Comparing Least Squares Methods for Three-Dimensional Modelling

Utrecht University

Utrecht University

Bachelorthesis

Wink M. van Zon (5651387)

under supervision of
Dr. Tristan van Leeuwen

June 7, 2019

Abstract

In this thesis we will compare three numerical methods that are used for function approximation based on sampled input points, within the context of three-dimensional modelling, namely the Least Squares, Weighted Least Squares and Moving Least Squares methods. Firstly, we shall explain the workings of the three methods, find the algorithmic complexity of each method and show that finding a bound for the theoretical stability of the methods is difficult, but that we can numerically compute one. Next, we will examine how the methods are used in practice for three-dimensional modelling, by looking at their role in point cloud and polygon soup reconstruction. Finally, we will implement the methods and perform numerical experiments. These show how the accuracy, stability and speed of each method hold up in practice. We also implement the point cloud reconstruction algorithm discussed earlier, which shows us how the methods compare in a three-dimensional modelling situation. We conclude that the methods have different benefits in specific situations, but that further research is needed to definitively state which method is the best for certain applications within the field of three-dimensional modelling.

Contents

Chapter 1

Introduction

In three-dimensional modelling, a recurring problem is approximating objects within three-dimensional space. This problem can occur during the reconstruction of an object in real life, but it can also appear in less obvious situations, for example when interpolating an existing mesh or when constructing a new mesh by deforming some initial mesh. Many of these problems can be solved if we know the solution to the following numerical problem: If we have an unknown function $f : \Omega \to \mathbb{R}$ with $\Omega \subseteq \mathbb{R}^3$ and a set of N sampled points $\{\boldsymbol{x_i}\}_{i=1}^{N}$ together with their respective scalar values $\{y_i\}_{i=1}^{N}$, so that for each $i \in \{1, ..., N\}$ we have $f(\boldsymbol{x_i}) = y_i$, can we then find an approximation $\tilde{f}$ for f? Solving this problem using a fast and accurate method is essential for all of the tasks mentioned earlier.

Example 1.1. If we look in a single dimension on the domain $[-1, 1]$ and take the function $f(x) = x^2$, we can take $x_i = -1 + \frac{2}{5}(i-1)$ and $y_i = f(x_i) = x_i^2$ for $i \in \{1, ..., 5\}$ as equidistant input points and their respective scalar values. Figure 1 then shows the points that need to be interpolated and the result of a successful Least Squares interpolation.

The *Least Squares* (LS) method and its successors, the global *Weighted Least Squares* (WLS) and the *Moving Least Squares* (MLS) methods, are methods which have been developed to solve these problems. These methods have all been applied within the three-dimensional modelling field and it is generally known that the Weighted and Moving variants are more accurate than the original Least Squares method. We know this from earlier reasearch, as WLS was built on the theory of LS and MLS was built on the theory of WLS [7]. The accuracy of Moving Least Squares is also why it is used the most in practical situations. However, as the accuracy increases, the algorithmic complexity of these methods seems to

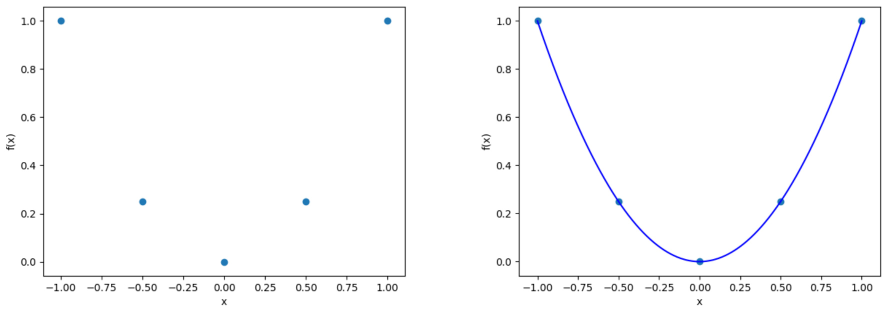


Figure 1.1: A set of equidistant points on the domain $[-1, 1]$ sampled from the function $f(x) = x^2$ is shown in the left figure. The right figure shows a Least Squares interpolation of order 2 of these input points.

increase as well. As WLS and MLS are originally local methods, they are never implemented naively within global implementations, seeing as this seems to greatly increase the needed computing time. It seems necessary to explore how the increased accuracy weighs off against the decrease in speed, which has not explicitly been done before.

In this thesis, we will examine and compare all three methods. We first check the quality of the approximation given by each method. Seeing as the error of the different approximations can be very dependent on the given set of sampled points, we will focus on the stability of each approximation method rather than on the actual error. Then we shall look at the algorithmic complexity of each method. Through this, a proper comparison of the accuracy versus the speed of each method can then be made. Through several numerical experiments, we will then establish if the theoretical values we have found for the complexity and accuracy of each method hold in practice. Finally, we shall test all three methods on a three-dimensional problem, to see how they compare in speed and accuracy when used on a common real-world scenario.

Chapter 2

Least Squares Methods and their Properties

In this chapter we will introduce the Least Squares, Weighted Least Squares and Moving Least Squares methods. For each method we will first see how a function is approximated from the input by explaining how the method works, using an explanation based on [10]. We then look at its stability by figuring out in which way the approximation changes when the input is shifted slightly. Finally, we will look at the algorithmic complexity of each method to see how fast it can produce an approximation, given the size of the input and values of certain parameters.

2.1 Least Squares

2.1.1 Method

The Least Squares Method seeks to minimise the error at each sampled point $\boldsymbol{x_i}$ in the least-squares sense, meaning that for a global approximation function $\tilde{f}(\boldsymbol{x})$, the sum of each squared error at an input point should be the smallest for $\tilde{f}$ compared to all functions within some chosen space of functions.

Definition 2.1. The space of polynomial functions of degree m in the pth dimension is written as Π_m^p.

We will take $\tilde{f}$ from the space of polynomials of degree m in three dimensions, so $\tilde{f} \in \Pi_m^3$. This is the most commonly used space for least-squares methods [10]. This results in the following definition:

Definition 2.2. The approximation $\tilde{f}$ produced by the Least Squares method is the $\tilde{f} \in \Pi_m^3$ so that the error $E_{LS}(\tilde{f}) = \sum_{0<i\leq N} ||\tilde{f}(\boldsymbol{x_i}) - y_i||^2$ is lesser or equal than $E_{LS}(h)$ for any other function $h \in \Pi_m^3$.

The approximation $\tilde{f}$ is therefore found by minimising the error function, which leads to the following equation:

$$\tilde{f} = \underset{h\in\Pi_m^3}{\operatorname{argmin}} \sum_{0<i\leq N} ||h(\boldsymbol{x_i}) - y_i||^2. \tag{2.1}$$

Notice that this minimum exists, as Π_m^3 is not empty. We will later show that the minimum is unique as long as N is greater than or equal to the amount of coefficients of h and all input points are distinct. We can now define each function $h \in \Pi_m^3$ as a dot product of a basis vector of the polynomial space Π_m^3, written as $\boldsymbol{\pi_m^3}(\boldsymbol{x})$ and a vector of unknown coefficients $\boldsymbol{c}$. Because $\boldsymbol{\pi_m^3}(\boldsymbol{x})$ is a known vector and $h(\boldsymbol{x}) = \boldsymbol{\pi_m^3}(\boldsymbol{x}) \cdot \boldsymbol{c}$, finding the $\boldsymbol{c_{min}} \in \mathbb{R}^k$ for which 2.1 holds is enough to calculate $\tilde{f}$. (The number k is the number of elements in $\boldsymbol{\pi_m^3}(\boldsymbol{x})$.)

Definition 2.3. The number of elements in $\boldsymbol{\pi_m^3}(\boldsymbol{x})$ is given by $\frac{1}{6}\prod_{i=1}^{3}(m+i)$. More generally, the vector $\boldsymbol{\pi_m^k}(\boldsymbol{x})$ contains $\frac{1}{k!}\prod_{i=1}^{k}(m+i)$ elements [5].

Example 2.4. For $m = 1$, we have $\boldsymbol{\pi_1^3}(\boldsymbol{x}) = [1, x, y, z]^T$. If $m = 2$, we have $\boldsymbol{\pi_2^3}(\boldsymbol{x}) = [1, x, y, z, x^2, xy, xz, y^2, yz, z^2]^T$ (if $\boldsymbol{x} = [x, y, z]^T$).

We can now take the partial derivatives of the error functional E_{LS}, with respect to the unknown coefficients $c_1, \ldots, c_k$, i.e. $\nabla E_{LS}(\boldsymbol{c})$, where $\nabla = [\partial/\partial c_1, \ldots, \partial/\partial c_k]$. For the coefficients $\boldsymbol{c_{min}}$ of the function $\tilde{f}$, the error functional should be minimal, which means that $\nabla E_{LS}(\boldsymbol{c_{min}}) = 0$. This allows us to compute $\boldsymbol{c_{min}}$ as follows: We know for each $0 < j \leq k$ that

$$\delta E_{LS}(\boldsymbol{c_{min}})/\delta c_j = \sum_i 2\boldsymbol{\pi_m^3}(\boldsymbol{x_i})_j[\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T\boldsymbol{c_{min}} - y_i] = 0. \tag{2.2}$$

This leads to the following in matrix-vector notation:

$$\sum_i 2\boldsymbol{\pi_m^3}(\boldsymbol{x_i})[\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T\boldsymbol{c_{min}} - y_i] = 2\sum_i \boldsymbol{\pi_m^3}(\boldsymbol{x_i})\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T\boldsymbol{c_{min}} - \boldsymbol{\pi_m^3}(\boldsymbol{x_i})y_i = 0, \tag{2.3}$$

which can then be rewritten as

$$\sum_i \boldsymbol{\pi_m^3}(\boldsymbol{x_i})\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T\boldsymbol{c_{min}} = \sum_i \boldsymbol{\pi_m^3}(\boldsymbol{x_i})y_i. \tag{2.4}$$

From this equation, $\boldsymbol{c_{min}}$ can be computed:

$$\boldsymbol{c_{min}} = [\sum_i \boldsymbol{\pi_m^3}(\boldsymbol{x_i})\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T]^{-1}\sum_i \boldsymbol{\pi_m^3}(\boldsymbol{x_i})y_i. \tag{2.5}$$

Assuming that $\sum_i \boldsymbol{\pi_m^3}(\boldsymbol{x_i})\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T$ is nonsingular, this method provides us with a definition for $\boldsymbol{c_{min}}$ (and therefore a definition for $\tilde{f}$) that can easily be calculated using numerical methods by performing the inversion and multiplications in equation 2.5.

We can simplify these equations by introducing the matrix A and the vector $\boldsymbol{y}$, where

$$A = \begin{pmatrix} \boldsymbol{\pi_m^3}(\boldsymbol{x_1})^T \\ \vdots \\ \boldsymbol{\pi_m^3}(\boldsymbol{x_N})^T \end{pmatrix} \text{ and } \boldsymbol{y} = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix}.$$

This allows us to write equation 2.5 in matrix-vector notation, which results in the equation

$$\boldsymbol{c_{min}} = [A^TA]^{-1}A^T\boldsymbol{y}. \tag{2.6}$$

Notice that we are solving the system of linear equations $A\boldsymbol{c_{min}} = \boldsymbol{y}$. From this we see that our earlier claim that the minimum described in 2.1 is only unique if $N \geq k$ is true (given that all the input points are distinct), as there have to be more than k independent rows in A to find a unique solution to this system. Because of this, A^TA will be singular if $N < k$, which means that the method cannot produce a result if there are too little input points for the given order of the polynomial space.

Example 2.5. Let us work out a one-dimensional example to demonstrate the Least Squares method. The function that is being approximated is $f(x) = x^2$ on the domain $[-1, 1]$. We sample this function at three points: $x_1 = -1, x_2 = 0$, $x_3 = 1$ and $N = 3$. This implies that

$y_1 = y_3 = 1$ and $y_2 = 0$. As f is a second-order function, we will approximate using a second-order polynomial space, so $m = 2$, meaning that our polynomial basis is $\boldsymbol{\pi_2^1} = [1, x, x^2]$. From this we can calculate that

$$A = \begin{pmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \end{pmatrix} = \begin{pmatrix} 1 & -1 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}, \; A^T A = \begin{pmatrix} 3 & 0 & 2 \\ 0 & 2 & 0 \\ 2 & 0 & 2 \end{pmatrix} \text{ and } [A^T A]^{-1} = \begin{pmatrix} 1 & 0 & -1 \\ 0 & \frac{1}{2} & 0 \\ -1 & 0 & \frac{3}{2} \end{pmatrix}.$$

Using equation 2.5 gives us that

$$\boldsymbol{c_{min}} = [A^T A]^{-1} A^T \boldsymbol{y} = \begin{pmatrix} 0 & 1 & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} \\ \frac{1}{2} & -1 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix}.$$

Finally, we know that $\tilde{f}(x) = \boldsymbol{\pi_2^1} \cdot \boldsymbol{c_{min}} = 0 + 0x + 1x^2 = x^2$. Our approximation is successful and equal to the original function.

2.1.2 Stability

In this section we will show the behaviour of the Least Squares approximation if the scalar values are slightly adjusted. We will show that the error is linear with respect to the adjusted input of the scalar values and we will examine the stability of the approximation by looking at the relative error of the coefficients.

We assume that applying the Least Squares method on a set of N sampled points $\{\boldsymbol{x_i}\}$ and their scalar values $\{y_i\}$ produces a certain approximation $\tilde{f}$. We now examine the result of using the Least Squares method on the same set of N sampled points $\{\boldsymbol{x_i}\}$ and slightly adjusted scalar values $\{y_i + \delta_i\}_{i=1}^N$, where each δ_i is a scalar value so that at least for one i we have $|\delta_i| > 0$. With this, we can explore the effect of errors within the scalar values on the resulting approximation. If the new errors are incorporated into the calculation, we get a new coefficient vector $\boldsymbol{c^*_{min}}$:

$$\boldsymbol{c^*_{min}} = [\sum_i \boldsymbol{\pi^3_m}(\boldsymbol{x_i}) \boldsymbol{\pi^3_m}(\boldsymbol{x_i})^T]^{-1} \sum_i \boldsymbol{\pi^3_m}(\boldsymbol{x_i})(y_i + \delta_i)$$

or in the matrix-vector notation we have

$$\boldsymbol{c^*_{min}} = [A^T A]^{-1} A^T (\boldsymbol{y} + \boldsymbol{\delta})$$

where

$$\boldsymbol{\delta} = \begin{pmatrix} \delta_1 \\ \vdots \\ \delta_N \end{pmatrix}.$$

We can rewrite this as

$$\boldsymbol{c^*_{min}} = ([A^T A]^{-1} A^T \boldsymbol{y}) + ([A^T A]^{-1} A^T \boldsymbol{\delta}) = \boldsymbol{c_{min}} + ([A^T A]^{-1} A^T \boldsymbol{\delta}). \tag{2.7}$$

This results in the following for the new approximation $\tilde{f}^*$:

$$\begin{aligned} \tilde{f}^*(\boldsymbol{x}) &= \boldsymbol{\pi^3_m}(\boldsymbol{x}) \cdot \boldsymbol{c^*_{min}} \\ &= \boldsymbol{\pi^3_m}(\boldsymbol{x}) \cdot (\boldsymbol{c_{min}} + ([A^T A]^{-1} A^T \boldsymbol{\delta})) \\ &= \boldsymbol{\pi^3_m}(\boldsymbol{x}) \cdot \boldsymbol{c_{min}} + \boldsymbol{\pi^3_m}(\boldsymbol{x}) \cdot ([A^T A]^{-1} A^T \boldsymbol{\delta}) \\ &= \tilde{f}(\boldsymbol{x}) + \boldsymbol{\pi^3_m}(\boldsymbol{x}) \cdot ([A^T A]^{-1} A^T \boldsymbol{\delta}). \end{aligned} \tag{2.8}$$

From this we can see that the difference between the original approximation $\tilde{f}$ and the new approximation $\tilde{f}^*$ is given by $|\tilde{f} - \tilde{f}^*| = |\boldsymbol{\pi^3_m}(\boldsymbol{x}) \cdot ([A^TA]^{-1}\sum_i A\boldsymbol{\delta})|$. This is exactly the absolute value of a Least Squares approximation function of the original input points and $\{\delta_i\}_{i=1}^N$ as the given scalar values.

To get an idea of the stability of the approximation as a whole we can observe the stability of the coefficients. We do this by calculating the relative error of the coefficient vector:

$$\frac{||\boldsymbol{c_{min}} - \boldsymbol{c^*_{min}}||}{||\boldsymbol{c_{min}}||} = \frac{||[A^TA]^{-1}A^T\boldsymbol{\delta}||}{||\boldsymbol{c_{min}}||}. \tag{2.9}$$

From equation 2.1 we know that

$$\boldsymbol{c_{min}} = \underset{\boldsymbol{c}\in\mathbb{R}^k}{\operatorname{argmin}} ||A\boldsymbol{c} - \boldsymbol{y}||^2.$$

This means that $A^TA\boldsymbol{c_{min}} = A^T\boldsymbol{y}$. From this we find that $||A^T\boldsymbol{y}|| = ||A^TA\boldsymbol{c_{min}}|| \leq ||A^TA|| \cdot ||\boldsymbol{c_{min}}||$ and that therefore

$$||\boldsymbol{c_{min}}|| \geq \frac{||A^T\boldsymbol{y}||}{||A^TA||}. \tag{2.10}$$

If we now combine equation 2.9 with our new lower bound on $||\boldsymbol{c_{min}}||$, we can acquire an upper bound on the relative error between $\boldsymbol{c_{min}}$ and $\boldsymbol{c^*_{min}}$, namely

$$\begin{aligned}\frac{||\boldsymbol{c_{min}} - \boldsymbol{c^*_{min}}||}{||\boldsymbol{c_{min}}||} &\leq \frac{||A^TA|| \cdot ||[A^TA]^{-1}A^T\boldsymbol{\delta}||}{||A^T\boldsymbol{y}||} \leq \frac{||[A^TA]|| \cdot ||[A^TA]^{-1}|| \cdot ||A^T|| \cdot ||\boldsymbol{\delta}||}{||A^T\boldsymbol{y}||} \\ &= \frac{||A^T|| \cdot ||\boldsymbol{\delta}||}{||A^T\boldsymbol{y}||} \leq \frac{||(A^T)^{-1}|| \cdot ||A^T|| \cdot ||\boldsymbol{\delta}||}{||(A^T)^{-1}A^T\boldsymbol{y}||} = ||(A^T)^{-1}|| \cdot ||A^T|| \frac{||\boldsymbol{y} - (\boldsymbol{y} + \boldsymbol{\delta})||}{||\boldsymbol{y}||}.\end{aligned} \tag{2.11}$$

From this we see that the condition number of A^T, written as $\kappa(A^T) = \kappa(A) = ||(A^T)^{-1}|| \cdot ||A^T||$, determines the stability of the coefficients. Notice that this is only true if A^T is nonsingular, which means that $N = m$. If $N \neq m$, we know that $\kappa(A^T) = \frac{\sigma_{max}(A^T)}{\sigma_{min}(A^T)}$ where $\sigma_{max}(A^T)$ and $\sigma_{min}(A^T)$ are the maximal and minimal singular values of A^T. (This holds for the Euclidean norm.) In this case, we still find that

$$\frac{||A^T|| \cdot ||\boldsymbol{\delta}||}{||A^T\boldsymbol{y}||} \leq \frac{\sigma_{\max}(A^T)||\boldsymbol{\delta}||}{\sigma_{\min}(A^T)||\boldsymbol{y}||} = \kappa(A)\frac{||\boldsymbol{y} - (\boldsymbol{y} + \boldsymbol{\delta})||}{||\boldsymbol{y}||}. \tag{2.12}$$

Example 2.6. If the problem is reduced to one dimension and we assume that $N = k$, which can be realised in some situations where it is possible to choose the polynomial space so that k is equal to the amount of sampled points, A^T takes on the form of

$$A^T = \begin{pmatrix} 1 & \dots & 1 \\ x_1 & \dots & x_N \\ x_1^2 & \dots & x_N^1 \\ \vdots & \vdots & \vdots \\ x_N^N & \dots & x_N^N \end{pmatrix}.$$

We see that this is a Vandermonde matrix $V(x_1, \dots, x_N)$ (or the transpose of a Vandermonde matrix according to some authors). For these Vandermonde matrices, lower bounds for the condition number $\kappa(V(x_1, \dots, x_N))$ exist in literature [6, 13]. Tyrtyshnikov [13] states that $\kappa(V(x_1, \dots, x_N)) \geq \frac{2^{N-2}}{N^{\frac{1}{2}}}$ for the Euclidean norm, for example. These lower bounds are useful for finding the optimal stability of the approximation, but they are not upper bounds, meaning that we have no clue as to how bad the stability can get when faced with unfortunate input points. An exception to this is the situation in which we have so little sample points that $N < k$, which means that $\kappa(A)$ will be infinite; in this situation we know that the stability is as bad as can be. This is not very useful information, as the algorithm can not produce a solution in this situation.

A problem is that when $N \neq k$ or in higher dimensions, the matrix A^T (or A) is not a Vandermonde matrix, which makes calculating the same lower bound for the number $\kappa(A)$ impossible. However, it is possible to numerically approximate $\kappa(A)$ to get an indication of the stability of the approximation, even in three dimensions.

In the final calculation of $\tilde{f}$, the coefficients are combined with the polynomial basis to create the approximated function. The stability of the coefficients gives an indication of the stability of the function, but it should be said that the general stability of polynomials might also have a big influence on the stability of the final approximation.

2.1.3 Complexity

To determine the speed of the Least Squares method, we shall calculate its theoretical algorithmic complexity. For this we will again examine equation 2.5. We know that naive matrix multiplication of two matrices of unequal dimensions has a complexity of $\mathcal{O}(nmp)$, where n is the height of the first matrix, m is the width of the first matrix and p is the width of the second matrix. For square $n \times n$ matrices, this results in the complexity $\mathcal{O}(n^\omega)$, where ω can be between 2.376 and 3, depending on if the naive method or other improved algorithms are used [1].

Theorem 2.7. *The complexity of the Least Squares algorithm is $\mathcal{O}(Nk^2)$ if $N \geq k^{\omega-2}$, otherwise the complexity is $\mathcal{O}(k^\omega)$.*

Remember that there are N sampled points and that $\boldsymbol{\pi_m^3}(\boldsymbol{x_i})$ has a size of k, where $k = \frac{1}{6}\prod_{i=1}^{3}(m+i)$ [5]. This means that multiplying $\boldsymbol{\pi_m^3}(\boldsymbol{x_i})\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T$ for any i has an algorithmic complexity of $\mathcal{O}(k^2)$ and multiplying $\boldsymbol{\pi_m^3}(\boldsymbol{x_i})y_i$ has an algorithmic complexity of $\mathcal{O}(k)$. This leads to algorithmic complexities of $\mathcal{O}(Nk^2)$ and $\mathcal{O}(Nk)$ for calculating $\sum_i \boldsymbol{\pi_m^3}(\boldsymbol{x_i})\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T$ and $\sum_i \boldsymbol{\pi_m^3}(\boldsymbol{x_i})y_i$ respectively. Inverting the square $k \times k$ matrix $\sum_i \boldsymbol{\pi_m^3}(\boldsymbol{x_i})\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T$ has a complexity of $\mathcal{O}(k^\omega)$ and multiplying this matrix with $\sum_i \boldsymbol{\pi_m^3}(\boldsymbol{x_i})y_i$ has a complexity of $\mathcal{O}(Nk^2)$. Finally, we multiply $\boldsymbol{\pi_m^3}(\boldsymbol{x})$ and $\boldsymbol{c_{min}}$ to obtain $\tilde{f}$, an operation with algorithmic complexity of $\mathcal{O}(k^2)$. From this we see that the exact complexity of the Least Squares method varies depending on the size of N, k and ω and we have proven the theorem.

If we assume that all matrix calculations are done naively, we can now say that the complexity of the Least Squares method is $\mathcal{O}(k^3)$ if $N < k$. However, as we know there is no solution to the algorithm if $N < k$, so the complexity is always $\mathcal{O}(Nk^2)$ if the calculations are done naively.

2.2 Weighted Least Squares

2.2.1 Method

The Weighted Least Squares method works similar to the Least Squares method, in that it seeks to minimise the error at each point $\boldsymbol{x_i}$ in the least-squares sense, but it adds a weighting function dependent on the distance of each point to a certain pivot-point $\boldsymbol{\hat{x}}$. This leads to the following definition:

Definition 2.8. The approximation $\tilde{f}$ produced by the Weighted Least Squares method is the $\tilde{f} \in \Pi_m^3$ so that the error $E_{WLS}(\tilde{f}) = \sum_{0<i\leq N} ||\tilde{f}(\boldsymbol{x_i}) - y_i||^2\theta(||\boldsymbol{\hat{x}} - \boldsymbol{x_i}||)$ is lesser than or equal to $E_{WLS}(h)$ for any other function $h \in \Pi_m^3$, where θ is a chosen weighting function.

The approximation $\tilde{f}$ is again found by minimising the error function, which leads to the following equation:

$$\tilde{f} = \underset{h\in\Pi_m^3}{\operatorname{argmin}} \sum_{0<i\leq N} ||h(\boldsymbol{x_i}) - y_i||^2\theta(||\boldsymbol{\hat{x}} - \boldsymbol{x_i}||) \tag{2.13}$$

As we can see, this does not have to produce an accurate result, especially if just a single pivot point $\boldsymbol{\hat{x}}$ is chosen at random for the entire function. This method is therefore a local method. It can however be extended to a global method [12, 10]. If n fixed points $\boldsymbol{\hat{x}_j}$ with $0 < j \leq n$ are chosen, we can construct a global approximation over the entire domain $\mathbb{R}^3$, by ensuring that each point within the domain $\Omega \subseteq \mathbb{R}^3$ is covered by one of the local approximations. If we define θ_i as the weight function chosen to calculate the WLS approximation around pivot-point $\boldsymbol{\hat{x}_i}$ (meaning that $\theta_i(\boldsymbol{x}) = \theta(||\boldsymbol{x} - \boldsymbol{\hat{x}_i}||)$), then the condition that

$$\Omega = \bigcup_j \operatorname{supp}(\theta_j)$$

will lead to a global approximation. The weighting of these local approximations can be done properly by construction a partition of unity from the θ_i [12]. This leads to

$$\phi_j(\boldsymbol{x}) = \frac{\theta_j(\boldsymbol{x})}{\sum_{k=1}^{k=n}\theta_k(\boldsymbol{x})} \tag{2.14}$$

so that $\sum_i \phi_i = 1$ over the entire domain $\Omega \subseteq \mathbb{R}^3$ of the original function f.

Similar to Least Squares, if we define $\tilde{f}_j$ as the local approximation corresponding to pivot-point $\boldsymbol{\hat{x}_j}$, we can say that $\tilde{f}_j = \boldsymbol{\pi_m^3}(\boldsymbol{x})\cdot\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})$. The coefficients $\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})$ are now weighted by their distance to $\boldsymbol{\hat{x}_j}$ and therefore a function of $\boldsymbol{\hat{x}_j}$. If we take the partial derivatives of E_{WLS} (from equation 2.13) with respect to the unknown coefficients $\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})$, we find that

$$\begin{gathered}\sum_i \theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_i}||)2\boldsymbol{\pi_m^3}(\boldsymbol{x_i})[\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) - y_i] = \\ 2\sum_i \theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_i}||)\boldsymbol{\pi_m^3}(\boldsymbol{x_i})\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) - \theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_i}||)\boldsymbol{\pi_m^3}(\boldsymbol{x_i})y_i = 0.\end{gathered} \tag{2.15}$$

From this we obtain

$$\sum_i \theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_i}||)\boldsymbol{\pi_m^3}(\boldsymbol{x_i})\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) = \theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_i}||)\boldsymbol{\pi_m^3}(\boldsymbol{x_i})y_i. \tag{2.16}$$

Similar to 2.5, we then get an expression for $\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})$:

$$\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) = [\sum_i \theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_i}||)\boldsymbol{\pi_m^3}(\boldsymbol{x_i})\boldsymbol{\pi_m^3}(\boldsymbol{x_i})^T]^{-1}\sum_i \theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_i}||)\boldsymbol{\pi_m^3}(\boldsymbol{x_i})y_i. \tag{2.17}$$

If we use the notation used in equation 2.6 and define a new matrix

$$\Theta(\boldsymbol{\hat{x}_j}) = \begin{pmatrix} \theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_1}||) & \dots & 0 \\ \vdots & \ddots & \dots \\ 0 & \dots & \theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_N}||) \end{pmatrix},$$

we can also simplify equation 2.17. This gives us

$$\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) = [A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{y}) \tag{2.18}$$

This provides us with a way to calculate each local set of coefficients $\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})$. To then compute the global function $\tilde{f}$, we combine 2.17 and the weighting function in 2.14 to find that

$$\begin{aligned}\tilde{f}(\boldsymbol{x}) &= \sum_j \phi_j(\boldsymbol{x})\boldsymbol{\pi^3_m}(\boldsymbol{x})^T\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) \\ &= \sum_j \phi_j(x)\boldsymbol{\pi^3_m}(\boldsymbol{x})^T([A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{y})).\end{aligned} \tag{2.19}$$

Just as with Least Squares, we now have an expression that can be calculated numerically and provides an approximation $\tilde{f}$. Notice that the method can only produce a result if k or more unique input points contribute to each local set of coefficients $\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})$, as we are again solving a linear system. This means, if all input points are unique, that for each $\boldsymbol{\hat{x}_j}$, N minus the amount of weighting functions $\theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_i}||)$ that are equal to zero (for $0 < i \leq N$) needs to be greater than or equal to k.

Example 2.9. To demonstrate Weighted Least Squares, we will look at the same problem as in example 2.5. We take the input points as pivot points with $\hat{x}_i = x_i$ for $0 < i \leq 3$ and a simple weighting function such as the Wendland function [14]: $\theta(d) = (1 - d/h)^4(4d/h + 1)$. This function is well-defined when $d \in [0, h]$, so it is good if h is bigger than the biggest distance between a pivot point and a sample point. We therefore take $h = 4$. This gives us $\theta(d) = (1 - d/4)^4(d + 1)$, from which we can calculate that

$$\boldsymbol{\Theta}(\hat{x}_1) = \begin{pmatrix}1 & 0 & 0\\ 0 & 81/128 & 0\\ 0 & 0 & 3/16\end{pmatrix}, \boldsymbol{\Theta}(\hat{x}_2) = \begin{pmatrix}81/128 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 81/128\end{pmatrix}, \boldsymbol{\Theta}(\hat{x}_3) = \begin{pmatrix}3/16 & 0 & 0\\ 0 & 81/128 & 0\\ 0 & 0 & 1\end{pmatrix}.$$

Using equation 2.17 and the matrices A and A^T we calculated in 2.5, we find that

$$\boldsymbol{c_{min}}(\hat{x}_1) = \begin{pmatrix}233/128 & -13/16 & 19/16\\ -13/16 & 19/16 & -13/16\\ 19/16 & -13/16 & 19/16\end{pmatrix}^{-1}\begin{pmatrix}1 & 81/128 & 3/16\\ -1 & 0 & 3/16\\ 1 & 0 & 3/16\end{pmatrix}\begin{pmatrix}1\\0\\1\end{pmatrix} = \begin{pmatrix}0\\0\\1\end{pmatrix}.$$

Similarly we can calculate that

$$\boldsymbol{c_{min}}(\hat{x}_2) = \boldsymbol{c_{min}}(\hat{x}_3) = \begin{pmatrix}0\\0\\1\end{pmatrix}.$$

Using 2.14 we know that $\sum_i \phi_i(x) = 1$ for any x and so we know that $\sum_i \phi_i(x)\boldsymbol{\pi^1_2}(\boldsymbol{x})^T\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_i}) = x^2$, as

$$\boldsymbol{\pi^1_2}(\boldsymbol{x})^T\begin{pmatrix}0\\0\\1\end{pmatrix} = x^2.$$

Normally we would calculate each ϕ_i individually, but this tends to get complicated quickly and is not necessary for this example, as we already see that we get the correct approximation.

Example 2.10. In figure 2.2.1 we can see how the final result of the Weighted Least Squares method is a weighting of the functions that are created per pivot point using the coefficients $\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_i})$. It is very visible that each of these functions is only a locally weighted approximation. The figure shows that as input points get further away from the pivot point, they contribute less to the local function. This is why the local function for the middle pivot point is so similar to the final approximation.

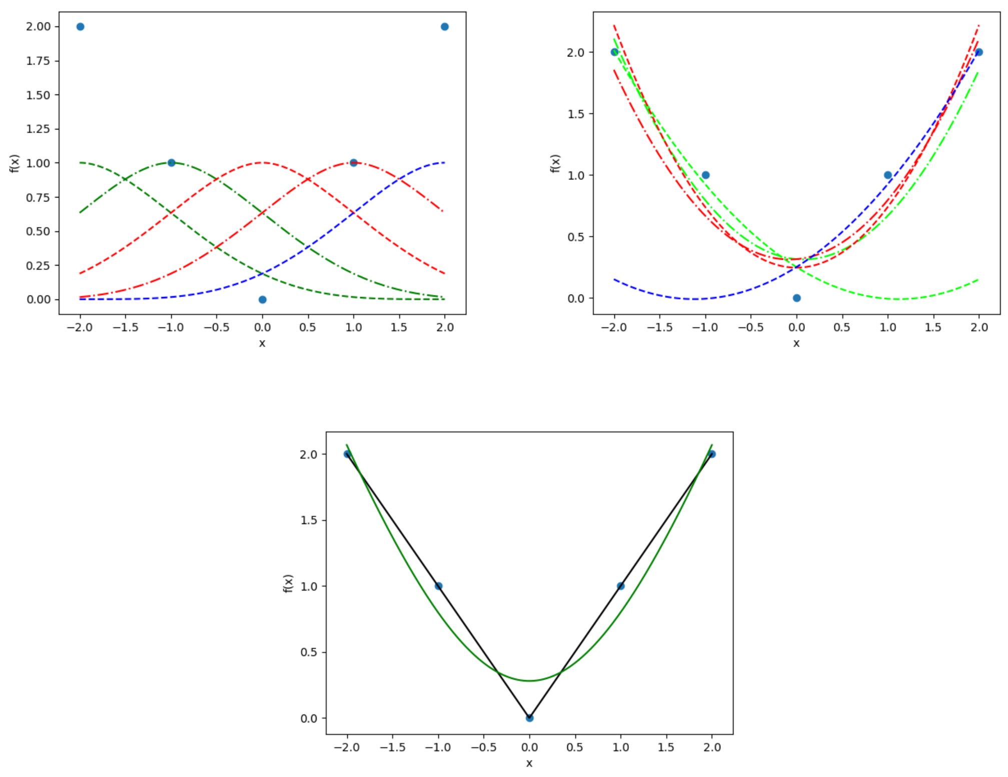


Figure 2.1: A set of equidistant points on the domain $[-2, 2]$ sampled from the function $f(x) = |x|$ is shown in both figures. The bottom figure shows the Weighted Least Squares interpolation of order 2 of these input points, with a Wendland function using $h = 4$, using the input points as pivot points. The top left figure plots the weighting function $\theta_j(x)$ for each pivot point. The top right figure shows the individual functions generated by multiplying the coefficient vector of each pivot point with the polynomial basis vector.

2.2.2 Stability

As we did with the Least Squares method, we will now look at the behaviour of the Weighted Least Squares approximation as the scalar values are slightly adjusted. We will show that the error is again linear with respect to the adjusted input of the scalar values. We will then look at the relative error of each coefficient vector belonging to a pivot point to determine the stability of the approximation.

To examine the stability, let us again observe the behaviour of this approximation on input points $\{\boldsymbol{x_i}\}$ and slightly adjusted scalar values $\{y_i + \delta_i\}$. First we find a new coefficient vector for each pivot point $\boldsymbol{\hat{x}_j}$:

$$\begin{aligned}\boldsymbol{c^*_{min}}(\boldsymbol{\hat{x}_j}) &= [A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})(\boldsymbol{y}+\boldsymbol{\delta}))\\ &= [A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{y}T + A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{\delta})\\ &= [A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{y}) + [A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{\delta})\\ &= \boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) + [A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{\delta}).\end{aligned} \tag{2.20}$$

We then find that the new approximation $\tilde{f}^*$ is given by

$$\begin{aligned}\tilde{f}^*(\boldsymbol{x}) &= \sum_j \phi_j(\boldsymbol{x})[\boldsymbol{\pi^3_m}(\boldsymbol{x})]^T(\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) + [A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{\delta}))\\ &= \sum_j \phi_j(\boldsymbol{x})[\boldsymbol{\pi^3_m}(\boldsymbol{x})]^T\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) + \sum_j \phi_j(\boldsymbol{x})[\boldsymbol{\pi^3_m}(\boldsymbol{x})]^T([A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{\delta}))\\ &= \tilde{f}(\boldsymbol{x}) + \sum_j \phi_j(\boldsymbol{x})[\boldsymbol{\pi^3_m}(\boldsymbol{x})]^T([A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{\delta})).\end{aligned} \tag{2.21}$$

As seen before with Least Squares, the Weighted Least Squares approximation is linear in its error, meaning that the difference between the initial approximation and the new approximation $|\tilde{f}(\boldsymbol{x}) - \tilde{f}^*(\boldsymbol{x})|$ is equal to a Weighted Least Squares approximation at the same points $\{\boldsymbol{x_i}\}$ with the adjustment values as the scalar values $\{\delta_i\}_{i=1}^N$.

To determine the stability, we use the stability of the coefficient vectors by examining the relative error of each coefficient vector. For a random $0 < j \leq N$, we have

$$\frac{||\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) - \boldsymbol{c^*_{min}}(\boldsymbol{\hat{x}_j})||}{||\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})||} = \frac{||[A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{\delta})||}{||\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})||}. \tag{2.22}$$

We know from the definition of Weighted Least Squares that

$$\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) = \underset{\boldsymbol{c}\in\mathbb{R}^k}{\operatorname{argmin}} ||\Theta(\boldsymbol{\hat{x}_j})A\boldsymbol{c} - \Theta(\boldsymbol{\hat{x}_j})\boldsymbol{y}||^2,$$

where $\boldsymbol{y}$ is the vector composed of the values y_i. This gives us that $A^T\Theta(\boldsymbol{\hat{x}_j})A\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) = A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{y}$. From this we see that $||A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{y}|| = ||A^T\Theta(\boldsymbol{\hat{x}_j})A\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})|| \leq ||A^T\Theta(\boldsymbol{\hat{x}_j})A|| \cdot ||\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})||$ and this gives us a result similar to 2.10:

$$||\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})|| \geq \frac{||A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{y}||}{||A^T\Theta(\boldsymbol{\hat{x}_j})A||}. \tag{2.23}$$

If we now combine equation 2.22 with this result, we see that

$$\begin{aligned}\frac{||\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j}) - \boldsymbol{c^*_{min}}(\boldsymbol{\hat{x}_j})||}{||\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})||} &\leq \frac{||A^T\Theta(\boldsymbol{\hat{x}_j})A|| \cdot ||[A^T\Theta(\boldsymbol{\hat{x}_j})A]^{-1}(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{\delta})||}{||A^T\boldsymbol{y}||} \leq \frac{||(A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{\delta})||}{||A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{y}||}\\ &\leq \frac{||(A^T\Theta(\boldsymbol{\hat{x}_j}))^{-1}|| \cdot ||A^T\Theta(\boldsymbol{\hat{x}_j})|| \cdot ||\boldsymbol{\delta}||}{||(A^T\Theta(\boldsymbol{\hat{x}_j}))^{-1}A^T\Theta(\boldsymbol{\hat{x}_j})\boldsymbol{y}||} \leq \kappa(A^T\Theta(\boldsymbol{\hat{x}_j}))\frac{||\boldsymbol{y} - (\boldsymbol{y}+\boldsymbol{\delta})||}{||\boldsymbol{y}||}.\end{aligned} \tag{2.24}$$

The stability of the coefficient vector is dependent on $\kappa(A^T\Theta(\boldsymbol{\hat{x}_j})) = \kappa(\Theta(\boldsymbol{\hat{x}_j})A)$. This immediately reveals the probable benefit of using Weighted Least Squares over Least Squares. Notice that $\kappa(\Theta(\boldsymbol{\hat{x}_j})A) = ||\Theta(\boldsymbol{\hat{x}}_j)A|| \cdot ||(\Theta(\boldsymbol{\hat{x}}_j)A)^{-1}||$. We know that a weighting function θ is generally chosen so that $|\theta(d)| \leq 1$ for any d. Therefore, if we look at either the 1-norm or 2-norm of matrices, we know that $||\Theta(\boldsymbol{\hat{x}_j})||_1$ is equal to the maximum absolute column sum of $\Theta(\boldsymbol{\hat{x}_j})$, which is therefore equal to 1 and because $||\Theta(\boldsymbol{\hat{x}_j})||_2$ is equal to the maximum absolute row sum of $\Theta(\boldsymbol{\hat{x}_j})$, it is also equal to 1. We therefore get both

$$\kappa(\Theta(\boldsymbol{\hat{x}_j})A) \leq ||\Theta(\boldsymbol{\hat{x}_j})||_1 \cdot ||A||_1 \cdot ||(\Theta(\boldsymbol{\hat{x}}_j)A)^{-1}||_1 \leq ||A||_1 \cdot ||(\Theta(\boldsymbol{\hat{x}}_j)A)^{-1}||_1$$

and

$$\kappa(\Theta(\boldsymbol{\hat{x}_j})A) \leq ||\Theta(\boldsymbol{\hat{x}_j})||_2 \cdot ||A||_2 \cdot ||(\Theta(\boldsymbol{\hat{x}}_j)A)^{-1}||_2 \leq ||A||_2 \cdot ||(\Theta(\boldsymbol{\hat{x}}_j)A)^{-1}||_2.$$

If we then find that $||(\Theta(\boldsymbol{\hat{x}}_j)A)^{-1}||_1 \leq ||A^{-1}||_1$ or $||(\Theta(\boldsymbol{\hat{x}}_j)A)^{-1}||_2 \leq ||A^{-1}||_2$, we have that $\kappa(A) \leq \kappa(\Theta(\boldsymbol{\hat{x}}_j)A)$ under the 1-norm or 2-norm respectively. This means that the Weighted Least Squares method has the potential to have more stable coefficients than the Least Squares method, dependent on θ and the different pivot points $\{\boldsymbol{\hat{x}_j}\}_{j=1}^n$.

Finally, as we saw in equation 2.19, all coefficient vectors are combined with the polynomial base to create a polynomial per pivot point. As with Least Squares, the general stability of polynomials will have an influence on the stability of the approximation due to this step. After this, all the polynomials per pivot point are combined by averaging them using the $\phi_j(\boldsymbol{x})$ functions from equation 2.14. This means that if some of the pivot point polynomials have instabilities, the averaging will help even out these instabilities.

2.2.3 Complexity

To determine the complexity of Weighted Least Squares, we first examine equation 2.17 to find the algorithmic complexity of calculating $\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})$ for one single pivot-point $\boldsymbol{\hat{x}_j}$. As we can see, the equation is similar to 2.5, but we can see that calculating $\theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_i}||)\boldsymbol{\pi^3_m}(\boldsymbol{x_i})\boldsymbol{\pi^3_m}(\boldsymbol{x_i})^T$ has an algorithmic complexion of $\mathcal{O}(k^3)$. From this we can conclude that the complexity of the calculation of $\sum_i \theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_i}||)\boldsymbol{\pi^3_m}(\boldsymbol{x_i})\boldsymbol{\pi^3_m}(\boldsymbol{x_i})^T$ is $\mathcal{O}(Nk^3)$. As other operations such as the inversion or the multiplication of the results of both sums in equation 2.17 have the same complexity as in 2.5 and calculating $\sum_i \theta(||\boldsymbol{\hat{x}_j} - \boldsymbol{x_i}||)\boldsymbol{\pi^3_m}(\boldsymbol{x_i})y_i$ has a complexity of $\mathcal{O}(Nk^2)$, we see that the complexity of calculating $\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})$ is of the order $\mathcal{O}(Nk^3)$. This is already a lot larger than the calculation of $\boldsymbol{c_{min}}$ in the Least Squares method.

Theorem 2.11. *The complexity of the global Weighted Least Squares algorithm is $\mathcal{O}(nNk^3)$.*

We know this because we have to calculate $\phi_j(\boldsymbol{x})\boldsymbol{c_{min}}(\boldsymbol{\hat{x}_j})$ for each $0 < j \leq n$, meaning that we have to perform the calculation as seen in 2.17 n times. The size of n depends on the points that are chosen as pivots. This shows that it might be necessary, before starting the actual approximation, to invest resources in pre-computing an efficient partition of unity of the domain with well-chosen pivot points to speed up computation for the Weighted Least Squares method.

2.3 Moving Least Squares

2.3.1 Method

Finally, the Moving Least Squares method is very similar to the Weighted Least Squares method, but instead of taking one fixed pivot-point within $\mathbb{R}^3$, the method moves this point continuously through $\mathbb{R}^3$. This means that for each point, an approximation pivoted around this point is calculated individually. This leads to the following definition:

Definition 2.12. The approximation $\tilde{f}$ produced by the Moving Least Squares method is the $\tilde{f} \in \Pi_m^3$ so that $\tilde{f}(\boldsymbol{x}) = \tilde{f}_{\boldsymbol{x}}(\boldsymbol{x})$ and the error $E_{MLS}(\tilde{f}_{\boldsymbol{x}}) = \sum_{0<i\leq N} ||\tilde{f}_{\boldsymbol{x}}(\boldsymbol{x_i}) - y_i||^2\theta(||\boldsymbol{x} - \boldsymbol{x_i}||)$ is lesser or equal than $E_{MLS}(h)$ for any other function $h \in \Pi_m^3$, where θ is a chosen weighting function.

Instead of constructing a global Weighted Least Squares method through a partition of unity, the Moving least Squares method provides a global method by continuously approximating over the entire domain $\mathbb{R}^3$ - that is, with each point within it as a pivot point.
We can now use 2.17 to calculate $\boldsymbol{c_{min}}(\boldsymbol{x})$ for each $\boldsymbol{x} \in \mathbb{R}^3$ continuously to extract each local function $\tilde{f}_{\boldsymbol{x}}(\boldsymbol{x}) = \boldsymbol{\pi_m^3}(\boldsymbol{x}) \cdot \boldsymbol{c_{min}}(\boldsymbol{x})$. As before, we now have a way to calculate the approximation $\tilde{f}$, this time using Moving Least Squares. Again, each function $\tilde{f}_{\boldsymbol{x}}$ needs to have k or more unique input points contributing to its coefficients for the method to produce a result.

Example 2.13. If we look at the problem described in examples 2.5 and 2.9 the difference between Moving Least Squares and the previous methods becomes clear. In previous methods, we could calculate an elementary expression for the approximation $\tilde{f}(x)$. With MLS however, the approximation is always dependent on the point that we are sampling. To keep this example simple we therefore choose a point in the approximation that we would like to sample, say $x = \frac{1}{2}$ and calculate $\tilde{f}(\frac{1}{2}) = \tilde{f}_{\frac{1}{2}}(\frac{1}{2})$. We use the same weighting function as we used with example 2.9, so

$$\Theta(\frac{1}{2}) = \begin{pmatrix} \frac{3125}{8192} & 0 & 0 \\ 0 & \frac{7203}{8192} & 0 \\ 0 & 0 & \frac{7203}{8192} \end{pmatrix}$$

which gives us

$$\boldsymbol{c_{min}}(\frac{1}{2}) = \begin{pmatrix} a+2b & b-a & a+b \\ b-a & a+b & b-a \\ a+b & b-a & a+b \end{pmatrix}^{-1} \begin{pmatrix} a & b & b \\ -a & 0 & b \\ a & 0 & b \end{pmatrix} \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix},$$

where $a = \frac{3125}{8192}$ and $b = \frac{7203}{8192}$. We know that $\tilde{f}(\frac{1}{2}) = \tilde{f}_{\frac{1}{2}}(x) = \boldsymbol{\pi_2^1}(x) \cdot \boldsymbol{c_{min}}(x) = x^2$, so $\tilde{f}(\frac{1}{2}) = (\frac{1}{2})^2$. We again find a precise approximation.

2.3.2 Stability

Just as with the two earlier methods, we will observe the behaviour of the Moving Least Squares approximation as the scalar values are adjusted. The error is linear in the adjusted scalar input, as before, which we will show. Finally, we will look at the relative error of the coefficient vector at each point to get an idea of the stability of the Moving Least Squares approximation.

We will again look at the approximation on input points $\{\boldsymbol{x_i}\}$ and slightly adjusted scalar values $\{y_i + \delta_i\}$. As the calculation of the coefficient vector is exactly the calculation of the coefficient vector for a pivot point in the Weighted Least Squares method, we can use 2.20 to find that

$$\boldsymbol{c^*_{min}}(\boldsymbol{x}) = \boldsymbol{c_{min}}(\boldsymbol{x}) + [A^T\Theta(\boldsymbol{x})A]^{-1}(A^T\Theta(\boldsymbol{x})\boldsymbol{\delta}).$$

Because we know that $\tilde{f}(\boldsymbol{x})^* = \boldsymbol{\pi_m^3}(\boldsymbol{x}) \cdot \boldsymbol{c^*_{min}}(\boldsymbol{x})$, similar to Least Squares, we find that

$$\begin{aligned} \tilde{f}(\boldsymbol{x})^* &= \boldsymbol{\pi_m^3}(\boldsymbol{x}) \cdot (\boldsymbol{c_{min}}(\boldsymbol{x}) + [A^T\Theta(\boldsymbol{x})A]^{-1}(A^T\Theta(\boldsymbol{x})\boldsymbol{\delta})) \\ &= \boldsymbol{\pi_m^3}(\boldsymbol{x}) \cdot \boldsymbol{c_{min}}(\boldsymbol{x}) + \boldsymbol{\pi_m^3}(\boldsymbol{x}) \cdot ([A^T\Theta(\boldsymbol{x})A]^{-1}(A^T\Theta(\boldsymbol{x})\boldsymbol{\delta})) \qquad (2.25) \\ &= \tilde{f}(\boldsymbol{x}) + \boldsymbol{\pi_m^3}(\boldsymbol{x}) \cdot ([A^T\Theta(\boldsymbol{x})A]^{-1}(A^T\Theta(\boldsymbol{x})\boldsymbol{\delta})), \end{aligned}$$

which means that, as we have seen with the Least Squares and Weighted Least Squares methods, Moving Least Squares is linear in its error. The difference between the original approximation and the shifted approximation $|\tilde{f}(\boldsymbol{x})^* - \tilde{f}(\boldsymbol{x})|$ is a Moving Least Squares approximation on the same points $\{\boldsymbol{x_i}\}$, with scalar values $\{\delta_i\}_{i=1}^N$.

We can examine the stability of the coefficient vector by using equation 2.24, as we know that the coefficient vector is calculated in the same manner as a coefficient vector for a pivot point in Weighted Least Squares. This means that

$$\frac{||\boldsymbol{c_{min}}(\boldsymbol{x}) - \boldsymbol{c^*_{min}}(\boldsymbol{x})||}{||\boldsymbol{c_{min}}(\boldsymbol{x})||} \leq \kappa(\Theta(\boldsymbol{x})A)\frac{||\boldsymbol{y} - (\boldsymbol{y} + \boldsymbol{\delta})||}{||\boldsymbol{y}||}. \tag{2.26}$$

Notice that similar to what we saw with Weighted Least Squares, the Moving Least Squares coefficient vectors have the potential to be more stable than the Least Squares coefficient vector. Furthermore, the coefficient vectors will have a stability equal to those in the Weighted Least Squares method. If it is later shown that the Moving Least Squares method is more stable than the Weighted Least Squares method, it must be a result of the fact that Moving Least Squares generates a coefficient vector for each input $\boldsymbol{x}$ instead of for a limited set of pivot points.

As before, the analysis above is only an analysis of the stability of the coefficients and not of the entire approximation. This can, just as we have seen with Least Squares, give an indication of the stability of the approximation, but it is not the exact stability as there is still the general stability of polynomials that will have an influence on the stability of the approximation.

2.3.3 Complexity

Finally, we examine the algorithmic complexity of the Moving Least Squares method.

Theorem 2.14. *The complexity of the Moving Least Squares algorithm is dependent on the amount of times that $\tilde{f}(\boldsymbol{x})$ is sampled at a certain point $\boldsymbol{x}$. Calculating one of these samples has a complexity of $\mathcal{O}(Nk^3)$.*

As Moving Least Squares works continuously over the entire domain of the function, the complexity of this method depends heavily on its implementation. We know from the complexity of Weighted Least Sqaures that calculating the function at one point, $\tilde{f}_{\boldsymbol{x}}(\boldsymbol{x})$, has a complexity of $\mathcal{O}(Nk^3)$, the same as the complexity of the calculation in 2.17. The amount of times that we sample one of these functions in one point is, unlike in the situation of Weighted Least Squares, not certain as we cannot calculate $\tilde{f}_{\boldsymbol{x}}(\boldsymbol{x})$ for every single point $\boldsymbol{x}$ of a continuous domain.

Chapter 3

Applications in Three-Dimensional Modelling

In this chapter we will explain how the Least Squares, Weighted Least Squares and Moving Least Squares methods can be used for solving practical problems. We will do this by explaining two implementations of the methods in the three-dimensional modelling field: point cloud and polygon soup reconstruction.

3.1 Point Cloud Reconstruction

Many problems in three-dimensional modelling involve computing a model from a set of known points, which is known as a *point cloud.* These points are often acquired by measuring a real world object, with some examples being a statue or an artwork, a human body or an anatomical part, or even a landscape. Reconstructing a point cloud into a three-dimensional model is called *surface reconstruction.*

3.1.1 Approximation

Cuomo et al. describe a four step algorithm for reconstructing a surface [2]. If we have the set of N surface points $\{\boldsymbol{x_i}\}_{i=1}^N$ and their respective normals $\{\boldsymbol{n_i}\}_{i=1}^N$ (which we can compute if they are not known), we can first construct a so-called extended data set. This set of points consists of the original point set $\{\boldsymbol{x_i}\}$ together with two sets of off-surface points $\{\boldsymbol{x_i} + \delta\boldsymbol{n_i}\}_{i=1}^N$ and $\{\boldsymbol{x_i} - \delta\boldsymbol{n_i}\}_{i=1}^N$ which can be generated by picking a small surface offset $\delta > 0$. This extended point set is necessary for the second step. We can now define a function $f : \Omega \to \mathbb{R}$ on the domain of our model $\Omega \subset \mathbb{R}^3$ which we define as follows:

$$f(\boldsymbol{x}) = \begin{cases} 1 & \text{if } \boldsymbol{x} \text{ is outside of our model.} \\ \text{-1} & \text{if } \boldsymbol{x} \text{ is inside of our model.} \\ 0 & \text{if } \boldsymbol{x} \text{ is on the surface of our model.} \end{cases} .$$

We can now generate the accompanying scalar values for all the points in our extended data set: all original points all have a scalar value of 0, the positive off-surface points have scalar values of 1 and the negative off-surface points have scalar values of -1. The second step of the algorithm is approximating the function f using the points and their scalar values. If we approximate this function, we can then find the reconstruction by looking at the isosurface on which our approximation is equal to zero. This is where either the Least Squares, Weighted Least Squares or Moving Least Squares method can be used to generate an approximation of f, which we will call $\tilde{f}$.

3.1.2 Sampling

The third step is the evaluation of $\tilde{f}$ to examine where the isosurface of $\tilde{f} = 0$ lies. This is often done along a grid-like domain to create a mesh of the surface compatible with three-dimensional rendering software. An efficient algorithm that can triangulate the surface is the marching cubes algorithm [9]. This algorithm samples the function along a three-dimensional grid composed of cubes; by calculating how the surface intersects each cube the algorithm can triangulate the part of the surface within that cube. This process will transform the entire surface to a geometrical representation. Modern software tools such as skimage provide an efficient implementation of this algorithm that also returns topologically sound meshes [8]. A benefit of using marching cubes algorithms is that the number of samples that is needed to provide a mesh of a certain resolution is known beforehand. This means that we can use the theoretical complexity of the Moving Least Squares methods to estimate the complexity of reconstructing a surface, as the complexity of using these methods is dependent on the amount of samples and we now know how many samples will be calculated.

The fourth and final step is rendering the surface. This can be done in any three-dimensional rendering software as all implementations of the marching cubes algorithm return a triangulated mesh of the reconstructed surface.

3.2 Polygon Soup Reconstruction

Another example of scattered data from which a mesh can be reconstructed is *polygon soup*. Instead of a set of points, we have a set of polygons. This data can be obtained from other polygonal meshes, which might be too detailed or not topologically closed. Therefore, reconstructing scattered polygon soup can have many uses. C. Shen, J. F. O'Brian and J. R. Shewchuk provide a way to rewrite Least Squares based methods so that the input is a set of polygons with their respective scalar values instead of a set of points [11]. Equation 2.5 is rewritten so that for a set of polygons $\{\Omega_i\}_{i=1}^N$ we have

$$\boldsymbol{c_{min}} = [\sum_i \boldsymbol{A_i}]^{-1} \sum_i \boldsymbol{a_i}, \tag{3.1}$$

where

$$\boldsymbol{A_i} = \int_{\Omega_i} \boldsymbol{\pi^3_m}(\boldsymbol{x})\boldsymbol{\pi^3_m}(\boldsymbol{x})^T d\boldsymbol{x} \tag{3.2}$$

and

$$\boldsymbol{a_i} = \int_{\Omega_i} \boldsymbol{\pi^3_m}(\boldsymbol{x}) y_i d\boldsymbol{x}. \tag{3.3}$$

This way we assign an infinite set of points to each polygon on which we can apply the Least Squares method. This can be applied in the same manner to equation 2.19 for Weighted Least Squares by defining the new variables as follows:

$$\boldsymbol{A_i} = \int_{\Omega_i} \theta(||\boldsymbol{x} - \boldsymbol{\hat{x}_j}||)\boldsymbol{\pi^3_m}(\boldsymbol{x})\boldsymbol{\pi^3_m}(\boldsymbol{x})^T d\boldsymbol{x}, \tag{3.4}$$

$$\boldsymbol{a_i} = \int_{\Omega_i} \theta(||\boldsymbol{x} - \boldsymbol{\hat{x}_j}||)\boldsymbol{\pi^3_m}(\boldsymbol{x}) y_i d\boldsymbol{x}. \tag{3.5}$$

These equations can then be extended to Moving Least Squares in the same way that Moving Least Squares extends Weighted Least Squares when interpolating a point set. The methods can be extended to add normal constraints. Similar to point cloud surface reconstruction, we can then produce an approximation function $\tilde{f}$ which can be sampled with a method that provides a topologically sound mesh such as the marching cubes algorithm. This makes it possible to transform polygon soup into a topologically correct geometric representation of the original model.

Chapter 4

Numerical Experiments

In this chapter we will show implementations of the Least Squares, Weighted Least Squares and Moving Least Squares methods. Using these we will conduct several numerical experiments to verify the theoretical results that were obtained with respect to the accuracy, stability and complexity of each individual method in chapter 2. We will also use these results to further compare the methods when they are used in practice. Finally, we will conduct an experiment where each method will be used to reconstruct a point cloud as described in section 3.1. The results of this experiment can then be combined with both the theoretical assumptions and the results of earlier experiments to determine the quality of each method when applied on such a real-world three-dimensional modelling problem.

4.1 Implementation

In this section we will show and explain our implementation of each method in Python. Each method has been implemented both in a one-dimensional space and in three-dimensional space, with small differences between these two implementations. The full implementation together with the code for all further experiments in this thesis, can be found at `https://github.com/desparito/comparing-ls-methods`.

4.1.1 Least Squares

We shall now look at the implementation of Least Squares in one-dimension. Notice that in the Python listings, the *Numpy* package is shorted to `np`.

Listing 4.1: The one-dimensional Least Squares algorithm.

```python
def least_squares_R(points, values, m):
   temp = lambda x : [x ** i for i in range(m + 1)]
   pi_t = lambda x : np.array([temp(x)])
   pi = lambda x : np.transpose(pi_t(x))
   k = len(temp(0))

   leftsum = np.zeros((k,k))
   rightsum = np.zeros((k, 1))
   for i in range(len(points)):
      leftsum += np.matmul(pi(points[i]), pi_t(points[i]))
      rightsum += pi(points[i])*values[i]
   cmin = np.matmul(np.linalg.inv(leftsum), rightsum)
   return np.vectorize(lambda x : np.dot(pi(x).flatten(), cmin.flatten()))
```

We see in the first line that the function takes three inputs: `points`, which is an array of scalars of length N, representing the one-dimensional points $\{x_i\}$; `values`, which is an array of length N, representing the respective scalar values $\{y_i\}$; and `m`, which is the dimension of the polynomial space as given in definition 2.1.

Lines 3 until 6 construct the vector $\boldsymbol{\pi}_{\boldsymbol{m}}^{\boldsymbol{1}}(x)$, its transpose and its length k. This is simple in one dimension, as $\boldsymbol{\pi}_{\boldsymbol{m}}^{\boldsymbol{1}}(x) = [1, x^1, \dots, x^m]^T$. Lines 8 until 13 then calculate $\boldsymbol{c_{min}}$ according to equation 2.5, with `leftsum` representing the left sum in this equation and `rightsum` representing the right. The `leftsum` is then inverted and multiplied with `rightsum` to acquire $\boldsymbol{c_{min}}$. In line 14 the final step is performed: $\tilde{f}(x) = \boldsymbol{\pi}_{\boldsymbol{m}}^{\boldsymbol{1}}(x) \cdot \boldsymbol{c_{min}}$.

Listing 4.2: The three-dimensional Least Squares algorithm.

```
1  def least_squares_R3(points, values, m):
2     vectors = []
3     for i in range(m + 1):
4        vectors += [[i - q - p, q, p] for p in range(i + 1) for q in range(i + 1)
             if i - q - p >= 0]
5
6     temp = lambda x, y, z : [(x**i)*(y**j)*(z**k) for [i,j,k] in vectors]
7     pi_t = lambda x, y, z : np.array([temp(x,y,z)])
8     pi = lambda x, y, z : np.transpose(pi_t(x,y,z))
9     k = len(vectors)
10
11    leftsum = np.zeros((k,k))
12    rightsum = np.zeros((k, 1))
13    for i in range(len(points)):
14       leftsum += np.matmul(pi(points[i][0], points[i][1], points[i][2]),
             pi_t(points[i][0], points[i][1], points[i][2]))
15       rightsum += pi(points[i][0], points[i][1], points[i][2])*values[i]
16    cmin = np.matmul(np.linalg.inv(leftsum), rightsum)
17    return np.vectorize(lambda x, y, z : np.dot(pi(x,y,z).flatten(),
         cmin.flatten()))
```

As we can see, the three-dimensional algorithm is quite similar. It has the same input (`points`, `values` and `m`), except for the fact that an element of `points` is now a three-dimensional array. In lines 2 to 9 we now compute $\boldsymbol{\pi}_{\boldsymbol{m}}^{\boldsymbol{3}}(\boldsymbol{x})$, its transpose and its length k. This is more complicated than in one dimension, something which we have seen with example 2.4. In lines 11 to 16 we again compute `leftsum` and `rightsum` as the left and right sum as seen in 2.5. Finally in line 17 we again use the dot product of $\boldsymbol{\pi}_{\boldsymbol{m}}^{\boldsymbol{3}}(\boldsymbol{x})$ and $\boldsymbol{c_{min}}$ to find $\tilde{f}(\boldsymbol{x})$.

4.1.2 Weighted Least Squares

We can now look at the one-dimensional implementation of Weighted Least Squares and see some similarities.

Listing 4.3: The one-dimensional Weighted Least Squares algorithm.

```
1  def weighted_least_squares_R(points, values, m, theta):
2     temp = lambda x : [x ** i for i in range(m + 1)]
3     k = len(temp(0))
4     pi_t = lambda x : np.array([temp(x)])
5     pi = lambda x : np.transpose(pi_t(x))
6
7     relative_theta = lambda x, j : theta(np.abs(x - points[j]))
8     relative_theta_sum = lambda x : sum([relative_theta(x, j) for j in
         range(len(points))])
9
10    def cmin(x):
```

```
11       leftsum = np.zeros((k,k))
12       rightsum = np.zeros((k, 1))
13       for i in range(len(points)):
14          leftsum += theta(np.abs(x - points[i]))*np.matmul(pi(points[i]),
               pi_t(points[i]))
15          rightsum += theta(np.abs(x - points[i]))*pi(points[i])*values[i]
16       return np.matmul(np.linalg.inv(leftsum), rightsum)
17
18    return np.vectorize(lambda x : sum(psi(x, j)*np.dot(pi(x).flatten(),
         cmin(points[j]).flatten()) for j in range(len(points))))
```

Weighted Least Squares has the same parameters as Least Squares, with `theta` as an extra parameter, which represents the weighting function; a lambda function can be given to the method to weigh the distances from each pivot point. Notice that both in our one-dimensional and three-dimensional implementation we have chosen the sampling points $\{x_i\}$ as our pivot points for our numerical experiments.

Lines 2 to 5 are the same as in listing 4.1, as the calculation of the polynomial basis vector does not change. In lines 7 to 8 we calculate what is needed for $\phi_j(x)$, the partition of unity which we saw in 2.14. In lines 10 until 16 we calculate a $\boldsymbol{c_{min}}(\hat{x_j})$ dependent on pivot point $\hat{x_j}$, given as parameter `x`. We again use a variable `leftsum` and a variable `rightsum`, this time to represent the left and right sums in equation 2.17. Finally, in line 18, we implement equation 2.19 to sum the local weighted functions and provide a global approximation. Notice that we do not directly calculate each ϕ_j but that we instead take the factor of $\frac{1}{\sum_{k=1}^{k=n}\theta_k(x)}$ out of the sum, which saves a lot of calculation time for each time we sample the approximation.

Listing 4.4: The three-dimensional Weighted Least Squares algorithm.

```
1  def weighted_least_squares_R3(points, values, m, theta):
2     vectors = []
3     for i in range(m + 1):
4     vectors += [[i - q - p, q, p] for p in range(i + 1) for q in range(i + 1)
         if i - q - p >= 0]
5
6     temp = lambda x, y, z : [(x**i)*(y**j)*(z**k) for [i,j,k] in vectors]
7     k = len(vectors)
8     pi_t = lambda x, y, z : np.array([temp(x,y,z)])
9     pi = lambda x, y, z : np.transpose(pi_t(x,y,z))
10
11    #Pre-calculate static values outside of the main cmin calculation:
12    pi_pi_t_at_point = [None] * len(points)
13    pi_values_at_point = [None] * len(points)
14    for i in range(len(points)):
15       pi_at_point = pi(points[i][0], points[i][1], points[i][2])
16       pi_pi_t_at_point[i] = np.matmul(pi_at_point, pi_t(points[i][0],
            points[i][1], points[i][2]))
17       pi_values_at_point[i] = pi_at_point*values[i]
18
19    norm = lambda x, y, z : np.sqrt(x**2 + y**2 + z**2)
20
21    relative_theta = lambda x, y, z, j : theta(norm(x - points[j][0], y -
         points[j][1], z - points[j][2]))
22    relative_theta_sum = lambda x, y, z : sum([relative_theta(x, y, z, j) for j
         in range(len(points))])
23
24    def cmin(x, y, z):
25       leftsum = np.zeros((k,k))
```

```
26        rightsum = np.zeros((k, 1))
27        for i in range(len(points)):
28          weight = theta(norm(x - points[i][0], y - points[i][1], z -
               points[i][2]))
29          leftsum += weight*pi_pi_t_at_point[i]
30          rightsum += weight*pi_values_at_point[i]
31        return np.matmul(np.linalg.inv(leftsum), rightsum)
32
33      #Precaculate the cmin for each (pivot) point:
34      cmin_at_point = [None] * len(points)
35      for i in range(len(points)):
36        cmin_at_point[i] = cmin(points[i][0],points[i][1],points[i][2])
37
38      def result(x,y,z):
39        local_pi = pi(x,y,z).flatten()
40        return (sum([relative_theta(x,y,z,j)*np.dot(local_pi,
              cmin_at_point[j].flatten()) for j in
              range(len(points))]))/relative_theta_sum(x,y,z)
41      return np.vectorize(result)
```

The three-dimensional calculation extends the one-dimensional calculation very similarly to how listing 4.1 extends listing 4.2. There are two sections that are interesting however; those in lines 11 until 17 and 33 to 36. Here we respectively pre-calculate the values of $\boldsymbol{\pi}_{\boldsymbol{m}}^{\mathbf{3}}(\boldsymbol{x_i})$, $\boldsymbol{\pi}_{\boldsymbol{m}}^{\mathbf{3}}(\boldsymbol{x_i})\boldsymbol{\pi}_{\boldsymbol{m}}^{\mathbf{3}}(\boldsymbol{x_i})^T$,$\boldsymbol{\pi}_{\boldsymbol{m}}^{\mathbf{3}}(\boldsymbol{x_i})y_i$ and the $\boldsymbol{c_{min}}(\boldsymbol{x_i})$ for each sample point from $0 < i \leq N$ and store the values in memory. This saves us quite a lot of computation time. In the final three lines we also pre-calculate and store $\boldsymbol{\pi}_{\boldsymbol{m}}^{\mathbf{3}}(\boldsymbol{x})$ when using it in the final sum instead of calculating it j times. This speeds up the method in three dimensions as calculating $\boldsymbol{\pi}_{\boldsymbol{m}}^{\mathbf{3}}(\boldsymbol{x})$ becomes more computationally expensive than in one dimension.

4.1.3 Moving Least Squares

The one-dimensional implementation of Moving Least Squares looks like a combination of listings 4.1 and 4.3. It calculates the $\boldsymbol{c_{min}}(x)$ for a certain point x similar to how the $\boldsymbol{c_{min}}$ vector would be calculated for a pivot point in Weighted Least Squares, but in line 15 the final dot product is calculated as described in section 2.3.1.

Listing 4.5: The one-dimensional Moving Least Squares algorithm.

```
1  def moving_least_squares_R(points, values, m, theta):
2    temp = lambda x : [x ** i for i in range(m + 1)]
3    k = len(temp(0))
4    pi_t = lambda x : np.array([temp(x)])
5    pi = lambda x : np.transpose(pi_t(x))
6
7    def cmin(x):
8      leftsum = np.zeros((k,k))
9      rightsum = np.zeros((k, 1))
10     for i in range(len(points)):
11       leftsum += theta(np.abs(x - points[i]))*np.matmul(pi(points[i]),
            pi_t(points[i]))
12       rightsum += theta(np.abs(x - points[i]))*pi(points[i])*values[i]
13     return np.matmul(np.linalg.inv(leftsum), rightsum)
14
15   return np.vectorize(lambda x : np.dot(pi(x).flatten(), cmin(x).flatten()))
```

Listing 4.6: The three-dimensional Moving Least Squares algorithm.

```
1  def moving_least_squares_R3(points, values, m, theta):
2     vectors = []
3     for i in range(m + 1):
4     vectors += [[i - q - p, q, p] for p in range(i + 1) for q in range(i + 1)
          if i - q - p >= 0]
5
6     temp = lambda x, y, z : [(x**i)*(y**j)*(z**k) for [i,j,k] in vectors]
7     k = len(vectors)
8     pi_t = lambda x, y, z : np.array([temp(x,y,z)])
9     pi = lambda x, y, z : np.transpose(pi_t(x,y,z))
10
11    #Pre-calculate static values outside of the main cmin calculation:
12    pi_pi_t_at_point = [None] * len(points)
13    pi_values_at_point = [None] * len(points)
14    for i in range(len(points)):
15       pi_at_point = pi(points[i][0], points[i][1], points[i][2])
16       pi_pi_t_at_point[i] = np.matmul(pi_at_point, pi_t(points[i][0],
             points[i][1], points[i][2]))
17       pi_values_at_point[i] = pi_at_point*values[i]
18
19    norm = lambda x, y, z : np.sqrt(x**2 + y**2 + z**2)
20
21    def cmin(x, y, z):
22       leftsum = np.zeros((k,k))
23       rightsum = np.zeros((k, 1))
24       for i in range(len(points)):
25          weight = theta(norm(x - points[i][0], y - points[i][1], z -
                points[i][2]))
26          leftsum += weight*pi_pi_t_at_point[i]
27          rightsum += weight*pi_values_at_point[i]
28       return np.matmul(np.linalg.inv(leftsum), rightsum)
29
30    return np.vectorize(lambda x, y, z : np.dot(pi(x,y,z).flatten(),
          cmin(x,y,z).flatten()))
```

Finally, the three-dimensional Moving Least Squares algorithm extends this to three dimensions. We see that it also pre-calculates the same values that were pre-calculated in listing 4.4 to lower the total computation time. Just as in the one-dimensional algorithm, the final step in line 30 completes the Moving Least Squares approach.

4.2 Examples of Approximation

To see what type of functions are interesting to look into in terms of stability, it is important that it is known what functions produce interesting and accurate approximations when approximated by the three methods. We also need a grasp of the behaviour of the different methods to choose the correct parameters when later comparing the stability of the methods, because some of these parameters might influence the results. We will therefore use this section to look at a few examples and explore a few different interesting approximations that show the general behaviour of the methods.

Example 4.1. If we approximate polynomial functions $f(x) = 10^{-5}x^6$ and $f(x) = 10^{-5}x^7$ on the domain $[-10, 10]$ by sampling ten points evenly across this domain and providing these points with their scalar values to the methods, we see in figure 4.1 that all three methods converge to a near-perfect approximation. We numerically approach the maximal error on

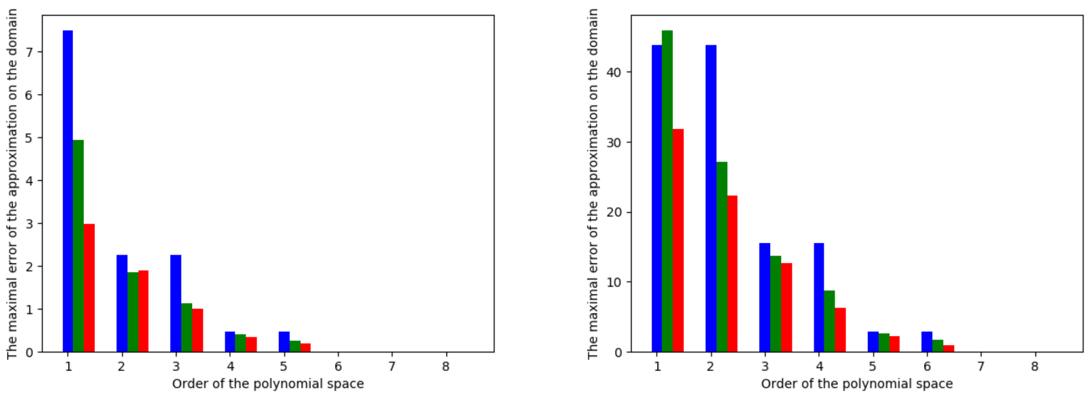


Figure 4.1: The maximal error (sampled across 1000 points) of the the approximation of two functions (left: $f(x) = 10^{-5}x^6$ and right: $f(x) = 10^{-5}x^7$) on the domain $[-10, 10]$ produced by LS (blue), WLS (green) and MLS (red) as the order of the polynomial space increases. WLS and MLS use the Wendland function as the weighting function with $h = 20$.

the domain $\max_{x\in[-10,10]} |f(x) - \tilde{f}(x)|$ by sampling 1000 points for $x \in [-10, 10]$ and taking the maximal error.

What example 4.1 shows us can be extrapolated to all polynomials; generally a polynomial of order n can be approximated almost perfectly by each method if the method uses a polynomial space of order n, sometimes even with as little samples as in example 4.1. Figure 4.1 also nicely shows that the error of a Weighted Least Squares approximation is generally lower than that of a Least Squares approximation and that the error of a Moving Least Squares approximation is generally even lower than that of a Weighted Least Squares approximation. This is expected, as we know from theory that each method should improve upon the previous.

Example 4.2. If we look at the first-order approximations of all methods as described in example 4.1, we can plot them against the original functions on the domain to compare the accuracy of each method. The result of this is shown in figure 4.2. Notice that a benefit of Moving and Weighted Least Squares over normal Least Squares is that a first-order Least Squares approximation is globally first order, whereas a first-order Weighted or Moving Least Squares approximation is only locally first-order and can therefore take on a higher order globally.

Example 4.2 can also be generalised to higher order approximations; we see that the global order of the approximation that Least Squares produces is limited by the order of the polynomial space. The Weighted and Moving Least Squares methods solve this issue by combining many local polynomials, either centred around the pivot points for WLS or around the points where the function is sampled for MLS.

Example 4.3. Another interesting example involves Runge's phenomenon. If we generate a standard polynomial interpolation of order n of the Runge function, $f(x) = \frac{1}{1+25x^2}$, through $n + 1$ equispaced points on the domain $[-1, 1]$, the error will increase as n increases. The interpolated function will oscillate when close to $x = -1$ and $x = 1$ and as the degree of the polynomial interpolation increases, the error may diverge to infinity [4]. With the Least Squares based methods however, the approximations stay well-conditioned as long as the order of the approximation is less than $2\sqrt{n}$ [3]. In figure 4.3 we see that if we break this rule with $n = 10$ and take an approximation of order 8, it starts to present the same problems

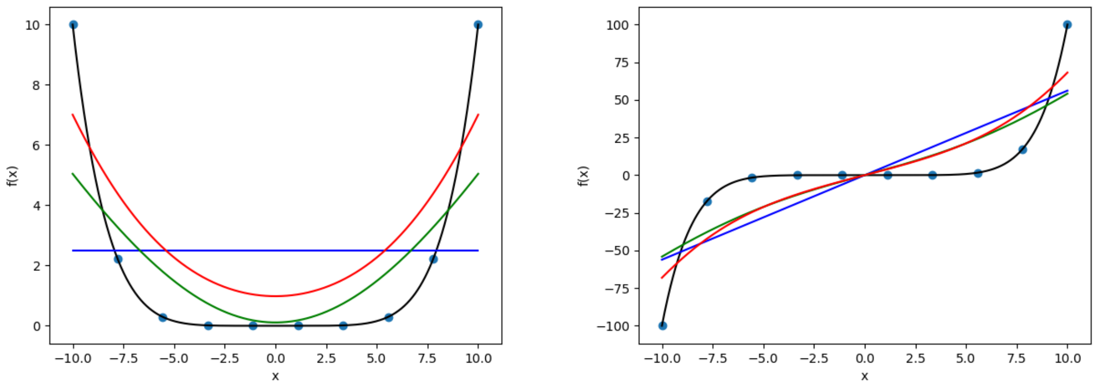


Figure 4.2: The first-order approximations of LS (blue), WLS (green) and MLS (red) of two functions (left: $f(x) = 10^{-5}x^6$ and right: $f(x) = 10^{-5}x^7$), compared to the original functions (black) on the domain $[-10, 10]$. WLS and MLS use the Wendland function as the weighting function with $h = 20$. The input points and their scalar values are represented by the blue dots. (The graph is sampled at 1000 points.)

as the polynomial interpolation. The approximation with order 6 is therefore better in this situation. We can also see that we can even slightly improve the interpolation by adding an extra input point at $x = 0$ to better represent the original function.

Example 4.3 shows us two final things to take into consideration when comparing the different approximation methods. The first is that there might be an inclination to think that using a higher order polynomial space produces approximations that are more accurate. This is true for some functions, as we saw in example 4.1 that polynomial functions have near-perfect approximations for any of the three methods when using an approximation with a polynomial space of the same order. But the Runge function shows that there are also functions where Least Squares based methods have a benefit over simple polynomial interpolation methods, as the order of the polynomial space is a choice and a more beneficial order can be chosen if the function calls for such a thing. When comparing the three methods, we should choose a function that has a relatively simple behaviour when increasing the order of the polynomial space, so that effects similar to Runge's phenomenon do not influence the results. The second consideration is that we choose a function that is well-represented when taking an equilateral set of input points over the domain. As we saw in 4.3, the equilateral distribution of points on the Runge function does not work very well, as we tend to miss the peak at $x = 0$. We would like to avoid this in the function that is approximated when comparing the methods, as this may also influence the results.

4.3 Stability

In this section we will examine the accuracy and usability of the theoretical bound on the stability calculated in chapter 2. The result is that in practice this bound is generally not an accurate bound: it is way higher than the actual relative error will be when shifting the scalar values. Because of this result, we experiment in this section to find out what the stability will be in practical situations. We will see that generally the Weighted and Moving Least Squares methods seem to be more stable than the Least Squares method and we will examine some special cases to get a better overview of how the stability behaves under different circumstances.

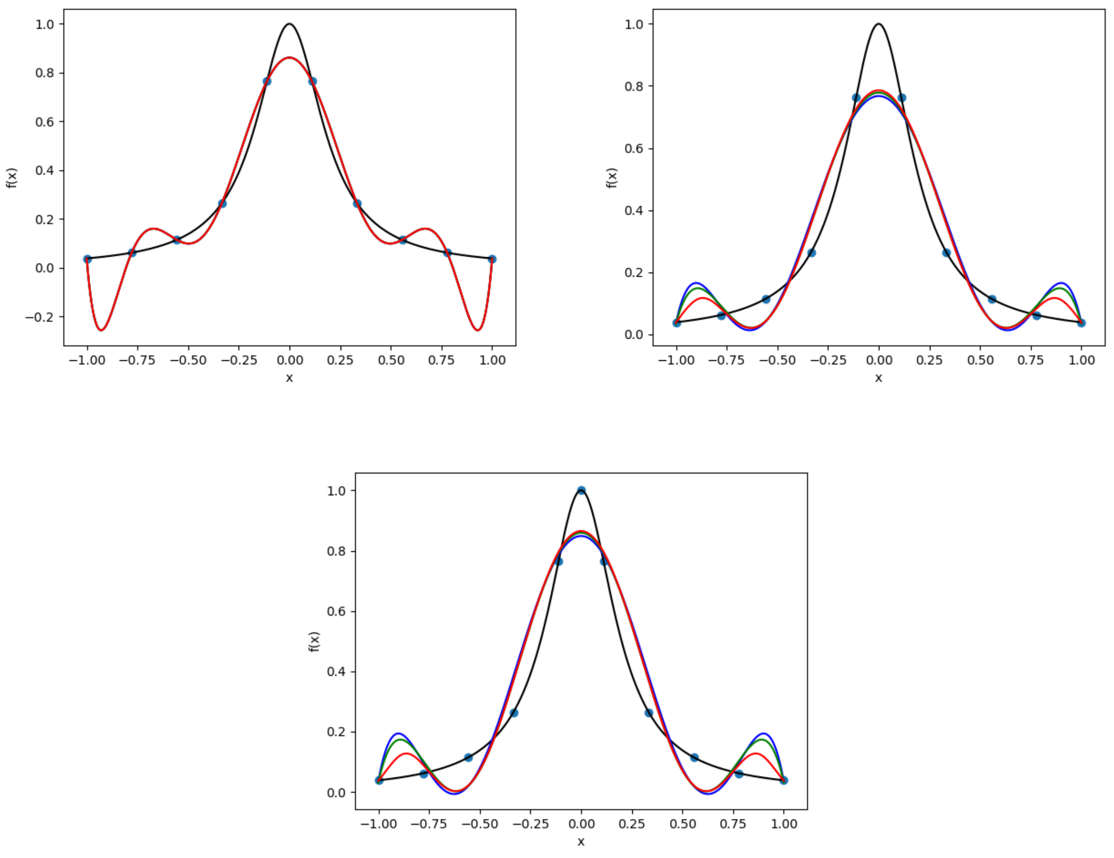


Figure 4.3: The approximations of LS (blue), WLS (green) and MLS (red) of the Runge function, compared to the original function (black) on the domain $[-1, 1]$. The top-left graph shows the eighth-order approximations, the top-right shows the sixth-order approximations. Both these graphs have ten equilateral input points. The bottom graph shows the sixth-order approximations with an extra input point at $x = 0$. WLS and MLS use the Wendland function as the weighting function with $h = 2$. The input points and their scalars are represented by the blue dots. (The graph is sampled at 1000 points.)

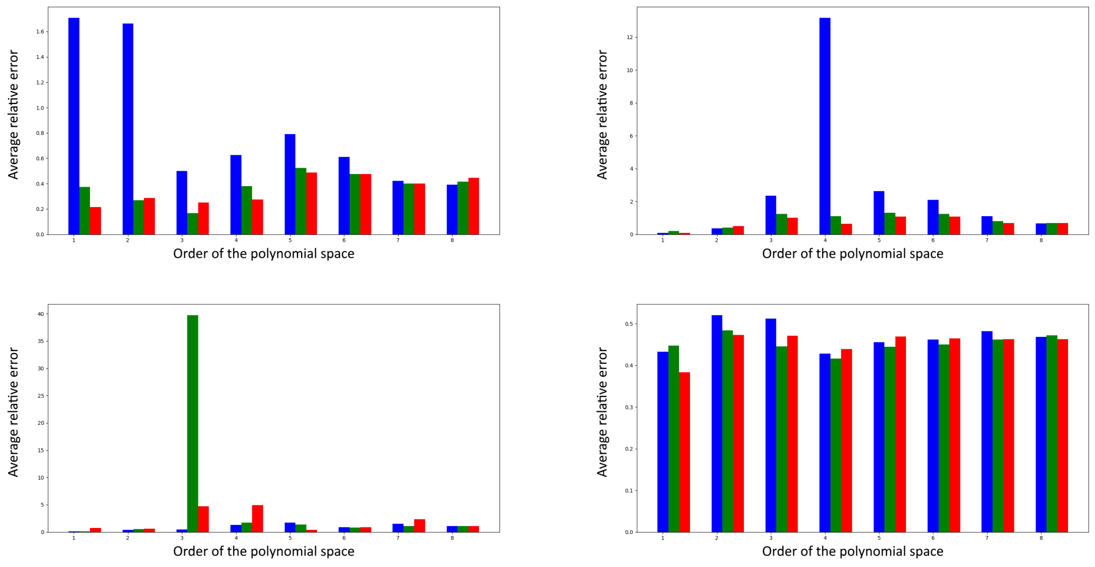


Figure 4.4: The average relative error (sampled over 1000 points) between a correct approximation and an approximation skewed by $\delta_i = a_i y_i$ (with a_i chosen randomly with a uniform distribution on the domain $[-1, 1]$) as the order of the polynomial space m with ten evenly sampled points on the domain $[0, 10]$ or $[-1, 1]$ for the bottom left graph is increased, for LS (blue), WLS (green) and MLS (red). Weighting is done using the Wendland function with $h = 10$ or $h = 2$ for the bottom left graph. From left to right, top to bottom the functions approximated are $f(x) = 10^{-5}x^6+1$, $f(x) = 10^{-5}x^7+1$, $f(x) = \frac{1}{25x^2+1}$ and $f(x) = |x-5|+1$.

In figure 4.4 we look at the stability of several functions when increasing the order of the polynomial space used. For the top two graphs, we used two functions based on the polynomial functions examined in example 4.1, but slightly adjusted so that $f(x) \neq 0$, which is easier for calculating the relative error. As we know for these experiments that $\delta_i = a_i y_i$ with a_i chosen randomly from a uniform distribution on $[-1, 1]$, we know that $\frac{||\boldsymbol{y}-(\boldsymbol{y}+\boldsymbol{\delta})||}{||\boldsymbol{y}||} \leq 1$. The results show that the relative errors between the skewed and original approximation are not too far from 1 except for some outliers, so the methods are generally quite stable. Even in the bottom two figures there is a low relative error, despite the fact that the functions used for these figures are more difficult to approximate than the polynomial functions in the top two figures.

If we look at the difference between the methods, it seems that the Weighted and Moving Least Squares methods seem to be more stable in most situations. As said before there do seem to be outliers, such as in the top-right figure where the Least Squares approximation peaks and in the bottom-left figure where the Weighted Least Squares method has a high peak and the Moving Least Squares method also has two relatively high peaks. This shows that the stability of each method heavily depends on the circumstances and that unexpected instabilities might occur. It is therefore difficult to say whether or not the Moving Least Squares method is more stable than the Weighted Least Squares method, as the difference in relative error is less clear than the difference between these methods and the Least Squares method.

Finally, if we take the sine function, we see another interesting result. In figure 4.5 we see that the stability decreases heavily as the order of the polynomial space increases past four. This means that for some functions there may be a point where taking an order that is too high may damage the quality of the approximation, something that was also shown

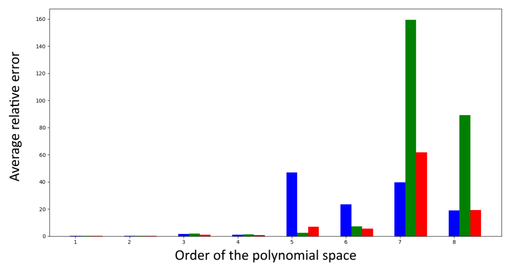


Figure 4.5: The average relative error (sampled over 1000 points) between a correct approximation and an approximation skewed by $\delta_i = a_i y_i$ (with each a_i chosen randomly with a uniform distribution on the domain $[-1, 1]$) as the order of the polynomial space m with ten evenly sampled points on the domain $[0, 2\pi]$ is increased, for LS (blue), WLS (green) and MLS (red). Weighting is done using the Wendland function with $h = 2\pi$. The function approximated is $f(x) = \sin(x) + 1$.

in the previous section. This is not a strange result, as higher order approximations are intuitively less stable than lower order approximations; a linear line will for example shift less when changing the scalar values of the points that it interpolates than a polynomial of a higher order. Interestingly, this effect is not visible with the stability of the Runge function, whereas in the previous section this effect did occur on the error of the Runge function. It is clear that this effect can get very large, as the highest peak in relative error in the figure is around 180 times larger than the lowest relative error of the same method.

The mostly good stabilities seen in 4.4 are a result that is a lot more positive than what was found theoretically. Looking at example 4.4 shows that the theoretical upper bound on the conditioning of the matrix A is a lot worse than the conditioning of the final result in practice. This means that in theory the relative error can get fairly big, which is what happens when increasing the order above $m = 4$ with the sine function, but in practice we often find that the approximations are all quite well-conditioned.

Example 4.4. Imagine that we try to approximate the function $f(x) = \sin(x) + 1$ on the domain $[0, 2\pi]$. We sample ten points from the original function, giving us the point set $\{x_i\}_{i=1}^{10}$ with $x_i = (i-1)\frac{2\pi}{9}$ with scalar values $\{y_i\}_{i=1}^{10}$ with $y_i = \sin(x_i) + 1$. If we linearly approximate this function, meaning that $m = 1$, we have

$$A = \begin{pmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_{10} \end{pmatrix}.$$

Numerical approximation using Singular Value Decomposition gives us $\kappa(A) \approx 7.26$. If we use the Wendland function for weighting with $h = 2\pi$, the same numerical approximation gives us $\kappa(\Theta(x_1)A) \approx 2.35$, $\kappa(\Theta(x_{10})A) \approx 41.21$ and if we numerically approximate a few more points we also see that $\kappa(\Theta(x)A) < \kappa(\Theta(y)A)$ if $x < y$.

In figure 4.6 we take the functions we have examined in figures 4.4, but we look at how the stability varies when the amount of input points is changed. We start at an amount of four points so that the matrix is not singular. In this figure the stability grows as the bar of each method gets smaller compared to the black bar; we have to compare to the black bar as we cannot compare the individual bars of each method because of the randomness involved in choosing each $\boldsymbol{\delta}$ vector. Generally, the stability seems to improve as the amount of points

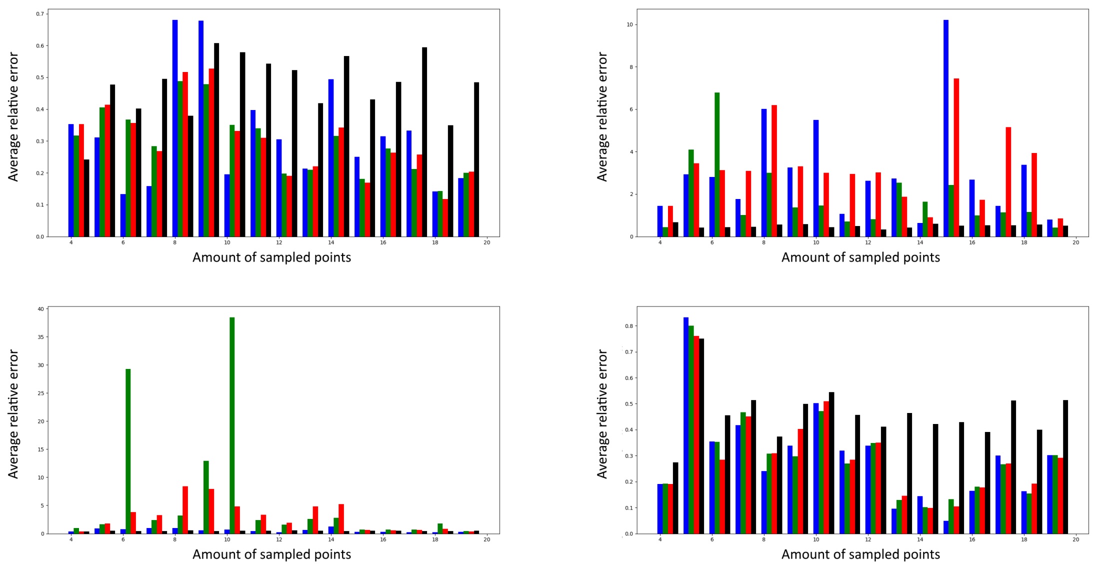


Figure 4.6: The average relative error (sampled over 1000 points) between a correct approximation and an approximation skewed by $\delta_i = a_i y_i$ (with a_i chosen randomly with a uniform distribution on the domain $[-1, 1]$ and all a_i re-chosen for each amount of points) with a polynomial order of $m = 3$ as the amount of equidistant points sampled on the domain $[0, 10]$ or $[-1, 1]$ for the bottom left graph is increased, for LS (blue), WLS (green) and MLS (red). The average over all i of the relative error between y_i and $\delta_i + y_i$ is shown in black. Weighting is done using the Wendland function with $h = 10$ and $h = 2$ for the second-row left graph. From left to right, top to bottom the functions approximated are $f(x) = 10^{-5}x^6 + 1$, $f(x) = 10^{-5}x^7 + 1$, $f(x) = \frac{1}{25x^2+1}$ and $f(x) = |x - 5| + 1$.

increases, which is a logical result; a larger set of input points means that a big fluctuation of a single scalar value has a smaller impact on the approximation as a whole. There are of course a few outliers within the data, just as in figure 4.4 and 4.5, which is to be expected considering the randomness involved in this experiment.

It is very difficult to compare the three methods in stability using figure 4.6. All three seem to preform better than the other two in different situations; in the top-left graph Least Squares seems to be the least stable method, whereas it is clearly the most stable in the bottom-left graph. This leads us to conclude that the amount of different parameters makes it quite difficult to accurately compare the stability of these methods in practice: all three might have parameters under which they have an optimal stability, such as a specific set of input points and a certain weighting function.

To conclude this section, we discovered that the methods are all three quite well conditioned, both when using a small amount of input points and a small order for the polynomial space. We also found that there might be big differences in the stability of each method given different circumstances under which it is tested. This makes it difficult to state which method objectively has the best stability. Fortunately, the theoretical bound we found earlier seems too high to accurately predict the stability of the approximations, as both $\kappa(A)$ and $\kappa(A\Theta(\boldsymbol{x}))$ for any $\boldsymbol{x}$ seem to grow too large in most realistic situations.

4.4 Complexity

In this section we examine the speed of each algorithm in practice under different conditions to see if the complexity of our implementation of the methods is in line with the theoretical complexities seen in chapter 2. The result is that the theoretical complexity and the speed of the algorithms in practice seem to mostly match each other for all three methods, with some changes in the implementation even improving the speed.

In figure 4.7 we can view how the speed of each algorithm in three dimensions scales against the amount of input points. The function and the domain were chosen to be relatively simple to calculate and so that it is easily verifiable if the approximations are relatively accurate. According to the complexity we found in section 2.1.3, the approximation speed should scale linearly as N increases, which is what happens in the figure. Notice that the slope is not very steep, especially when compared to the increase in computation time in the other two graphs. The greatest increase is in the approximation time. This is logical if we remember our implementation: computing the $\boldsymbol{c_{min}}$ vector is done when approximating and scales more heavily with N than calculating the dot product of $\boldsymbol{\pi_3^3}(\boldsymbol{x})$ and $\boldsymbol{c_{min}}$, which is what is done for each sample point.

If we look at the results in figure 4.7 for WLS, we see that it is a lot less fast than LS. The theoretical complexity calculated earlier was $\mathcal{O}(nNk^2)$, which is equal to $\mathcal{O}(N^2k^2)$ as $N = n$ in this situation. It is difficult to see if the computational time of WLS has a linear or quadratic increase as N grows. If we assume that the increase is quadratic, we can clearly state that as the amount of input points gets very large, it becomes beneficial to choose less points as pivot points, so that the complexity decreases to $\mathcal{O}(nNk^2)$. Notice that almost all of the computation time comes from sampling. This is interesting, because it means that the computation of all the coefficient vectors is a lot faster than the final weighting of each coefficient vector with the ϕ functions. This means that it might be beneficial to choose less pivot points and more simple weighting functions so that the calculation of each $\phi_j(\boldsymbol{x})$ becomes faster.

Finally, looking at the results for MLS in figure 4.7, we find that MLS is slightly slower than the WLS method. However, comparing these methods in speed through this figure is unreliable as the load of the machine that the calculations were performed on differs and performing certain tasks may be slightly faster or slower because of this. The result that is seen in the figure is what we would expect from the theoretical stability: if we look at what we stated in sector 2.3.3, the computation time should increase linearly as N increases. This is what is visible in the figure. Almost all of the time is spent in the sampling phase, which is logical: In our implementation generating the approximation has a small pre-calculation cost, but the bulk of the computation happens during the sampling, as seen in the first sector of this chapter.

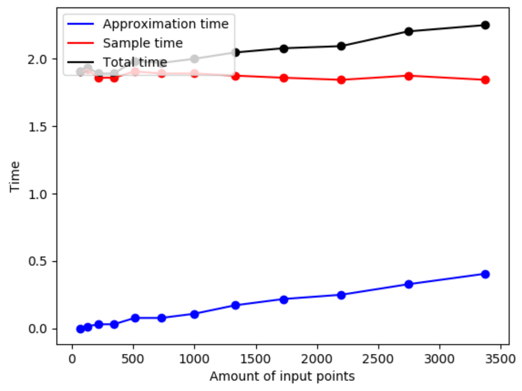
In figure 4.8, we examine what happens to the speed of the different methods as the order of the polynomial space is increased. It shows that the speed of the Least Squares algorithm increases at around the rate found when calculating the theoretical complexity. When we look at the Weighted Least Squares method, it seems that the order seems to have a linear influence on the speed. This does not coincide with our theoretical results, but it can be easily explained when looking at the implementation of the algorithm. As we explained earlier in this chapter, both our implementation of Weighted and Moving Least Squares pre-calculate the values of $\boldsymbol{\pi}_{\boldsymbol{m}}^{\boldsymbol{3}}(\boldsymbol{x_i})$, $\boldsymbol{\pi}_{\boldsymbol{m}}^{\boldsymbol{3}}(\boldsymbol{x_i})\boldsymbol{\pi}_{\boldsymbol{m}}^{\boldsymbol{3}}(\boldsymbol{x_i})^T$ and $\boldsymbol{\pi}_{\boldsymbol{m}}^{\boldsymbol{3}}(\boldsymbol{x_i})y_i$ for each sample point from $0 < i \leq N$. From what we saw in sections 2.2.3 and 2.3.3, these calculations contributed a factor of around k^2 to the algorithmic complexity of our algorithm. By moving these calculations to the beginning of the algorithm and storing them in memory when sampling, the theory is that the algorithmic complexity should drop to around $\mathcal{O}(nNk)$ for Weighted Least Squares and to $\mathcal{O}(Nk)$ for a single sample in Moving Least Squares. In practice, this seems to be true for Weighted Least Squares. It is not true for Moving Least Squares, as we can see from the figure that its complexity still seems to be equal to our found theoretical complexity of $\mathcal{O}(Nk^3)$ per sampled point. We could also add such a pre-calculation to the Least Squares algorithm, but we chose not to, as Least Squares is already relatively fast when compared to the other two methods and is therefore less in need of such an improvement.

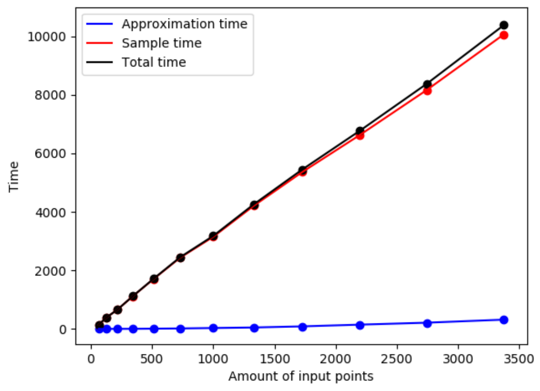
4.5 Three-dimensional modelling

In this section we will implement the method for point cloud reconstruction explained in section 3.1 and see how the methods perform in this implementation. The results will help us compare the three methods in a realistic scenario, but they will mostly illustrate that it is very difficult to accurately compare the three models due to the large amount of parameters involved in the process.

As original input points we have taken meshes that represent the famous Utah teapot [16]. As described in 3.1, we generate a function f and two sets of off-surface points with $\delta = 0.22$. This is around 1 percent of the bounding box of the model, which is described as an ideal value for δ by Wendland [15]. By now approximating this function f with the three different methods and then generating a new triangular mesh using the marching cubes algorithm, we can now create three-dimensional interpolations of the original set of points. The input points are shown in figure 4.5. Notice the small sets of three: these are the original points in combination with their two off-surface points.

The result of interpolating the low-resolution set of points using the different methods is shown in figure 4.9. It is immediately noticeable that the teapot is smoothed by all three methods as small details such as the handle and the rim of the teapot are not well-represented





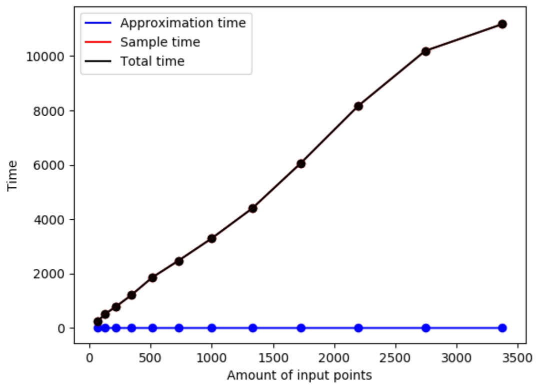


Figure 4.7: The amount of time in seconds that an approximation of the three dimensional function $f(x, y, z) = \cos(x) + \cos(y) + \cos(z)$ on the domain $\{-4 \leq x \leq 4, -4 \leq y \leq 4, -4 \leq z \leq 4 | (x, y, z) \in \mathbb{R}^3\}$ and a marching cubes sampling of the approximation takes as we increase the amount of input points. The marching cubes sampling samples a grid of 51 by 51 by 51 equidistant points (the isosurface used is $f(x, y, z) = -0.3$). The order of the polynomial space $m = 3$. Weighting is done using the Wendland function with $h = 8$. The top-left graph shows the results of LS, those of WLS are shown in the top-right graph and those of MLS are shown in the bottom graph.

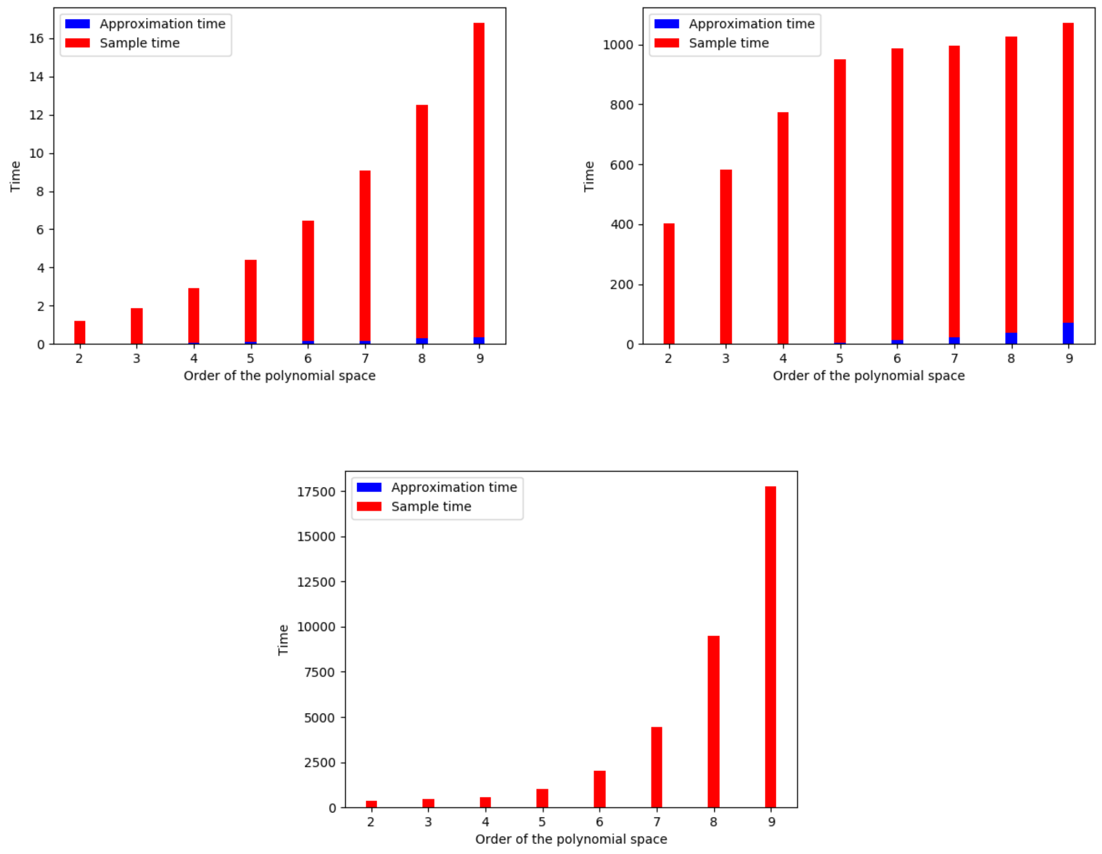


Figure 4.8: The amount of time in seconds that an approximation of the three dimensional function $f(x, y, z) = \cos(x) + \cos(y) + \cos(z)$ on the domain $\{-4 \leq x \leq 4, -4 \leq y \leq 4, -4 \leq z \leq 4 | (x, y, z) \in \mathbb{R}^3\}$ and a marching cubes sampling of the approximation takes as the order of the polynomial space increases. The marching cubes sampling samples a grid of 51 by 51 by 51 equidistant points (the isosurface used is $f(x, y, z) = -0.3$). The input points are a grid of 7 by 7 by 7 equidistant points. Weighting is done using the Wendland function with $h = 8$. The top-left graph shows the results of LS, those of WLS are shown in the top-right graph and those of MLS are shown in the bottom graph.

in the approximations. Another detail that can clearly be noticed from the figure is that all three methods have to create a continuous approximation out of the discontinuous function f. As f is generated from only a small amount of points, we can see that the interpolated models have different closed surfaces than the original model, which is one closed surface. This is why holes are found near the top and the handle in the interpolated models and why there seems to be another surface on top of the teapot in the interpolated model generated by the LS method. This surface is a result of the hole on top of the teapot opening up and as the approximated function is continuous, the method extends this hole into a larger surface. This effect is visible to a lesser extent near the handle and the spout in the MLS and WLS models.

If we compare the results in figure 4.9, it seems as if the Least Squares interpolation is the worst and the Moving Least Squares interpolation is the best. The comparison is a lot more complex in reality. First of all, just one weighting function has been chosen, albeit with an informed choice of the parameter h, but a wide range of different weighting functions exists, which may produce better or worse results. The Wendland function is a function more designed for smoothing, whereas other weighting functions are better for interpolation. These functions all have their own parameters that can be tuned. Next to this, the parameter δ can be changed, which might improve the performance of one or more of the functions. The order of the polynomial space in each approximation has been hand-picked to produce the most beneficial result. Figure 4.11 shows how picking the order of the polynomial space differently can affect the WLS and MLS methods so that they are outperformed by the interpolation generated by the LS method with the same order in figure 4.9. Finally, this experiment shows just one simple set of input points. All these factors show that there is a large difficulty when comparing the performance of the methods and although figure 4.9 is a good indication of the quality of each method, it does not definitively prove which method has the best performance.

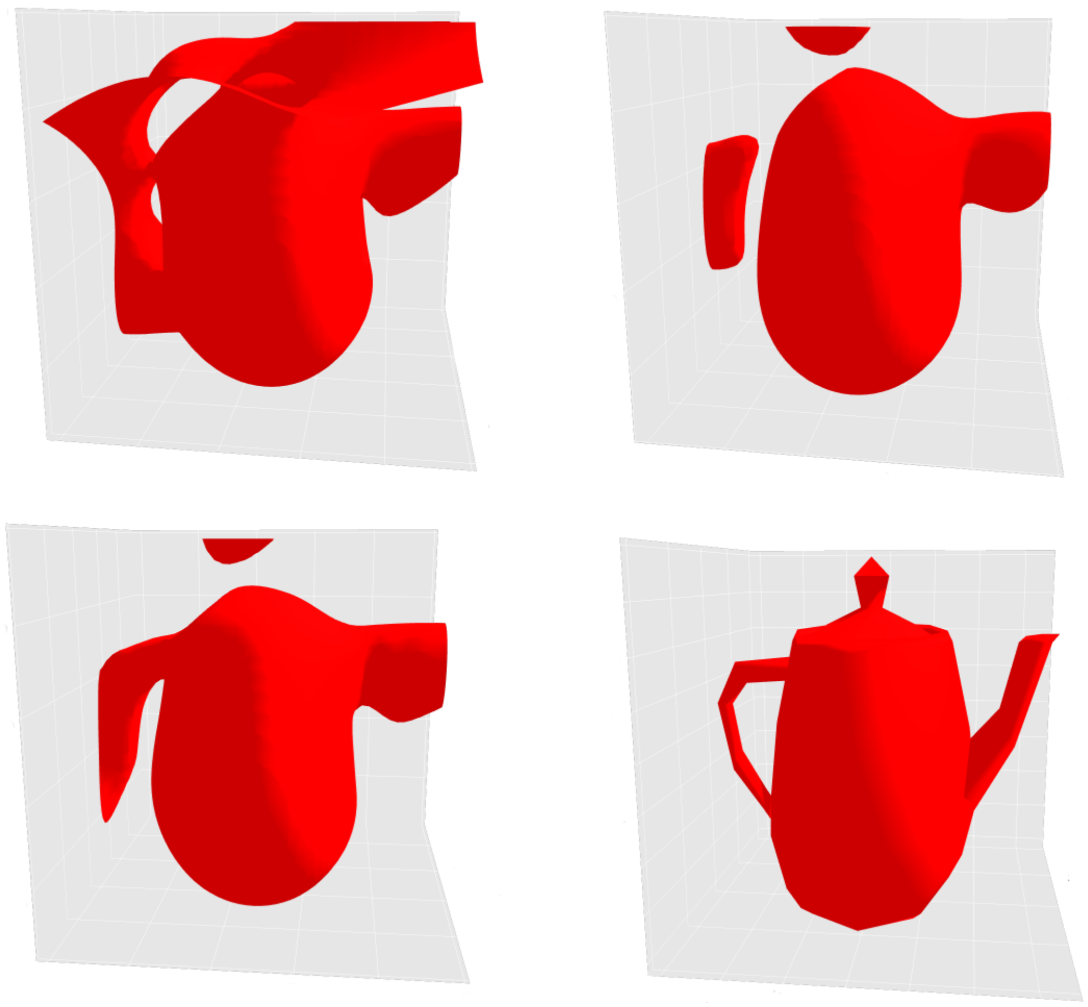


Figure 4.9: The rendering of different approximations of the Utah Teapot model on a bounded box domain in $\mathbb{R}^3$ made using a marching cubes sampling of the approximation. The marching cubes sampling samples a grid of 51 by 51 by 51 equidistant points (the isosurface used is $f(x, y, z) = 0$, where f is the function as described in section 3.1). Weighting is done using the Wendland function with $h = 20$. The top-left graph shows the results of LS with an polynomial space of order 5, the results of WLS are shown in the top-right graph with order 4 and those of MLS are shown in the bottom-left graph with order 4. The bottom-right graph shows the triangulation of the input points used. (There are 137 input points, the δ as described in section 3.1 has a value of 0.22.)

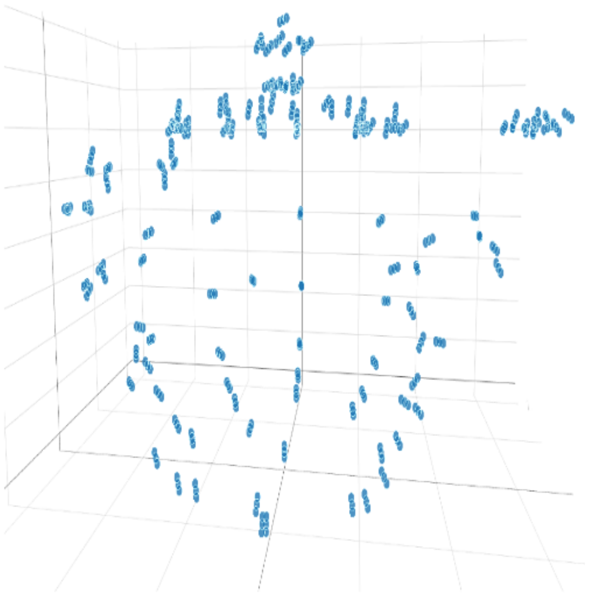


Figure 4.10: The input points for the Utah Teapot model on a bounded box domain, after defining the off-surface points.

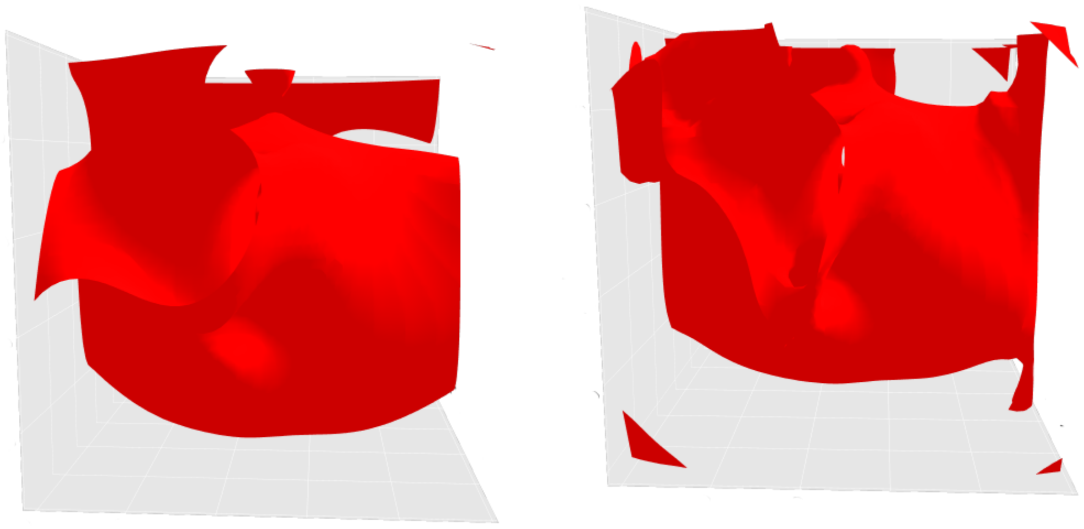


Figure 4.11: The rendering of different approximations of the Utah Teapot model on a bounded box domain in $\mathbb{R}^3$ made using a marching cubes sampling of the approximation. The marching cubes sampling samples a grid of 51 by 51 by 51 equidistant points (the isosurface used is $f(x, y, z) = 0$, where f is the function as described in section 3.1). Weighting is done using the Wendland function with $h = 20$. The left figure shows the WLS method with a polynomial space of order 5 and the right figure shows the MLS method with order 5. (There are 137 input points, the δ as described in section 3.1 has a value of 0.22.)

Chapter 5

Conclusion

All three methods have been examined in detail, by looking at their stability and complexity in theory and comparing this with results obtained in practice. Examples have also been shown of how the different methods can be used when working on the interpolation or reconstruction of three-dimensional models. Such a method has been implemented to observe the behaviour of each method in a practical situation.

If we use these results to compare all three methods to each other, we can conclude that such a comparison is a very difficult one to make. The results give a good idea of what qualities each method has under certain circumstances, but there are still a lot of circumstances unaccounted for. This is due to the fact that there are still a lot of variables that can be changed. Each method produces different results when these are changed, something which we have seen many times throughout this thesis. For example, we have only compared the stability of the methods in practice when approximating a few one-dimensional functions, with just a single weighting function. A more thorough analysis of the stability would examine more functions, in different dimensions, using different weighting functions and so forth. This is clearly out of the scope of this thesis.

Further suggested research would therefore be to perform a more in-depth comparison centred around a single attribute of the methods, such as the stability. For this thesis, we chose to examine the methods in the context of three-dimensional modelling, but this can only give a very wide overview. For a deeper analysis, one can compare the three methods through their quality in a specific implementation from chapter 3, for example. This significantly lowers the amount of different parameters involved in an experiment, which provides the opportunity to pick more interesting parameters, e.g. a weighting function can be chosen so that the approximation of a function is suddenly more accurate.

Comparing the methods in a very specific context with interesting parameters are eventually how a definitive comparison between the three methods can come to light. This comparison will most likely mean that different applications within the three-dimensional modelling field each have a best method, with specific parameters. It is therefore also important to research each method on its own, to examine under what circumstances it behaves the best and under what circumstances it seems to fail. This can give good information as to which parameters can best be used to compare the different methods in further research.

To conclude, we have shown a good overview of the properties of the methods and found that in certain situations there is a more preferable method in terms of speed or stability. To give a definitive answer as to which method is the best in the context of three-dimensional modelling, more research on the behaviour of the methods in specific situations is needed.

Bibliography

[1] Coppersmith, D., and Winograd, S. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation 9*, 3 (1990), 251–280.

[2] Cuomo, S., Galletti, A., Giunta, G., and Starace, A. Surface reconstruction from scattered point via rbf interpolation on gpu. In *2013 federated conference on computer science and information systems* (2013), IEEE, pp. 433–440.

[3] Dahlquist, G., and Bjork, A. Equidistant interpolation and the runge phenomenon. *Numerical Methods* (1974), 101–103.

[4] Epperson, J. F. On the runge example. *The American Mathematical Monthly 94*, 4 (1987), 329–341.

[5] Fries, T.-P., Matthies, H., et al. Classification and overview of meshfree methods (revised). *Informatikbericht Nr.:2003-3* (2004).

[6] Gautschi, W., and Inglese, G. Lower bounds for the condition number of vandermonde matrices. *Numerische Mathematik 52*, 3 (1987), 241–250.

[7] Lancaster, P., and Salkauskas, K. Surfaces generated by moving least squares methods. *Mathematics of computation 37*, 155 (1981), 141–158.

[8] Lewiner, T., Lopes, H., Vieira, A. W., and Tavares, G. Efficient implementation of marching cubes' cases with topological guarantees. *Journal of graphics tools 8*, 2 (2003), 1–15.

[9] Lorensen, W. E., and Cline, H. E. Marching cubes: A high resolution 3d surface construction algorithm. In *ACM siggraph computer graphics* (1987), vol. 21, ACM, pp. 163–169.

[10] Nealen, A. An as-short-as-possible introduction to the least squares, weighted least squares and moving least squares methods for scattered data approximation and interpolation. *URL: http://www. nealen. com/projects 130*, 150 (2004), 25.

[11] Shen, C., O'Brien, J. F., and Shewchuk, J. R. Interpolating and approximating implicit surfaces from polygon soup. In *ACM Siggraph 2005 Courses* (2005), ACM, p. 204.

[12] Shepard, D. A two-dimensional interpolation function for irregularly-spaced data. In *Proceedings of the 1968 23rd ACM national conference* (1968), ACM, pp. 517–524.

[13] Tyrtyshnikov, E. E. How bad are hankel matrices? *Numerische Mathematik 67*, 2 (1994), 261–269.

[14] Wendland, H. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in computational Mathematics 4*, 1 (1995), 389–396.

[15] Wendland, H. *Scattered data approximation*, vol. 2. Cambridge University Press, 2005.

[16] Yuksel, C. Ray tracing for graphics course, project 5. https://graphics.cs.utah.edu/courses/cs6620/fall2013/?prj=5, 2013. Accessed: 2019-05-30.