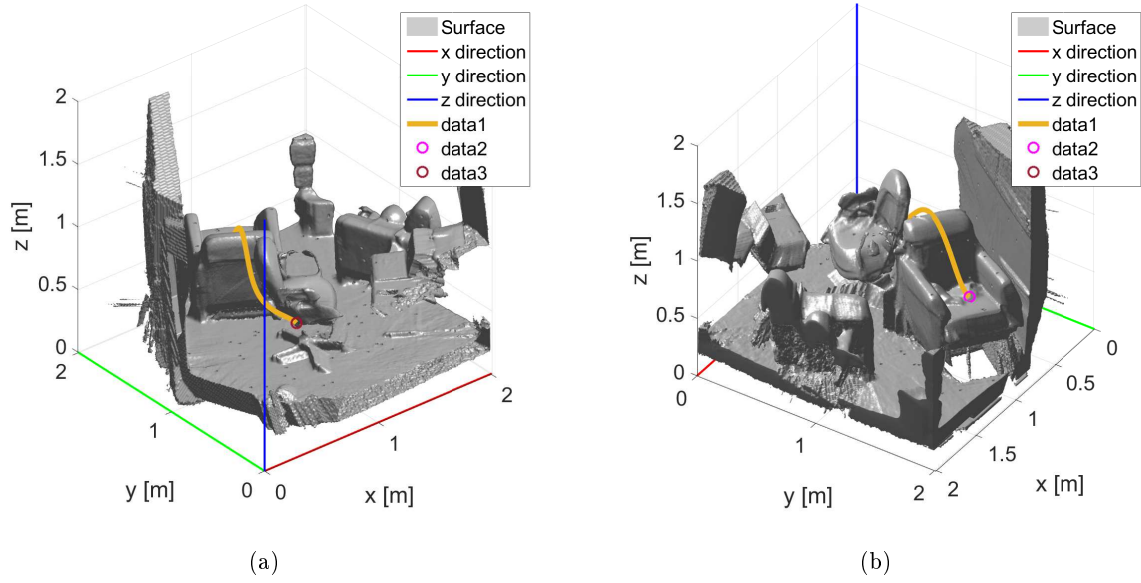
The simplest solution to solve the hollowness problem is to increase the truncation threshold of the KinFu algorithm. The obstacle will not be hollow when the truncation threshold is larger than half the thickness of the largest obstacle. However, the KinFu algorithm has some problems with this increase of truncation threshold. First, the number of points within the point cloud increase linearly with the size of the truncation threshold. Therefore, the point cloud buffer has to be increased as well. Moreover, the tracking of the camera is influenced by an increase of the truncation threshold. The depth camera is tracked by an ICP algorithm, as explained in Chapter 2.



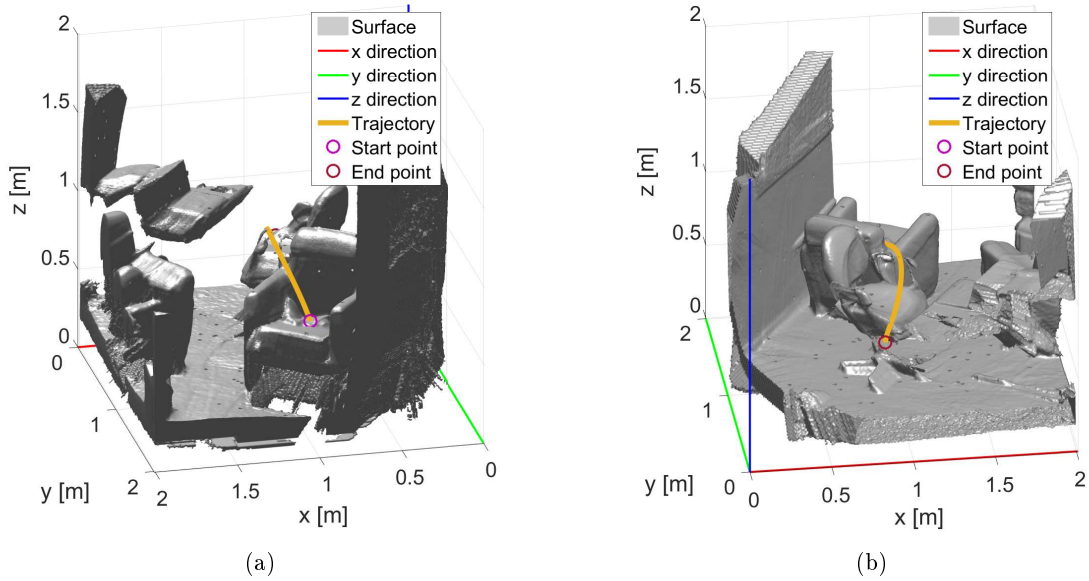
**Figure 4.11:** *An example of a too large truncation threshold. Here positive truncation signed distance functions (TSDF) are shown by blue to green ( $TSDF \in (1, 0]$ ), negative truncation distances are shown by green to red ( $TSDF \in [0, -1)$ ) and the black line represent the obstacle surface ( $TSDF = 0$ ). Here the black camera is the original point of view and the gray camera is the second point of view.*

Figure 4.11 shows a thin obstacle which has been scanned by the KinFu solely from one camera position, the black camera in the Figure. As can be seen in Figure 4.11, the voxels behind an obstacle are sensed as a voxel which is inside of an obstacle while this clearly is not true. In other words, the sensor thinks that the obstacle is much thicker than the obstacle actually is. Therefore, the depth image of Figure 4.11 results in false information in the global discrete TSDF. When the camera move to the second location, the gray camera in Figure 4.11, the second depth image senses a different environment than it expected. Hence, the false information of the first measurement is noise for the ICP algorithm. In case of the hollowness problem, the truncation threshold should be at least larger than half the size of the largest obstacle, as discussed earlier in this section. However, Figure 4.11 shows that the truncation threshold should not be larger than the smallest obstacle. Therefore, the hollowness problem and the tracking problem contradict. Hence, the truncation threshold should be chosen with caution and is a practical solution for the hollowness problem.

To test the KinFu data in optimal motion planning, the same state cost, dynamics, and terminal cost as in Section 4.2.1 are used. Moreover, the same dPRONTO algorithm is used as in Section 4.2.1. However, The obstacle avoidance constraint is created by interpolation the modified KinFu data instead of a predefined discrete TSDF. The parameters of the dPRONTO algorithm are slightly different for the KinFu TSDF than the predefined discrete TSDF. The parameters of the dPRONTO algorithm for the KinFu based optimal motion planning are given in Appendix F. Figures 4.12 and 4.13 show the optimal trajectory based on modified KinFu data. As can be seen, the optimal motion planning finds two different trajectories around the obstacles. The starting points of both trajectories has been chosen to be the same, both  $x(0) = [1.4414, 2.3906, 1.0547]$ . However, the difference of the trajectories is caused by the different desired end point. The desired end point of Figure 4.12 is  $x(T) = [0.66, 0.47, .70]$  and the desired end point of Figure 4.13 is  $[0.86, 0.23, 0.70]$ . Both trajectories are initialized with a constant velocity over the entire trajectory.



**Figure 4.12:** The optimal trajectory around Kinect Fusion data. (a) and (b) show the same trajectory and obstacles form a different angle.



**Figure 4.13:** The optimal trajectory around Kinect Fusion data. (a) and (b) show the same trajectory and obstacles form a different angle. The obstacle in this Figure is the same obstacle as Figure 4.12.

### 4.3 Summary

This chapter started with the verification of the requirement that the interpolation method should be as fast as possible. The calculation time of the proposed interpolation method is compared with a CPU version of the proposed method and a MATLAB interpolation. As in Section 4.1, the proposed interpolation algorithm calculates the coefficients significantly faster than the other two interpolation method. For a

discrete TSDF of the size  $128 \times 128 \times 128$ , the GPGPU interpolated method is over 275 times faster than a default MATLAB interpolation method and over 40 times faster than a CPU implementation of the interpolation method. The CPU interpolation is faster in calculating the output based on the coefficients when the number of outputs is less than 50. When the number of outputs is more than 50 than the proposed interpolation method is faster than the CPU version. However, the combination of calculating the coefficients using a GPU and calculating the outputs using a CPU is not ideal. The coefficients have to be transferred from the GPU memory to the CPU memory, resulting in time loss due to restoring the coefficients. The MATLAB version is in both cases significantly slower than the proposed interpolation method. Concluding, the GPGPU interpolation should be used for calculating the coefficients and the outputs of the interpolation method.

In Section 4.2, the proposed interpolation method is implemented in an optimal motion planning algorithm. First, the discrete TSDF is calculated using an exact SDF. A few example are shown where the optimal motion planning algorithm provides an optimal trajectory. Moreover, this section showed that linear interpolation fails in some cases of optimal motion planning. The default open-source implementation of KinectFusion (KinFu) provides a different discrete TSDF than expected. In KinFu, an object is hollow inside, where the assumption was that these voxels inside of an obstacle are  $-1$ . This difference in the discrete TSDF results in a problem in the optimal motion planning algorithm. Therefore, KinFu is tuned in order to get a usable discrete TSDF. The optimal motion planning algorithm is able to calculate an optimal trajectory based on modified KinectFusion data, as shown in Section 4.2.





## 5 Conclusions and recommendations

### 5.1 Conclusions

In Chapter 1, three different goals have been set for this research. The first goal is to implement and a validated a fast interpolation method of a 3D scalar field for optimal motion planning. The second goal is to combine the interpolation method with an SDF to calculate the obstacle avoidance constraint for optimal motion planning. The last goal is to compare the zero level set of the interpolated SDF with the zero level set of the exact SDF. This section summarises the report based on these three goals.

The interpolation method should satisfy the smoothness and pass-through requirement as discussed in Chapter 1. Furthermore, the interpolation method should be as fast as possible. A literature study on interpolation methods has been conducted to find the fastest interpolation method. The literature concluded that the interpolation method of Unser et al. [19] [20] should be the fastest. However, no mathematical proof was given in the papers of Unser et al. Section 3.1 shows that the digital filter method is faster in the conducted experiments than a time reduced version of cubic b-spline interpolation method which is based on the tridiagonal theory. The smoothness and pass-through requirements have been validated for the digital filter method in Section 3.3. Section 3.2 discusses the implementation of the proposed interpolation method. The proposed interpolation method is a CUDA implementation of the digital filter method of Unser et al.

Section 2.3.4 compares the proposed interpolation with a CPU implementation of the digital filter and a default MATLAB interpolation algorithm. The interpolation methods have been compared based on their calculation time of two parts of the interpolation. The first part of the interpolation is calculating the coefficients from the SDF. The second part of the interpolation is calculating the obstacle avoidance constraint and the derivatives of multiple points. Section 2.3.4 shows that the proposed interpolation method is significantly faster than the other two methods. When the SDF has a size of  $128 \times 128 \times 128$ , the proposed interpolated method is over 275 times faster than a default MATLAB interpolation method and over 40 times faster than a CPU implementation of the digital filter interpolation method. The difference in second part of the interpolation method was not as clear as the first part. The CPU implementation is faster when the number of outputs is lower than 50. The coefficients calculating of the CPU implementation is significantly slower and converting the coefficients from the GPU memory to the CPU memory costs time as well. Furthermore, the number of outputs of the interpolation method is probably larger than 50. Concluding, the proposed interpolation method is the fastest method of interpolating the discrete SDF.

Section 3.4 compares the zero level set of the interpolated SDF with the zero level set of the exact SDF. This comparison has been done for several different obstacle radius to grid size ratios. The validation showed a relation between the error and the obstacle radius to grid size ratio. The largest error was visible when the interpolation method had to recreate an edge. The error in the edge reconstruction was smaller than 1% of the grid size when the radius to grid size ratio is larger than 4. When more accuracy is required, the radius to grid size ratio can be increased.

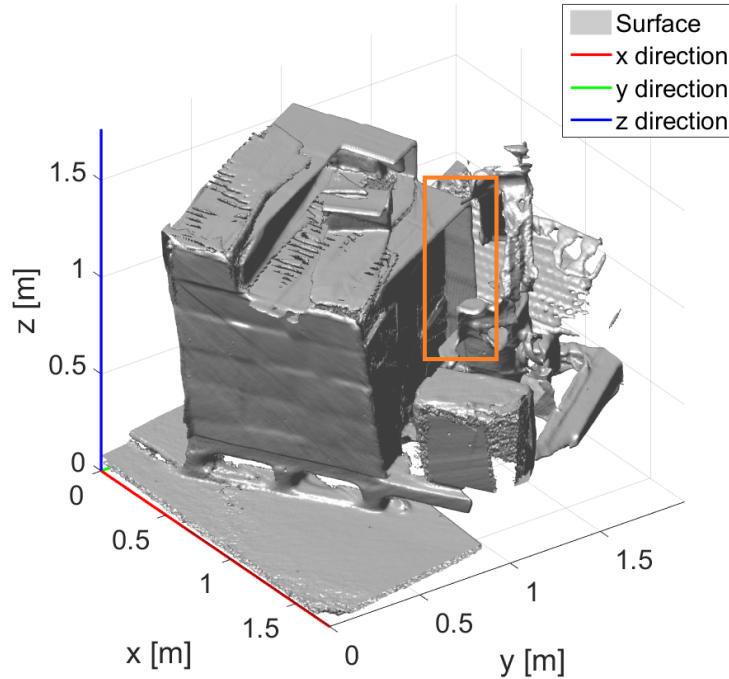
Last, the proposed interpolation method has been implemented in an in-house optimal motion planning algorithm. This research provided images which showed that the combination of the proposed interpolation method and optimal motion planning was possible for different simulated environments. This research combined the proposed interpolation method with KinectFusion in optimal motion planning as well. The KinectFusion algorithm is originally designed to recreate the surface of an environment and locate the camera position. Consequently, KinectFusion is solely interested in points which are close to a surface. Consequently, the points which are closer to a surface than the truncation threshold are

used to recreate the environment. However, hollow obstacles can result in an optimal control problem which cannot be used in optimal motion planning as discussed in Section 4.2.2. Hence, the truncation threshold of the KinectFusion algorithm has to be increased. This increase in truncation threshold reduces the accuracy of the Iterative Closest Point algorithm which tracks the camera position. This ICP algorithm already has difficulties to track the camera in low detailed environments or when the camera is moving fast. Hence, the increase of the truncation threshold should be done with caution. The proposed interpolation method combined with the modified KinectFusion algorithm was able to provide an obstacle avoidance constraint for optimal motion planning. Furthermore, this research provided proof that sensor data can be used to calculate an obstacle avoidance constraint for optimal motion planning.

## 5.2 Recommendations

1. **Smoothness of the interpolation method.** This research proposed a fastest interpolation method which satisfied the requirement and assumptions for optimal motion planning. However, the fastest interpolation method might not result in the fastest convergence rate in the optimal motion planning algorithm. Section 2.4 shows that a Newton-like minimization method has a region in which it has a quadratic convergence rate. Increasing the smoothness of the obstacle avoidance constraint might result in a larger region with a quadratic convergence rate. Hence, the optimal motion planning should use fewer iterations to find the minimizer. The obstacle avoidance constraint can be smoother by using, for example, smoothing splines. This thesis assumed that smoothing splines are not necessary for optimal motion planning, in Chapter 3. The same idea holds for higher order splines. Higher order splines might increase the smoothness of the obstacle avoidance constraint as well. However, this thesis was solely concerned with satisfying the requirements with the fastest interpolation method. I do not have the expertise to prove or disprove this concept. Therefore, an investigation might give more insight whether this concept improves the calculation time. Even when this concept improves the calculation time, the reducing the calculation time is not the highest priority.
2. **Not knowing the environment a-priori.** An assumption which has not been discussed in this report is the fact that this research assumed that the entire environment is known a-priori. To use this thesis, the entire environment has to be scanned before the robot is able to apply optimal motion planning. A more realistic/autonomous case would be that a robot should plan the optimal trajectory while scanning the environment. I do not have the background to state whether the current method will or will not work in a finite optimal motion planning. Therefore, the environment description should be implemented in a finite horizon approach. This problem is not of top priority but should be investigated in the near future.
3. **Dynamic environments.** This thesis was solely concerned with a static environment. However, a static environment is not a realistic assumption for the operating environment of most robots. Most robots will encounter moving obstacles in their environment which it should avoid. KinFu is able to reconstruct moving obstacles. However, the KinFu algorithm needs to have a scan of the static environment before it is able to segment the moving obstacles. Therefore, the KinFu algorithm needs to scan an environment with solely static obstacles. Furthermore, the KinFu algorithm needs to scan again for the moving obstacles. If the KinFu algorithm is used, static obstacles cannot be added or removed from the environment. Therefore, an updated KinFu algorithm or a different algorithm is required to deal with dynamic obstacles in the environment. An alternative method for KinFu is given by Keller et al. [34]. Keller et al. state that their method is able to segment dynamic obstacles from the scan. This problem is not a top priority. However, robots require an optimal motion planning method which is able to cope with dynamic environments. Therefore, a research on how to describe a dynamic environment has to be done in the near future.

4. **Tracking of the camera.** The KinFu algorithm relies on an ICP algorithm to track the camera position. The tracking error has an influence on the obstacle reconstruction. However, the ICP algorithm is not accurate in tracking the camera. Keller et al. [34] have investigated the tracking error of KinFu algorithm and their scanning method. Both algorithms had a nonnegligible tracking error in a controlled experiment. The tracking error of KinFu algorithm peaked at  $0.1[m]$  in the controlled experiment. Moreover, Keller et al. noticed that both methods have problems with sensor drift. This sensor drift was also noticed in this thesis. Figure 5.1 shows a KinFu representation of a pallet which is a box. However, the pallet does not seem to be a box in Figure 5.1. Moreover, the edge shown inside the orange square shows that the edge is misaligned. This misalignment is the result of the sensor drift.



**Figure 5.1:** *An example of a failing scan due to a large error in the tracking.*

Concluding, the camera tracking of the KinFu algorithm is not accurate enough for optimal motion planning. Therefore, the tracking of the camera needs to be improved. A method would be to fuse the ICP tracking with a different tracking method. This tracking problem is of high priority and an investigation into improving the tracking should be done as fast as possible.

5. **Hollow obstacles.** Section 4.2.2 discusses the problem of obstacle representation of the default KinFu algorithm. The truncation of the SDF results in hollow obstacles when the truncation threshold is smaller than half the thickness of the thickest obstacles. This thesis solved this problem by increasing the truncation threshold. However, the tracking of the camera was not able to track the camera when the truncation threshold was larger than the thickness of the thinnest obstacle. Therefore, the truncation threshold has to be chosen with caution when using this method. Although, the tracking of the camera already has to be improved, increasing the truncation threshold is not the most intelligent solution for solving this problem. Hence, a more intelligent solution should be investigated for the hollowness problem. The hollowness problem is not as important as

the tracking error. Therefore, the hollowness problem can wait at least until the tracking error is resolved.

6. **Fast output calculation based on coefficients.** This research used (2.22) for calculating the output of a desired position. However, Ruijters et al. [35] propose a faster method of calculating the desired output. The method of Ruijters et al. has not been used in this thesis because it would not work in a CUDA/MATLAB implementation. Ruijters et al. propose their method using CUDA together with *C*-code. Therefore, the problem could be that the method of Ruijters et al. does not work with MATLAB. As far as I understand their solution, Ruijters et al. propose to store the coefficients differently in a specific GPU memory type. This type of memory has the benefit that linear interpolating between two points costs significantly less time than normal interpolation. Therefore, the output of a desired point can be calculated using 8 linear interpolations instead of 64 nearest neighbour lookups. Hence, the calculation time will be reduced using the method of Ruijters et al. According to Ruijters et al., their method is over 20% faster than the method which has been used in this thesis. Each loop of the optimal motion planning requires a calculation of the output of the interpolated obstacle avoidance constraint and its derivatives. Therefore, the method of Ruijters et al. should be beneficial for optimal motion planning. However, this is not as important as the tracking and hollowness problem described before.

## References

- [1] Roland Jan Geraerts. Sampling-based motion planning: Analysis and path quality. 2006.
- [2] Matt Zucker, Nathan Ratliff, Anca D Dragan, Mihail Pivtoraiko, Matthew Klingensmith, Christopher M Dellin, J Andrew Bagnell, and Siddhartha S Srinivasa. CHOMP: Covariant Hamiltonian optimization for motion planning. *The International Journal of Robotics Research*, 32(9-10):1164–1193, 2013.
- [3] Mrinal Kalakrishnan, Sachin Chitta, Evangelos Theodorou, Peter Pastor, and Stefan Schaal. STOMP: Stochastic trajectory optimization for motion planning. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 4569–4574. IEEE, 2011.
- [4] John Schulman, Jonathan Ho, Alex X Lee, Ibrahim Awwal, Henry Bradlow, and Pieter Abbeel. Finding locally optimal, collision-free trajectories with sequential convex optimization. In *Robotics: science and systems*, volume 9, pages 1–10, 2013.
- [5] Jim Mainprice, Nathan Ratliff, and Stefan Schaal. Warping the workspace geometry with electric potentials for motion optimization of manipulation tasks. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*, pages 3156–3163. IEEE, 2016.
- [6] C.T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics, 1995.
- [7] Car De Boor. *A practical guide to splines*, volume 27. Springer-Verlag New York, 1978.
- [8] Brian Curless and Marc Levoy. A volumetric method for building complex models from range images. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312. ACM, 1996.
- [9] Shahram Izadi, David Kim, Otmar Hilliges, David Molyneaux, Richard Newcombe, Pushmeet Kohli, Jamie Shotton, Steve Hodges, Dustin Freeman, Andrew Davison, et al. KinectFusion: real-time 3d reconstruction and interaction using a moving depth camera. In *Proceedings of the 24th annual ACM symposium on User interface software and technology*, pages 559–568. ACM, 2011.
- [10] John P Morgan and Richard L Tutwiler. Real-time reconstruction of depth sequences using signed distance functions. In *SPIE Defense+ Security*, pages 909117–909117. International Society for Optics and Photonics, 2014.
- [11] Paul J Besl, Neil D McKay, et al. A method for registration of 3-D shapes. *IEEE Transactions on pattern analysis and machine intelligence*, 14(2):239–256, 1992.
- [12] John H Mathews, Kurtis D Fink, et al. *Numerical methods using MATLAB*, volume 4. Pearson London, UK:, 2004.
- [13] Philippe Thévenaz, Thierry Blu, and Michael Unser. Interpolation revisited [medical images application]. *IEEE Transactions on medical imaging*, 19(7):739–758, 2000.
- [14] Carl De Boor. *Spline toolbox for use with MATLAB: user’s guide*, version 3. MathWorks, 2005.
- [15] Brian A Barsky. End conditions and boundary conditions for uniform b-spline curve and surface representations. *Computers in Industry*, 3(1-2):17–29, 1982.
- [16] Francois Lekien and J Marsden. Tricubic interpolation in three dimensions. *International Journal for Numerical Methods in Engineering*, 63(3):455–471, 2005.

- 
- [17] Larry Schumaker. Spline functions: basic theory. Cambridge University Press, 2007.
  - [18] Richard H Bartels, John C Beatty, and Brian A Barsky. An introduction to splines for use in computer graphics and geometric modeling. Morgan Kaufmann, 1995.
  - [19] Michael Unser, Akram Aldroubi, and Murray Eden. Fast B-spline transforms for continuous image representation and interpolation. *IEEE Transactions on pattern analysis and machine intelligence*, 13(3):277–285, 1991.
  - [20] Michael Unser. Splines: A perfect fit for signal and image processing. *IEEE Signal processing magazine*, 16(6):22–38, 1999.
  - [21] Ahmed I Zayed. A convolution and product theorem for the fractional fourier transform. *IEEE Signal processing letters*, 5(4):101–103, 1998.
  - [22] Daniel Ruijters and Philippe Thévenaz. GPU prefilter for accurate cubic B-spline interpolation. *The Computer Journal*, page 086, 2010.
  - [23] Stephen J Wright and Jorge Nocedal. Numerical optimization. Springer Science, 35(67-68):7, 1999.
  - [24] Samuel Daniel Conte and Carl W De Boor. Elementary numerical analysis: an algorithmic approach. McGraw-Hill Higher Education, 1980.
  - [25] Dan Zuras, Mike Cowlshaw, Alex Aiken, Matthew Applegate, David Bailey, Steve Bass, Dileep Bhandarkar, Mahesh Bhat, David Bindel, Sylvie Boldo, et al. IEEE standard for floating-point arithmetic. *IEEE Std 754-2008*, pages 1–70, 2008.
  - [26] How CUDA’s abstractions map to a GPGPU implementation. <http://15418.courses.cs.cmu.edu/spring2013/article/11>.
  - [27] Robert C James. Advanced calculus belmont. Calif.: Wadsworth, 1966.
  - [28] Christian Knauer, Maarten Löffler, Marc Scherfenberg, and Thomas Wolle. The directed Hausdorff distance between imprecise point sets. In *ISAAC*, pages 720–729. Springer, 2009.
  - [29] Stuart E Dreyfus. Dynamic programming and the calculus of variations, volume 21. Academic Press New York, 1965.
  - [30] John Hauser and Alessandro Saccon. A barrier function method for the optimization of trajectory functionals with constraints. In *Decision and Control, 2006 45th IEEE Conference on*, pages 864–869. IEEE, 2006.
  - [31] T.J.G Damen. Discrete-time optimal control algorithms for online trajectory planning. Master Thesis, Eindhoven University of Technology.
  - [32] A Pedro Aguiar, Florian A Bayer, John Hauser, Andreas J Häusler, Giuseppe Notarstefano, Antonio M Pascoal, Alessandro Rucco, and Alessandro Saccon. Constrained optimal motion planning for autonomous vehicles using pronto. In *Sensing and Control for Autonomous Vehicles*, pages 207–226. Springer, 2017.
  - [33] Inigo Quilez. Signed distance functions of different primitives and combinations. <http://http://iquilezles.org/www/articles/distfunctions/distfunctions.htm>.
  - [34] Maik Keller, Damien Lefloch, Martin Lambers, Shahram Izadi, Tim Weyrich, and Andreas Kolb. Real-time 3d reconstruction in dynamic scenes using point-based fusion. In *3DTV-Conference, 2013 International Conference on*, pages 1–8. IEEE, 2013.
-

- [35] Daniel Ruijters, Bart M ter Haar Romeny, and Paul Suetens. Efficient GPU-based texture interpolation using uniform B-splines. *Journal of Graphics, GPU, and Game Tools*, 13(4):61–69, 2008.



## A B-splines is a special case of piecewise polynomial

To determine whether b-spline interpolation is a specific form of piecewise polynomial interpolation, the following equation has to be solved

$$S_i(x) = a_i + b_i(x - x_i) + c_i(x - x_i)^2 + d_i(x - x_i)^3 = \sum_{j=0}^n c_j \phi_j(x - x_j), \quad (\text{A.1})$$

where  $S_i$  is the piecewise polynomial which describes the interpolation between points  $x_i$  and  $x_{i+1}$ ,  $a_i$ ,  $b_i$ ,  $c_i$  and  $d_i$  are the coefficients of the piecewise polynomial interpolation,  $c_j$  are the coefficients of the b-splines. Solely for readability, the following equation will be used

$$u_i = x - x_i, \quad \Delta u = x_{i+1} - x_i, \quad u_{i+1} = u_i - \Delta u, \quad (\text{A.2})$$

where  $\Delta u$  is the grid size. To reduce complexity of the derivation, the grid size will be fixed here at one. Cubic b-splines have a small window where they have a value. Therefore, the equation (A.1) can be rewritten into

$$S_i(x) = \sum_{j=i-1}^{i+2} c_j \phi_j(u_j) = c_{i-1} \phi_{i-1}(u_i + \Delta u) + c_i \phi_i(u_i) + c_{i+1} \phi_{i+1}(u_i - \Delta u) + c_{i+2} \phi_{i+2}(u_i - 2\Delta u). \quad (\text{A.3})$$

This equation can be rewritten into vector notation

$$S_i(x) = \begin{bmatrix} \phi_{i-1}(u_i + \Delta u) & \phi_i(u_i) & \phi_{i+1}(u_i - \Delta u) & \phi_{i+2}(u_i - 2\Delta u) \end{bmatrix} \begin{bmatrix} c_{i-1} \\ c_i \\ c_{i+1} \\ c_{i+1} \end{bmatrix}. \quad (\text{A.4})$$

To solve this equation, four basis function have to be calculated in term of  $u_i$ . Therefore, equation (2.16) is determine the influence of each basis function. Moreover, these equation are written in vector notation.

$$\phi_{i-1}(u_i + \Delta u) = \frac{1}{6}(2 - u_i - 1)^3 = -\frac{1}{6}u_i^3 + \frac{1}{2}u_i^2 - \frac{1}{2}u_i + \frac{1}{6} = \begin{bmatrix} u_i^3 & u_i^2 & u_i & 1 \end{bmatrix} \begin{bmatrix} -\frac{1}{6} \\ \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{6} \end{bmatrix}, \quad (\text{A.5})$$

$$\phi_i(u_i) = \frac{2}{3} - \frac{1}{2}(u_i)^2(2 - u_i) = -\frac{1}{2}u_i^3 + u_i^2 + \frac{2}{3} = \begin{bmatrix} u_i^3 & u_i^2 & u_i & 1 \end{bmatrix} \begin{bmatrix} -\frac{1}{2} \\ 1 \\ 0 \\ \frac{2}{3} \end{bmatrix}, \quad (\text{A.6})$$

$$\phi_{i+1}(u_i - \Delta u) = \frac{2}{3} - \frac{1}{2}(u_i - 1)^2(2 + u_i - 1) = \frac{1}{2}u_i^3 - \frac{1}{2}u_i^2 - \frac{1}{2}u_i + \frac{1}{6} = \begin{bmatrix} u_i^3 & u_i^2 & u_i & 1 \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{6} \end{bmatrix}, \quad (\text{A.7})$$

$$\phi_{i+2}(u_i - 2\Delta u) = \frac{1}{6}(2 + u_i - 2)^3 = \frac{1}{6}u_i^3 = \begin{bmatrix} u_i^3 & u_i^2 & u_i & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}. \quad (\text{A.8})$$

Combining equations (A.5), (A.6), (A.7), (A.8) and (A.4) together creates the following equation

$$S_i(x) = \begin{bmatrix} u_i^3 & u_i^2 & u_i & 1 \end{bmatrix} \begin{bmatrix} -\frac{1}{6} & -\frac{1}{2} & \frac{1}{2} & \frac{1}{6} \\ \frac{1}{2} & 1 & -\frac{1}{2} & 0 \\ -\frac{1}{2} & 0 & -\frac{1}{2} & 0 \\ \frac{1}{6} & \frac{2}{3} & \frac{7}{6} & 0 \end{bmatrix} \begin{bmatrix} c_{i-1} \\ c_i \\ c_{i+1} \\ c_{i+1} \end{bmatrix}. \quad (\text{A.9})$$

## B Convergence rate Newton's Method

The proof of the local quadratic convergence can be proven using the three assumptions. When the standard assumption hold, there exist a ball ( $B(x||e_i|| < r)$ ) around the actual root in which the following equations hold

$$||f'(x_i)|| \leq 2||f'(x^*)||, \quad (\text{B.1})$$

$$||f'(x_i)^{-1}|| \leq 2||f'(x^*)^{-1}||, \quad (\text{B.2})$$

$$||f(x^*)^{-1}||^{-1}||e_i||/2 \leq ||f(x_i)|| \leq 2||f'(x^*)||||e_i||. \quad (\text{B.3})$$

Moreover, when  $x_i$  is close to  $x^*$  then the following relation holds

$$f(x_i) = \int_0^1 f'(x^* + te_i)e_i dt. \quad (\text{B.4})$$

Combining (2.39), (2.46) and (B.4) results in the following equation

$$\begin{aligned} e_+ &= f'(x_i)^{-1}(f'(x_i)e_i + f(x_i)) \\ &= f'(x_i)^{-1} \int_0^1 (f'(x_i) - f'(x^* + te_i))e_i dt \end{aligned} \quad (\text{B.5})$$

Based on the second standard assumption, the following relation holds

$$\begin{aligned} ||f'(x_i) - f'(x^* + te_i)|| &\leq \gamma ||x_i - x^* - te_i|| \\ &\leq \gamma ||(1-t)e_i||. \end{aligned} \quad (\text{B.6})$$

When (B.6) is implemented into (B.5) results in

$$\begin{aligned} ||e_+|| &\leq ||f'(x_i)^{-1}|| ||e_i|| \int_0^1 \gamma ||(1-t)e_i|| dt \\ &\leq ||f'(x_i)^{-1}|| ||e_i|| [\gamma ||(t - 1/2t^2)e_i||]_0^1 \\ &\leq ||f'(x_i)^{-1}|| ||e_i|| \gamma \frac{1}{2} ||e_i|| \\ &\leq 2||f'(x^*)^{-1}|| \gamma \frac{1}{2} ||e_i||^2. \end{aligned} \quad (\text{B.7})$$

## C CUDA source code

### C.1 1D interpolation

```

1 // include function of Ruijters
2 #include "math_func.cu"
3 #include "cutil_math_bugfixes.h"
4
5 #define Pole (sqrt(3.0f)-2.0f) // Pole of cubic b-splines is defined as sqrt(3)-2
6
7 // Initial condition for causal filter
8 __host__ __device__ float InitialCausalCoefficient(
9     float* c,           // coefficients
10    uint DataLength,     // number of coefficients
11    int step)           // element interleave in bytes
12 {
13     const uint Horizon = UMIN(12, DataLength); // More than 12 has no influence
14
15     // this initialization corresponds to clamped boundaries
16     // accelerated loop
17     float zn = Pole; // Weighting factor based on the ...
18     pole // Initializing sum with c(0)
19     float Sum = *c;
20     for (uint n = 0; n < Horizon; n++) {
21         Sum += zn * *c; // Weighting factor multiplied with ...
22         coefficient
23         zn *= Pole; // Update weighting factor
24         c = (float*)((uchar*)c + step); // Update coefficients
25     }
26     return(Sum); // Return intial condition for ...
27     causal filter
28 }
29
30 // Initial conditon for the anti causal filter
31 __host__ __device__ float InitialAntiCausalCoefficient(
32     float* c,           // pointer to last coefficient
33     uint DataLength,     // number of samples or coefficients
34     int step)           // element interleave in bytes
35 {
36     // this initialization corresponds to clamping boundaries
37     return((Pole / (Pole - 1.0f)) * *c);
38 }
39
40 __host__ __device__ void ConvertToInterpolationCoefficients(
41     float* coeffs,       // input samples --> output coefficients
42     uint DataLength,     // number of samples or coefficients
43     int step)           // element interleave in bytes
44 {
45     // compute the overall gain
46     const float Lambda = (1.0f - Pole) * (1.0f - 1.0f / Pole); // given formula for ...
47     b-splines
48
49     // causal initialization
50     float* c = coeffs;
51     float previous_c; //cache the ...
52     previously calculated c rather than look it up again (faster!)
53     *c = previous_c = Lambda * InitialCausalCoefficient(c, DataLength, step); // ...
54     Initial condition for causal filter

```

```

49 // causal recursion
50 for (uint n = 1; n < DataLength; n++) {
51     c = (float*)((uchar*)c + step);
52     *c = previous_c = Lambda * *c + Pole * previous_c;
53 }
54 // anticausal initialization
55 *c = previous_c = InitialAntiCausalCoefficient(c, DataLength, step);
56 // anticausal recursion
57 for (int n = DataLength - 2; 0 ≤ n; n--) {
58     c = (float*)((uchar*)c - step);
59     *c = previous_c = Pole * (previous_c - *c);
60 }
61 }

```

## C.2 Interpolate 3D grid in x-direction

```

1 #include "interpolate.cu" //include 1D interpolation functions
2
3 // interpolation in x-direction
4 __global__ void SamplesToCoefficients3DX(
5     float* volume, int N_x, int N_y, int N_z )
6 // volume: input the current space -> output coefficients
7 // N_x size of space in x-direction, N_y size of space in y-direction, N_z size of ...
8 // space in z-direction
9 {
10     uint width  = (unsigned int) N_x;           // width of the space
11     uint height = (unsigned int) N_y;           // height of the space
12     uint depth  = (unsigned int) N_z;           // depth of the space
13     uint pitch  = sizeof(float) * width;        // width in bytes
14
15     // process lines in x-direction
16
17     const uint y = blockIdx.x * blockDim.x + threadIdx.x;
18     // starting points in y-direction
19     const uint z = blockIdx.y * blockDim.y + threadIdx.y;
20     // starting points in z-direction
21     const uint startIdx = (z * height + y) * pitch;
22     // starting points in y- and z-direction
23
24     float* ptr = (float*)((uchar*)volume + startIdx);
25     // pointer to starting point in the vector volume
26     ConvertToInterpolationCoefficients(ptr, width, sizeof(float));
27     // 1D interpolation in x direction and next point is sizeof(float) away
28 }

```

## C.3 Interpolate 3D grid in y-direction

```

1 #include "interpolate.cu" //include 1D interpolation functions
2
3 // interpolation in y-direction
4 __global__ void SamplesToCoefficients3DY(
5     float* volume, int N_x, int N_y, int N_z )
6 // volume: input the current space -> output coefficients

```

```

7 // N_x size of space in x-direction, N_y size of space in y-direction, N_z size of ...
  space in z-direction
8 {
9     uint width = (unsigned int) N_x;           // width of the space
10    uint height= (unsigned int) N_y;           // height of the space
11    uint depth = (unsigned int) N_z;           // depth of the space
12    uint pitch  = sizeof(float) *width;        // width in space
13
14    // process lines in y-direction
15
16    const uint x = blockIdx.x * blockDim.x + threadIdx.x;
17    // starting points in x-direction
18    const uint z = blockIdx.y * blockDim.y + threadIdx.y;
19    // starting points in z-direction
20    const uint startIdx = z * height * pitch;
21    // starting points in x- and z-direction
22
23    float* ptr = (float*)((uchar*)volume + startIdx);
24    // pointer to starting point in the vector volume
25    ConvertToInterpolationCoefficients(ptr + x, height, pitch);
26    // 1D interpolation in y-direction and next point is pitch away
27 }

```

## C.4 Interpolate 3D grid in z-direction

```

1 #include "interpolate.cu" //include 1D interpolation functions
2
3 // interpolation in z-direction
4 __global__ void SamplesToCoefficients3DZ(
5     float* volume, int N_x , int N_y , int N_z )
6 // volume: input the current space -> output coefficients
7 // N_x size of space in x-direction, N_y size of space in y-direction, N_z size of ...
  space in z-direction
8 {
9
10    uint width  = (unsigned int) N_x;           // width of the space
11    uint height = (unsigned int) N_y;           // height of the space
12    uint depth  = (unsigned int) N_z;           // depth of the space
13    uint pitch  = sizeof(float) *width;        // width in bytes
14
15    // process lines in z-direction
16
17    const uint x = blockIdx.x * blockDim.x + threadIdx.x;
18    // starting points in x-direction
19    const uint y = blockIdx.y * blockDim.y + threadIdx.y;
20    // starting points in y-direction
21    const uint startIdx = y * pitch;
22    // step in bytes in order to get from y(i) to y(i+1)
23    const uint slice = height * pitch;
24    // step in bytes in order to get from z(i) to z(i+1)
25
26    float* ptr = (float*)((uchar*)volume + startIdx);
27    // pointer to starting point in the vector volume
28    ConvertToInterpolationCoefficients(ptr + x, depth, slice);
29    // 1D interpolation in y-direction and next point is slice away
30 }

```

## C.5 Function which calculates the number of threads and blocks

```

1  function [n_t,n_b] = number_thread_blocks( n )
2  %NUMBER_THREAD_BLOCKS Calculate ideal number of threads and blocks for CUDA
3  %calculation
4  % input n is a vector with number of theads in x,y,z-direciont
5  % n(1) is the number of CUDA cores needed in x direction
6  % n(2) is the number of CUDA cores needed in y direction
7  % n(3) is the number of CUDA cores needed in z direction
8  % when number of theads are defined in less than 3 direction, user can sent smaller ...
   vector as well.
9  % ouput n_t is a vector with number threads per block in all three directions and n_b...
   is a vector with the number of blocks in the grid in all three directions
10
11  gpu_device = gpuDevice; % get information on gpu ...
   devices in pc
12  n_t = [0,0,0]; % initiate number of threads
13  n_b = [0,0,0]; % initiate number of blocks
14  dim = length(n); % dimension of vector n
15
16  if dim == 3
17      %% 3D
18      if max(n) < gpu_device.MaxThreadsPerBlock^(1/3) % when the number of ...
         calculation in each direction is smaller than the maximum theas per ...
         block
19          n_t = [n(1),n(2),n(3)]; % size of threads is equal to...
         number of calculcaltion
20          n_b = [1,1,1]; % block size is set to 1
21      else
22          for i = 1:3 % loop over all directions
23              n_t(i) = 8; % 8 still chosen on my gpu ...
                 because it is gpu_device.MaxThreadsPerBlock^(1/3) / should be ...
                 dividable by 2
24              while mod(n(i),n_t(i)) ~=0 % are the number of ...
                 calculation dividable by the number of threads
25                  n_t(i) = n_t(i)/2; % update number of theads
26                  if n_t(i) <1 % not possible
27                      return;
28                  end
29              end
30          end
31          n_b = [n(1)/n_t(1),n(2)/n_t(2),n(3)/n_t(3)]; % setting number of ...
                 blocks and threads
32      end
33  elseif dim == 2
34      %% 2D
35      if max(n) < sqrt(gpu_device.MaxThreadsPerBlock) % when the number of ...
         calculation in each direction is smaller than the maximum theas per ...
         block
36          n_t = [n(1),n(2),1]; % size of threads is ...
         equal to number of calculcaltion
37          n_b = [1,1,1]; % block size is set ...
         to 1
38      else
39          for i =1:2
40              n_t(i) = sqrt(gpu_device.MaxThreadsPerBlock); % max number of ...
                 threads in 2D case
41              while mod(n(i),n_t(i)) ~=0 % are the number of ...
                 calculation dividable by the number of threads

```

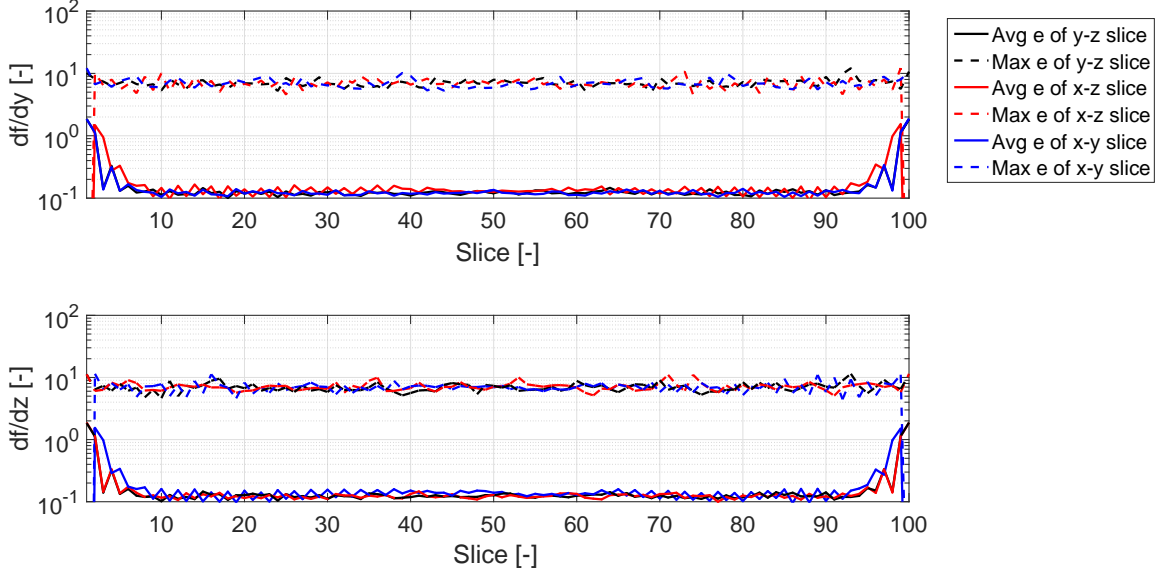
```

42         n_t(i) = n_t(i)/2;                                % update number of ...
43         threads                                     % not possible
44         if n_t(i) < 1
45             return;
46         end
47     end
48     n_t = [n_t(1),n_t(2),1];                                % setting number of ...
49     threads
50     n_b = [n(1)/n_t(1),n(2)/n_t(2),1];                        % setting number of ...
51     blocks
52 end
53 else
54     %% 1D
55     if n < gpu_device.MaxThreadBlockSize(1)                  % when the number of ...
56         calculation in each direction is smaller than the maximum threas per ...
57         block
58         n_t = [n,1,1];                                        % size of threads is equal ...
59         to number of calculcaltion
60         n_b = [1,1,1];                                        % block size is set ...
61         to 1
62     else
63         if mod(n,32) == 0                                     % take warp into account
64             n_t = [32,1,1];
65             n_b = [n/32,1,1];
66         else
67             n_t = [1,1,1];                                    % last resort
68             n_b = [n,1,1];
69         end
70     end
71 end
72 end
73 end

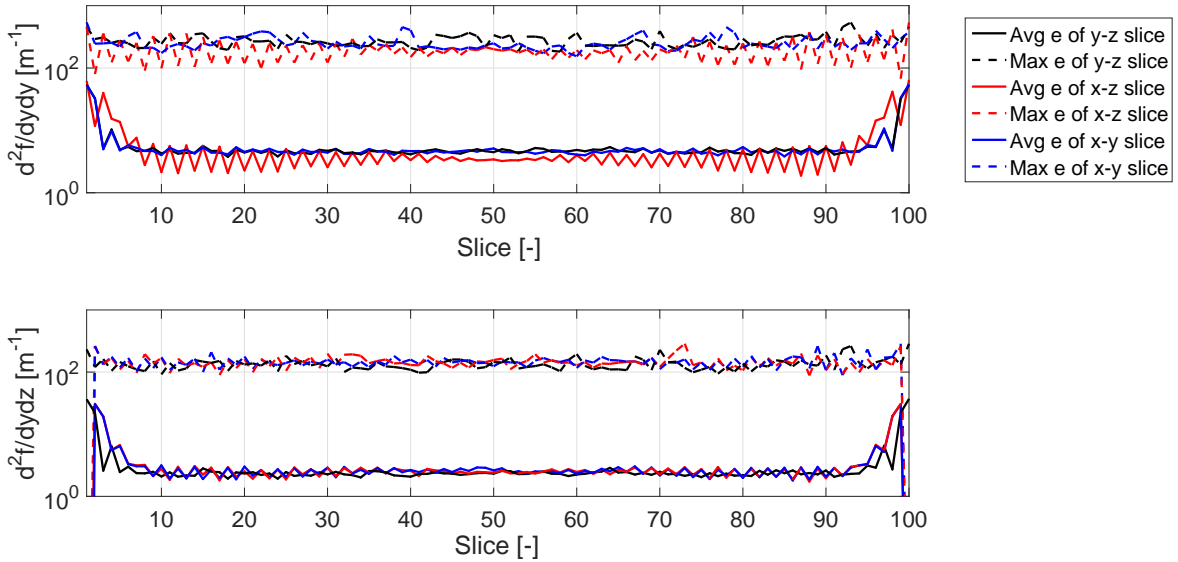
```

## D The addition images of the multivariate interpolation validation

The images of the other derivatives of the comparison which included all the refined points.

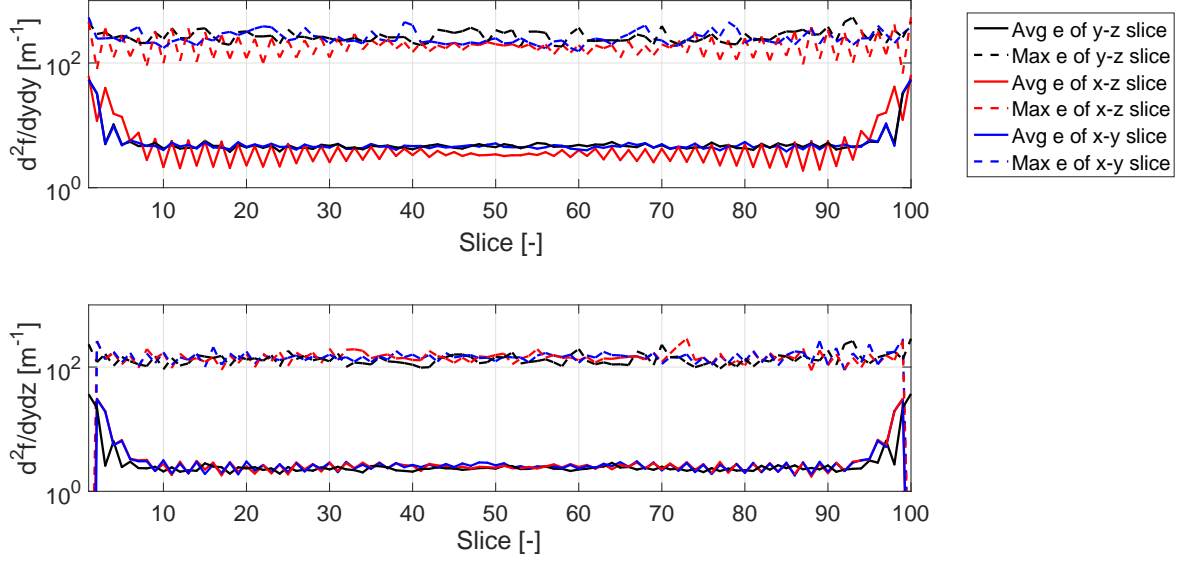


**Figure D.1:** The absolute difference between the first derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice. The maximum values of the functions  $df/dy$  and  $df/dz$  are  $\pm 20$  which results in a floating point precision of  $1.191e - 7$ .



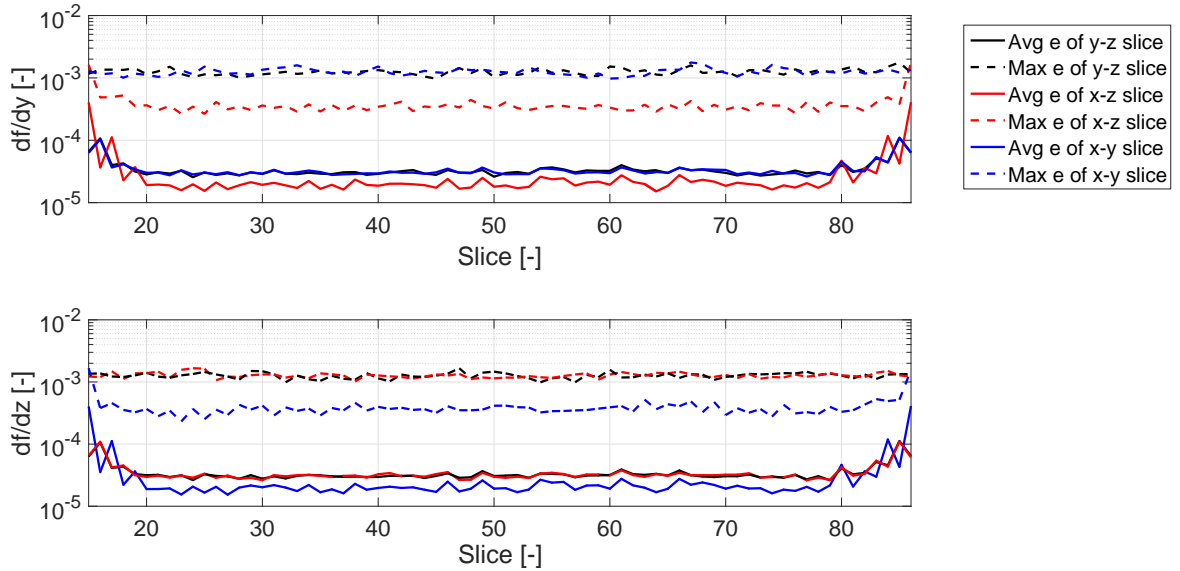
**Figure D.2:** The absolute difference between the second derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice. The maximum values of the second derivatives are  $\pm 900$  which results in a floating point precision of  $6.1e - 5$ .



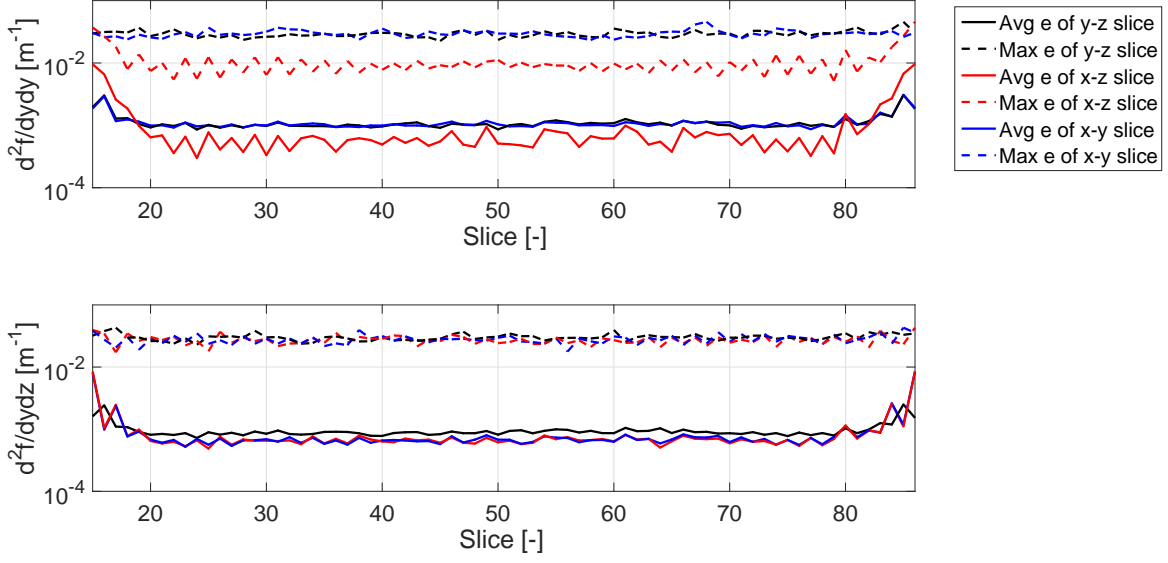


**Figure D.3:** The absolute difference between the second derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice. The maximum values of the second derivatives are  $\pm 900$  which results in a floating point precision of  $6.1e - 5$ .

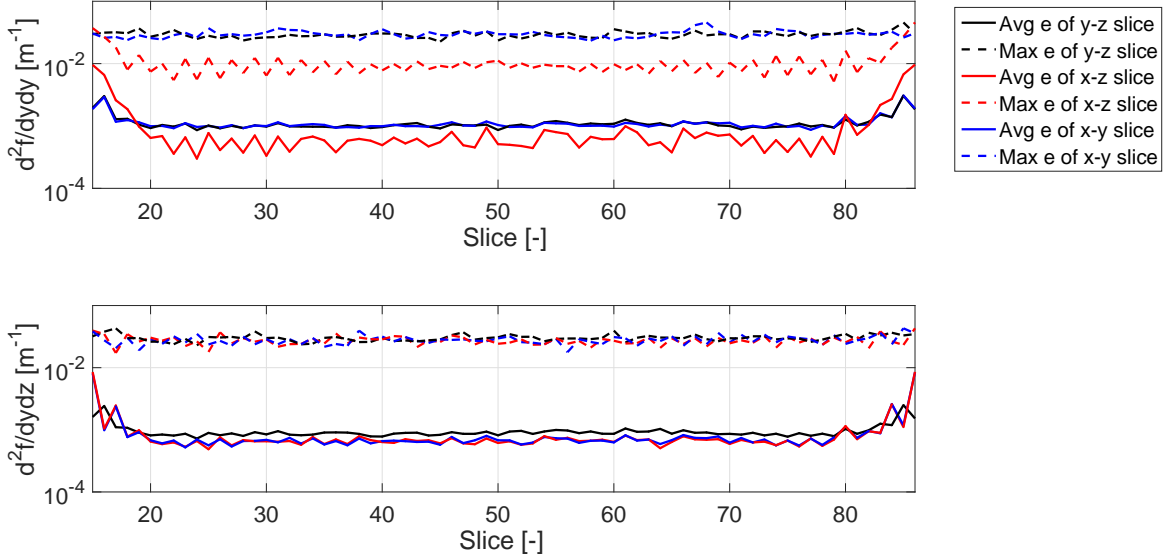
The images of the other derivatives of the comparison which neglected the 14 refined grid points close the the boundaries.



**Figure D.4:** The absolute difference between the first derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice. The maximum values of the functions  $df/dy$  and  $df/dz$  are  $\pm 20$  which results in a floating point precision of  $1.191e - 7$ .

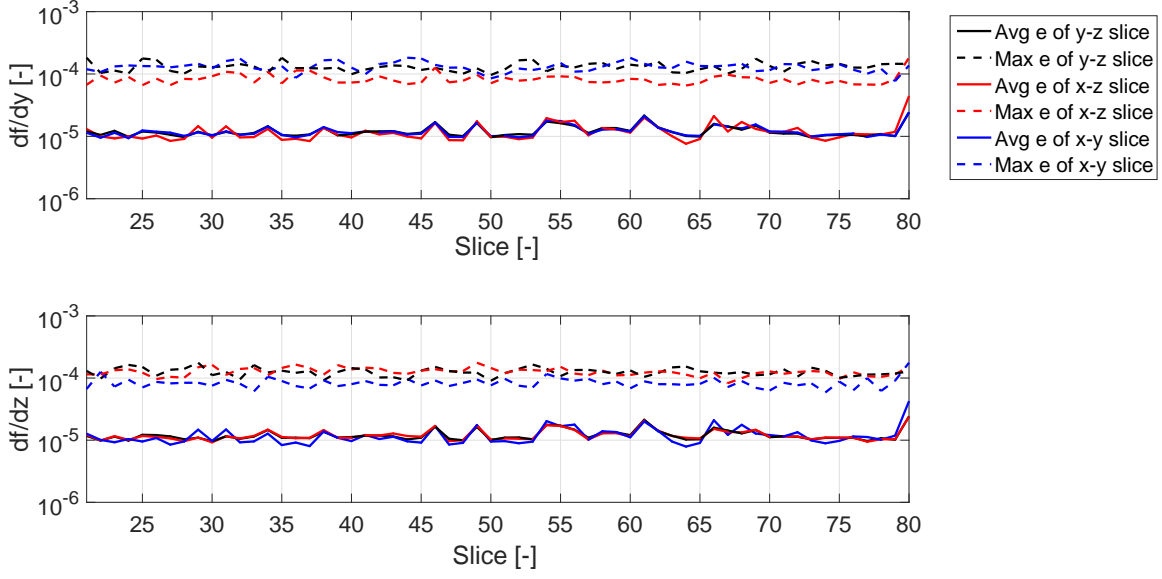


**Figure D.5:** The absolute difference between the second derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice. The maximum values of the second derivatives are  $\pm 900$  which results in a floating point precision of  $6.1e - 5$ .

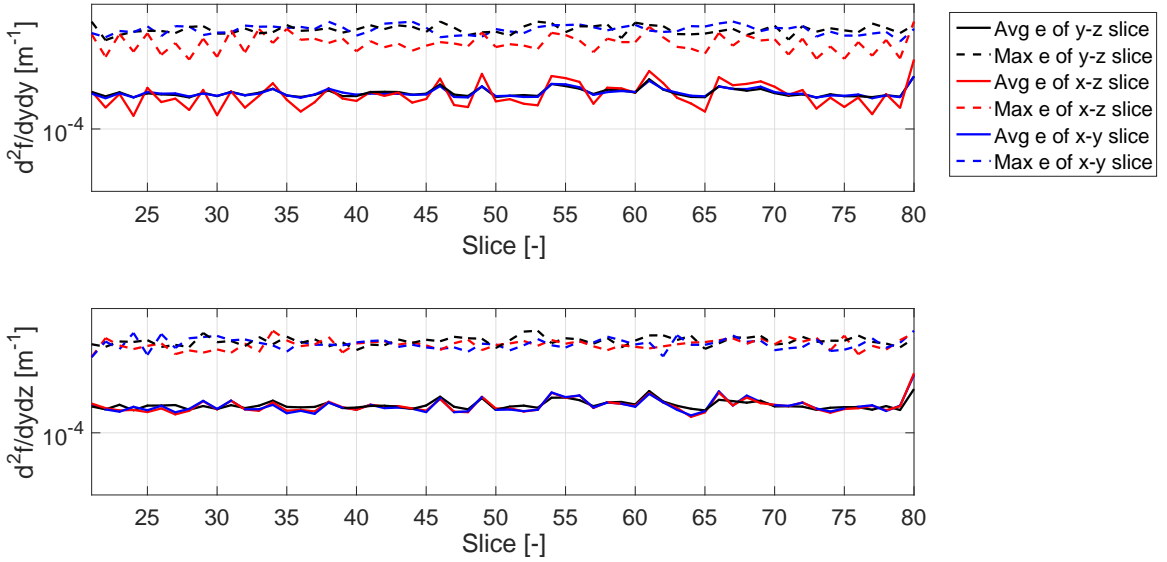


**Figure D.6:** The absolute difference between the second derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice. The maximum values of the second derivatives are  $\pm 900$  which results in a floating point precision of  $6.1e - 5$ .

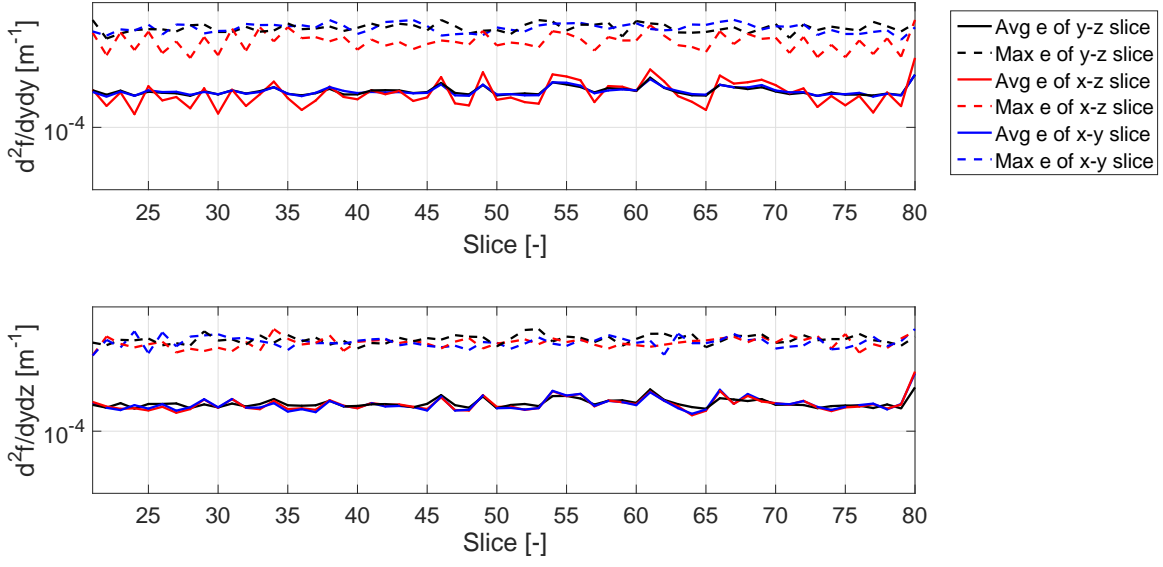
The images of the other derivatives of the comparison which neglected the 20 refined grid points close the the boundaries.



**Figure D.7:** The absolute difference between the first derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice. The maximum values of the functions  $df/dy$  and  $df/dz$  are  $\pm 20$  which results in a floating point precision of  $1.191e - 7$ .



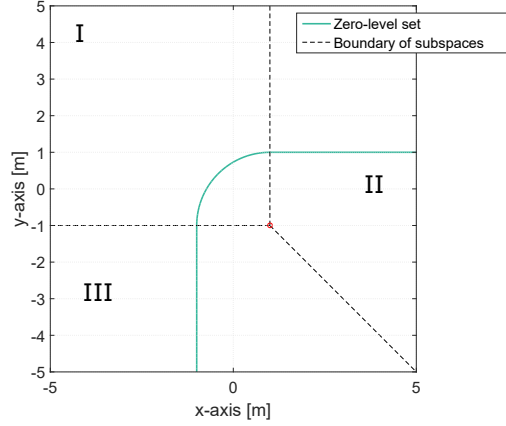
**Figure D.8:** The absolute difference between the second derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice. The maximum values of the second derivatives are  $\pm 900$  which results in a floating point precision of  $6.1e - 5$ .



**Figure D.9:** The absolute difference between the second derivatives of MATLAB and proposed interpolation visualized by a mean error and the maximum error of a slice. The maximum values of the second derivatives are  $\pm 900$  which results in a floating point precision of  $6.1e - 5$ .

## E Signed Distance Field of an edge

The Signed Distance Field (SDF) of the edge case can not be calculated using a single formula. Lets define the edge to be in  $z$ -direction and can be neglected in the SDF. Therefore, the SDF can be written as multiple functions in  $x$ - and  $y$ -direction. To calculate the SDF, the 2D space is divided into three different subspace which together describe the entire space. Figure E.1 shows the zero-level set of an edge together with the three different subspaces.



**Figure E.1:** The zero-level set of an edge and the three subspaces.

The first subspace describes the distance between the rounded part of the edge. In the Figure E.1 indicated as I. A point is within this subspace when

$$x \leq x_c \quad \text{and} \quad y \geq y_c, \quad (\text{E.1})$$

where  $x$  and  $y$  is the position and  $x_c$  and  $y_c$  is the center of the edge. The center of the edge is given in Figure E.1 as a red circle. The SDF in this subspace can be calculated with the following equation

$$SDF_I = \sqrt{(x - x_c)^2 + (y - y_c)^2} - r_e, \quad (\text{E.2})$$

where  $r_e$  is the radius of the edge. The second space is the subspace with the horizontal surface of the edge. In the Figure E.1 indicated as II. The boundaries of this subspace is fined as

$$x > x_c \quad \text{and} \quad y > y_c - (x - x_c) \quad (\text{E.3})$$

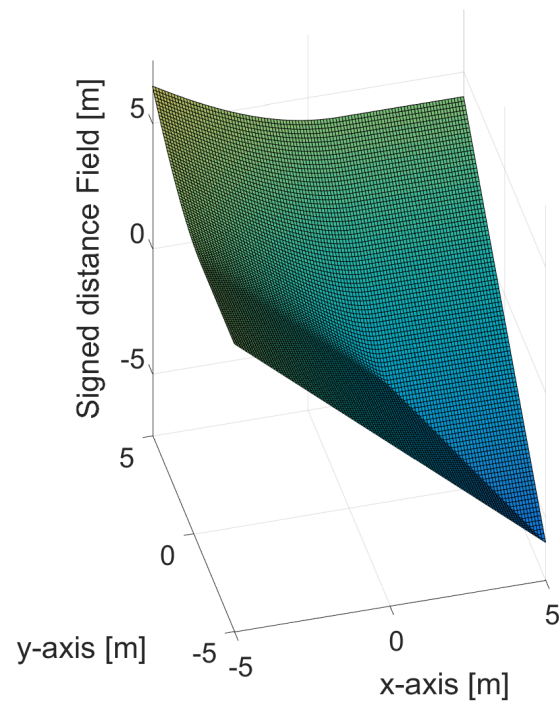
All the distances of this subspace can be calculated with the following equation

$$SDF_{II} = y - y_c - r_e. \quad (\text{E.4})$$

All the other points are in the last subspace, in Figure E.1 indicated with III. The distance to the vertical surface to these points are given by the following formula

$$SDF_{III} = x_c - x - r. \quad (\text{E.5})$$

These equation and subspaces result in a SDF for the entire 2D space. Figure E.2 shows an example of the SDF as a function of the position in space. If the edge is in  $z$ -direction, the SDF is the same for each position in  $z$ -direction.



**Figure E.2:** *The SDF of the edge.*

## F Parameters of dPRONTO algorithm

The values of the dPRONTO parameters for simulations are given in Table F.1. The values of the dPRONTO parameters for the KinectFusion data are given in Table F.2.

Parameter	Value
$Q_n$	$h^{-2}diag([10, 1, 10, 1, 10, 10])$
$t_f$	5s
$R$	$h^{-2}10^{-2}I$
$N$	100
$AbsTol$	$1e^{-1}$
$\delta_{step}$	0.1
$\delta_0$	0.1
$\alpha$	.4
$\beta$	0.3
$\epsilon_{step}$	0.1
$\epsilon_0$	0.1
$\epsilon_{min}$	$1e^{-3}$

**Table F.1:** The parameters for the dPRONTO algorithm for the simulations.

Parameter	Value
$Q_n$	$h^{-2}diag([100, 10, 100, 10, 100, 10])$
$t_f$	5s
$R$	$h^{-2}10^{-1}I$
$N$	100
$AbsTol$	$1e^{-1}$
$\delta_{step}$	0.1
$\delta_0$	0.01
$\alpha$	.4
$\beta$	0.3
$\epsilon_{step}$	0.1
$\epsilon_0$	0.1
$\epsilon_{min}$	$1e^{-3}$

**Table F.2:** The parameters for the dPRONTO algorithm for the KinFu data.

## Declaration concerning the TU/e Code of Scientific Conduct for the Master's thesis

I have read the TU/e Code of Scientific Conduct<sup>1</sup>.

I hereby declare that my Master's thesis has been carried out in accordance with the rules of the TU/e Code of Scientific Conduct

Date

25-1-2018

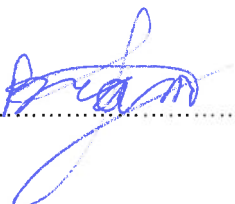
Name

Bram van de Schoot

ID-number

08739157

Signature



*Submit the signed declaration to the student administration of your department.*

<sup>1</sup> See: <http://www.tue.nl/en/university/about-the-university/integrity/scientific-integrity/>

The Netherlands Code of Conduct for Academic Practice of the VSNU can be found here also.

More information about scientific integrity is published on the websites of TU/e and VSNU