Investigations on Improving the SEE-GRID Optimization Algorithm

Masterarbeit zur Erlangung des akademischen Grades

Diplomingenieur

im *Masterstudium Technische Mathematik*

Angefertigt am Institut für Symbolisches Rechnen (RISC)

Eingereicht von:

Johannes Watzl

Betreuung:

A. Univ. Prof. DI Dr. Wolfgang Schreiner

Linz, Juni *2008*

Eidesstattliche Erklärung

Ich erkläre an Eides statt, daß ich die vorliegende Diplomarbeit selbständig und ohne fremde Hilfe verfaßt habe. Ich habe dazu keine weiteren als die angeführten Hilfsmittel benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet.

Linz, 10. Juni 2008

Unterschrift

Acknowledgements

I would like to thank Prof. Schreiner for supervising this work and his support. Michael Buchberger, Thomas Kaltofen and the people at the Upper Austrian Research GmbH have provided assistance in numerous ways. I am grateful for the help of Johann Messner and Christian Schaubschläger at the supercomputer. I want to thank my parents for facilitating this work. Finally, I would like to express my gratitude to my family and friends for their inspiration and mental support.

Abstract

In this work various ways to accelerate the computation process in the SEE-KID/SEE-GRID software for the biomechanical simulation of the human eye are investigated. Both sequential and parallel strategies are discussed. To improve the sequential optimizer, the Broyden update method is utilized. A strategy to interpolate function values is first found on the basis of the Delaunay algorithm. An enhanced strategy was developed based on the special regularity of the mesh used in the interpolation program. The strategies were implemented and the results of the benchmarks are presented together with the source code.

In dieser Arbeit werden Wege vorgestellt, die Berechnungszeit der mathematischen Kernprozesse in SEE-KID/SEE-GRID zur biomechanischen Simulation des menschlichen Auges zu verkürzen. Sowohl sequentielle als auch parallele Strategien zur Beschleunigung der Berechnung werden aufgezeigt. Um den sequentiellen Optimierungsalgorithmus zu verbessern, wird die Methode des Broyden Updates eingeführt. Eine Strategie die Funktionswerte zu interpolieren, stellt die Benützung des Delaunay Algorithmus dar. Aufbauend auf der Idee der Interpolation wird ein Modell entwickelt, das sich die Regularität, des in unserem Falle verwendeten Gitters für die Interpolation zunutze macht. Alle diese Strategien wurden implementiert, beschrieben und getestet. Die jeweiligen Berechnungszeiten wurden, wie auch der Quelltext der entwickelten Programmabschnitte, in der Arbeit ausgeführt.

Contents

List of Figures

Chapter 1

Introduction

In this thesis different approaches of accelerating the computation kernel of the SEE-KID/SEE-GRID software ([7], [6], [26], [27], [1], [8], [9], [10]) are investigated. The software is designed to help doctors treat eye motility disorders. The major fields of application of the program are corresponding the simulation of pathologies of the human eye concerning strabism and eye surgeries based on a biomechanical model of the human eye.

SEE-KID is developed at the Upper Austrian Research GmbH (UAR) headed by Michael Buchberger and Thomas Kaltofen; SEE-GRID is a grid variant of the software developed in cooperation with the Research Institute for Symbolic Computation (RISC) of the Johannes Kepler University Linz by Károly Bósa and Wolfgang Schreiner. In SEE-KID/SEE-GRID results are gained by optimization of a certain function called *torque function*. This nonlinear function is used to describe eye positions with given eye parameters of vice versa. The optimization is performed by the *Levenberg Marquardt algorithm* ([3], [13]) which combines robustness in calculation with nearly quadratic convergence.

The goal of this work is to find a possible strategy to accelerate the computation kernel. First, the investigations were focused on the optimization process. The idea of parallelizing the optimizer by decomposing the matrices was deemed as an impossible approach because the dimensions of the utilized matrices are too small. A way to enhance the sequential optimization process is the so called *Broyden update method* ([2], [3], [13], [12]). Making use of this method leads to decreasing computation times by just updating the Jacobian and Hessian matrices used by the optimizer instead of recomputing it in every step.

A strategy for using parallelizm was found in generating and decomposing a grid over the torque function's domain. The gridpoints store torque function values which are used to interpolate values of points between the

gridpoints. The first approach for implementing this idea was the *Delaunay algorithm* ([17]). With this algorithm a cluster of points can easily be triangulated. The function value of a certain point is then computed by interpolation from the three corner points of the surrounding triangle.

The usage of the Delaunay algorithm represents an overkill because it can deal with irregular mesh structures as well. However, an improved interpolation method was found by making use of the regularity of the mesh. The regular mesh structure in this approach replaces the Delaunay algorithm's triangle search by a lookup in the regular mesh structure.

Finally, methods for parallelizing the regular mesh strategy were developed. For the implementation of the parallel parts of the program the functionality of the programming APIs POSIX ([22], [23], [24], [5], [29], [15]) and OpenMP ([14], [11], [18], [4]) was used.

The remainder of the thesis is structured as follows: In Chapter 2 a detailed description of the biomechanical model of the human eye, the torque function and the structure of the optimization process is given. Chapter 3 first introduces nonlinear optimization, describes the Levenberg Marquardt algorithm used in the software and shows a way to accelerate the sequential optimization – the Broyden update. The Delaunay triangulation algorithm together with interpolation is discussed in Chapter4. Chapter 5 deals with an interpolation method using the regular structure of the mesh over the torque function's domain. Chapter 6 outlines possible ways for parallelizing the interpolation method developed in Chapter 5.

Chapter 2

Description of the Problem

2.1 Medical Overview

This chapter gives an introduction in the anatomy of the human eye with attention to the extraocular eye muscles. An overview of strabismus is shown and the treatments of eye motility disorders are explained. The description of mathematical model of the human eye takes the main part of this chapter. This chapter is based on the PhD thesis by Michael Buchberger [7].

2.1.1 Basic Anatomical definitions

To control the movements of the human eye we have a set of six extraocular muscles surrounding the eyeball (Figure 2.1). The following list shows the names and the types of movements performed by each of the six muscles (Figure 2.2):

- rectus superior muscle for upward movements
- rectus inferior muscle for downward movements
- rectus lateralis muscle for sideways outside movements
- rectus medialis muscle for sideways inside movements
- obliquus superior muscle for upward and outside movements
- obliquus inferior muscle for downward and inside movements

To avoid the slipping off the eyeball during certain movements the eye muscles have to be stabilized. This stabilization is done by a special tissue

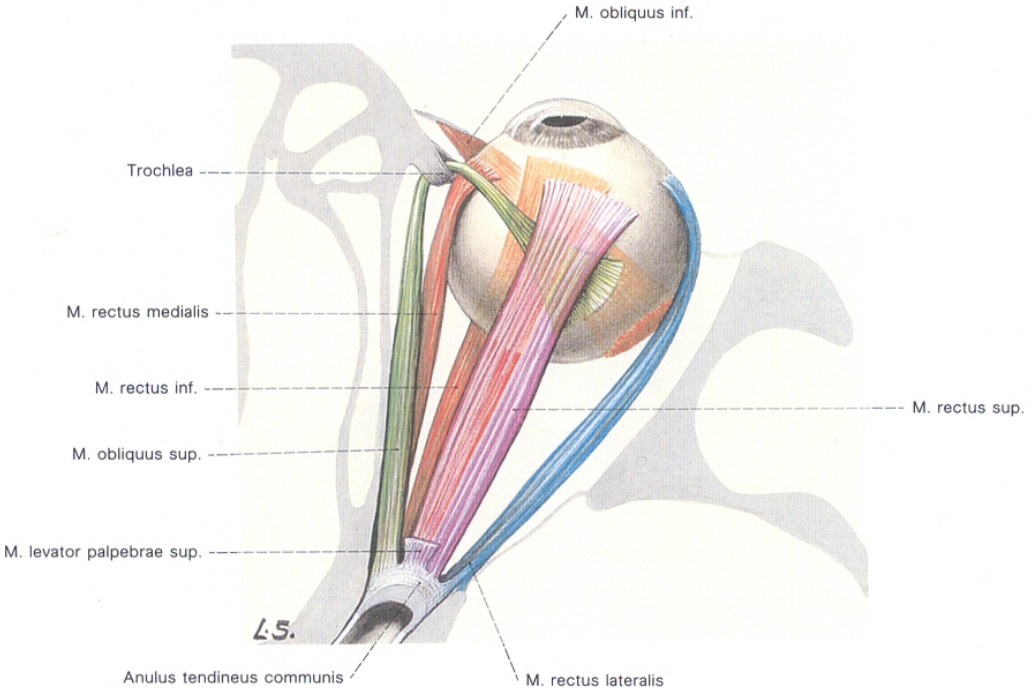

Figure 2.1: The eye muscles [20]

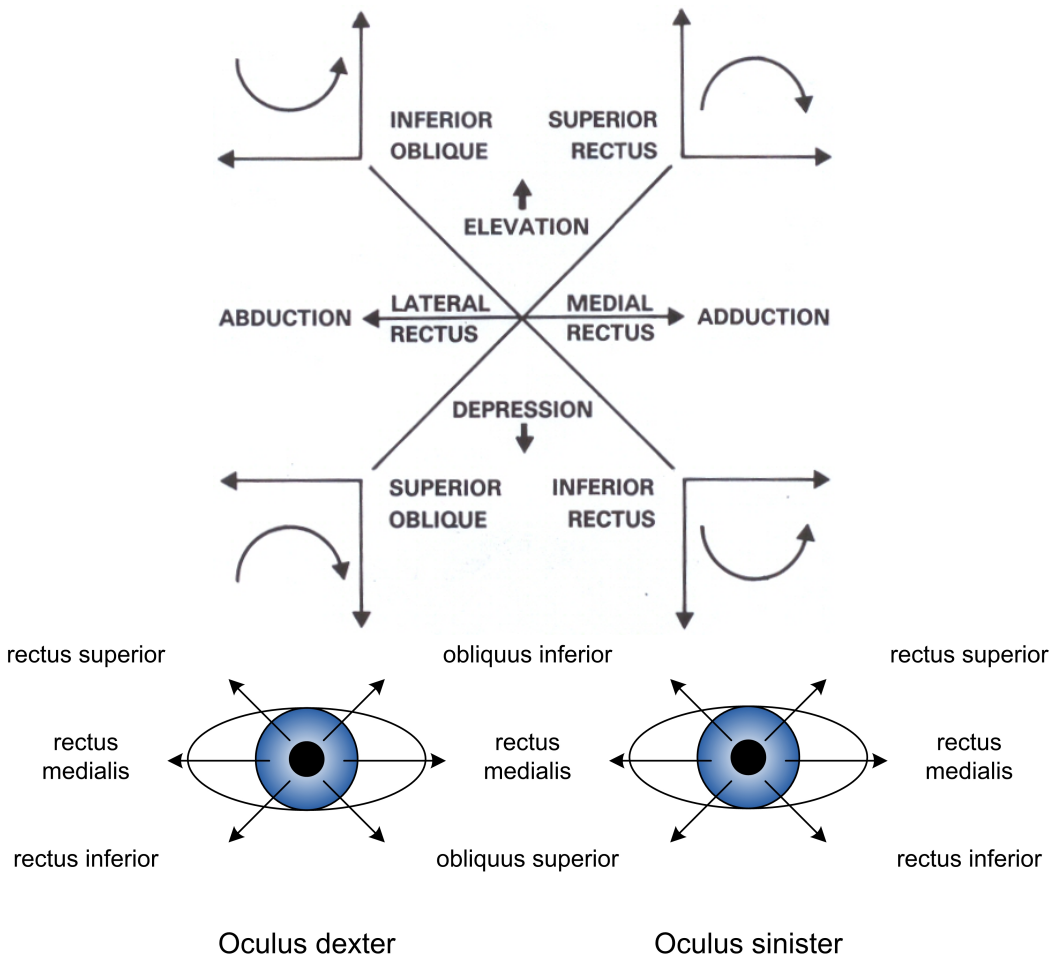

Figure 2.2: Relationship between the eye muscles and the types of movements (Top picture: [32])

represented by a set of pulleys surrounding every muscle at a specific position. The length of the muscles during moving the eyeball is eminently influenced by these pulleys. So they have to be treated carefully during surgical corrections of the muscle length to avoid negative side affects of the operation.

2.1.2 Strabismus - Forms, Classes, Tests and Treatment

Normally the eyes can only be moved together. The eyes' movements from one position to another are performend very fast and certain positions can be fixed for longer periods of time without getting exhausted. During the process of seeing, the light reflected by the surrounding (three dimensional) objects goes through the lens and the vitreous body leading to an upside down picture on the retina. So we get two slightly different upside down two dimensional pictures on the two retinas. These two pictures are flipped and merged in the brain to produce one picture experienced as a three dimensional one. If the difference between the two pictures is to big the brain cannot merge them, which represents one form of strabismus. In early childhood the brain is able to "delete" one of the two pictures delivered by the pathological seeing mechanism. The big problem now is that one eye is permanently "switched off" and is not used anymore. So children suffering from strabismus should be treated very early. Otherwise the result can lead to partial or even total blindness of one eye.

In general there are four main forms of strabismus which can be observed in patient's eye positions (Figure 2.3).

- Esotropia (a): one eye looks inwards while the other eye looks straight on
- Exotropia (b): one eye looks outwards while the other eye looks straight on
- Hypertropia (c): one eye looks upwards while the other eye looks straight on
- Hypotropia (d): one eye looks downwards while the other eye looks straight on

The sources of strabismus can be divided in three classes.

- **Concomitant Strabismus**

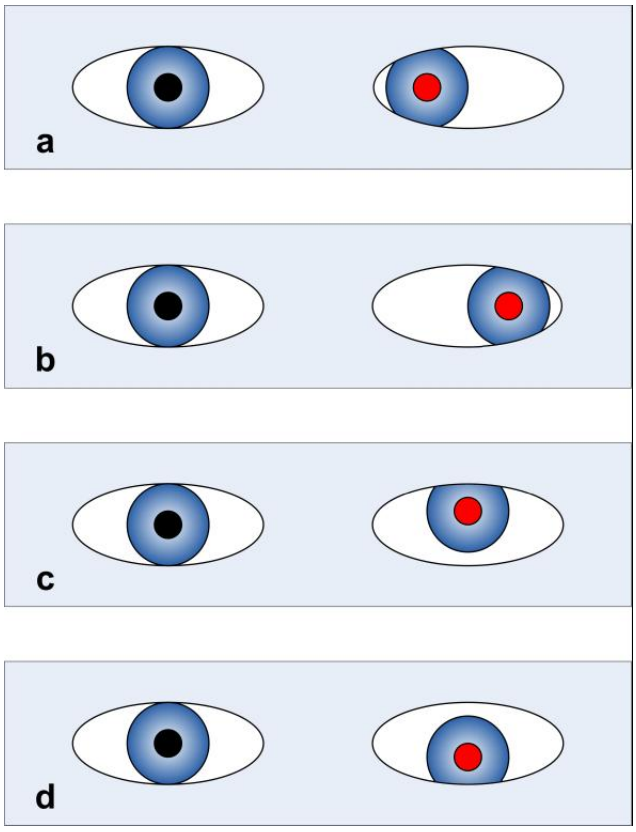

Figure 2.3: The forms of strabismus

- **Inconcomitant Strabismus**
- **Paralytic Strabismus**

Concomitant Strabismus Congenital eye motility disorders are called concomitant strabismus.

Inconcomitant Strabismus The symptoms of incomitant strabismus are dissociations of the ocular movements. The brain commands the muscles correctly, but due to defects of the motor path of the binocular reflexes, the movements are not carried out in the right way.

Paralytic Strabismus Paralytic strabismus shows in the paralyzation of one or more muscles or nerves. These palsies can be congenital or the result of certain diseases. There are several tests for diagnosing strabismus. This tests usually work with lenses, patterns and/or the coverage of one eye. One of the tests, the Hess-Landaster test carried out in the following way: The patient wearing red-green glasses gets a green light pointer and a green filter in front of one eye which is the fixing eye in this situation whereas the doctor performing the test gets a red light pointer. Now the doctor projects a red light spot on a certain canvas called "Hess screen". The patient now has to put its green light spot at the position of the red one. This procedure is done a certain number of times and leads to a certain pattern called Hess diagram. The test is repeated with the other eye and a red filter. So we get two Hess diagrams, one for each of the two eyes. If the two eyes are healthy the red and green points overlay in all measured positions.

Hess diagrams are very helpful for doctors because the diagnosis depends on the form of the Hess diagrams and can even derived directly from them.

Treatment There are three forms of treating strabismus.

- Eyeglasses
- Occlusion treatment
- Surgery

The therapy with eyeglasses is usually done when squinting children are far-sighted. The squinting of these children is the result of trying to fixate near objects. Occlusion treatment is applied not only as is but as additional treatment to glasses or after operations. This form of treatment often lasts a period of several years. Both the squinting and the healthy eye are covered

alternately with a special plaster to exercise the squinting eye. This treatment works with children in the growth phase. The third form of strabismus treatment are surgical operations of the extraocular eye muscles. This operations are only applied if the children can be examined sufficiently, glasses are worn reliably and the eyesight of both eyes is nearly equal. There are three main surgical goals - the weakening surgery, the strengthening surgery and the transposition surgery. Weakening or recession surgery is done in order to reduce the traction of a certain muscle and strengthening or resection surgery is performed to raise its traction. If one needs to change the pulling direction of a certain muscle transposition surgery is applied.

2.2 Specification of the Biomechanical Model

2.2.1 Mathematical Description of Eye Positions and Rotational Movements

The current position of the eye can be interpreted as two or three dimensional. These position is represented by (α, β) for the 2D position and (α, β, γ) for the 3D position. The components of the vectors are angles containing the two or three angles. To describe 2D or 3D eye positions, we first have to define the position of the coordinate systems for the eyes. The center of this coordinate systems lies in the center of the eyeball. Movements of the eye are interpreted as rotations around these axes (Figure 2.4).

- x-axis (X): elevation, depression
- y-axis (Y): ab-, adduction
- z-axis (Z): intorsion, extorsion

One part of our model is the calculation of 3D eye positions out of 2D eye positions. This calculation can be deduced from the so called *Listing's Law*. As explained before 2D eye positions are represented by a vector (α, β) with the torsional component γ missing. The relationship between 2D and 3D eye positions can be expressed by the mapping

$$(\alpha, \beta) \xrightarrow{Rel_{Listing}} (\alpha, \beta, \otimes(\alpha, \beta))$$

with the torsional component $\gamma = \otimes(\alpha, \beta)$ which is defined by

$$\otimes(\alpha, \beta) = \cos^{-1} \frac{\cos\alpha + \cos\beta}{1 + \cos\alpha\cos\beta}$$

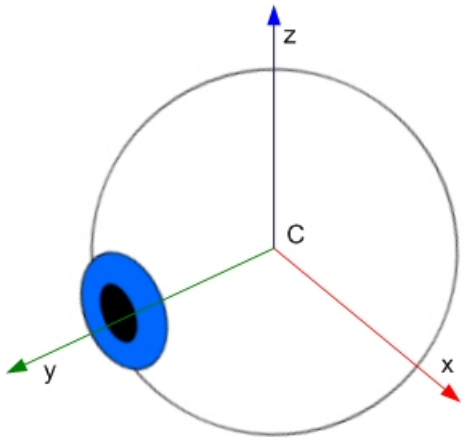

Figure 2.4: Eye coordinate system

There are two eye muscles per axis carrying out the movements. Let (α, β, γ) be the angles rotating around the corresponding axes of the coordinate system (Z, X, Y). (The order of the axes was defined by Fick; we will speak of a Fick sequence in further definitions. For expressing an eye position one needs only the vector of the three angles (α, β, γ). In our model quaternions are used to deal with rotations instead of rotational matrices. Quaternions are a generalization of complex numbers. We use quaternions of the form

$$q = a + bI + cJ + dK$$

to describe eye positions in our model. This quaternion can be written as

$$q = [a, (b, c, d)]$$

where $Scal(q) = a$ is the scalar part and $Vect(q) = (b, c, d)$ denotes the vector part of q. For quaternions there exist operations for the basic arithmetic operations, for inversion, conjugation and for the quaternion power operation. These operations are used for expressing rotations represented by quaternions. Let q be a rotation quaternion, then the quaternion v_p can be rotated by multiplication on the right side with the rotation quaternion and on the left with the inverse of the rotation quaternion.

$$v'_p = Rot(v_p, q) = q^{-1} v_p q$$

For defining the complete rotation quaternion

$$q_{Listing} = q_{rx} q_{ry} q_{rz}$$

we need the base vectors of our eye coordinate system $\overrightarrow{XA}(1,0,0)$, $\overrightarrow{YA}(0,1,0)$ and $\overrightarrow{ZA}(0,0,1)$. The rotations quaternions q_{rz}, q_{ry} and q_{rx} are computed as follows.

$$q_{rx} = [\beta, (\overrightarrow{XA'})]$$

$$q_{ry} = [\gamma, (\overrightarrow{YA'})]$$

$$q_{rz} = [\alpha, (\overrightarrow{ZA})]$$

with $\overrightarrow{XA'} = Vect(Rot(\overrightarrow{XA}, q_{rx}))$ and $\overrightarrow{YA'} = Vect(Rot(\overrightarrow{YA}, q_{ry}))$

2.2.2 Mathematical Description of Extraocular Eye Muscle Actions

To describe extraocular eye muscle actions we have to make some definitions (Figure 2.5).

- *insertion point* (I) of a muscle: The point on the eyeball where this muscle is attached to the eyeball.
- *origin* (O): the end of the muscle fixed on the cranial bone
- *tangency point* (T): the point on the eyeball where the muscle first touches the eyeball.
- *action circle* (ac) of the muscle: a circle around the eyeball lying in the plane of muscle force operating.
- *arc of contact*: the line on the eyeball from insertion to tangency point.

There are some historical models to describe eye muscle actions. In these models the pulleys are not treated adequate, which means that the slipping off of the muscles from the eyeball is less or even not implemented in these models. A complete description of these models can be found in [7] starting on page 107. In the implementation of the SEE-KID/SEE-GRID software, the so called pulley model is used. This model considers the action of pulleys. In the pulley model a new poithe force directing reference position.

The center C_c of the action circle has to be computed separately for every eye position because in certain eye positions the original center of the action circle is different from the center in the pulley model (Figure 2.6). This center is calculated with I, P and the center denoted by C as described below. First we need three vectors $(\overrightarrow{SX}, \overrightarrow{SY}, \overrightarrow{SZ})$ with

$$\overrightarrow{SX} = \overrightarrow{SY} \times \overrightarrow{SZ}$$

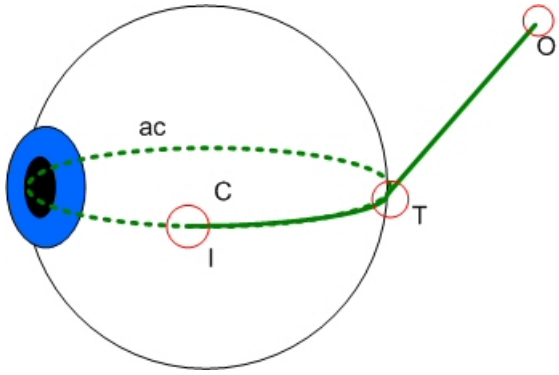

Figure 2.5: Defined points inside and around the eyeball

$$\overrightarrow{SY} = \frac{\overrightarrow{P} \times \overrightarrow{I}}{|\overrightarrow{P} \times \overrightarrow{I}|}$$

$$\overrightarrow{SZ} = \frac{\overrightarrow{I}}{|\overrightarrow{I}|}$$

where $\overrightarrow{I}$ denotes the vector from the center of the coordinate system to the insertion point. These three vectors are needed for the computation of $(\overrightarrow{TX}, \overrightarrow{TY}, \overrightarrow{TZ})$.

$$\overrightarrow{TX} = Vect(Rot(\overrightarrow{SX}, q))$$

$$\overrightarrow{TY} = Vect(Rot(\overrightarrow{SY}, q))$$

$$\overrightarrow{TZ} = Vect(Rot(\overrightarrow{SZ}, q))$$

For the calculation of the center of the action circle, the rotation quaternion q describing the current eye position is used to define three vectors $(\overrightarrow{GX}, \overrightarrow{GY}, \overrightarrow{GZ})$ with

$$I' = Vect(Rot(\overrightarrow{I}, q))$$

$$\overrightarrow{GX} = \overrightarrow{GY} \times \overrightarrow{GZ}$$

$$\overrightarrow{GY} = \frac{\overrightarrow{P} \times \overrightarrow{I'}}{|\overrightarrow{P} \times \overrightarrow{I'}|}$$

$$\overrightarrow{GZ} = \frac{\overrightarrow{I'}}{|\overrightarrow{I'}|}$$

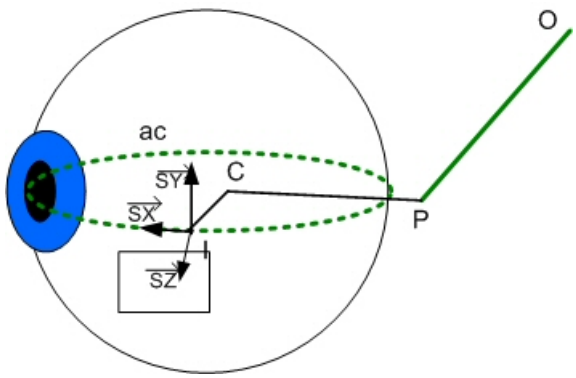

Figure 2.6: Defined points inside and around the eyeball in the pulley model

For the final computation of the muscle action circle in the pulley model, we need an angle value (side-slip angle) called θ calculated as

$$\theta = -\alpha \tan^{-1} \frac{\overrightarrow{GX}\overrightarrow{TY}}{\overrightarrow{GX}\overrightarrow{TX}}$$

where α is a so called *side-slip scalar* which is different for each of the six eye muscles and is derived from the EyeLab model [19]. The exact values for α can be found in [7] on page 115. With vector

$$\overrightarrow{D} = \overrightarrow{TY} \sin\theta - \overrightarrow{TX} \cos\theta$$

and vector

$$\overrightarrow{N} = \frac{\overrightarrow{D} \times \overrightarrow{I'P}}{|\overrightarrow{D} \times \overrightarrow{I'P}|}$$

we are able to define the center

$$C_c = (\overrightarrow{I'}\overrightarrow{N})\overrightarrow{N}$$

As a further major thing we have to deal with the eye muscles. First we define the length of the eye muscles denoted by l.

$$m_l = d_{arc} + l_1 + l_2$$

$$d_{arc} = rad \cdot \alpha$$

with

$$rad = |\overrightarrow{I'}|$$

and

$$\alpha = \cos^{-1} \overrightarrow{I'\overrightarrow{T}}$$
$$l = |\overrightarrow{P} - \overrightarrow{C_C}|$$

The length from the origin point of the muscle to the pulley point l_1 and the length from the pulley point the point of tangency l_2 is given by

$$l_1 = |\overrightarrow{P}O|$$

$$l_2 = \sqrt{\overrightarrow{P}\overrightarrow{P} - \overrightarrow{I'}\overrightarrow{I'}} - T_l$$

with T_l denoting the length of the muscle tendon. The passive length change of a muscle dl can be computed with

$$dl = \frac{100(m_l - L_0)}{L_0}$$

where L_0 is the muscle length of a relaxed muscle without tendon. To describe the muscle force we make use of the muscle length and tension with the *Length-Tension Relationship*:

$$F_i(dl_i, e_i) = \lambda_i \left(\frac{k}{2}(dl_i + e_i) + \sqrt{\frac{k^2}{4}(dl_i + e_i)^2 + a^2} \right)$$

$i = 1, 2, \ldots, 6$ (The index i represents the ith eye muscle.) The parameters k and a denote the asymptotic slope and the sharpness of the curvature. e_i is the elastic strength which scales the passive force function influencing elastic properties of a muscle. dl_i denotes the changes in muscle length and λ scales the total force function (passive and active force). In addition to the parameters above we need the contractile strength c_s scaling the active force function, the total strength t_s scaling the total force function in relation to the current muscle and the tendon length T_l which gives us the tendon length. Innervations of the eye muscles are defined by

$$iv_i(e_i) = F_i(0, e_i) - F(0, e_0)$$

with a certain constant value e_0 where F_i is the total isometric force and $F(0, e_0)$ is the passive isometric force for zero length change with e_0 given as constant value. $F(dl, e_0)$ is defined by

$$F(dl, e_0) = \lambda \left(\frac{k}{2}(dl + e_0) + \sqrt{\frac{k^2}{4}(dl + e_0)^2 + a^2} \right)$$

where λ, k and a denote constant values. The relationship between elastic force $F_E(dl) = F(dl, e_0)$ and contractile force $F_C(dl, i_v) = F(dl, i_v) - F_E(dl)$ is defined in the following equation:

$$F_T(dl, iv) = F_E(dl) + F_C(dl, iv)$$

By using the scaling parameters e_s, c_s and t_s we define the scaled force function

$$F_T(dl, iv, e_s, c_s, t_s) = t_s(F_E(dl)e_s) + F_C(dl, iv)c_s))$$

For solving kinematic operations, we require a so called stable eye position. To express the stable eye position we make use of the torque imbalance vector

$$\overrightarrow{t} = \sum_{i=1}^{6} F_{T_i}(dl_i, iv_i, e_{si}, c_{si}, t_{si}) \cdot \overrightarrow{n_i}$$

where n_i denotes the ith element of the unit moment vector computed by

$$\overrightarrow{N} = \frac{\overrightarrow{D} \times \overrightarrow{I'P}}{|\overrightarrow{D} \times \overrightarrow{I'P}|}$$

with

$$\overrightarrow{D} = \overrightarrow{TY} \sin\theta - \overrightarrow{TX} \cos\theta$$

A stable eye position is achieved when:

$$|\overrightarrow{t}| \approx 0$$

2.2.3 Refined Mathematical Model

In this basic version of the torque imbalance vector mentioned above, orbital restoring forces and globe translations are not considered. An orbital restoring force denoted by $\overrightarrow{P}$ is applied by elastic tissues in the eye orbit. $\overrightarrow{P}$ depends on torsional stiffness constants. Globe translations are very useful for the diagnosis. They describe a movement of the eyeball during eye movements. For describing globe translation, we have to define a new coordinate system lying between the origin points of the rectus superior and the rectus inferior muscle denoted by O_{sr} and O_{ir}. The apex point in the posterior region of the orbit is defined as

$$\overrightarrow{V} = \frac{\overrightarrow{O_{sr}} + \overrightarrow{O_{ir}}}{2}$$

and

$$\overrightarrow{A_y} = \overrightarrow{V}$$
$$\overrightarrow{A_x} = \overrightarrow{H_z} \times \overrightarrow{A_y}$$
$$\overrightarrow{A_z} = \overrightarrow{A_x} \times \overrightarrow{A_y}$$

denotes the new apex coordinate system. For the definition of the rotation quaternions for the transformation between the apex and the head-fixed coordinate system we need two rotation angles.

$$\psi = \cos^{-1} \overrightarrow{A_y} \cdot \overrightarrow{H_y}$$

$$\omega = \cos^{-1} \overrightarrow{A_z} \cdot \overrightarrow{H_z}$$

Now the quaternion for the forward transformation (from the head-fixed to the apex coordinate system) is denoted by

$$q_{apex} = [\omega, \overrightarrow{H_x}] \cdot [\psi, \overrightarrow{H_z}]$$

and the other way (from apex to head-fixed coordinates) by

$$q_{head} = q_{apex}^{-1}$$

For using the globe translation to transform the torque imbalance vector $\overrightarrow{t}$ to apex coordinates we use the translation vector $\overrightarrow{G_{trans}}$. To be able to do this we have to introduce a stiffness vector called F_a containing three constant values. This leads us to the definition of the translation vector $\overrightarrow{G_{trans}}$ and the amount of globe translation gt.

$$\overrightarrow{G_{trans}} = \frac{Vect(Rot(\overrightarrow{t}, q_{apex}))}{2\overrightarrow{F_a}}$$

$$gt = |\overrightarrow{G_{trans}}|$$

The translation vector to modify ocular geometry using the head-fixed coordinate system is denoted by

$$\overrightarrow{Trans} = Vect(Rot(\overrightarrow{G_{trans}}, q_{head}))$$

The transformation of the muscle rotations axis in the pulley model to a displaced muscle and pulley origin is done by

$$G_t(\overrightarrow{Trans}, \overrightarrow{RA}) := \overrightarrow{RA} \rightarrow -\overrightarrow{Trans}$$

with the muscle rotation axis $\overrightarrow{RA} = (ra_1, ra_2, ra_3)$. In the pulley model $\overrightarrow{RA} = \overrightarrow{N}$. $-\overrightarrow{Trans}$ translates the origin point O and pulley point P of each muscle in the opposite direction.

The notation $\overrightarrow{RA} \rightarrow$ is for expressing that the axes are calculated with modified pulley and origin data. With this new definitions we can give the refined torque imbalance equation

$$\overrightarrow{T} = \overrightarrow{P} + \sum_{i=1}^{6} \overrightarrow{F_{T_i}}(dl_i, iv_i, e_{si}, c_{si}, t_{si}) \cdot G_t(\overrightarrow{Trans}, \overrightarrow{ra_i})$$

To control the value of $\overrightarrow{T}$ we define the two vectors $\overrightarrow{I_v}$ and $\overrightarrow{E_p}$. $\overrightarrow{I_v} = (iv_1, iv_2, iv_3, iv_4, iv_5, iv_6)$ contains the innervation values for each eye muscle and $\overrightarrow{E_p} = (e_x, e_y, e_z)$ contains the position values based on rotation quaternions. The stable eye position can be found by the minimization of the following function called *torque function*.

$$L_T(\overrightarrow{I_v}, \overrightarrow{E_p}, e_{si}, c_{si}, t_{si}) = \overrightarrow{P} + \sum_{i=1}^{6} \overrightarrow{F_{T_i}}(dl_i, iv_i, e_{si}, c_{si}, t_{si}) \cdot G_t(\overrightarrow{Trans}, \overrightarrow{ra_i})$$

There is no function parameter dl because the six values dl_i are computed using E_p.

We introduce

$$L_T(\overrightarrow{I_v}, \overrightarrow{E_p})$$

as a shortcut for

$$L_T(\overrightarrow{I_v}, \overrightarrow{E_p}) = L_T(\overrightarrow{I_v}, \overrightarrow{E_p}, e_{si}, c_{si}, t_{si})$$

For expressing the following minimization problem the standard form of [13] is used. The goal is to find values for $\overrightarrow{I_v}$ and $\overrightarrow{E_p}$ with

$$\underset{(\overrightarrow{I_v}, \overrightarrow{E_p})}{min}\ L_T(\overrightarrow{I_v}, \overrightarrow{E_p})$$

The minimization problem is a least squares problem which is solved in the SEE-KID/SEE-GRID software with the *Levenberg-Marquardt algorithm* ([3], [13]).

2.2.4 Kinematic Operations, Torque function, Optimization

Now we are able to define the terms **forward** and **inverse kinematics**.

- Forward kinematics: to compute an eye position out of a given set of innervations.
- Inverse kinematics: to compute a set of innervation out of a given eye position.

2.2.5 Forward Kinematics

In forward kinematics (find a stable eye position with a given set of innervations) the minimization problem is of the form:

$$E_{p_{min}}(\overrightarrow{I_v}) := \underset{\overrightarrow{E_p}}{min} L_T(\overrightarrow{I_v}, \overrightarrow{E_p})$$

In this case $\overrightarrow{I_v}$ is constant in the minimization.

2.2.6 Inverse Kinematics

If we solve the problem for inverse kinematics ($\overrightarrow{E_p}$ is given and constant) we want to determine the innervation vector $\overrightarrow{I_v}$. In this case we get the following minimization problem:

$$I_{v_{min}}(\overrightarrow{E_p}) := \underset{\overrightarrow{I_v}}{min} L_T(I_{v_o}(\overrightarrow{I_v}), \overrightarrow{E_p})$$

In the inverse kinematics problem we are able to reduce the degrees of freedom from 6 to 3 with the usage of the odd innervation vector I_{v_0}. The even innervation vector can be calculated with the help of Sherrington's law of reciprocal innervations:

$$I_{v_e}(I_{v_o}) = \frac{(h+w)^2}{I_{v_o} + w} - w$$

where the form

$$I_v = (I_{vo(1)}, I_{ve(1)}, I_{vo(2)}, I_{ve(2)}, I_{vo(3)}, I_{ve(3)})$$

of I_v is used with $I_{vo(i)}$ representing the three innervations with odd muscle indices and $I_{ve(i)}$ denoting the innervations with even muscle indices. In the SEE-KID model it is also possible to use brainstem data (e.g. for nerve palsies). There a matrix D_n is used which scales the innervation vector $\overrightarrow{I_v}$. The dimension of the matrix is 6×6 and the matrix elements s_{ij} express the scaling parameters for the nerve i and the muscle j. So the influence of the matrix D_n is given by

$$\overrightarrow{I'_v} = (\overrightarrow{I_v} \cdot D_n)^T$$

where I'_v is the scaled version of I_v.

2.2.7 Hess-Lancaster Test Simulation

The minimization of the torque function is used for the simulation of the Hess-Lancaster test. For this simulation a so called reference eye (healthy) is used. First we need a starting set of innervations corresponding to $\overrightarrow{P}$ describing a 2D eye position. If the innervations of the reference eye lead the following eye's 2D position to a matching position to $\overrightarrow{P}$, then a valid 3D eye position is found. The fixing eye parameters and the 3D eye position is used to compute the innervations for the six eye muscles. With this innervations the fixing eye is brought in the corresponding position. Then the innervations are used to compute the reference eye's position. If the calculated positions of the fixing eye and the reference eye are different, the fixing eye is pathological. To get the position of the following eye the the intended fixation position of the fixing eye is mirrored.

Chapter 3

Accelerating the Sequential Optimization Algorithm

In the following chapter we give a general overview of nonlinear optimization algorithms starting with the Newton method. Based on this method we introduce trust-region methods which lead us to the explanation of the currently implemented Levenberg-Marquardt algorithm. Finally an approach to accelerate the implementation with the Broyden update is discussed.

3.1 Nonlinear Optimization

First we introduce the basics of nonlinear optimization. The general structure of an optimization algorithm can be seen in Algorithm 3.1.1. The steps of the algorithm are performed in a loop. Before each step we have to check if a certain *convergence criterion* is fulfilled. In every step a *search direction* and a *step size* is computed. These two values are needed for the iteration rule which is also applied in every step.

The search direction is the direction in which we go in the argument space to find the minimum. The length of one step gone in the search direction is called *stepsize*. The convergence criterion is usually built in one of the two ways: The first one is to choose a (small) value ϵ. If the difference of the result of iteration k and $k-1$ is smaller than ϵ: $|x^k - x^{k-1}| < \epsilon$ x^k is returned as the solution. The second way is to choose a certain number n of iterations. After performing these n iterations the result x^n is returned.

In the following sections, we describe existing optimization methods.

Algorithm 3.1.1: OPTIMIZATION$(f, startingvalue)$

Input: $f : \mathbb{R}^n \to \mathbb{R}, x^1$
Output: y such that $f(y)$ is minimum

$k \leftarrow 1$
while !(convergence criterion)
do $\begin{cases} \text{compute search direction } p^k \in \mathbb{R}^n \\ \text{compute step size } \alpha^k > 0, \text{ with } f(x^k + \alpha^k p^k) < f^k \\ x^{k+1} := x^k + \alpha^k p^k \\ k \leftarrow k + 1 \end{cases}$
return (x^k)

3.2 Steepest-Descent Method

One method for optimization is the steepest-descent method ([13], [3]). In this "simple" method the search direction is given by

$$p^k = -g^k$$

with

$$g^k = \nabla f(x) := \left(\frac{\partial f}{\partial x_1}(x), \frac{\partial f}{\partial x_2}(x), \ldots, \frac{\partial f}{\partial x_n}(x) \right)^T$$

denoting the gradient of $f(x)$. The steepest descent method has global convergence. The disadvantage in the steepest-descent method is the very slow convergence. (beg. In order to get one more correct decimal place, the number of iterations has to be increased by the factor 10.) The slow convergence is the main reason for not using the steepest-descent method in general.

3.3 The Newton method

The classical method for solving optimization problems is the *Newton method* ([3], [13], [12]). Here the search direction is:

$$p^k = -G^{k^{-1}} g^k$$

where g^k denotes the gradient of $f(x)$ as in Section 3.2 and

$$G^k = \nabla^2 f(x) := \left(\frac{\partial^2 f}{\partial x_i x_j}(x) \right)_{i,j=1,\ldots,n}$$

denotes the Hessian matrix. The Newton method converges quadratically, i.e. the number of correct decimal places doubles in every iteration step.

An improvement of the Newton method is the *Gauss-Newton method* [3] where the Hessian matrix is approximated by the Jacobian matrix (here denoted as J) which contains only the first derivatives.

$$G^k \approx J^{kT} J^k$$

This approximation is also used in the Levenberg-Marquardt algorithm explained in Section 3.5.

The problem of the Newton or Gauss-Newton method is the local convergence, which means that the algorithm returns a local minimum as global solution. In order to achieve global quadratic convergence, we have to modify the Newton method as described in the following section.

3.4 Trust-Region Methods

An approach to get global convergence can be found in so called trust-region methods ([13], [3]). The Newton method converges only locally because

$$(q^k(p) \not\approx f(x^k))$$

To gain global convergence we compute p^k by solving the minimization problem

$$\underset{p}{min}(q^k(p))$$

i.e. we have to find that p such that

$$q^k(p) = f^k(p) + g^{k^T} p + \frac{1}{2} p^T G^k p$$

is a minimum. The goal of trust-region methods is to construct a set Ω with $p \in \Omega$ and $q^k(p) \approx f(x^k + p)$. The trust-region can be seen as a sphere with a certain radius r adjusted during the optimization process. For the algorithm, depicted in Figure 3.4.1, we need a starting value X^1 and six values (describing the trust region) as input parameters. The value r^1 denotes the initial radius of the trust-region. r is changed during the optimization by multiplication with τ_1 for decreasing the size of the trust-region or with τ_2 for increasing its size. The other values are the following constant values:

- μ_0: the least demanded descent.

Algorithm 3.4.1: TRUSTREGION$(f, x^1, r^1, \mu_0, \mu 1, \mu_2, \tau_1, \tau_2)$

Input: $f \in C^2$
Output: y such that $f(y)$ is minimum

$k \leftarrow 1$
while !(convergence criterion)
do
compute search direction p^k: solve $q^k(p) \rightarrow min, \|p\| \leq r^k$
$\triangle q_k := q^k(p^k) - q^k(0) = q^k(p^k) - f^k$,
$\triangle f^k := f(x^k + p^k) - f^k$
$x^{k+1} := x^k + \alpha^k p^k$
if $\triangle f^k \geq \mu_0 \triangle q^k$
  then $\{x^{k+1} = x^k$
  else $\{x^{k+1} = x^k + p^k$
if $\triangle f^k \geq \mu_1 \triangle q^k$
  then $\{r^{k+1} = \tau_1 \|p^k\|$

  else
    if $\triangle f^k < \mu_2 \triangle q^k$ **and** $\|p^k\| = r^k$
      then $\{r^{k+1} = \tau_2 r^k$
      else $\{r^{k+1} = r^k$
$k \leftarrow k + 1$
return (x^k)

- μ_1: to measure if there is a descent or if is the descent to small. If there is no descent at all or the descent is to small in the current step, we have to decrease the trust-region's size in the following step.
- μ_2: to express if the descent is approximately $\triangle q^k$. In this case we have to increase the size of the trust-region in the next step.
- τ_1: a factor to make the trust-region smaller.
- τ_2: a factor to enlarge the trust region.

The conditions for the described values are as follows: $r^1 < 0$, $0 < \mu_0 < \mu_1 < \mu_2 < 1)$, $0 < \tau_1 < 1 < \tau_2$. In actual implementations the trust-region method typically switches to the Newton method near minimums.

Algorithm 3.5.1: TRUSTREGIONTESTLM$(x^k, x^t, f, \nu, \mu_0, \mu 1, \mu_2, \tau_1, \tau_2)$

Input: $x^k, x^t, f, \nu, \mu_0, \mu 1, \mu_2, \tau_1, \tau_2$
Output: x^s

$r \leftarrow x^k$
while $r = x^k$
do $\begin{cases} ared = f(x^k) - f(x^t) \\ s^t = x^t - x^k \\ pred = -\nabla f(x^k)^T s^t / 2 \\ \textbf{if } ared/pred < \mu_0 \\ \quad \textbf{then } \begin{cases} z = x^k \\ \nu = max(\tau_2 \nu, \nu^0) \\ \text{compute } x^t \text{ with the new value } \nu \end{cases} \\ \textbf{if } \mu_0 \leq ared/pred < \mu_1 \\ \quad \textbf{then } \begin{cases} z = x^t \\ \nu = max(\tau_2 \nu, \nu^0) \end{cases} \\ \textbf{if } \mu_1 \leq ared/pred \\ \quad \textbf{then } \{ z = x^t \\ \textbf{if } \mu_2 < ared/pred \\ \quad \textbf{then } \{ \nu = \tau_1 \nu \\ \textbf{if } \nu < \nu^0 \\ \quad \textbf{then } \{ \nu = 0 \end{cases}$
$x^s = z$
return (x^s)

3.5 Levenberg-Marquardt Method

The Levenberg-Marquardt algorithm ([3], [13]) is used for solving *nonlinear least squares problems*. These problems form a special class of optimization problems that minimize functions of namely those form:

$$f(x) = \frac{1}{2} \sum_{i=1}^{M} \|r_i(x)\|_2^2 = \frac{1}{2} R(x)^T R(x)$$

with the residual vector $R = (r_1, \cdots, r_M)$.

The pseudocode of the Levenberg-Marquardt algorithm is depicted in Algorithm 3.5.2 and 3.5.1. To perform an optimization with this algorithm we again need the trust region parameters $\mu_0, \mu 1, \mu_2, \tau_1, \tau_2$ with $\mu_0 \leq \mu 1 < \mu_2$

Algorithm 3.5.2: LEVENBERGMARQUARDT$(x^1, R, kmax, \mu_0, \mu 1, \mu_2, \tau_1, \tau_2, \nu^0)$

Input: $x^1, R, kmax$
Output: minimum $f(x^k)$

$\nu \leftarrow \nu^0$
for $k \leftarrow 1$ **to** $kmax$
do $\begin{cases} x^k = x^1 \\ \text{compute } R(x^k), f(x^k), R'(x^k) \text{ and } \nabla f(x^k) \\ \text{ with test for termination} \\ x^t = x^k - \left(R'(x^k)^T R(x^k) + \nu^k I\right)^{-1} R(x^k)^T R(x^k) \\ x^k = \text{TRUSTREGIONTESTLM}(x^k, x^t, f, \nu, \mu_0, \mu 1, \mu_2, \tau_1, \tau_2) \end{cases}$
return $(f(x^k))$

and $0 < \tau_1 < 1 < \tau_2$. Moreover an additional value, the so called Levenberg-Marquardt parameter ν, is needed. During the optimization certain values have to be computed: the trial point x^t, the actual reduction in f denoted by *ared* and the predicted reduction denoted by *pred*.

The algorithm is a combination of the Gauss-Newton and a certain trust-region method. With this combination we achieve the fast convergence (nearly quadratic) of the Newton method and global convergence as in the steepest descent method.

The Levenberg-Marquardt method is very robust and it converges nearly quadratic like the Gauss-Newton method. The Levenberg-Marquardt method is used in the SEE-GRID/SEE-KID software system to minimize the Torque function. The implementation of the Levenberg-Marquardt algorithm in SEE-GRID is based on a MatLab implementation of a software system called EyeLab [19]. The MatLab-code was transcripted into C++ as part of the SEE-KID project.

3.6 Timing the Existing Implementation

We benchmarked the implementation of the Levenberg-Marquardt algorithm that is currently used in the SEE-KID/SEE-GRID project.The benchmarks were performed on a PC with 1.4 GHz P4 processor and 512 MB of RAM and generated by using the existing testsoftware for the mathematic parts of the SEE-KID software.

In the benchmarks, we have computed

Test	Torque Evaluations	Computation Time
Test 1	8532	5.735s
Test 2	37390	24.484s
Test 3	55565	43.906s

Figure 3.1: Number of torque function evaluations

Operation	Computation Time
Matrix multiplication	0.014ms
Matrix inversion	0.015ms
Matrix-Vector Multiplication	0.006ms

Figure 3.2: Timings of the existing algorithm

- the number of evaluations of this function (Figure 3.1) and
- average computation time of basic matrix and vector operations used in every step of the Levenberg-Marquardt algorithm (Figure 3.2).

The computation time of whole optimization processes is depicted in Figure 3.3 and the number of evaluations of the torque function can be seen in Figure 3.1 for three different sets of input data:

1. healthy eye.
2. length of the rectus lateralis changed to 30.5 mm.
3. lengths of rectus lateralis, rectus medialis, rectus superior and rectus inferior changed to 30.5 mm, 32 mm, 32 mm and 29 mm.

3.7 Broyden Update

One possible approach improving the optimization algorithm is the so called Broyden update ([2], [3], [13], [12]). In general one has to compute the Jacobian or Hessian matrix in each step of the optimization process. This is one of the time consuming computations in our implementation of the Levenberg-Marquardt algorithm. The idea of the Broyden update is to use a Jacobian matrix computed in the step before and build an approximated new one by updating the previous one. The Broyden update starts with an initial Jacobian matrix (often the identity matrix is taken as initial matrix)

Algorithm 3.7.1: BroydenUpdate($f, startingvalue$)

Input: $f : \mathbb{R}^n \to \mathbb{R}, x^1$
Output: y such that $f(y)$ is minimum

$k \leftarrow 1$
while !(convergence criterion)

$$\textbf{do}\begin{cases} \vdots \\ \textbf{if } \text{Broyden update criterion holds} \\ \quad \textbf{then} \begin{cases} r^{k+1} = \tau_2 r^k \\ v^k = x^{k+1} + x^k \\ y^k = F(x^{k+1}) - F(x^k) \\ u^k = \frac{1}{{v^k}^T v^k}(y^k - J^k v^k) \\ J^{k+1} = J^k + u^k v^k \end{cases} \\ \quad \textbf{else} \begin{cases} \text{restart: normal computation of the} \\ \text{Jacobian matrix} \end{cases} \\ \vdots \\ k \leftarrow k+1 \end{cases}$$

return (x^k)

and updates this matrix a certain number of (usually three or four) times. The number of updates during two exact computations can either be defined statically as a constant number or dynamically by setting an error bound. The constant number depends on the problem and has to be determined by testing the optimizer. The exact computation of the Jacobian matrix after one or more updates is called "restart" and the optimizer runs one optimization step with this matrix. After this restart we can update the just now computed matrix again a certain number of times.

The basic structure of the Broyden update is

$$v^k = x^{k+1} + x^k$$

$$y^k = F(x^{k+1}) - F(x^k)$$

$$u^k = \frac{1}{v^{k^T} v^k}(y^k - J^k v^k)$$

$$J^{k+1} = J^k + u^k v^k$$

where $F(x^k)$ in our case denotes the Torque function evaluated in every step k in x^k and J^k is the Jacobian matrix in the k-th step.

3.8 Timing the New Implementation

We have developed an implementation of the Broyden update ([30]). In this implementation we use an update counter initially set to zero. If this counter is zero, the Jacobian matrix is computed exactly. For every number greater than zero the update of the Jacobian matrix is performed. The counter is increased by one at every step and if it reaches a certain number (the number of updates to be done), it gets zero again. The source code of this prototype implementation is depicted in Appendix A.

In our specific optimization of the torque function we get a problem with the Broyden update such that we update in our tests the Jacobian matrix once. The problem is that the process of optimization lasts even longer as without performing the update; in some cases the algorithm even does not terminate or the computation times grow explosively (see Figure 3.3). All in all the nearly quadratic convergence of the Levenberg Marquardt method cannot be reached.

Analyzing the problem, it turns out that its root cause is the the loss of positive definiteness of most of the updated Jacobian matrices (a matrix is positive definite if all of its eigenvalues are positive). As an example we picked one of the Jacobian matrices computed during the optimization process and computed the eigenvalues.

Test 1	Iterations	Time	Position of Minimum
LM	1427	5.735s	$\begin{pmatrix} -0.14097 \\ -0.262964 \\ 0.000077 \end{pmatrix}$
LM+B	1908	10.093s	$\begin{pmatrix} 0.002623 \\ -0.26737 \\ -0.005205 \end{pmatrix}$

Test 2	Iterations	Time	Position of Minimum
LM	6202	25.484s	$\begin{pmatrix} -0.133243 \\ -0.309130 \\ -0.002774 \end{pmatrix}$
LM+B	14847	74.11s	$\begin{pmatrix} 0.170384 \\ 0.242074 \\ -0.000019 \end{pmatrix}$

Test 3	Iterations	Time	Position of Minimum
LM	9003	43.906s	$\begin{pmatrix} 0.154371 \\ -0.198040 \\ -0.039909 \end{pmatrix}$
LM+B	49365	243.407s	$\begin{pmatrix} -0.080549 \\ -0.153042 \\ -0.019640 \end{pmatrix}$

Figure 3.3: Benchmarks (Levenberg Marquardt, Levenberg Marquardt with with Broyden update)

The eigenvalues of

$$\begin{pmatrix} 1.1026 & -0.145282 & -0.027819 \\ 13.4197 & -17.9943 & -3.63714 \\ 1.91109 & -2.70498 & 0.482035 \end{pmatrix}$$

are given by the vector

$$(-18.4096, 1, 1)$$

One of the elements of this vector is negative, so this matrix is not positive definite. This causes a very unstable behavior of the optimization.

For benchmarking the Broyden implementation we adapted the existing testsoftware.

Figure 3.3 compares the timings and computed minimums both of the exact Levenberg Marquardt method and the Levenberg Marquardt method combined with a Broyden update of the Jacobian matrix and shows the number of iterations performed by the optimizer. It shows three tests: The first test is done with a healthy eye, in the second test the length of the rectus lateralis was changed to 30.5 mm and in the third the lengths of rectus lateralis, rectus medialis, rectus superior and rectus inferior were changed to 30.5, 32, 32 and 29.

In addition to the benchmarks, graphics were generated with the same parameters. These are depicted in Figure 3.4, 3.5, 3.6. Each of the figures consists of three graphics. The first (top) shows the function values in every optimization step with Levenberg Marquardt only (LM), the second picture in the middle shows the function values from the optimization with Levenberg Marquardt combined with the Broyden update (LM+B) and the third picture shows the torque function plot over $-30 \leq x, y \leq 30$. It can be seen that the optimization with the Broyden update performs more steps in a wider range inside the domain.

3.9 Conclusions

In this chapter we first introduced the basics of optimization and showed several algorithms which led us to the currently implemented Levenberg Marquardt algorithm. The ability of acceleration of this sequential optimization was discussed and a certain method, the Broyden update, was described and implemented. During testing it turned out that for our special problem the Broyden method does not work properly – it produces wrong results and needs more computation time.

Testing the existing implementation we found out that the evaluations of the torque function are the most time consuming part. This discovery leads us to the next chapters which deal with strategies to interpolate the torque function both sequential and parallel.

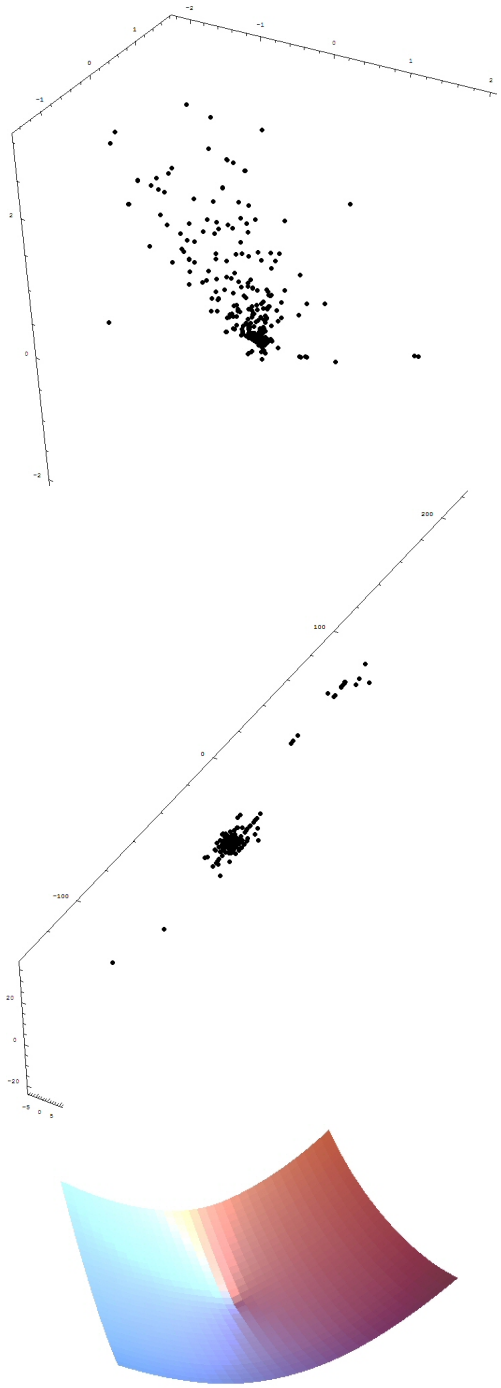

Figure 3.4: Test 1, LM, LM+B, function

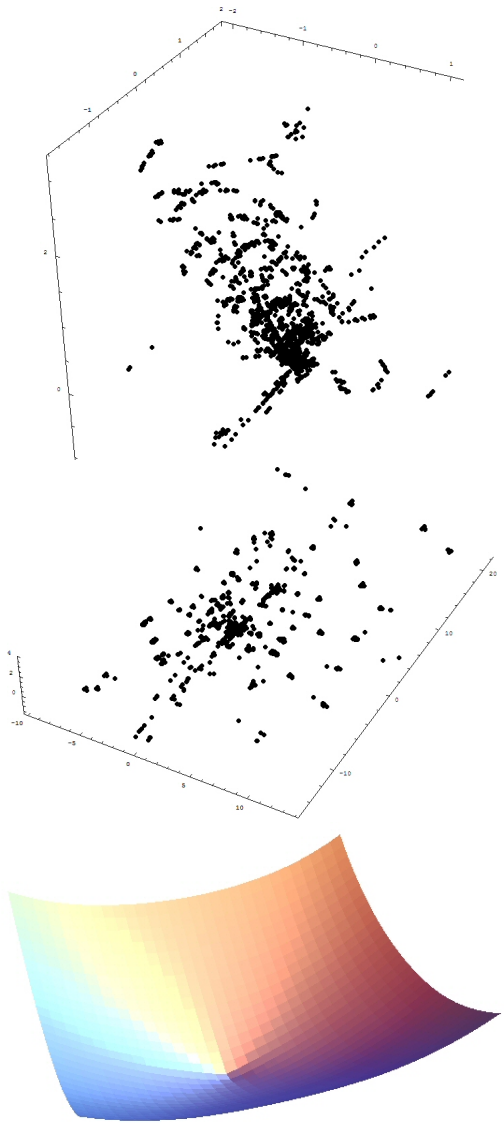

Figure 3.5: Test 2, LM, LM+B, function

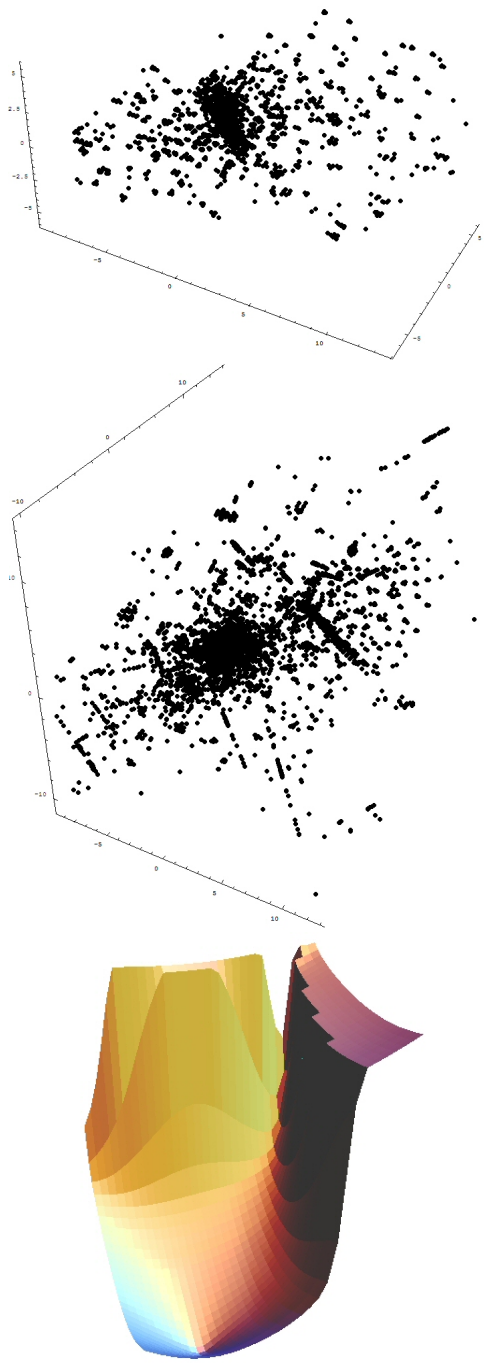

Figure 3.6: Test 3, LM, LM+B, function

Chapter 4

Triangulation and Interpolation

During the analysis of the existing algorithm it was detected, that the evaluations of the Torque function (the function to be minimized) needs more than half of of the whole computation time. In our optimization algorithm, we need thousands of evaluations in total. Thus, one idea to improve the efficiency of the algorithm is to reduce this large number of function evaluations by computing function values of certain points in advance and interpolate the points in between.

One way to manage a certain set of grid points and use these points for interpolations is the Delaunay triangulation.

4.1 Delaunay Triangulation

The interpolation we use in our project works by the triangulation of the function in a certain surface.

When triangulating the function, we decompose the domain into a set of triangles with our input points as vertices. Every time a function value is requested, the algorithm has to interpolate this value from the tree vertex points of the surrounding triangle which should be faster than the exact computation of the function value.

To implement this idea, we first have to triangulate the function with the Delaunay algorithm [17]. The Delaunay algorithm is used to describe 3-dimensional surfaces as sets of triangles. This algorithm takes as input a set of points we get from evaluating the Torque function. It then computes the triangulation. Afterwards, every function value we have not computed exactly before can be interpolated. A triangulation is basically computed by creating edges between the given points. This builds up a mesh of triangles.

For the visualizations of the torque function, the grid points, the triangle

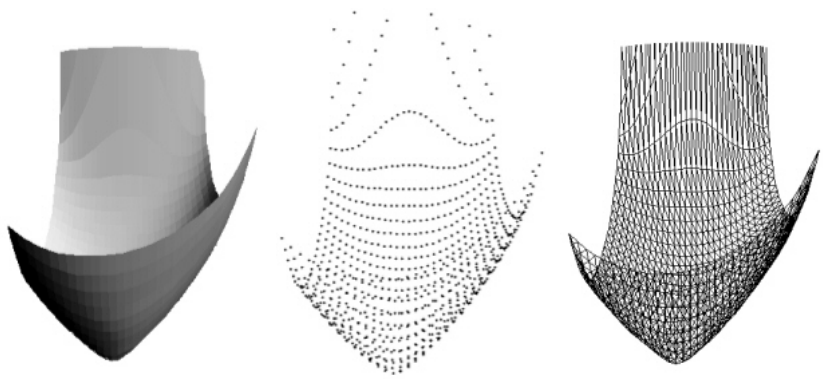

Figure 4.1: The Torque Function (original - set of points - triangulated)

mesh and the timing we used the so called *Listing's Law* ([7]) which let us compute the z parameter out of x and y. The z axis shows the Euclidian norm of the three dimensional torque function result vector.

As an example, Figure 4.1 represents the Torque function for a certain pathology. We changed some data concerning muscle force and length. The first graphic shows the original Torque function. In the second one, one can see a set of points (in this case a homogeneous mesh) created by evaluating the torque function in all of the points chosen before. The last picture represents a triangulation of this set of points.

4.2 The Delaunay Algorithm

The input of the Delaunay triangulation consists of a set of points $P = \{p_1, p_2, \ldots, p_n\}$ (later representing the vertices of our triangles). The output of the algorithm delivers us a triangulation T. The pseudo-code of the algorithm is depicted as Algorithm 4.2.1. The "overall" algorithm containing the generation of P and the following Delaunay triangulation is depicted in algorithm 4.2.2 with the function f and the range of the mesh (l_x, l_y, u_x, u_y) as input.

The first step in the Delaunay triangulation is to create a triangle containing all points of the set P. The triangulations are performed in a loop iterating over every point p_r of P. First p_r has to be inserted into the triangulation and then a triangle containing p_r has to be found. After that,

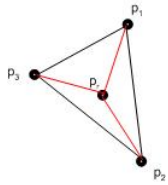

Figure 4.2: Inserting a point located in a triangle

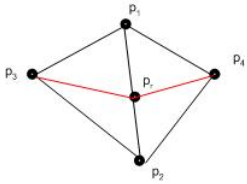

Figure 4.3: Inserting a point located on an edge of a triangle

we check if p_r lies in a triangle of the triangulation. If this is the case, the algorithm splits the triangle into three parts by inserting three new edges from every vertex of the triangle to p_r located in the triangle (see Figure 4.2). We have to check now if the circumcircle of the new triangles does not contain any other points of P (circumcircle condition). In the case of a point lying inside the circumcircle of a triangle we have to do an "edge flip" (see Figure 4.4). This is done by deleting the inner edge and inserting a new one to create two triangles again. Then the circumcircle condition will hold. Next, we have to check if p_r lies on an edge of a triangle. If this check is true we have to split the two triangles having this edge in common into four (see Figure 4.3). After inserting the new edges to split the triangles we have to validate the circumcircle condition and do edge flipping as described before.

As the last step in the algorithm the initial triangle has to be removed and the finished triangulation T can be returned.

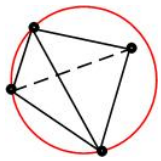

Figure 4.4: Edge flip

Algorithm 4.2.1: DELAUNAY(P)

Input: Set of points $P = \{p_1, p_2, \ldots, p_n\}$
Output: Triangulation T

```
generate a triangle surrounding the whole set P
generate a random permutation of P
for r ← 1 to n
    do { insert p_r into the triangulation
         find a triangle in the triangulation containing p_r
         if p_r lies in a triangle of the triangulation
           then { split the triangle into thee triangles by creating
                  edges from p_r to every vertex
                  check edges if the circumcircle condition holds
                  otherwise do an edge flip
           else if (p_r lies on an edge of a triangle)
           then { split the two triangles
                  which have that edge in common into four
                  check edges if the circumcircle condition holds
                  otherwise do an edge flip
remove the initial triangle from the triangulation
return (T)
```

Algorithm 4.2.2: TRIANGULATIONWITHDELAUNAY(l_x, l_y, u_x, u_y, f)

Input: $l_x \leq 0, l_y \leq 0, u_x \geq 0, u_y \geq 0, f$
Output: Triangulation T

```
P :=
for i ← l_x to u_x
  do { for j ← l_y to u_y
       do {P := P ∪ f(i, j)
T =DELAUNAY(P)

return (T)
```

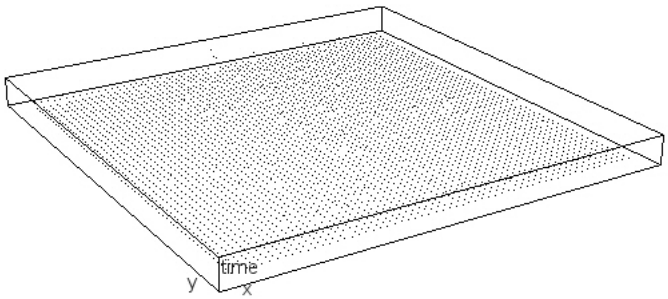

Figure 4.5: Timings of the Delaunay interpolation

4.3 Implementation and Benchmarks

The SEE-KID/SEE-GRID software actually uses a GPL implementation of the Delaunay triangulation [25] for dealing with surfaces in another part of the program. This implementation was used for our benchmarks together with the existing test software ([30]).

For performing the benchmarks we had to change the test software. The source code snippet is depicted in Appendix B. The range of the mesh in x and y goes from -30 to 30. For generating the mesh we used two loops with a stepping of 2. In every step we computed the torque function value in the current x and y. This produced a set of points which is needed for the Delaunay algorithm.

After the triangulation of our mesh, we tested and measured the interpolation procedure. For this we used again two loops running over the whole mesh (this time with a stepping of 1) and computed 3600 interpolated values.

The average computation time for one interpolation is 0.33 ms. This value was computed as the mean value of the timings of these 3600 interpolations. The measured computation times lie between $0,0023883026$ ms and $3,8312207175$ ms. A visualization of the timings can be found in the Figure 4.5. In this figure the timings (in ms) for the interpolations can be seen for every gridpoint.

4.4 Application of Parallelism

We have the following possibilities for the parallelization of the algorithm:

1. *Before the triangulation:* We can compute the function values in par-

Algorithm 4.4.1: ParallelDelaunay(P)

Input: Set of points $P = \{p_1, p_2, \ldots, p_n\}$
Output: Triangulation T

divide P into disjoint subsets $P_1, P_2, \ldots, P_n$
such that $P_1 \cup P_2 \cup \cdots \cup P_n = P$
for each P_i
do in parallel $\{$Delaunay(P_i)
combine the $T_1, T_2, \ldots, T_n$ to the triangulation T of P
return (T)

allel: We have to select a certain number of (x, y) sample points and determine the corresponding z value by evaluating the Torque function in these points in parallel.

2. *Parallel Delaunay:* The triangulation of the set of points can be done in parallel. This means a *Divide and Conquer* strategy is possible. This strategy is sketched in Algorithm 4.4.1.

3. *Concurrent computation of Delaunay triangulations of different sets of points:* During the optimization we can compute the triangulation of different subsets of the domain of the Torque function in parallel.

4. *Performing triangulation and optimization in parallel:* The triangulation is needed for the function evaluations in the optimization algorithm. By considering triangulation not in isolation but in combination with optimization, the efficiency may be improved and the potential for parallelization may be increased. This way of parallelization is described in Chapter 6.

All in all, the Delaunay triangulation is an overkill for our problem because it is designed for general meshes. That's why an implementation of the parallel Delaunay algorithm was not carried out. In our specific problem we have a regular mesh (generated before). We can save computation time by looking for a strategy making use of our "special" mesh structure. A possible way to deal with our regular mesh structure is given in the next chapter.

Chapter 5

Interpolation using the Regular Mesh Structure

5.1 Introduction

The Delaunay algorithm is is necessary for general meshes but not for regular ones. In a regular mesh, the triangulation is given by the mesh structure and moreover the triangle search can be replaced by a simple lookup. This chapter describes a possible way to deal with the regular mesh structure in our specific problem.

First we give a description of the mesh itself and the ideas of making use of its structure for interpolation.

To introduce the idea we need to look at the input and output parameters of the torque function. In Section 2.2.3 the torque function was given by

$$L_T(\overrightarrow{I_v}, \overrightarrow{E_p}, e_{si}, c_{si}, t_{si}) = \overrightarrow{P} + \sum_{i=1}^{6} \overrightarrow{F_{T_i}}(dl_i, iv_i, e_{si}, c_{si}, t_{si}) \cdot G_t(\overrightarrow{Trans}, \overrightarrow{ra_i})$$

To decrease the six parameters of $\overrightarrow{I_v}$ to three we make use of Sherrington's Law to compute the three elements with even indices $\overrightarrow{I_{v_o}}$ out of the other elements with odd indices. A detailled description of this law can be found in Section 2.2.4. Basically, the torque function now can be seen as

$$f_{torque} : \mathbb{R}^3 \rightarrow \mathbb{R}^3$$

The domain of input parameters of the torque function can be interpreted as a cube. For our model we need to create a three dimensional grid in this cube. In every node point of this grid we have to store a vector of three numbers representing the three dimensional output vector of the torque function in

this point. The point $(0, 0, 0)$ is the center of this cube. For generating the grid we need need a step size which is the space between the node points. In each of the node points the return vector of the torque function is calculated and stored in a certain data structure.

Interpolating the function value needs three points surrounding the the requested point. Basically the interpolation is done in a linear way.

In this chapter we describe the implementation of a standalone version of the interpolation which was used for testing the data structures and the interpolation method and an object oriented implementation which was actually integrated in the SEE-KID/SEE-GRID software.

Performance tests and benchmarking were done for the object oriented implementation.

5.2 Description of the Basic Solution

First, we introduce some terms used to describe our model ([31]). The Torque function gets three double values as input such that the domain of this function can be interpreted as a *cube*. This cube is (virtually) divided into *subcubes* of which only some are actually represented at any time in memory during the execution of the algorithm (Figure 5.1). There is no data structure for the cube itself because the values are stored in the subcubes which are organized as a *queue*. To save memory, we use a *queue of subcubes*, where we store a certain number of subcubes. For the interpolation, we need information about the position of each subcube in the domain (the cube) residing in the subcube data structure as a vector of three elements. One more thing we need to know is the position of the subcube in the queue. This is done by a *cube of integers* (Figure 5.2). In this cube of integers there exists a value for every virtual or existing subcube. If the subcube has not been calculated before or is out of the cube, the dedicated value in the cube of integers is -1. If the cube exists, we store the subcube's queue position as integer value in the cube of integers $(0 \ldots n)$. This cube of integers has to be updated at any change of the queue.

If the optimization procedure requires a certain point, we check its position and return the function values interpolated from the function values of the corners of a pre-computed triangle surrounding the point. Our subcube is virtually divided into planes, so we first have to check the plane of the requested point and then do the interpolation with three surrounding points lying in the same plane. Three interpolations have to be performed because the Torque function returns a vector with three elements.

In the beginning our domain has to be virtually divided into subcubes.

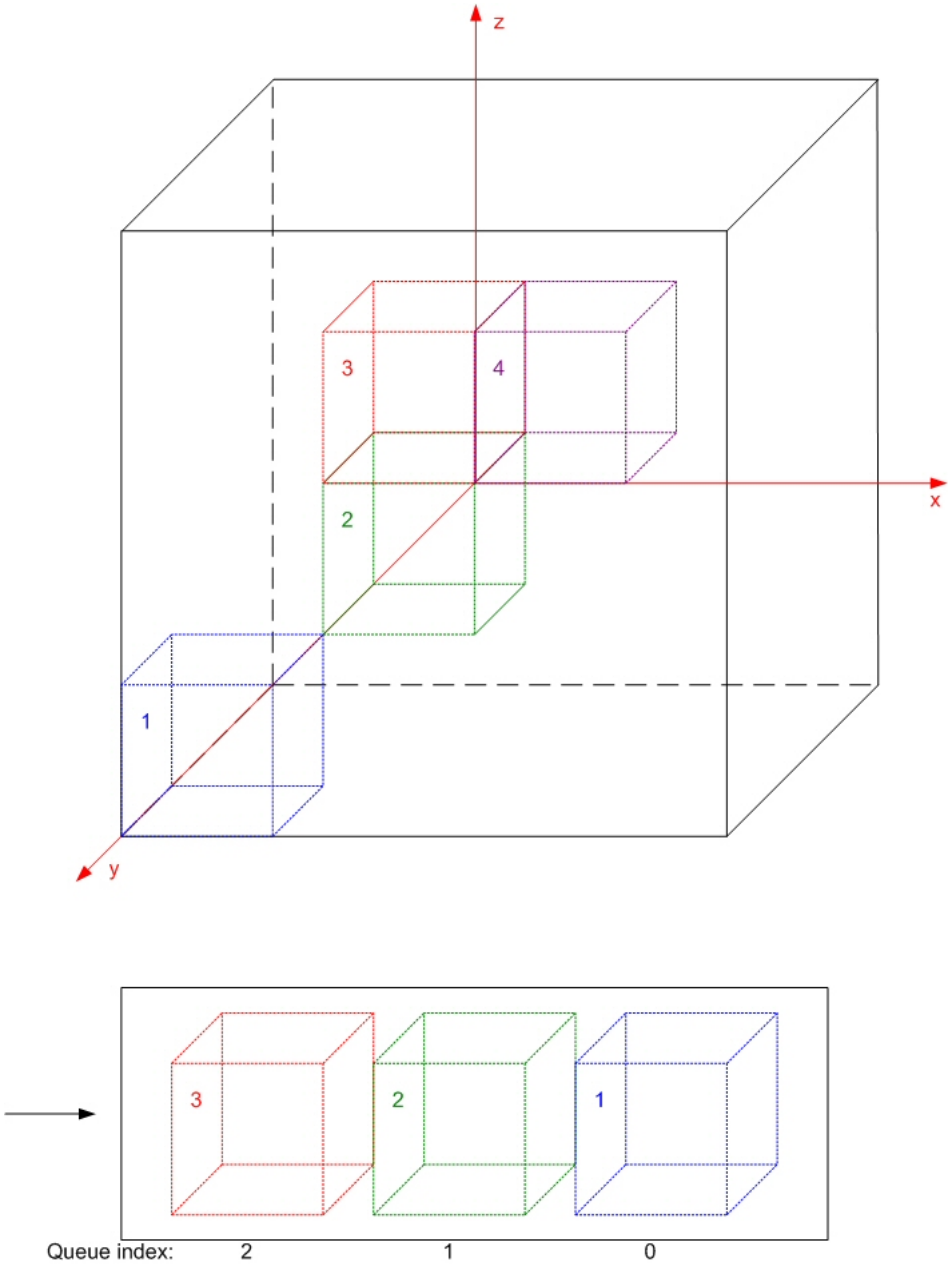

Figure 5.1: The domain of the Torque function divided into subcubes added to a queue

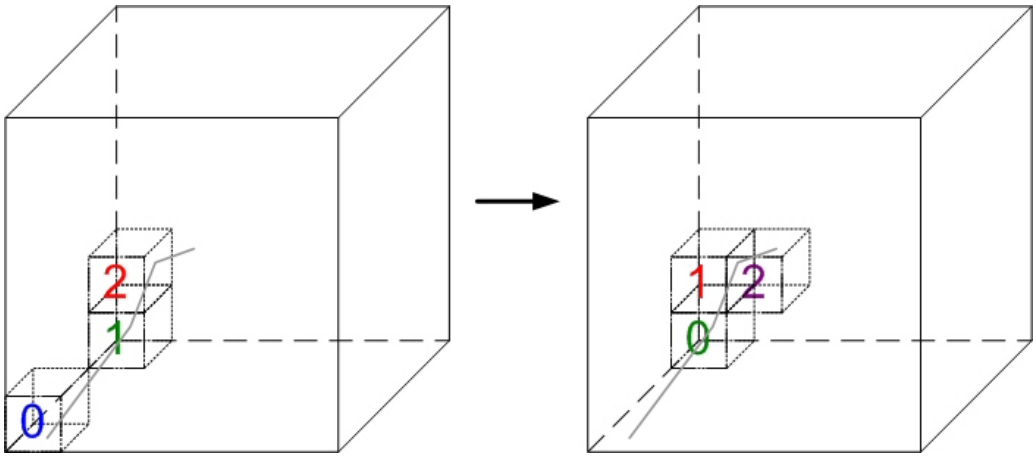

Figure 5.2: The cube with the positions of the subcubes in the queue when adding a new subcube

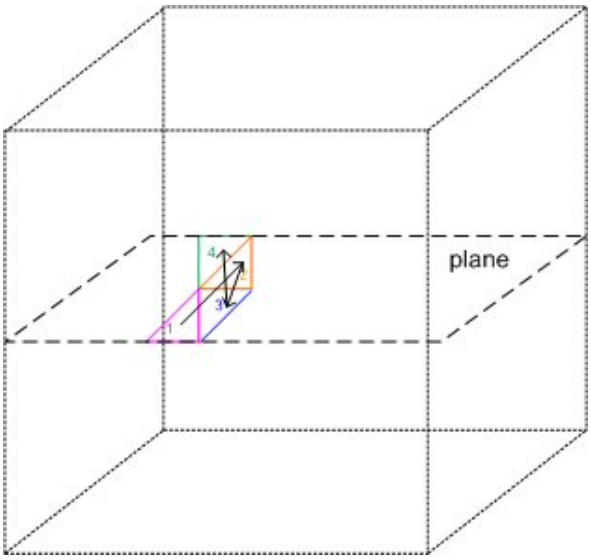

Figure 5.3: Subcube with some points already computed

Algorithm 5.3.1: INTERPOLATE(x, y, z)

1. Set the dimensions of the *cube of integers* and of the *subcubes* and the *stepsize*
2. Create the *cube of integers* and the *queue*
3. $\tilde{f}$:= CHECKCUBE(*cube of integers, x, y, z, stepsize*)

return $\tilde{f}$

After that we check the position of the point to be interpolated. In the very beginning, we have no subcube calculated before and so we have to create the first subcube and put it into the queue. In the following steps we check if the subcube containing the point to be interpolated exists and return the subcube calculated some time before. If the new point is outside the subcubes already created, we have to create a new one and also put it into the queue.

For the interpolation of the first point in a subcube, we need to evaluate the Torque function three times to get a triangle around the point (Figure 5.3). After computing the interpolated function value, a vector containing the three interpolated Torque function values is returned to the optimizer.

If a point resides in an already existing triangle (the best case), no function evaluation has to be done. In other cases (the point is "near" an already existing triangle), we have to compute one or two new function values to build a new triangle with the point inside. In Figure 5.3 one can see that for the first point we have to compute three function evaluations, for the second one two. For the third one, we do not have to compute any points, and for the fourth point we need one function evaluation.

5.3 Pseudocode

In this section we describe the process of calculating an interpolated point of the Torque function (see Algorithm 5.3.1). The optimizer calls the interpolation function instead of the exact Torque function. As global input we get a vector of three elements from the optimizer. The domain dimensions, the sizes of the cube and the subcubes together with the stepsize have to be set before doing any interpolation.

General Structure of the Algorithm

Algorithm 5.3.2: CHECKCUBE(*cube of integers, x ,y, z, stepsize*)

1. Determine the position P of the *subcube*
containing the requested point in the domain
2. Check in the *cube of integers* if subcube P is computed

if *subcube is computed*

then $\begin{cases} \text{Get the queue position } i \text{ of the subcube} \\ \tilde{f} := \text{RETURNVALUE}(queue[i],\ x,\ y,\ z,\ stepsize) \end{cases}$

else $\begin{cases} \text{Create the subcube} \\ \text{add the subcube to the queue} \\ \text{update the } (\textit{cube of integers}) \\ \tilde{f} := \text{RETURNVALUE}(subcube,\ x,\ y,\ z,\ stepsize) \end{cases}$

return $\tilde{f}$

- INTERPOLATE (Algorithm 5.3.1) In this function first the datastructures are created if not existing. Then the method CHECKCUBE is called.

- CHECKCUBE (Algorithm 5.3.2) To interpolate a point, we have to determine its surrounding subcube. For this we first need the position in the *cube of integers* to check if the subcube is already computed (the value has to be $1 \ldots n$, depending on the size of the queue). In the case where the subcube is already computed, we call the method RETURNVALUE (Algorithm 5.3.3) described later in this section. If the subcube is not computed (represented by the value -1 in the *cube of integers*), we have to create a subcube, add this new cube to the queue, perform an update to the *cube of integers* and call the RETURNVALUE method with the new subcube.

- RETURNVALUE (Algorithm 5.3.3) The real work, the interpolation of the Torque function values, has to be done by the method RETURNVALUE. The input for this method is a subcube, the coordinates of the requested point and the stepsize. First we have to determine the plane index of the point within a set of virtual planes into which we have divided our subcube. As the next step, we have to calculate the three triangle vertices on the plane. A 3×3 matrix has to be filled with the coordinates of the vertex points in the first two rows and 1 in the third row (the structure of this matrix can be seen in the pseudocode). For further calculation we need the inverse of this matrix. The calculation

Algorithm 5.3.3: RETURNVALUE(*subcube, x, y, z, stepsize*)

1. Check the z position in the subcube to get the plane index i using the stepsize
2. Compute the coordinates of the three triangle vertex points surrounding the point in the plane i using the stepsize

$$P_1 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}, P_2 = \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}, P_3 = \begin{pmatrix} x_3 \\ y_3 \end{pmatrix}$$

3. Build a matrix M and fill in the coordinates of the triangle vertices

$$M = \begin{pmatrix} x_1 & x_2 & x_3 \\ y_1 & y_2 & y_3 \\ 1 & 1 & 1 \end{pmatrix}$$

4. Compute the inverse of this matrix
5. Compute the scalarproduct r of M^{-1} and the point coordinate vector with $z = 1$

$$r = M^{-1} \times \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

6. Check for each triangle point $P_{i,1 \leqslant i \leqslant 3}$, if the *computed flag* is false

if *computed flag* is false

then $\begin{cases} \text{evaluate the Torque function in this point} \\ \text{save the vector } \begin{pmatrix} f_1 \\ f_2 \\ f_3 \end{pmatrix} \text{ in the subcube} \\ \text{set the } \textit{computed flag} \text{ of } P_i \text{ to true} \end{cases}$

7. Interpolate the three values:

$\tilde{f}_1 = r_1 * f_1(P_1) + r_2 * f_1(P_2) + r_3 * f_1(P_3)$
$\tilde{f}_2 = r_1 * f_2(P_1) + r_2 * f_2(P_2) + r_3 * f_2(P_3)$
$\tilde{f}_3 = r_1 * f_3(P_1) + r_2 * f_3(P_2) + r_3 * f_3(P_3)$
return $\tilde{f}$

of the inverse matrix is done by dividing the transposed cofactor matrix by the determinate which is computed by Sarrus' scheme. Next, we need a vector called r which is computed as the inverse matrix times the point's coordinate vector with the third coordinate $z = 1$.

In the next steps we check for each triangle point if the function values have been computed already. For this, we have set a flag (the "computed flag") in our data structure. If the point has been calculated before, we are lucky. Otherwise, we have to perform an evaluation of the Torque function to get our three function values in this triangle point. After the evaluation, the output vector of the Torque function is saved in our data structure and the computed flag of this point is set to true. If we have our complete set of function values, we can start to interpolate. We use a linear interpolation as one can see in our pseudocode. Finally, the interpolated values are returned to the optimizer as a three dimensional vector.

5.4 Prototype Implementation

For this rapid prototyping implementation we used the programming language C. Especially for the data structures we used the C construct of *structs* for the subcubes, the grid points in the subcubes, the queue elements and the integer cube.

As input parameter we get a three dimensional vector from the optimizer representing the input values of the Torque function. Parameters like cube and subcube dimensions together with the queue size are set in the interpolation routines. These parameters do not effect the optimizer and are not changed from the optimizer.

Our algorithm returns a three dimensional vector. This vector contains the interpolated function values of the Torque function.

5.4.1 Sourcecode

In our sourcecode we use methods for creating the cube, the subcubes and the queue (Appendix C lists the sourcecode of a version with a simple function for testing). The cube and the subcube are realized as pointers to a matrix of pointers (eg. **** int) and the queue as pointer to a scalar value.

For every structure used, there exists a method to create this structure and fill it with a set of initial values (**createSubcube, createSubcubeQueue, createCube**). For adding subcubes to the queue we use the method

addQueue. In the **addQueue** method each subcube is put into a datastructure together with its global position in the domain represented by (x, y, z). The method **checkCube** gives us the position of the subcube in the queue or creates a new subcube. Afterwards it calls the **returnValue** method either with the new subcube or a certain subcube in the queue. The **func** method is a dummy function used for testing.

The main part of the computation is done by the **returnValue** method. Here the algorithm determines the position of the point in the subcube and in the plane in the subcube and checks if the surrounding triangle is complete. If there are points of the triangle missing, the triangle is completed by the method by performing Torque function evaluations. With the three surrounding points we can compute the three interpolation values by basically solving a linear equation system in three dimensions. In the case of an error during the calculation, an exception is thrown and the exact function values are computed and returned because otherwise the whole optimization would crash.

Finally, (if there was no error) this method returns a vector consisting of the interpolated values.

This implementation was tested as standalone program with a dummy function to check the functionality of the data structures and the interpolation method. In the next section the description of the object oriented implementation integrated in the existing software is given.

5.5 Object Oriented Implementation

To integrate the idea of mesh interpolation into the existing SEE-KID/SEE-GRID software we had to develop an object oriented version of the program. The source code can be seen in Appendix D.

During the design and implementation several improvements were done such as the cube of integers is not needed any more. There is a difference in the creation of the subcubes in this implementation. In the standalone version a subcube is created empty and the function values in the nodepoints only are computed if they are necessary for an interpolation. In this version the subcube is filled with the torque function values in every nodepoint at creation time.

The basic structure of the implementation consists of three classes:

- Interpol
- Subcube

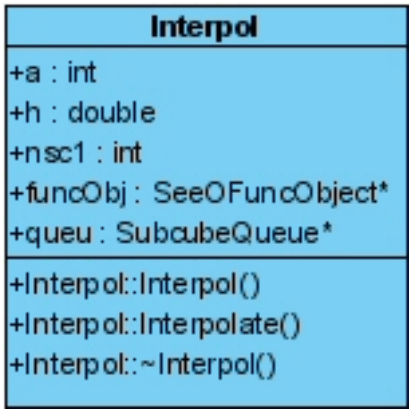

Figure 5.4: UML diagram of the Interpolation class

- SubcubeQueue

5.5.1 Interpol

The Interpol class is the main class of the project. In the SEE-KID/SEE-GRID software one needs to create an instance of this class to get interpolated function values. The class is depicted in Figure 5.4 as an UML diagram.

The class contains the following variables (public):

- a (integer): This variable represents half of the domain size in one dimension. (e.g. If the domain is the cube from $(-30, -30, -30)$ to $(30, 30, 30)$, a is 30).
- h (double): h measures the stepsize between the grid points.
- nsc1 (integer): nsc1 represents the number of subcubes in one dimension of our domain. (e.g. If a is 30 and nsc1 is 30, the dimension of the subcube would be 2.)
- funcObj (SeeOFuncObject*): This is a pointer to the function object for computing the torque function values. This variable is initially set from the optimizer with the constructor of Interpol.
- queue (SubcubeQueue*): The queue is used to store the subcubes.

The constructor of the class is used to set the class variables and create a queue filled with initial empty subcubes.

Algorithm 5.5.1: INTERPOL$((x, y, z))$

1. compute the left-down-front corner of the subcube
if subcube is in queue
  then {get the subcube from the queue
  else {create new subcube
3., 4. 5. and 7. as in Algorithm 5.3.3
return $\tilde{f}$

The main method *Interpolate* of the project resides in the class Interpol. This function gets a three dimensional vector as input and returns a three dimensional vector. The pseudo code of this function is depicted as Algorithm 5.5.1. First we need to determine the left-down-front corner of the subcube. If this subcube was computed before and is in the queue, it is used to fetch the three triangle points. If the subcube is not in the queue we have to create a new one with the corner values computed before. The next step is to retrieve the triangle points surrounding the requested function value, perform a linear interpolation and return the interpolated function vector.

If there is any error in *Interpolate* like the requested point is out of the borders of our domain of the matrix inversion leads to an error, the function value of the torque function is computed and returned.

5.5.2 Subcube

Basically the Subcube class handles the creation of subcubes (filling the subcubes with torque function values). Moreover the class contains the methods to get the three triangle points for the interpolation. The UML diagram can be found in Figure 5.5.

The following variables are used in the Subcube class:

- public
  - ax (integer): The value of the left-down-front corner on the x-axis.
  - ay (integer): The value of the left-down-front corner on the y-axis.
  - az (integer): The value of the left-down-front corner on the z-axis.
  - dim (integer): The dimension of the subcube.
  - h (double): The stepsize.

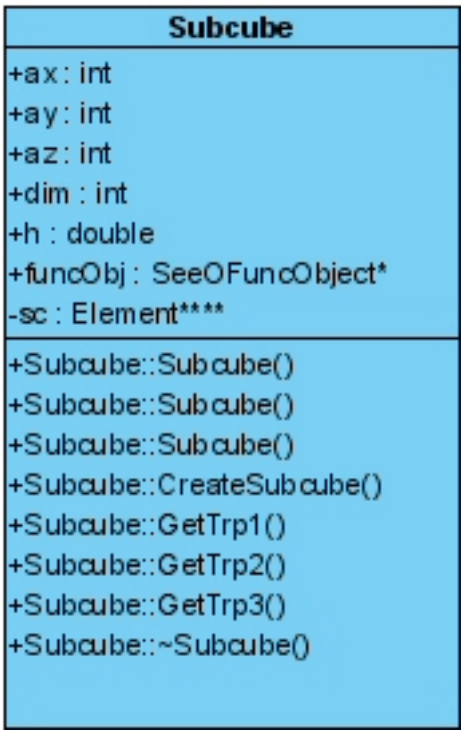

Figure 5.5: UML diagram of the Subcube class

  - funcobj (SeeOFuncObject*): The pointer to the function object performing the torque evaluations.

- private
  - Element (structure containing three double values): This user defined variable is used in every grid point of the subcube to store the torque function vector.
  - sc (Element ****) This pointer represents the subcube itself.

For the creation of a Subcube object there are three different constructors in the class.

- default constructor: for creating an Subcube object without setting any variables.
- constructor with a Subcube pointer as input: This constructor creates a new Subcube object out of the pointer to a given one.
- constructor with the whole set of class variables as input parameters: In this constructor all of the class variables are set to the input parameter values.

The filling of a subcube with function values is done by calling the *CreateSubcube* method. In this method the necessary memory for the subcube data structure is reserved and filled with the torque function vectors.

For determining the three triangle points there are the methods *GetTrp1*, *GetTrp2* and *GetTrp3*. These methods get the input vector of the torque function as input to compute the positions of the triangle points. Each of these functions returns a vector containing seven values: the first three elements are the torque function values in the trianglepoint, the second three values are the global (not the subcube dimension) coordinates of the triangle point and the last value is a flag which is 0 if there was any error in the computation and one if the point is valid.

5.5.3 SubcubeQueue

The class SubcubeQueue represents a queue for storing computed subcubes. The *First in First out* principle is used because in general the optimizer works in one direction.

The SubcubeQueue has two variables (public):

- numOfsc (int): This variable resents the number of subcubes in the queue.

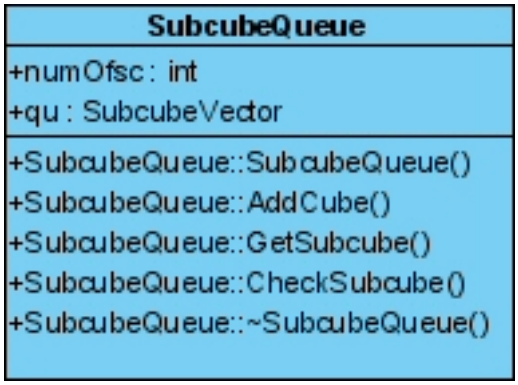

Figure 5.6: UML diagram of the SubcubeQueue class

- qu (vector of subcubes): qu is the variable to store the subcubes. It uses the standard C++ vector library to store Subcube objects.

The constructor to create a SubcubeQueue object needs the queuesize and an object of the Subcube class as input parameters. In the constructor the queuesize is set and the vector is filled with the Subcube object.

There are three public methods for using the queue: The first one is the *AddSubcube* method. This method gets a pointer to a certain subcube as input and adds the according subcube to the queue. In this method the oldest subcube in the queue is deleted.

The second method is the *GetSubcube* method. This method returns the subcube with the left-down-front corner (x, y, z) which is given as input parameter. The method works correctly only in combination with the *Check-Subcube* method which returns true if a requested subcube with (x, y, z) as left-down-front corner is stored in the queue and false if not.

5.6 Testing and Benchmarks

The tests were done with different sets of input parameters both of the interpolation class and the torque function. The input data of the torque function was set to the parameters used in Chapter 3.7:

1. Model 1: healthy eye.
2. Model 2: length of the rectus lateralis changed to 30.5 mm.
3. Model 3: lengths of rectus lateralis, rectus medialis, rectus superior and rectus inferior changed to 30.5 mm, 32 mm, 32 mm and 29 mm.

The tests were performed with the following three sets of input data of the interpolation class:

1. Parameter 1:
   (a) Size of the domain in one dimension: 30
   (b) Stepsize: 2
   (c) Number of subcubes: 10
   (d) Queuesize: 10
2. Parameter 2:
   (a) Size of the domain in one dimension: 40
   (b) Stepsize: 5
   (c) Number of subcubes: 4
   (d) Queuesize: 10
3. Parameter 3:
   (a) Size of the domain in one dimension: 10
   (b) Stepsize: 0.25
   (c) Number of subcubes: 10
   (d) Queuesize: 10

The computation time for a single interpolation is approximately 0.0297ms. A torque function needs $\approx$ 0.67ms.

The computation time for one subcube can be described by the following formula:

$$time_{subcube} = n_{gridpoints} * time_{torque} + overhead$$

In this formula $n_{gridpoints}$ is the number of grid points in the subcube and $time_{torque}$ gives the time of one evaluation of the torque function. The overhead contains the computation time for creating the data structure and grows with the number of elements in the subcube because there has to be done more memory allocation.

The timings of the whole optimization processes can be seen in Figure 5.7. Comparing the number of iterations and the timings of the original computation to the interpolated version a growth both of the computation time and the number of iterations can be seen. The reason of the higher number of iterations can be found in the influenced optimization process

whereas the biggest amount of the higher computation time is not consumed by the optimization process but by the computation of the subcubes.

In Figure 5.8 a comparison of the exact result function vector with the interpolated result function vector can be found.

The tests show that the values computed by using the interpolation of the torque function are the same (very close to) as the values computed with the exact torque function evaluations in the first two tests whereas in the third test the result with interpolating the function values differs from the exact result. The measured computation times are very high because the creation (filling with torque function vectors) of the subcubes is very time consuming. Moreover the convergence of the optimization algorithm is influenced by the approximated values which can be seen in *Test 3*. Comparing *Set 2* of *Test 2* and *Test 3* one can see, that in the second test the computation time is higher for less iterations than in the third test where $\approx$ 150s of computation time is saved because there are done more interpolations in the same subcube.

To measure the quality of the result we calculated the Euclidean norm of the result vector (error function) as depicted in Figure 5.9. In this table it can be seen clearly that in the first two models the results are of the same quality. The the computation with the third model gives incorrect results. Looking at the second parameter set in *Model 3* we get $\|f_{torque}\|_2 > \|f_{interpolate}\|_2$. But comparing the position of the minimum to the original computation we see that in this case the result is better "by accident".

5.7 Conclusions

The approach of interpolating the torque function values implemented in the actual program delivers on the one hand nearly exact results and on the other hand unusable results. The problem is that in some cases the optimizer is influenced by the "errors" done by interpolating the function values. However, even if there are unusable results we get highly correct results in other cases, we have the chance of parallelization of parts of the algorithm. The ideas and possible strategies for parallelization to reduce the computation time can be seen in the next chapter.

Model 1	Iterations	Time	Position of Minimum
Original	1427	5.735s	$\begin{pmatrix} -0.14097 \\ -0.262964 \\ 0.000077 \end{pmatrix}$
Parameter 1	1612	25.75s	$\begin{pmatrix} -0.140976 \\ -0.262964 \\ 0.000076 \end{pmatrix}$
Parameter 2	1657	32.516s	$\begin{pmatrix} -0.140835 \\ -0.262912 \\ 0.000075 \end{pmatrix}$
Parameter 3	1605	171.765s	$\begin{pmatrix} -0.140976 \\ -0.262964 \\ 0.000077 \end{pmatrix}$

Model 2	Iterations	Time	Position of Minimum
Original	6202	25.484s	$\begin{pmatrix} -0.133243 \\ -0.309130 \\ -0.002774 \end{pmatrix}$
Parameter 1	9186	154.797s	$\begin{pmatrix} -0.133243 \\ -0.309130 \\ -0.002774 \end{pmatrix}$
Parameter 2	10617	187.735s	$\begin{pmatrix} -0.133251 \\ -0.309126 \\ -0.002747 \end{pmatrix}$
Parameter 3	8286	812.344s	$\begin{pmatrix} -0.133243 \\ -0.309130 \\ 0.002774 \end{pmatrix}$

Model 3	Iterations	Time	Position of Minimum
Original	9003	43.906s	$\begin{pmatrix} 0.154371 \\ -0.198040 \\ -0.039909 \end{pmatrix}$
Parameter 1	7958	124.422s	$\begin{pmatrix} 0.045309 \\ 1.039755 \\ -0.008367 \end{pmatrix}$
Parameter 2	8849	152.609s	$\begin{pmatrix} -0.173883 \\ -0.124797 \\ -0.017115 \end{pmatrix}$
Parameter 3	11355	654.328s	$\begin{pmatrix} -0.007664 \\ -0.577341 \\ -0.011950 \end{pmatrix}$

Figure 5.7: Benchmarks for different sets of torque and interpolation parameters

Model 1	f_{torque}	$f_{interpolate}$	$f_{\Delta} = \lvert f_{torque} - f_{interpolate} \rvert$
Parameter 1	$\begin{pmatrix} 0.000004 \\ 0.000001 \\ -0.000003 \end{pmatrix}$	$\begin{pmatrix} 0.000004 \\ 0.000001 \\ -0.000003 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$
Parameter 2	$\begin{pmatrix} 0.000004 \\ 0.000001 \\ -0.000003 \end{pmatrix}$	$\begin{pmatrix} 0.000004 \\ 0.000001 \\ -0.000003 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$
Parameter 3	$\begin{pmatrix} 0.000004 \\ 0.000001 \\ -0.000003 \end{pmatrix}$	$\begin{pmatrix} 0.000004 \\ 0.000001 \\ -0.000003 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$

Model 2	f_{torque}	$f_{interpolate}$	$f_{\Delta} = \lvert f_{torque} - f_{interpolate} \rvert$
Parameter 1	$\begin{pmatrix} 0.000000 \\ 0.000064 \\ 0.000053 \end{pmatrix}$	$\begin{pmatrix} 0.000000 \\ 0.000064 \\ 0.000053 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$
Parameter 2	$\begin{pmatrix} 0.000000 \\ 0.000064 \\ 0.000053 \end{pmatrix}$	$\begin{pmatrix} 0.000000 \\ 0.000064 \\ 0.000053 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$
Parameter 3	$\begin{pmatrix} 0.000000 \\ 0.000064 \\ 0.000053 \end{pmatrix}$	$\begin{pmatrix} 0.000000 \\ 0.000064 \\ 0.000053 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$

Model 3	f_{torque}	$f_{interpolate}$	$f_{\Delta} = \lvert f_{torque} - f_{interpolate} \rvert$
Parameter 1	$\begin{pmatrix} -0.000000 \\ -0.000090 \\ -0.000009 \end{pmatrix}$	$\begin{pmatrix} 2.665771 \\ 25.094553 \\ -5.471609 \end{pmatrix}$	$\begin{pmatrix} 2.665771 \\ 25.0963 \\ 5.4716 \end{pmatrix}$
Parameter 2	$\begin{pmatrix} -0.000000 \\ -0.000090 \\ -0.000009 \end{pmatrix}$	$\begin{pmatrix} 0.000000 \\ 0.000002 \\ 0.000000 \end{pmatrix}$	$\begin{pmatrix} 0 \\ 0.000092 \\ 0.000009 \end{pmatrix}$
Parameter 3	$\begin{pmatrix} -0.000000 \\ -0.000090 \\ -0.000009 \end{pmatrix}$	$\begin{pmatrix} 3.540901 \\ 26.634500 \\ -4.324699 \end{pmatrix}$	$\begin{pmatrix} 3.540901 \\ 26.63459 \\ 4.32469 \end{pmatrix}$

Figure 5.8: Comparing the results of the optimization process both with and without interpolation

Model 1	$\|f_{torque}\|_2$	$\|f_{interpolate}\|_2$	$\|f_{torque}\|_2 - \|f_{interpolate}\|_2$
Parameter 1	0.000005	0.000005	0
Parameter 2	0.000005	0.000005	0
Parameter 3	0.000005	0.000005	0

Model 2	$\|f_{torque}\|_2$	$\|f_{interpolate}\|_2$	$\|f_{torque}\|_2 - \|f_{interpolate}\|_2$
Parameter 1	0.000083	0.000083	0
Parameter 2	0.000083	0.000083	0
Parameter 3	0.000083	0.000083	0

Model 3	$\|f_{torque}\|_2$	$\|f_{interpolate}\|_2$	$\|f_{torque}\|_2 - \|f_{interpolate}\|_2$
Parameter 1	0.00009	25.8221	-25.822
Parameter 2	0.00009	0.000002	0.000088
Parameter 3	0.00009	27.2147	-27.2146

Figure 5.9: Comparing the Euclidean norms of the results of the optimization process both with and without interpolation

Chapter 6

Parallelization

There are some possible approaches for accelerating our new implementation by parallelization. Moreover, we have the possibility to do parallelization on several levels i.e. we can make use of concurrent computation inside other parallel processes. The figures illustrating the strategies described in this chapter show the original three dimensional domain and the subdomains schematical as two dimensional pictures.

6.1 Parallel Interpolation

For obtaining the interpolated three dimensional result vector of the torque function, three interpolations have to be performed. These three interpolations can be done in parallel (Algorithm 6.1.1) in each of the possible strategies described before. Thus the parallel interpolation can be seen as a building block for later. The interpolation is an autonomous process only requiring three triangle vertices.

6.2 Parallel Grid Point Computation: Brute Force

The idea of the *brute force* method is to compute all the function values in a three dimensional grid (cube) in our domain of input values of the torque function.

The *brute force* method (Algorithm 6.2.1, Figure 6.1) is to create a regular set of grid (or mesh) points with a certain stepsize in our domain. Before starting the optimization we evaluate the Torque function for all possible triangle vertices (for all grid points) in parallel and save the computed values

Algorithm 6.1.1: RETURNVALUEPAR_BF(*subcube, x, y, z, stepsize*)

Steps 1 to 5 like in Algorithm 5.3.3
6. Perform a lookup in the pre-calculated *subcube*
to get the triangle points $P_{i,1\leq i\leq 3}$ of the surrounding triangle
7. Interpolate the three values in PARALLEL:
$\tilde{f}_1 = r_1 * f_1(P_1) + r_2 * f_1(P_2) + r_3 * f_1(P_3)$
$\tilde{f}_2 = r_1 * f_2(P_1) + r_2 * f_2(P_2) + r_3 * f_2(P_3)$
$\tilde{f}_3 = r_1 * f_3(P_1) + r_2 * f_3(P_2) + r_3 * f_3(P_3)$
return $\tilde{f}$

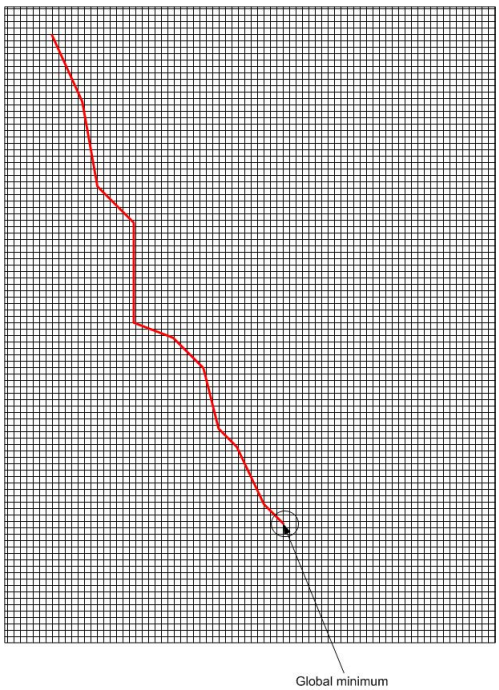

Figure 6.1: The Brute Force Strategy

Algorithm 6.2.1: INTERPOLATE_BF(x, y, z)

domaincube:=Pre-calculate the grid points over the
whole domain in PARALLEL using domain decomposition
2. Set the *stepsize*
$\tilde{f}$:=RETURNVALUE_BF(*domaincube, x, y, z, stepsize*)
return $\tilde{f}$

in a cubic data structure. If an interpolated value is requested, one has to do only three lookups in our data structure to get the three triangle vertex points and compute the interpolated result vector. For this approach, one needs the capability of a big supercomputer grid system because the Torque function has to be evaluated in every point of our mesh. With this approach, we dissipate computing power and memory by calculating many points not needed in further calculations.

The problem with such an implementation is that the bigger part of the computed values is not needed in the optimization algorithm. So this strategy is very inefficient. Nevertheless this algorithm is used as subalgorithm in later algorithms for smaller subsets.

6.3 Parallel Grid Point Computation: Subcubes

Dividing the Domain into Subcubes This strategy is based on the idea of dividing the domain into subcubes to restrict the torque function value computation to certain areas of interest (Figure 6.2). Like in the sequential case, we can make use of a queue. The queue management has to be performed by a master process.

To compute the function values in the subcube we have to use the *brute force* method for every subcube. This computation of all the grid (or mesh) points inside the subcube at the subcube's creation time can be done in parallel. There are some points computed which are not needed afterwards but in comparison to the *brute force* method, we narrow the calculations to subcubes really needed by the optimization process. With this method all the time-consuming evaluations of the Torque function can be sourced out to a supercomputer grid system before an interpolation is done. So, for every interpolation one only needs three simple lookups like in the *brute force*

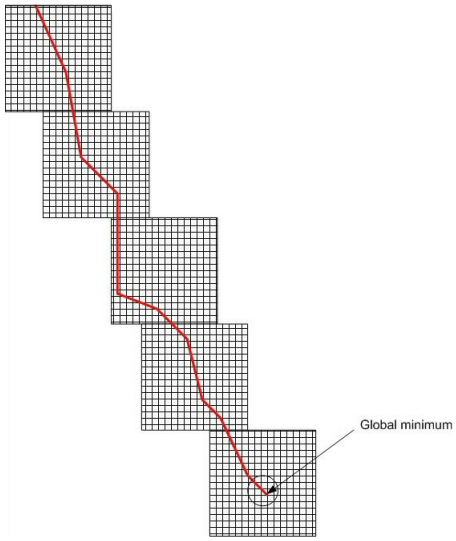

Figure 6.2: Subcube Strategy

Algorithm 6.3.1: INTERPOLATE_SC_BF(x, y, z)

1. Set the dimensions of the *cube of integers* and of the *subcubes* and the *stepsize*
2. Create the *cube of integers* and the *queue*
3. $\tilde{f}$:= CHECKCUBEPAR(*cube of integers*, x, y, z, *stepsize*)

return $\tilde{f}$

method. This option is depicted as pseudocode in the two Algorithms 6.3.1 and 6.3.2.

6.4 Parallel Grid Point Computation: Overlapping with Optimization

The problem with this strategy is that for each new subdomain the optimization algorithm has to wait during the interpolation is performed until it can continue. The idea to improve the solution is to decouple interpolation from optimization and let the interpolation process work with the new domain, while the optimizer still operates in the old domain.

Algorithm 6.3.2: CHECKCUBEPAR(*cube of integers, x ,y, z, stepsize*)

1. Determine the position P of the *subcube*
 containing the requested point in the domain
2. Check in the *cube of integers* if subcube P is computed

if *subcube is computed*

then $\begin{cases} \text{Get the queue position } i \text{ of the subcube} \\ \tilde{f} := \text{RETURNVALUE\_BF}(queue[i],\ x,\ y,\ z,\ stepsize) \end{cases}$

else $\begin{cases} \text{Create the subcube} \\ \text{Pre-compute all grid points in the } subcube \text{ in PARALLEL} \\ \text{add the } subcube \text{ to the } queue \\ \text{update the } (cube\ of\ integers) \\ \tilde{f} := \text{RETURNVALUE\_BF}(subcube,\ x,\ y,\ z,\ stepsize) \end{cases}$

return $\tilde{f}$

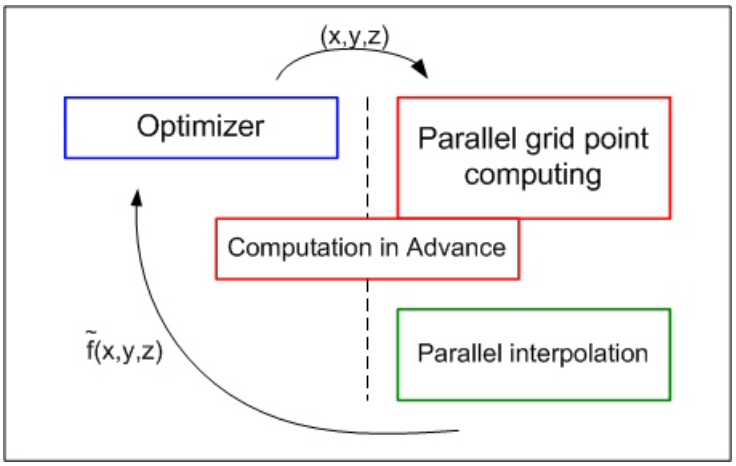

Figure 6.3: Schematical view of the parallel processes

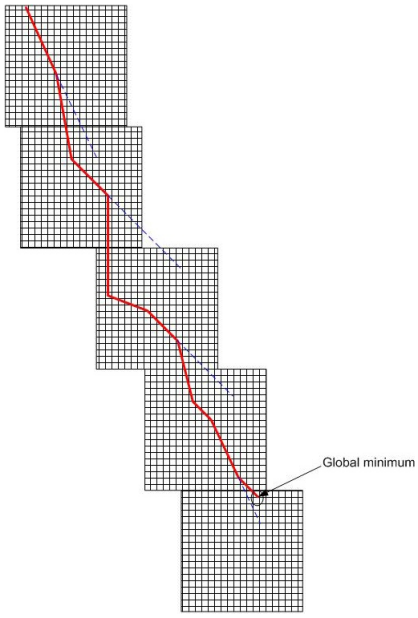

Figure 6.4: Overlapping Strategy

Computation in Advance A further approach calculating the subcubes is an overlapping strategy between the optimization and the grid point computation. The algorithm computes subcubes in advance during the optimization process. The current position of the optimization has to be checked and if this position is near the borders of the subcube, we start to compute subcubes touching these borders.

This strategy (Figure 6.4) combines optimization and interpolation best, because both can work nearly independently. For this purpose we need to know the current position of the optimizer. If this position is near the subcube border, we compute the subcube flanking this border.

An implementation of this algorithm needs two threads: one for the optimization algorithm and one for the interpolation. There has to be done some communication between the threads for providing the current position of the optimizer.

6.5 Implementation

In order to implement and test the program with parallelism the source code was ported to the current supercomputer environment (SGI Altix 4700 [28]) of the Johannes Kepler university. To compile the source code, the available

Intel Compiler (version 9.1) was used because it produces highly optimized code for the Intel Itanium processors integrated in the Altix system. Moreover this compiler is able to compile OpenMP commands. Basically speaking, OpenMP is an API for shared memory parallelism ([14], [11], [18], [4]). The implementation of the parallel parts of the program was performed using POSIX threads (pthreads) ([5], [29], [15], [22], [23], [24]) and OpenMP ([18], [4]).

In the parallel version the parallel fetching of the triangle points was implemented as an addition to Section 6.1.

6.5.1 Parallel Interpolation

The parallel interpolation of the three elements of the torque function's interpolated result vector is implemented using pthreads.

The implementation needs three additional methods in the Interpolation class. Each method of these three computes the interpolated value of one element of the result vector.

```
void * Interpol::fthr_1(void *arg){
        f[0]=r[0][0]*trp1[0]+r[1][0]*trp2[0]+r[2][0]*trp3[0];
        return(0);
}

void * Interpol::fthr_2(void *arg){
        f[1]=r[0][0]*trp1[1]+r[1][0]*trp2[1]+r[2][0]*trp3[1];
        return(0);
}

void * Interpol::fthr_3(void *arg){
        f[2]=r[0][0]*trp1[2]+r[1][0]*trp2[2]+r[2][0]*trp3[2];
        return(0);
}
```

The thread functions are called inside the Interpolate method with:

```
pthread_create(&fthr1, NULL, fthr_1, strings);
pthread_create(&fthr2, NULL, fthr_2, strings);
pthread_create(&fthr3, NULL, fthr_3, strings);

pthread_join(fthr1, NULL);
pthread_join(fthr2, NULL);
pthread_join(fthr3, NULL);
```

6.5.2 Parallel Triangle Fetching

The computation of the interpolated values of requires three points surrounding the requested value. In the sequential implementation three methods perform these triangle fetchings in a serial way.

To perform the fetching of the three triangles in parallel, again pthreads are used. To realize this idea three thread methods have to be added to the Interpolation class. Each method is responsible for one triangle point.

```
void * Interpol::thread_1(void *arg){
        trp1=subc->GetTrp1(x);
        return(0);
}

void * Interpol::thread_2(void *arg){
        trp2=subc->GetTrp2(x);
        return(0);
}

void * Interpol::thread_3(void *arg){
        trp3=subc->GetTrp3(x);
        return(0);
}
```

Again, the thread functions are called in the Interpolation method by:

```
pthread_create(&thread1, NULL, thread_1, strings);
pthread_create(&thread2, NULL, thread_2, strings);
pthread_create(&thread3, NULL, thread_3, strings);

pthread_join(thread1, NULL);
pthread_join(thread2, NULL);
pthread_join(thread3, NULL);
```

6.5.3 Parallel Subcube Computation

Basically, the computation of a subcube can be represented by filling a grid with function values in every node point. In the current sequential implementation this computation is performed by a triple nested for loop.

For the parallelization of this nested loop the OpenMP pragma *pragma omp parallel for* was used together with instancing private and shared variables. Inside this triple nested loop first the three dimensional input vector is computed out of the subcube's left down front corner point, the stepsize and the loop variables. The next step – the torque function evaluation – results in an three dimensional vector. The elements of this vector are inserted into the subcube at the associated point.

```
# pragma omp parallel for shared(subcube,N)
      private(i,j,l,func,hh,ax,ay,az,x,tmp)
for(i=0;i<N,i++){
  for(j=0;j<N;j++){
    for(l=0;l<N;l++){
      x[0][0]=(double)ax+i*h;
      x[0][1]=(double)ay+j*h;
      x[0][2]=(double)az+l*h;

      tmp=MatrixToColumnVector (this->funcObj->Evaluate(x));

      (*subcube[i][j][l]).f1=tmp[0];
      (*subcube[i][j][l]).f2=tmp[1];
      (*subcube[i][j][l]).f3=tmp[2];
    }
  }
}
```

6.5.4 Advance Computation

For the advance computation (the computation of a subcube during the optimization process) a border area of the subcube has to be defined. The border area is described by the subcube minus a cube inside the subcube with side length $a_innnersubc = a_subc - 2 * stepsize$ with the side length of the subcube a_subc. The optimization process requests interpolated function values. If a requested value resides in a predefined area near the side surfaces of the subcube, the subcube bordering this side surface is computed during the optimization process. The pseudocode is depicted in Algorithm 6.5.1.

For implementing the OpenMP structure of *omp parallel sections* was used. In our special case the number of threads is predefined and set to two (optimization thread, advance computation thread).

```
#pragma omp parallel sections num_threads(2) private (x)
  {
    #pragma omp section{
      interp->ComputeInAdvance(x,interp_border);
    }
    #pragma omp section
      {optimization}
```

6.6 Parallelizing the Software

During the tests of the the parallel subcube computation and the computation in advance it turned out that the computation of the torque function

Algorithm 6.5.1: ComputeInAdvance(x, *border*)

```
Determine the position of x in the subcube
 check if x is in the range of border near one of
the side surfaces
if x is in border area
  then {compute bordering subcube in parallel
        add subcube to the queue
  else {exit
```

evaluation is not thread save. The analysis of the torque function's source code showed problems in the Newmat library [16]. This library is highly optimized for single threaded computation but was not intended thread safe as the developer describes. The mathematical kernel of the SEE-KID/SEE-GRID software is based on this library. Therefore, parallel testing using the Newmat library causes memory access errors and crashes the running program.

To gain parallelism in the OpenMP parallelized parts of the program, every usage of the Newmat library has to be eliminated inside the OpenMP parallelizm. This is not possible for the computation in advance because the optimization process heavily uses the functionality of Newmat.

To achieve parallelizm in the parallel subcube computation of the program, two options are possible:

Option 1: Using another matrix library This option is not possible because of the lack of an interface for an easy change of the matrix library.

Option 2: Using a separate "server" process for the torque function evaluation The idea for option 2 is given by Unix process handling and communication possibilities and commands [21].

- The fork() command fully copies a process including its address space.
- POSIX semaphores allow to set up semaphores to lock and unlock certain ressources used by different processes or threads.
- POSIX shared memory can be applied to allocate memory for more than one process. This functionality affords inter process communication.

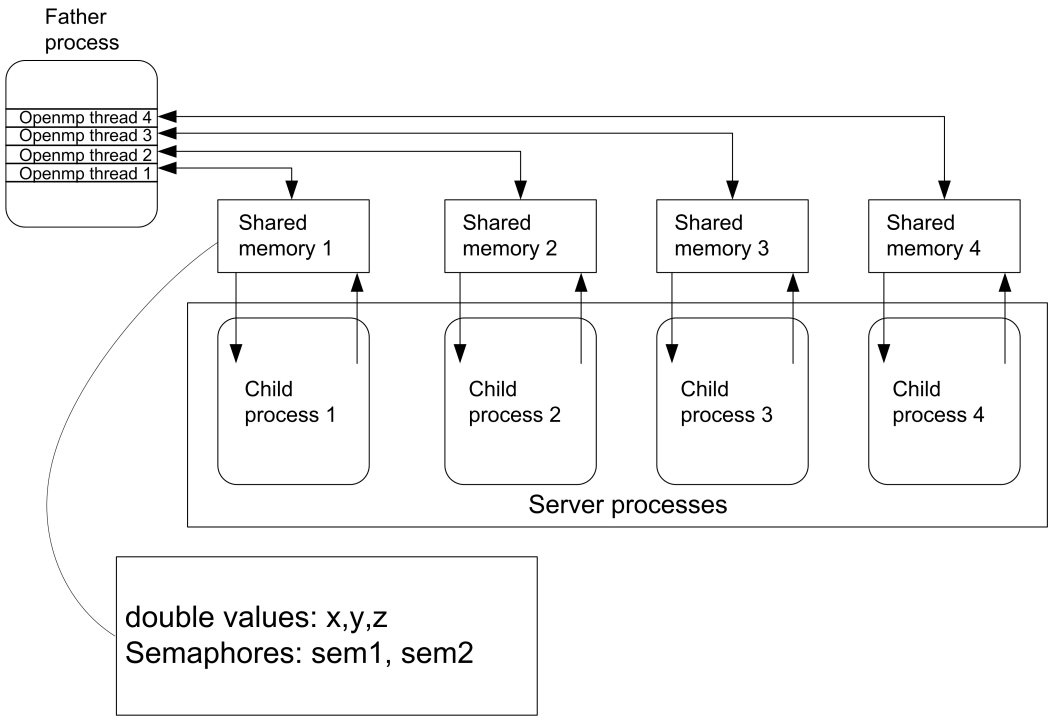

Figure 6.5: The structure of client server model (for four server processes)

First of all we have to get rid of everything using the Newmat library (e.g. matrices and column vectors). In our triple nested loop described in 6.5.3, both the matrix and the tmp column vector can be replaced using three dimensional double arrays (e.g. double * tmp = new double[3]). The subcube itself uses three double values inside a structure (per grid point) to store the torque function result vectors.

The evaluation method of the torque function is replaced by a new method handling the client-server model (Figure 6.5 showing the model for four server processes). OpenMP assigns each OpenMP thread a certain thread number. This thread number is used in the parallel program to identify the thread and evaluate the torque function on the right server process.

In this strategy we have to create a separate process for every OpenMP thread before starting the optimization. The process creating is realized by forking the original process *number of processes* times. After the fork call each process id is stored in a vector. The process ids are needed for killing the server processes.

As next step a separate shared memory space is assigned to every child process and the father process. Three double values and two semaphore values (one for the server and one for the client) reside in the shared memory structure for every process. The two semaphores are both initiated as blocking. After setting up the memory the child process transforms into a server process. Every server process first checks the server semaphore and waits till the semaphore's status is set to nonblocking. After waiting it fetches the three input values for the torque function out of the shared memory space and computes the three output values by calling the normal torque function evaluation method. The three output values are written into the shared memory, the client semaphore of the process is released and the server process blocks itself and waits for the next input values.

The client process writes the three input values into the shared memory, while the server process is waiting, releases the server process and blocks itself till the client semaphore is set to nonblocking from the server process after performing the computation. The the three values are popped out of the shared memory.

In the end the father process deallocates the shared memory of all server processes, sends the kill signal to each server process and receives (wait(&status)) the "terminated signal" of the children's (server) processes. The source code of the client-server model is shown in Appendix E.

6.7 Benchmarks

The test software is designed to switch the different parallel parts either on or off. The tests were performed on an SGI Altix 4700 system with 128 Intel Itanium2 Dualcore CPUs with 18 MB L3 Cache and 1 TB of RAM.

The benchmarks were done for the benchmark settings from Chapter 5.6 with (first without OpenMP parallelizm):

- parallel triangle fetching (triangle)
- parallel function interpolation (interpol)

Three combinations of the parallel methods were tested:

- triangle on, interpol off (par1)
- triangle off, interpol on (par2)
- triangle on, interpol on (par3)

Model 1	$time_o$ (s)	$time_s$ (s)	$time_{par1}$ (s)	$time_{par2}$ (s)	$time_{par3}$ (s)
Parameter 1	5.735	25.75	~~6.116~~	~~4.952~~	6.272
			~~6.512~~	4.976	~~6.52~~
			6.408	4.996	6.384
			6.244	~~5.132~~	~~5.736~~
			6.44	5.036	6.22
Average			6.364	5.003	6.292
Parameter 2	5.735	32.516	14.472	10.552	14.392
			~~14.664~~	10.668	14.656
			14.056	~~10.352~~	13.944
			~~13.68~~	~~10.752~~	~~14.764~~
			13.868	10.48	~~13.82~~
Average			14.132	10.567	14.331
Parameter 3	5.735	171.765	~~74.492~~	68.464	~~75.312~~
			72.412	68.52	~~72.924~~
			72.04	68.372	74.024
			~~71.708~~	~~68.668~~	73.648
			72.476	~~68.272~~	74.04
Average			72.31	68.452	73.904

Figure 6.6: Timings of the parallel computation (Model 1)

For every benchmark setting five test runs were performed. From the set of the achieved timings the minimum and maximum were eliminated and the average of the remaining three values was computed. The timings are depicted as Figures 6.6, 6.7 and 6.8. In the benchmarks the notations $time_o$ (original computation time) and $time_s$ (computation time of the sequential interpolation) are used.

Comparing the computation times of the sequential computation to the computation times of the parallel version shows that the computation time on the parallel system is remarkably faster than the sequential interpolation version but much slower than the original sequential version. One can see that the combination with parallel triangle fetching off and parallel interpolation on shows the best results in nearly all benchmark sets. The overhead for multithreading the triangle fetching methods is too high because the triangle fetching methods perform only lookups.

The benchmarks in Figures 6.9, 6.10 and 6.11 compare the timings of parallel subcube computation with parallel interpolation both switched on and off. In contrast to the benchmarks without using OpenMP, the utilization

Model 2	$time_o$	$time_s$ (s)	$time_{par1}$ (s)	$time_{par2}$ (s)	$time_{par3}$ (s)
Parameter 1	25.484	154.797	63.924	~~47.544~~	63.916
			~~64.984~~	46.904	~~61.04~~
			64.2	47.052	~~65.776~~
			~~61.852~~	~~46.364~~	62.616
			62.832	47.4	64.2
Average			63.652	47.119	63.577
Parameter 2	25.484	187.735	~~52.304~~	39.44	55.264
			52.648	~~38.08~~	52.368
			52.552	~~39.688~~	~~51.04~~
			52.48	39.16	52.068
			~~52.66~~	38.52	~~95.22~~
Average			52.56	39.04	53.234
Parameter 3	25.484	812.344	252.924	274.084	260.492
			~~245.936~~	~~274.296~~	~~246.316~~
			258.976	~~273.664~~	272.26
			260.988	274.024	277.368
			~~279.068~~	273.928	~~279.776~~
Average			257.629	274.012	270.04

Figure 6.7: Timings of the parallel computation (Model 2)

Model 3	$time_o$	$time_s$ (s)	$time_{par1}$ (s)	$time_{par2}$ (s)	$time_{par3}$ (s)
Parameter 1	43.906	124.422	68.52	48.8	65.176
			68.864	48.288	67.668
			~~67.176~~	~~48.932~~	~~62.752~~
			~~70.928~~	47.92	~~70.568~~
			68.616	~~47.824~~	67.264
Average			68.667	48.336	66.703
Parameter 2	43.906	152.609	~~35.612~~	~~28.212~~	38.928
			37.096	27.676	~~40.38~~
			35.86	28.12	~~35.608~~
			37.352	~~27.16~~	40.008
			~~37.808~~	27.536	39.688
Average			36.77	27.778	39.541
Parameter 3	43.906	654.328	~~263.4~~	~~237.076~~	246.756
			247.84	237.476	249.252
			245	237.424	~~250.768~~
			245.14	~~238.032~~	~~236.788~~
			~~242.528~~	237.368	239.372
Average			245.993	237.423	245.127

Figure 6.8: Timings of the parallel computation (Model 3)

of the POSIX parallelizm (parallel interpolation) leads higher computation times because of the overhead caused by creating and deleting the threads.

The benchmarks were accomplished using 2, 4 and 8 processes. Due to a faster sequential testing the dimensions of the subcubes were chosen 3, 4 and 8. In the triple nested loop every loop variable has to perform 3, 4 or 8 steps. On this account Parameter Set 1 and 2 are replaced by two other sets with subcube dimension 16:

1. *new* Parameter 1:
   (a) Size of the domain in one dimension: 64
   (b) Stepsize: 1
   (c) Number of subcubes: 8
   (d) Queuesize: 10
2. *new* Parameter 2:
   (a) Size of the domain in one dimension: 32
   (b) Stepsize: 0.25
   (c) Number of subcubes: 16
   (d) Queuesize: 10

These two sets are used to perform the benchmarks depicted in Figures 6.12, 6.13 and 6.14. The tests with Parameter set 1 and 2 were executed additionally with 16 processes. In these figures the increasing speed using more processes can be seen easily.

Model 1	$time_o$ (s)	$time_{par2}$ (s)	$time_2$ (s)	$time_4$ (s)	$time_8$ (s)
Parameter 3	5.735	68.452	27.160	15.133	~~12.116~~
			~~27.455~~	~~17.203~~	11.172
			~~26.158~~	17.094	11.176
			27.145	~~15.126~~	10.138
			27.168	15.129	~~10.128~~
Average			27.158	15.79	10.83
Parameter 3 + par2	5.735	68.452	30.15	21.135	~~13.165~~
			26.196	~~21.144~~	~~15.135~~
			~~26.068~~	20.137	16.186
			~~30.2~~	19.182	16.266
			26.242	~~17.131~~	16.213
Average			27.53	20.152	16.22

Figure 6.9: Timings of the parallel subcube computation (Model 1, Parameter Set 3)

Model 2	$time_o$ (s)	$time_{par2}$ (s)	$time_2$ (s)	$time_4$ (s)	$time_8$ (s)
Parameter 3	25.484	274.012	80.456	~~44.316~~	~~29.179~~
			79.226	45.293	~~30.102~~
			~~82.416~~	46.076	29.186
			81.29	~~50.184~~	29.197
			~~79.118~~	45.293	29.285
Average			80.324	45.554	29.223
Parameter 3 + par2	25.484	274.012	92.228	~~52.244~~	38.112
			92.401	56.352	39.237
			~~103.254~~	54.152	39.189
			~~88.09~~	~~62.520~~	~~36.152~~
			88.95	60.358	~~43.362~~
Average			91.193	56.954	38.846

Figure 6.10: Timings of the parallel subcube computation (Model 2, Parameter Set 3)

Model 3	$time_o$ (s)	$time_{par2}$ (s)	$time_2$ (s)	$time_4$ (s)	$time_8$ (s)
Parameter 3	43.906	237.423	102.33	~~52.213~~	~~41.279~~
			104.867	56.298	38.227
			~~106.542~~	~~58.354~~	39.275
			102.615	55.29	37.181
			~~95.344~~	55.267	~~36.241~~
Average			103.27	55.618	38.228
Parameter 3 + par2	43.906	237.423	117.493	79.141	~~54.164~~
			~~117.73~~	~~75.207~~	~~64.305~~
			115.497	77.58	63.333
			116.413	75.503	58.353
			~~108.194~~	~~83.395~~	77.313
Average			116.468	77.408	66.333

Figure 6.11: Timings of the parallel subcube computation (Model 3, Parameter Set 3)

Model 1	$time_o$ (s)	$time_s$ (s)	$time_2$ (s)	$time_4$ (s)	$time_8$ (s)	$time_{16}$ (s)
new Parameter 1	5.735	83.428	~~41.251~~	22.21	~~12.108~~	~~8.11~~
			43.222	~~24.174~~	12.129	9.096
			44.245	23.131	~~13.063~~	9.161
			42.276	~~22.091~~	12.11	9.202
			~~48.104~~	23.141	12.114	~~10.146~~
Average			43.248	22.827	12.118	9.153
new Parameter 2	5.735	139.131	~~83.478~~	45.349	23.155	~~17.234~~
			82.219	44.283	~~24.204~~	15.17
			82.44	~~44.16~~	23.239	16.116
			~~479.231~~	~~46.593~~	23.145	14.227
			82.345	44.279	~~22.224~~	~~14.199~~
Average			82.335	44.637	23.18	15.171

Figure 6.12: Timings of the parallel subcube computation (Model 1, new Parameter Set 1 and 2)

Model 2	$time_o$ (s)	$time_s$ (s)	$time_2$ (s)	$time_4$ (s)	$time_8$ (s)	$time_{16}$ (s)
new Parameter 1	25.484	209.848	159.586	~~85.143~~	45.942	~~26.189~~
			159.416	~~81.182~~	46.345	26.309
			159.758	82.19	47.291	27.212
			~~160.148~~	84.242	~~48.13~~	~~27.311~~
			~~156.198~~	82.144	~~42.191~~	27.219
Average		159.587	82.859	46.526	26.914	
new Parameter 2	25.484	402.067	266.155	~~137.181~~	~~80.49~~	~~54.306~~
			262.219	~~121.703~~	77.441	50.377
			~~272.404~~	136.5	80.132	53.478
			257.913	134.795	77.444	51.43
			~~248.278~~	135.096	~~74.508~~	~~47.241~~
Average			262.096	135.464	78.339	51.76

Figure 6.13: Timings of the parallel subcube computation (Model 2, new Parameter Set 1 and 2)

Model 3	$time_o$ (s)	$time_s$ (s)	$time_2$ (s)	$time_4$ (s)	$time_8$ (s)	$time_{16}$ (s)
new Parameter 1	43.906	261.054	178.256	~~91.456~~	53.229	34.286
			~~160.184~~	92.518	~~56.254~~	34.289
			174.476	93.303	~~48.234~~	~~31.236~~
			176.802	99.326	53.367	~~37.388~~
			~~197.602~~	~~99.416~~	55.34	35.171
Average			176.511	95.049	53.979	34.582
new Parameter 2	43.906	400.5	279.48	164.154	~~90.464~~	46.29
			~~288.326~~	~~168.7433~~	80.443	44.378
			286.247	163.689	77.425	47.217
			287.303	~~159.666~~	81.555	~~43.284~~
			~~278.977~~	163.214	~~76.203~~	~~47.358~~
Average			284.343	163.686	79.808	45.962

Figure 6.14: Timings of the parallel subcube computation (Model 3, new Parameter Set 1 and 2)

Chapter 7

Conclusions

In our work we both focused on the acceleration of the sequential computations and the parallel approach. We introduced the biomechanical model of the human eye which is an elementary part of the SEE-KID/SEE-GRID software. First, the improvement of the sequential optimization process was discussed. As a possible way to accelerate the optimizer the Broyden update (updating the Jacobian or Hessian matrices instead of recomputing it completely) was introduced. As a next strategy for speeding up the computation kernel, the Delaunay triangulation with a succeeding interpolation of the torque function values was described and implemented. To avoid the time-consuming triangle search necessary in the Delaunay algorithm caused by searching in the list of points, a new method based on the regularity of the mesh in our interpolation routines was introduced. We described a new approach to replace searching by simple lookups. Finally, possible parallelization strategies were developed and implemented. One strategy consisted of multithreading certain parts of the program whereas another strategy focused on gaining speedup using the capability of the OpenMP automatic parallelizer with domain decomposition. Due to problems with the linear algebra library called Newmat, we had to find a strategy to evaluate the torque function in parallel without crashing the program.

The Broyden update method was implemented and tested but did not yield to an improved version of the optimization process. The speedup is achieved at expense of precision. This loss of precision may lead to incorrect results.

The Delaunay approach was tested too but due to the time consuming triangle search this strategy had to be rejected.

The idea of interpolating the function values in the Delaunay approach was picked up in the strategy of interpolation using the regular mesh structure. Running tests using this strategy lead to very good results in most of

the cases but failed in tests with other benchmark settings. Moreover, the measured computation time using the regular mesh strategy together with interpolation highly increases due to the large number of torque function evaluations needed to create the subcubes.

To decrease the computation time, suggestions were made to parallelize certain parts of the program. Problems were encountered trying to parallelize parts of the program which use the Newmat library. For evaluating the torque function in parallel, a special client server model was developed and implemented. The computation time of the interpolation method decreases using this client server model but does not speedup relative to the original timings of the sequential program without interpolation.

Future work should concentrate both on implementing an interface to be able to use other linear algebra libraries and find additional interpolation methods. Using interpolation in combination with domain decomposition enables the application of parallelizm.

Appendix A

Source: Code Broyden update

This code snippet shows the Broyden update procedure. In the **for**-loop the Torque function is evaluated. The computation of the updated matrix is done as described in Section 3.7.

```
Matrix SeeOSimplex::Optimize (Matrix &xIn){
  Matrix x = xIn;
  ColumnVector params(20); values(params, 0);

  if(funcObj == 0) {
    // if no funcObj is set
    return Matrix(0,0);
  } // if
  if(paramObj == 0) {
    // if no paramObj is set
    return Matrix(0,0);
  } // if

  params[2] = paramObj->GetTerminationThreshold();
  params[3] = paramObj->GetResidualThreshold();
  params[4] = 1.0e-6;
  params[5] = 0; // LEVENBERG
  params[14] = paramObj->GetIterationMaximum();
  params[16] = paramObj->GetStepAccuracy(); // Stepsize
    when modifying Parameters (Default is 1e-8)
  params[17] = paramObj->GetStepSize(); // Default 0.1

  ColumnVector XOUT (xIn.Storage());
  XOUT = MatrixToColumnVector (xIn);
  int nvars = XOUT.Storage ();
  ColumnVector f (nvars);
  f = MatrixToColumnVector (funcObj->Evaluate(x));
  if (funcObj->EvaluationError())
    return Matrix(0,0);

  int nfun = f.Nrows ();
  Matrix GRAD (nvars, nfun);
  values (GRAD, 0);
  ColumnVector OLDX (XOUT.Storage()); OLDX << XOUT;

  Matrix MATX (3,1);
  values (MATX, 0);

  ColumnVector MATL (3);
  values (MATL, 0);
  MATL[0] = f.SumSquare();

  ColumnVector OLDF_2;
  double OLDF = MATL[0];
  double FIRSTF = OLDF;

  int Broydenc=0;

  OLDX = XOUT;
  OLDF_2 = f;
```

```
if(params [14] == 0) {
  params [14] = XOUT.Nrows()*100.0;
} // if

double PCNT = 0.0;
double EstSum = 0.5;
double GradFactor = 0.0;
double newstep = 0.0;
double fbest = 0.0;
double fnew = 0.0;
ColumnVector OX (xIn.Storage());

ColumnVector chg_ones (nvars); values (chg_ones,1);
ColumnVector CHG (nvars); CHG << prec7 *
  Abs (XOUT) + prec7 * chg_ones;

params [10] = 1;
int status = -1;
Matrix SD;
Matrix newf;
Matrix OLDG;
Matrix GDOLD;
Matrix FOLD;
Matrix OLDJ;
IdentityMatrix Iden(3);
Matrix uk,vk,wk,sk,yk,pk;
Matrix Hk (nvars,nfun); values (Hk,0);

while ((status != 1) && (!terminate)) {
  // Calculate Gradients
    OLDF_2.Inject (f);
  ColumnVector param16 (nvars);
    ColumnVector param17 (nvars);

  values(param16, params[16]);
  values(param17, params[17]);

  if(Broydenc==0){
    //cout<<"Broyden Restart begin"<<endl;
    //cout<<"~~~~~~~~~~~~~~~~~~~~~~~~~~"<<endl;
    CHG = Multiply(Signum (CHG+eps),
      Minimum(Maximum(Abs (CHG), param16), param17));

    for(int gcnt = 0; gcnt < nvars; gcnt++) {
      double temp = XOUT [gcnt];
      XOUT [gcnt] = temp + CHG [gcnt];

      x = XOUT;
      f = MatrixToColumnVector (funcObj->Evaluate(x));
        // fk:Get force imbalance for eye position x
      if (funcObj->EvaluationError())
        return Matrix(0,0);

      if (CHG [gcnt] == 0.0) {
        RowVector tempRow(nvars);
        GRAD.Row (gcnt+1) = values (tempRow,0.0);
      } // if
      else {
        GRAD.Row (gcnt+1) = ((f-OLDF_2)).t()/CHG [gcnt];
      } // else

      XOUT [gcnt] = temp;
    } // for

    f.Inject (OLDF_2);
    params [10] = params [10] + nvars;


    if(nfun == 1) {
      CHG = Rdevide (nfun*prec8,GRAD);
    } // if
    else {
      CHG = Rdevide (nfun*prec8,ColSum
        (Abs(GRAD).t()).t());
      } // else

    OLDJ.CleanUp();
    OLDJ=Iden;
    FOLD=f;
    OLDX=x;
    OLDF_2=f;

  }
```

```
    else{
      pk=OLDJ.i()*(-1)*FOLD;
      //neues x (xk+1) berechnen
      XOUT=OLDX+pk;

      //fk+1 berechnen
      for(int gcnt = 0; gcnt < nvars; gcnt++) {
        x = XOUT;
        f = MatrixToColumnVector (funcObj->Evaluate(x));
          // fk:Get force imbalance for eye position x
        if (funcObj->EvaluationError()){
          //cout<<"Function Error"<<endl;
          return Matrix(0,0);
        }
      }

      //BFGS-Verfahren
      //yk=f-FOLD;
      yk=f-FOLD;
      sk=XOUT-OLDX;
      wk=OLDJ*sk;
      vk=sk.t()*sk;
      uk=(1/vk.element(0,0))*(yk-wk);

      //upgedatete Matrix berechnen
      GRAD=OLDJ+(uk*sk.t());

      //Bk wird zu Bk+1
      OLDJ=GRAD;
      FOLD=f;
    }

    //Zï½½hler fï½½r Broyden-Update erhï½½hen
    Broydenc++;

    //Jedes zweite Mal (jetzt) die Matrix neu berechnen
    if(Broydenc==2){
      Broydenc=0;
    }

    Matrix gradf (nvars,nfun); values (gradf,0);
    gradf = 2*GRAD*f;
    Matrix hund = f.t() * f;
      fnew = hund[0][0];

    if(status == -1) {
      if(cond (GRAD) > prec80) {
        Matrix identity (nvars, nvars);
        eye (identity);
        Matrix a = (GRAD*GRAD.t()+(norm (GRAD)+1)*
          (identity));
        Matrix ai = a.i();
        Matrix b = GRAD*f;
        SD = -(ai*b);

        if (params [5] == 0) {
          GradFactor=norm(GRAD)+1.0;
        } // if
      } // if
      else {
        Matrix identity (nvars, nvars);
        eye (identity);
        Matrix a = (GRAD*GRAD.t()+GradFactor*(identity));
        Matrix ai;
        //try
        {
          ai = a.i();
        Matrix b = GRAD*f;
        SD = -(ai*b);
        }
        //catch (...) { SD = GRAD*f; }

      } // else
      FIRSTF = fnew;
      OLDG = GRAD;
      GDOLD = gradf.t()*SD;

      if(params [18] == 0) {
        params [18] = 1;
      } // if

//        printf("%5.0f %12.6f %12.3f %12.3f",
        params [10],fnew,params [18],GDOLD[0]);
```

```
      XOUT = XOUT+params[18]*SD;

      if(params [5] == 0) {
      newf = GRAD.t()*SD+f;
      GradFactor = newf.SumSquare();
      Matrix identity (nvars, nvars);
      eye (identity);
      Matrix a = (GRAD*GRAD.t()+GradFactor*(identity));
      Matrix ai;
      Matrix b;
      //try
      {
        ai = a.i();
        b= GRAD*f;
        SD = -(ai*b);
      }
      //catch (...)
      {
      //  SD = (GRAD*f);
      }
      } // if

   newf=GRAD.t()*SD+f;
   XOUT = XOUT + params [18]*SD;
   EstSum=newf.SumSquare();
   status=0;

   if(params [7]==0) {
     PCNT=1;
   } // if
} // if
else {
//-------------Direction Update-----------------

   Matrix gdnew(gradf.Nrows(), gradf.Ncols());
     gdnew << gradf.t()*SD;
   /*
   if(IsGreater(gdnew, 0.0) && (fnew>FIRSTF)) {
     // Case 1: New function is bigger than
       last and gradient w.r.t. SD -ve
     // ... interpolate.
     //[stepsize]=cubici1(fnew,FIRSTF,
         gdnew,GDOLD,OPTIONS(18));
     //OPTIONS(18)=0.9*stepsize;
   } // if
   else*/
   if(fnew<FIRSTF) {
     //  New function less than old fun.
       and OK for updating
     //  .... update and calculate new direction.

     cubinterp3(newstep,fbest,fnew,FIRSTF,
       gdnew,GDOLD,params[18]);
     if(fbest>fnew) {
       fbest=0.9*fnew;
     } // if
     if(gdnew[0][0] < 0) {
       if(newstep < params [18]) {
         newstep = (2*params[18]+1e-4);
         params [18]=fabs(newstep);
       } // if
       else {
         if(params [18] > 0.9) {
           params [18]=__MIN(1,fabs(newstep));
         } // if
       } // else
     } // if

     // SET DIRECTION.
     // Gauss-Newton Method

   double temp = 1;
     if(params [5] == 1) {
       if((params [18] > prec8) &&
         (cond (GRAD) < prec80)) {
         SD = GRAD.t().i()*(GRAD.t()*XOUT-f)-XOUT;
         Matrix h_ = SD.t()*gradf;
         if(h_[0][0]>eps) {
         } // if
         temp=0;
       } // if
       else {
```

```
/*          cout << "Conditioning of Gradient Poor -
            Switching To LM method";*/
            params [5]=0;
            params [18]=fabs(params[18]);
          } // else
        } // if

        if(temp) {
          // Levenberg_marquardt Method N.B.
          // EstSum is the estimated sum of squares.
          //  GradFactor is the value of lambda.
          // Estimated Residual:

          if(EstSum>fbest) {
            if (1.0+params[18] == 0.0)
              GradFactor = 0.0;
            else
              GradFactor=GradFactor/(1.0+params[18]);
          } // if
          else {
            if (params[18]+eps == 0.0)
              GradFactor = 0.0;
            else
              GradFactor=GradFactor+(fbest-EstSum)/
                (params[18]+eps);
          } // else

          Matrix identity (nvars, nvars);
          eye (identity);
          Matrix a = (GRAD*GRAD.t()+GradFactor*(identity));
          Matrix ai = a.i();
          Matrix b = GRAD*f;
          SD = -(ai*b);

          params [18]=1;
          Matrix estf=GRAD.t()*SD+f;
          EstSum=estf.SumSquare();
        } // if

        gdnew=gradf.t()*SD;

        OLDX=XOUT;
        // Save Variables
        FIRSTF=fnew;
        OLDG=gradf;
        GDOLD=gdnew;

        // If quadratic interpolation set PCNT
        if(params [7] == 0) {
          PCNT=1;
          values (MATX,0);
          MATL[0]=fnew;
        } // if
      } // else if
      else {
        // Halve Step-length
          if(fnew==FIRSTF) {
/*          cout << "No improvement in search
                direction: Terminating" << endl;*/
          status=1;
        } // if
        else {
          params [18] = params [18] / 8.0;
          if(params [18] < prec8) {
            params [18] = -params[18];
          } // if
        } // else
      } // else

      XOUT=OLDX+params [18]*SD;
    } // else
    //----------End of Direction Update----------------

    if(params [7] ==0) {
      PCNT=1;
      values (MATX,0);
      MATL[0]=fnew;
    } // if
    // Check Termination

    Matrix h_ = gradf.t()*SD;
    if((Abs(SD).Maximum()< params [2]) &&
      (h_[0][0] < params [3]) &&
```

```
      (Abs(gradf).Maximum() < 10*(params[3]+params[2])))
    {
        //cout << "Optimization Terminated Successfully"
            << endl;
        status=1;
    } // if
    else if(params [10] > params [14]) {
        status=1;
    } // else if
    else {
      // Line search using mixed polynomial
      // interpolation and extrapolation.
      if(PCNT != 0) {
        while ((PCNT > 0) && (!terminate)) {
          x = XOUT;
          f = MatrixToColumnVector (funcObj->Evaluate(x));
          if (funcObj->EvaluationError())
            return Matrix(0,0);

          params [10] = params [10]+1;
          fnew = f.SumSquare();

          // <= used in case when no improvement found.
          if(fnew <= OLDF_2.SumSquare()) {
            OX = XOUT;
            OLDF_2=f;
              } // if

          double steplen = params [18];
          quadsearch (PCNT,fnew,MATL,MATX,GDOLD,steplen);
          params [18] = steplen;

          XOUT=OLDX+steplen*SD;
          if(fnew==FIRSTF) {
            PCNT=0;
                } // if
        } // while
        XOUT = OX;
        f=OLDF_2;
      } // if
      else {
        x =XOUT;
        f = MatrixToColumnVector (funcObj->Evaluate(x));
        if (funcObj->EvaluationError())
          return Matrix(0,0);

        params [10] = params [10]+1;
      } // else
    } // else
  } // while
  params [8] = fnew;
  XOUT=OLDX;
  x=XOUT;
  cout<<"x "<<x<<endl;
  return (x.AsRow());
}
```

Appendix B

Source Code: Delaunay Test

```
DelaunayInterpolator xx (points1);
xx.TriangulationToMathematica ("D:\\Arbeit\\SEE_Optimizer\\trig.txt");
Matrix grid;
xx.CalculateGrid (30,30,grid);

xx.SetPoints(points1);
std::fstream str_grid;
str_grid.open("D:\\Arbeit\\SEE_Optimizer\\grid.txt", ios::out);
if(!str_grid.good()) {
 return(0);
} // if
str_grid << "{";
for (int i=0;i<grid.Nrows();i++)
{
  str_grid << "{";
  for (int j=0;j<grid.Ncols();j++)
    {
    char num[100];
    sprintf (num,"%.3f",grid[i][j]);
    str_grid << num;
    if (j < grid.Ncols()-1)
    str_grid << ",";
  }
  str_grid << "}"<<endl;
  if (i < grid.Nrows()-1)
    str_grid << ",";
}
str_grid << "}";
  str_grid.close();
bool valid;
for (double x=-xw;x<xw;x+=2){
  for (double y=-xw;y<xw;y+=2){
    z = gm->Listing (x,y);
    gm->SetHeadfixGazeAngles (y,x,z);
    mm->CalculateForces (MuscleModel::_6dof);
    t = mm->Torque (MuscleModel::_6dof, 1);
    //errVal = t.length();
    //errVal = t.z();
    errVal = t.length();
    errVal = pow(errVal,1.2);
    str_delau<<"("<<x<<", "<<y<<");"<<"delaunay: ;";
    str_delau<<xx.f(x,y,valid)<<";torque: ;"<<errVal<<endl;
  }
}
str_delau.close();
```

Appendix C

Source Code: Sequential Interpolation (Prototype Implementation)

```
#include <iostream>
#include <math.h>

using namespace std;

typedef struct element{
   double f1,f2,f3;
   bool calculated;
} Element;

typedef struct celement{
  int v;
}CElement;

typedef struct queueelement{
  Element****sc;
  int xc,yc,zc; //cube coordinates
}QueueElement;


Element**** createSubcube(int N){
  int i,j,l;
  Element **** subcube;

  subcube = (Element ****)
    malloc(sizeof(Element****) * N);
  for (i=0; i<N; i++){
    subcube[i] = (Element ***)
      malloc(sizeof(Element ***) *N);
    for (j=0; j<N; j++){
      subcube[i][j] = (Element **)
        malloc(sizeof(Element**) * N);
      for (l=0; l<N; l++){
        subcube[i][j][l] = (Element*)
          malloc(sizeof(Element));
      }
    }
  }
  for(i=0;i<N;i++){
    for(j=0;j<N;j++){
      for(l=0;l<N;l++){
        (*subcube[i][j][l]).f1=0.0;
        (*subcube[i][j][l]).f2=0.0;
        (*subcube[i][j][l]).f3=0.0;
        (*subcube[i][j][l]).calculated=false;

      }
```

```
    }
  }
  cout<<"subcube init complete"<<endl;
  return(subcube);
}

QueueElement* createSubcubeQueue(int queueElements){
  QueueElement * subcubeQueue;

  subcubeQueue = (QueueElement*)malloc(sizeof(QueueElement)
                 * queueElements);

  for(int i=0;i<queueElements;i++){
    subcubeQueue[i].xc=-1;
    subcubeQueue[i].yc=-1;
    subcubeQueue[i].zc=-1;
  }

  cout<<"subcubeQueue init complete"<<endl;
  return(subcubeQueue);
}


CElement**** createCube(int N){
  int i,j,l;
  CElement **** cube;

  cube = (CElement ****)
    malloc(sizeof(CElement****) * N);
  for (i=0; i<N; i++){
    cube[i] = (CElement ***)
      malloc(sizeof(CElement ***) *N);
    for (j=0; j<N; j++){
      cube[i][j] = (CElement **)
        malloc(sizeof(CElement**) * N);
      for (l=0; l<N; l++){
        cube[i][j][l] = (CElement*)
          malloc(sizeof(CElement));
      }
    }
  }
  for(i=0;i<N;i++){
    for(j=0;j<N;j++){
      for(l=0;l<N;l++){
        (*cube[i][j][l]).v=-1;
      }
    }
  }
  cout<<"cube init complete"<<endl;
  return(cube);
}

void addQueue(CElement **** cube,
              QueueElement * subcubeQueue,
              int queueElements, Element **** newsubcube,
              int newxc, int newyc, int newzc){
  if((subcubeQueue[queueElements-1].xc!=-1)&&
  (subcubeQueue[queueElements-1].yc!=-1)&&
  (subcubeQueue[queueElements-1].zc!=-1)){
        (*cube[subcubeQueue[queueElements-1].xc]
        [subcubeQueue[queueElements-1].yc]
        [subcubeQueue[queueElements-1].zc]).v=-1;
  }
  for(int i=queueElements-1;i>0;i--){
    if((subcubeQueue[i].xc!=-1)&&
    (subcubeQueue[i].yc!=-1)&&
    (subcubeQueue[i].zc!=-1)){
      (*cube[subcubeQueue[i].xc]
      [subcubeQueue[i].yc]
      [subcubeQueue[i].zc]).v=i ;
    }
    subcubeQueue[i].sc=subcubeQueue[i-1].sc;
    subcubeQueue[i].xc=subcubeQueue[i-1].xc;
    subcubeQueue[i].yc=subcubeQueue[i-1].yc;
    subcubeQueue[i].zc=subcubeQueue[i-1].zc;
  }
  subcubeQueue[0].sc=newsubcube;
  subcubeQueue[0].xc=newxc;
  subcubeQueue[0].yc=newyc;
  subcubeQueue[0].zc=newzc;
  (*cube[subcubeQueue[0].xc]
  [subcubeQueue[0].yc]
  [subcubeQueue[0].zc]).v=0;
```

```
}


double* func(double x, double y, double z){
  double * f;
  f = (double*)malloc(sizeof(double) * 3);
  f[0]=x*x+y*y+z*z+4;
  f[1]=x*x-y*y+z*z-3;
  f[2]=x*x+y*y-z*z+1;
  return f;
}


double* returnValue(Element**** subcube,
                    double x,double y, double z,
                    double h,int dimsc){
  double xl,xu,yl,yu,zl,zu;
  int*tr1,*tr2,*tr3;
  double plane;
  double deta;
  int i;
  tr1 = (int*)malloc(sizeof(int) * 2);
  tr2 = (int*)malloc(sizeof(int) * 2);
  tr3 = (int*)malloc(sizeof(int) * 2);

  xl=(int)x;
  xu=xl+h;
  yl=(int)y;
  yu=yl+h;
  zl=(int)z;
  zu=zl+h;
  cout<<xl<<" "<<xu<<" "<<yl<<" "<<yu<<" "<<zl<<" "<<zu<<endl;
  if((z-zl)<(zu-z)){
    plane=zl;
  }
  else{
    plane=zu;
  }
  //Dreieckspunkte
  tr1[0]=xl;
  tr1[1]=yl;
  if(x>y){
    tr2[0]=xu;
    tr2[1]=yl;
  }
  else{
    tr2[0]=xl;
    tr2[1]=yu;
  }
  tr3[0]=xu;
  tr3[1]=yu;

  double ** m;
  double **mi;

  m = (double**)malloc(sizeof(double*) * 3);
  for(i=0;i<3;i++){
    m[i] = (double*)malloc(sizeof(double) * 3);
  }

  mi = (double**)malloc(sizeof(double*) * 3);
  for(i=0;i<3;i++){
    mi[i] = (double*)malloc(sizeof(double) * 3);
  }

  m[0][0] = tr1[0]; m[0][1] = tr2[0]; m[0][2] = tr3[0];
  m[1][0] = tr1[1]; m[1][1] = tr2[1]; m[1][2] = tr3[1];
  m[2][0] =       1; m[2][1] =       1; m[2][2] =       1;

  cout<<"Matrix"<<endl;
  cout<<m[0][0]<<" "<<m[0][1]<<" "<<m[0][2]<<endl;
  cout<<m[1][0]<<" "<<m[1][1]<<" "<<m[1][2]<<endl;
  cout<<m[2][0]<<" "<<m[2][1]<<" "<<m[2][2]<<endl;
  deta=m[0][0]*(m[1][1]*m[2][2]-m[1][2]*m[2][1])
  -m[0][1]*(m[1][0]*m[2][2]-m[1][2]*m[2][0])+m[0][2]
  *(m[1][0]*m[2][1]-m[1][1]*m[2][0]);

  mi[0][0]=(m[1][1]*m[2][2]-m[1][2]*m[2][1])/deta;
  mi[0][1]=(m[0][2]*m[2][1]-m[0][1]*m[2][2])/deta;
  mi[0][2]=(m[0][1]*m[1][2]-m[0][2]*m[1][1])/deta;

  mi[1][0]=(m[1][2]*m[2][0]-m[1][0]*m[2][2])/deta;
```

```
mi[1][1]=(m[0][0]*m[2][2]-m[0][2]*m[2][0])/deta;
mi[1][2]=(m[0][2]*m[2][2]-m[0][0]*m[1][2])/deta;

mi[2][0]=(m[1][0]*m[2][1]-m[1][1]*m[2][0])/deta;
mi[2][1]=(m[0][1]*m[2][1]-m[0][0]*m[2][1])/deta;
mi[2][2]=(m[1][1]*m[0][0]-m[0][1]*m[1][1])/deta;

cout<<"Matrix^(-1)"<<endl;
cout<<mi[0][0]<<" "<<mi[0][1]<<" "<<mi[0][2]<<endl;
cout<<mi[1][0]<<" "<<mi[1][1]<<" "<<mi[1][2]<<endl;
cout<<mi[2][0]<<" "<<mi[2][1]<<" "<<mi[2][2]<<endl;

}
catch (...){
  return (func(x,y,z));
}

double r[3];

r[0]=mi[0][0]*x+mi[0][1]*y+mi[0][2];
r[1]=mi[1][0]*x+mi[1][1]*y+mi[1][2];
r[2]=mi[2][0]*x+mi[2][1]*y+mi[2][2];
cout<<"r"<<endl;
cout<<r[0]<<endl;
cout<<r[1]<<endl;
cout<<r[2]<<endl;


double * f;
f = (double*)malloc(sizeof(double) * 3);


if((*subcube[tr1[0]][tr1[1]][(int)plane]).
calculated==false){
  (*subcube[tr1[0]][tr1[1]][(int)plane]).
  f1=func(tr1[0],tr1[1],plane)[0];
  (*subcube[tr1[0]][tr1[1]][(int)plane]).
  f2=func(tr1[0],tr1[1],plane)[1];
  (*subcube[tr1[0]][tr1[1]][(int)plane]).
  f3=func(tr1[0],tr1[1],plane)[2];
  (*subcube[tr1[0]][tr1[1]][(int)plane]).
  calculated=true;
  cout<<"point one missing"<<endl;
  cout<<"functionvalue "<<tr1[0]<<" "
    <<tr1[1]<<" "<<func(tr1[0],tr1[1],plane)[0]<<endl;
  cout<<"functionvalue "<<tr1[0]
    <<" "<<tr1[1]<<" "<<func(tr1[0],tr1[1],plane)[1]<<endl;
  cout<<"functionvalue "<<tr1[0]
    <<" "<<tr1[1]<<" "<<func(tr1[0],tr1[1],plane)[2]<<endl;
}

if((*subcube[tr2[0]][tr2[1]][(int)plane]).
calculated==false){
  (*subcube[tr2[0]][tr2[1]][(int)plane]).
  f1=func(tr2[0],tr2[1],plane)[0];
  (*subcube[tr2[0]][tr2[1]][(int)plane]).
  f2=func(tr2[0],tr2[1],plane)[1];
  (*subcube[tr2[0]][tr2[1]][(int)plane]).
  f3=func(tr2[0],tr2[1],plane)[2];
  (*subcube[tr2[0]][tr2[1]][(int)plane]).
  calculated=true;
  cout<<"triangle point two missing"<<endl;
  cout<<"functionvalue "<<tr2[0]
    <<" "<<tr2[1]<<" "
    <<func(tr2[0],tr2[1],plane)[0]<<endl;
  cout<<"functionvalue "<<tr2[0]
    <<" "<<tr2[1]<<" "
    <<func(tr2[0],tr2[1],plane)[1]<<endl;
  cout<<"functionvalue "<<tr2[0]
    <<" "<<tr2[1]<<" "
    <<func(tr2[0],tr2[1],plane)[2]<<endl;
}

if((*subcube[tr3[0]][tr3[1]][(int)plane]).
calculated==false){
  (*subcube[tr3[0]][tr3[1]][(int)plane]).
  f1=func(tr3[0],tr3[1],plane)[0];
  (*subcube[tr3[0]][tr3[1]][(int)plane]).
  f2=func(tr3[0],tr3[1],plane)[1];
  (*subcube[tr3[0]][tr3[1]][(int)plane]).
  f3=func(tr3[0],tr3[1],plane)[2];
  (*subcube[tr3[0]][tr3[1]][(int)plane]).
  calculated=true;
```

```
    cout<<"tirangle point three missing"<<endl;
    cout<<"functionvalue "<<tr3[0]
      <<" "<<tr3[1]<<" "
      <<func(tr3[0],tr3[1],plane)[0]<<endl;
    cout<<"functionvalue "<<tr3[0]
      <<" "<<tr3[1]<<" "
      <<func(tr3[0],tr3[1],plane)[1]<<endl;
    cout<<"functionvalue "<<tr3[0]
      <<" "<<tr3[1]<<" "
      <<func(tr3[0],tr3[1],plane)[2]<<endl;
  }


  f[0]=r[0]*(*subcube[tr1[0]][tr1[1]][(int)plane]).f1+r[1]*
  (*subcube[tr2[0]][tr2[1]][(int)plane]).f1+r[2]*
  (*subcube[tr3[0]][tr3[1]][(int)plane]).f1;
  f[1]=r[0]*(*subcube[tr1[0]][tr1[1]][(int)plane]).f2+r[1]*
  (*subcube[tr2[0]][tr2[1]][(int)plane]).f2+r[2]*
  (*subcube[tr3[0]][tr3[1]][(int)plane]).f2;
  f[2]=r[0]*(*subcube[tr1[0]][tr1[1]][(int)plane]).f3+r[1]*
  (*subcube[tr2[0]][tr2[1]][(int)plane]).f3+r[2]*
  (*subcube[tr3[0]][tr3[1]][(int)plane]).f3;

  return(f);
}

int * checkCube(CElement **** cube, double x, double y,
 double z, int M, int N){
  int dim=N*M;
  int * back;
  back = (int*)malloc(sizeof(int) * 4);
  int xc=0,yc=0,zc=0;
  if (x!=0)
    xc=(int)dim/x;
  if (y!=0)
    yc=(int)dim/y;
  if (z!=0)
    zc=(int)dim/z;
  cout<<xc<<" "<<yc<<" "<<zc<<endl;
  back[0]=(*cube[(int)xc/N][(int)yc/N][(int)zc/N]).v;
  back[1]=xc;
  back[2]=yc;
  back[3]=zc;
  return(back);
}
```

Appendix D

Source Code: Sequential Interpolation (Object Oriented Implementation)

```
//Interpol.h

//.. includes ...
#include "SubcubeQueue.h"

#ifdef use_namespace
using namespace NEWMAT;
#endif

using namespace std;

namespace MathModel {

  class Interpol{

    public:
      int a;
      double h;
      int nsc1;
      SeeOFuncObject *funcObj;
      SubcubeQueue *queu;

      Interpol(int a, double h,
         int nsc1, int queuesize, SeeOFuncObject *funcObj);

      ColumnVector Interpolate(Matrix x);

      ~Interpol();
  };
}

//Interpol.cpp

//... includes ...

#include "Interpol.h"
#include "Subcube.h"
#include "SubcubeQueue.h"

using namespace MathModel;

Interpol::Interpol(int a, double h, int nsc1,
  int queuesize, SeeOFuncObject *funcObj){

  this->a=a;
  this->h=h;
  this->nsc1=nsc1;
```

```
  this->funcObj=funcObj;
  Subcube * sc;
  sc=new Subcube(-2*a,-2*a,-2*a,h,(int)(2*a*(1/h))/nsc1,funcObj);
  queu=new SubcubeQueue(queuesize,sc);
}

ColumnVector Interpol::Interpolate(Matrix x){

  double mx,my,mz;
  int scx,scy,scz;
  Subcube * subc;
  Subcube * scsc;
  ColumnVector trp1,trp2,trp3;

  mx=x[0][0];
  my=x[0][1];
  mz=x[0][2];

  //cout<<mx<<” ”<<my<<” ”<<mz<<endl;
  if (fabs((double)a)>fabs(mx)){
    if (mx<0){
      scx=(int)(mx/(2*a/nsc1)-1)*(2*a/nsc1);
    }
    if (mx>=0){
      scx=(int)(mx/(2*a/nsc1))*(2*a/nsc1);
    }
    if (fabs((double)a)>fabs(my)){
      if (my<0){
        scy=(int)(my/(2*a/nsc1)-1)*(2*a/nsc1);
      }
      if (my>=0){
        scy=(int)(my/(2*a/nsc1))*(2*a/nsc1);
      }
      if (fabs((double)a)>fabs(mz)){
        if (mz<0){
          scz=(int)(mz/(2*a/nsc1)-1)*(2*a/nsc1);
        }
        if (mz>=0){
          scz=(int)(mz/(2*a/nsc1))*(2*a/nsc1);
        }
        cout<<scx<<”␣”<<scy<<”␣”<<scz<<endl;
      }
      else{
        cout<<”z␣out␣of␣borders”<<endl;
        return(MatrixToColumnVector (this->funcObj->Evaluate(x)));
      }
    }
    else{
      cout<<”y␣out␣of␣borders”<<endl;
      return(MatrixToColumnVector (this->funcObj->Evaluate(x)));
    }
  }
  else{
    cout<<”x␣out␣of␣borders”<<endl;
    return(MatrixToColumnVector (this->funcObj->Evaluate(x)));
  }
  cout<<”interpol␣initiating”<<endl;
  if(queu->CheckSubcube(scx,scy,scz)){
    scsc=queu->GetSubcube(scx,scy,scz);
    scsc=new Subcube(scsc);
    trp1=scsc->GetTrp1(MatrixToColumnVector(x));
    trp2=scsc->GetTrp2(MatrixToColumnVector(x));
    trp3=scsc->GetTrp3(MatrixToColumnVector(x));
  }
  else{
    cout<<”generating␣new␣subcube”<<endl;
    subc=new Subcube(scx,scy,scz,h,(int)(2*a*(1/h))/nsc1,funcObj);
    subc->CreateSubcube((int)(2*a*(1/h))/nsc1);
    queu->AddCube(subc);
    trp1=subc->GetTrp1(MatrixToColumnVector(x));
    trp2=subc->GetTrp2(MatrixToColumnVector(x));
    trp3=subc->GetTrp3(MatrixToColumnVector(x));

  }
  //cout<<(int)(2*a*(1/h))/nsc1<<endl;
  //cout<<trp1[6]<<trp2[6]<<trp3[6]<<endl;
  if ((trp1[6]==1)&&(trp2[6]==1)&&(trp3[6]==1)){
    Matrix m(3,3);
    m[0][0] = trp1[3]; m[0][1] = trp2[3]; m[0][2] = trp3[3];
    m[1][0] = trp1[4]; m[1][1] = trp2[4]; m[1][2] = trp3[4];
    m[2][0] =       1; m[2][1] =       1; m[2][2] =       1;
    //cout<<”~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~”<<endl;
    //cout<<m<<endl;
```

```
      //cout<<"~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~"<<endl;
      try{
        m = m.i();
      }
      catch (...){
        cout<<"interpolate:␣matrix␣inversion␣error"<<endl;
        return(MatrixToColumnVector (this->funcObj->Evaluate(x)));
      }
      cout<<"matrix␣inversion␣successful"<<endl;
      cout<<"interpolation␣started"<<endl;
      Matrix n (3,1);
      n[0][0] = x[0][0];
      n[1][0] = x[0][1];
      n[2][0] =       1;
      Matrix r (3,1);
      r= m * n;
      cout<<r<<endl;
      ColumnVector f(3);
      cout<<r[0][0]*trp1[0]+r[1][0]*trp2[0]+r[2][0]*trp3[0]<<endl;
      f[0]=r[0][0]*trp1[0]+r[1][0]*trp2[0]+r[2][0]*trp3[0];
      f[1]=r[0][0]*trp1[1]+r[1][0]*trp2[1]+r[2][0]*trp3[1];
      f[2]=r[0][0]*trp1[2]+r[1][0]*trp2[2]+r[2][0]*trp3[2];
      cout<<"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"<<endl;
      cout<<"interpolated"<<endl<<f<<endl;
      cout<<"original"<<endl<<MatrixToColumnVector (this->funcObj->Evaluate(x))<<endl;
      cout<<"␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣"<<endl;
      return (f);
    }
    else{
      return(MatrixToColumnVector (this->funcObj->Evaluate(x)));
    }
}

Interpol::~Interpol(){
  delete(queu);
  delete(funcObj);
}

//Subcube.h

//... includes ...

#ifdef use_namespace
using namespace NEWMAT;
#endif

using namespace std;

namespace MathModel {

  class Subcube{

    public:
      int ax;
      int ay;
      int az;
      int dim;
      double h;
      SeeOFuncObject *funcObj;

    private:
      typedef struct element{
        double f1,f2,f3;
      } Element;
      Element **** sc;

    public:
      Subcube::Subcube();
      Subcube::Subcube(Subcube * s);
      Subcube::Subcube(int ax, int ay, int az, double h, int dim, SeeOFuncObject *funcObj);

      void Subcube::CreateSubcube(int N);
      ColumnVector Subcube::GetTrp1(ColumnVector xg);
      ColumnVector Subcube::GetTrp2(ColumnVector xg);
      ColumnVector Subcube::GetTrp3(ColumnVector xg);

      Subcube::~Subcube();
  };
}
#endif

//Subcube.cpp
```

```
//... includes ...

#include "Interpol.h"
#include "Subcube.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

using namespace MathModel;

Subcube::Subcube(){

}

Subcube::Subcube(Subcube * s){

  this->ax=s->ax;
  this->ay=s->ay;
  this->az=s->az;
  this->h=s->h;
  this->dim=s->dim;
  this->funcObj=s->funcObj;
  this->sc=s->sc;
}

Subcube::Subcube(int ax, int ay, int az,
   double h, int dim, SeeOFuncObject *funcObj){

  this->ax=ax;
  this->ay=ay;
  this->az=az;
  this->h=h;
  this->dim=dim;
  this->funcObj=funcObj;
}

void Subcube::CreateSubcube(int N)  {

  cout<<""<<N<<endl;
  int i,j,l;
  Element **** subcube;
  Matrix x(1,3);
  ColumnVector tmp;
  subcube = (Element ****)malloc(sizeof(Element****) * N);
  for (i=0; i<N; i++){
    subcube[i] = (Element ***)malloc(sizeof(Element ***) *N);
    for (j=0; j<N; j++){
      subcube[i][j] = (Element **)malloc(sizeof(Element**) * N);
      for (l=0; l<N; l++){
        subcube[i][j][l] = (Element*)malloc(sizeof(ColumnVector));
      }
    }
  }
  cout<<"subcube created"<<endl;
  for(i=0;i<N;i++){
    for(j=0;j<N;j++){
      for(l=0;l<N;l++){
        //cout<<i<<" "<<j<<" "<<l<<endl;
        //cout<<(double)i*h+a<<endl;
        x[0][0]=(double)ax+i*h;
        x[0][1]=(double)ay+j*h;
        x[0][2]=(double)az+l*h;
        tmp=MatrixToColumnVector (this->funcObj->Evaluate(x));
        //cout<<"torque "<<i<<j<<l<<endl;
        (*subcube[i][j][l]).f1=tmp[0];
        (*subcube[i][j][l]).f2=tmp[1];
        (*subcube[i][j][l]).f3=tmp[2];
      }
    }
  }
  this->sc=subcube;
  cout<<"subcube filled"<<endl;
}

ColumnVector Subcube::GetTrp1(ColumnVector xg){

  double x,y,z;
  double dx,dy,dz;
  int xi,yi,zi;
  ColumnVector p(7);
```

```
p[6]=1;
cout<<"gettrp1"<<endl;
x=xg[0];
y=xg[1];
z=xg[2];

if (x<0){
   xi=(int)(x+ax)/h+dim;
   dx=(xi-dim)*h-ax;
   if((dx-x)>=h){
      xi--;
   }
   if((dx-x)>=(h/2)){
      xi--;
   }
   dx=(xi-dim)*h-ax;
}
else{
   xi=(int)(x-ax)/h;
   dx=xi*h+ax;
   if((x-dx)>=h){
      xi++;
   }
   if((x-dx)>=(h/2)){
      xi++;
   }
   dx=xi*h+ax;
}
if (y<0){
   yi=(int)(y+ay)/h+dim;
   dy=(yi-dim)*h-ay;
   if((dy-y)>=(h)){
      yi--;
   }
   if((dy-y)>=(h/2)){
      yi--;
   }
   dy=(yi-dim)*h-ay;
}
else{
   yi=(int)(y-ay)/h;
   dy=yi*h+ay;
   if((y-dy)>=(h)){
      yi++;
   }
   if((y-dy)>=(h/2)){
      yi++;
   }
   dy=yi*h+ay;
}
if (z<0){
   zi=(int)(z+az)/h+dim;
   dz=(zi-dim)*h-az;
   if((dz-z)>=(h)){
      zi--;
   }
   if((dz-z)>=(h/2)){
      zi--;
   }
   dz=(zi-dim)*h-az;
}
else{
   zi=(int)(z-az)/h;
   dz=zi*h+az;
   if((z-dz)>=(h)){
      zi++;
   }
   if((z-dz)>=(h/2)){
      zi++;
   }
   dz=zi*h+az;
}
if((xi>dim-1)||(yi>dim-1)||(zi>dim-1)||(xi<0)||(yi<0)||(zi<0)){
   p[6]=0;
   cout<<"trp1 dimension error "<<endl;
   return p;
}
cout<<"trpcoord 1 "<<xi<<" "<<yi<<" "<<zi<<endl;
p[0]=(*sc[xi][yi][zi]).f1;
p[1]=(*sc[xi][yi][zi]).f2;
p[2]=(*sc[xi][yi][zi]).f3;
```

```
  p[3]=dx;
  p[4]=dy;
  p[5]=dz;
  return p;
}

ColumnVector Subcube::GetTrp2(ColumnVector xg){

  double x,y,z;
  double dx,dy,dz;
  int xi,yi,zi;
  ColumnVector p(7);

  p[6]=1;
  cout<<"gettrp2"<<endl;
  x=xg[0];
  y=xg[1];
  z=xg[2];

  if (x<0){
    xi=(int)(x+ax)/h+dim;
    dx=(xi-dim)*h-ax;
    if((dx-x)>=(h)){
      xi--;
    }
    if((dx-x)>=(h/2)){
      xi--;
    }
    dx=(xi-dim)*h-ax;
  }
  else{
    xi=(int)(x-ax)/h;
    dx=xi*h+ax;
    if((x-dx)>=(h)){
      xi++;
    }
    if((x-dx)>=(h/2)){
      xi++;
    }
    dx=xi*h+ax;
  }
  if (y<0){
    yi=(int)(y+ay)/h+dim;
    dy=(yi-dim)*h-ay;
    if((dy-y)>=(h)){
      yi--;
    }
    if((dy-y)>=(h/2)){
      yi--;
    }
    dy=(yi-dim)*h-ay;
  }
  else{
    yi=(int)(y-ay)/h;
    dy=yi*h+ay;
    if((y-dy)>=(h)){
      yi++;
    }
    if((y-dy)>=(h/2)){
      yi++;
    }
    dy=yi*h+ay;
  }
  if (z<0){
    zi=(int)(z+az)/h+dim;
    dz=(zi-dim)*h-az;
    if((dz-z)>=(h)){
      zi--;
    }
    if((dz-z)>=(h/2)){
      zi--;
    }
    dz=(zi-dim)*h-az;
  }
  else{
    zi=(int)(z-az)/h;
    dz=zi*h+az;
    if((z-dz)>=(h)){
      zi++;
    }
    if((z-dz)>=(h/2)){
      zi++;
    }
```

```
      dz=zi*h+az;
    }
    //I
    if((x>0)&&(y>0)&&(x>=y)){
      xi++;
      dx=xi*h+ax;
      cout<<"I+"<<endl;
    }
    if((x>0)&&(y>0)&&(x<y)){
      yi++;
      dy=yi*h+ay;
      cout<<"I-"<<endl;
    }
    //II
    if((x<0)&&(y>0)&&(-x>=y)){
      xi--;
      dx=(xi-dim)*h-ax;
      cout<<"II+"<<endl;
    }
    if((x<0)&&(y>0)&&(-x<y)){
      yi++;
      dy=yi*h+ay;
      cout<<"II-"<<endl;
    }
    //III
    if((x<0)&&(y<0)&&(-x>=-y)){
      xi--;
      dx=(xi-dim)*h-ax;
      cout<<"III+"<<endl;
    }
    if((x<0)&&(y<0)&&(-x<-y)){
      yi--;
      yi=(int)(y+ay)/h+dim;
      cout<<"II-+"<<endl;
    }
    //IV
    if((x>0)&&(y<0)&&(x>=-y)){
      xi++;
      dx=xi*h+ax;
      cout<<"IV+"<<endl;
    }
    if((x>0)&&(y<0)&&(x<-y)){
      yi--;
      yi=(int)(y+ay)/h+dim;
      cout<<"IV-"<<endl;
    }
    if((xi>dim-1)||(yi>dim-1)||(zi>dim-1)||(xi<0)||(yi<0)||(zi<0)){
      p[6]=0;
      cout<<"trp2 dimension error "<<endl;
      return p;
    }
    cout<<"trpcoord 2 "<<xi<<" "<<yi<<" "<<zi<<endl;
    p[0]=(*sc[xi][yi][zi]).f1;
    p[1]=(*sc[xi][yi][zi]).f2;
    p[2]=(*sc[xi][yi][zi]).f3;

    p[3]=dx;
    p[4]=dy;
    p[5]=dz;
    //cout<<"Trianglepoint 2: "<<endl<<p<<endl;
    return p;
  }

  ColumnVector Subcube::GetTrp3(ColumnVector xg){

    double x,y,z;
    double dx,dy,dz;
    int xi,yi,zi;
    ColumnVector p(7);

    p[6]=1;
    cout<<"gettrp3"<<endl;
    x=xg[0];
    y=xg[1];
    z=xg[2];

    if (x<0){
      xi=(int)(x+ax)/h+dim;
      dx=(xi-dim)*h-ax;
      if((dx-x)>=(h)){
        xi--;
      }
      if((dx-x)>=(h/2)){
```

```
        xi--;
      }
      xi--;
      dx=(xi-dim)*h-ax;
    }
    else{
      xi=(int)(x-ax)/h;
      dx=xi*h+ax;
      if((x-dx)>=(h)){
        xi++;
      }
      if((x-dx)>=(h/2)){
        xi++;
      }
      xi++;
      dx=xi*h+ax;
    }
    if (y<0){
      yi=(int)(y+ay)/h+dim;
      dy=(yi-dim)*h-ay;
      if((dy-y)>=(h)){
        yi--;
      }
      if((dy-y)>=(h/2)){
        yi--;
      }
      yi--;
      dy=(yi-dim)*h-ay;
    }
    else{
      yi=(int)(y-ay)/h;
      dy=yi*h+ay;
      if((y-dy)>=(h)){
        yi++;
      }
      if((y-dy)>=(h/2)){
        yi++;
      }
      yi++;
      dy=yi*h+ay;
    }
    if (z<0){
      zi=(int)(z+az)/h+dim;
      dz=(zi-dim)*h-az;
      if((dz-z)>=(h)){
        zi--;
      }
      if((dz-z)>=(h/2)){
        zi--;
      }
      dz=(zi-dim)*h-az;
    }
    else{
      zi=(int)(z-az)/h;
      dz=zi*h+az;
      if((z-dz)>=(h)){
        zi++;
      }
      if((z-dz)>=(h/2)){
        zi++;
      }
      dz=zi*h+az;
    }
    if((xi>dim-1)||(yi>dim-1)||(zi>dim-1)||(xi<0)||(yi<0)||(zi<0)){
      p[6]=0;
      cout<<"trp3 dimension error "<<endl;
      return p;
    }
    cout<<"trpcoord 3 "<<xi<<" "<<yi<<" "<<zi<<endl;
    p[0]=(*sc[xi][yi][zi]).f1;
    p[1]=(*sc[xi][yi][zi]).f2;
    p[2]=(*sc[xi][yi][zi]).f3;

    p[3]=dx;
    p[4]=dy;
    p[5]=dz;

    //cout<<"Trianglepoint 3: "<<endl<<p<<endl;
    return p;
  }

  Subcube::~Subcube(){
```

```
}

//SubcubeQueue.h

//... includes ...

#include "Subcube.h"

#ifdef use_namespace
using namespace NEWMAT;
#endif

using namespace std;

namespace MathModel {

  typedef vector<Subcube> SubcubeVector;

  class SubcubeQueue{

    public:
      unsigned int numOfsc;
      SubcubeVector qu;

      SubcubeQueue::SubcubeQueue(int numsc, Subcube* sc);

      void SubcubeQueue::AddCube(Subcube* sc);
      Subcube * SubcubeQueue::GetSubcube(int x, int y, int z);
      bool SubcubeQueue::CheckSubcube(int x, int y, int z);

      SubcubeQueue::~SubcubeQueue();
  };
}
#endif

//SubcubeQueue.cpp

//... includes ...

#include "Interpol.h"
#include "Subcube.h"
#include "SubcubeQueue.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

using namespace MathModel;

SubcubeQueue::SubcubeQueue(int numsc, Subcube *sc){

  numOfsc=numsc;
  qu.assign(numsc,*sc);
}

void SubcubeQueue::AddCube(Subcube * sc){

  for(int i=0;i<numOfsc-1;i++){
    qu.at(i)=qu.at(i+1);
  }
  qu.pop_back();
  qu.push_back(*sc);
}

Subcube *SubcubeQueue::GetSubcube(int x, int y, int z){

  bool flag=true;
  int i=0;

  Subcube s;
  Subcube *returnsc;
  while(flag){
    s=qu.at(i);
    if((x==s.ax)&&(y==s.ay)&&(z==s.az)){
      flag=false;
      returnsc=&s;
      cout<<"getsubcube "<<returnsc->ax<<" "<<returnsc->ay<<" "<<returnsc->az<<endl;
      return returnsc;
    }
    i++;
  }
```

```
  returnsc=&s;
  return returnsc;
}

bool SubcubeQueue::CheckSubcube(int x, int y, int z){

  bool flag=false;
  int i=0;
  Subcube* s;

  cout<<"checksubcube 1 "<<qu.size()<<endl;
  while((!flag)&&(i<(int)qu.size())){
    s=&qu.at(i);
    if((x==s->ax)&&(y==s->ay)&&(z==s->az)){
      flag=true;
    cout<<"checksubcube 4"<<endl;
    }
    i++;
  }
  cout<<"checksubcube 5"<<endl;
  return flag;
}

SubcubeQueue::~SubcubeQueue(){

}
```

Appendix E

Source Code: Parallel Subcube Computation

```
struct msg_s {
  double m1,m2,m3;
  sem_t sem,semm;
};

struct msg_s *shared_msg;

int shmfd;
int pid;
char memadr[9];
int shared_seg_size;

int Subcube::SetMemAndFork(){
  if (shm_unlink(memadr) != 0) {
    perror("In shm_unlink()");
  }
  close(shmfd);
  shared_seg_size = (1 * sizeof(struct msg_s));   // want shared segment capable of storing 1 message
  // the shared segment, and head of the messages list
// creating the shared memory object    --  shm_open()
  shm_unlink(memadr);
  shmfd = shm_open(memadr0, O_CREAT | O_EXCL | O_RDWR, S_IRWXU | S_IRWXG);
  if (shmfd < 0) {
    perror(" In shm_open()");
  }
  //fprintf(stderr, "Created shared memory object %s\n", memadr);
  ftruncate(shmfd, shared_seg_size);
  shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED, shmfd, 0);
  if (shared_msg == NULL) {
    perror("In mmap()");
  }
  sem_init(&shared_msg->sem, 1, 0);
  sem_init(&shared_msg->semm, 1, 0);

  pid = fork();
  if( pid < 0 ) {
    //fprintf( stderr, "Could not create a new process.\n" );
    exit( -1 );
  }
  if( pid == 0 ) {

    ColumnVector tmp;
    Matrix x(3,1);
    while(1){

      sem_wait(&shared_msg->sem);
      sem_init(&shared_msg->sem, 1, 0);
      x[0][0]=shared_msg->m1;
      x[0][1]=shared_msg->m2;
      x[0][2]=shared_msg->m3;
      tmp=MatrixToColumnVector (funcObj->Evaluate(x));
```

```
        //cout<<"calculating"<<endl;
        shared_msg->m1=tmp[0];
        shared_msg->m2=tmp[1];
        shared_msg->m3=tmp[2];
        //cout<<"values written..."<<endl;
        sem_post(&shared_msg->semm);
      }

    }
    if (pid!=0){
      return(pid);
    }
    return(-1);
}

void Subcube::UnsetMemAndKill(){
    int status;
    if (shm_unlink(memadr) != 0) {
            perror("In_shm_unlink()");
    }
    close(shmfd);
    if( pid != 0 ) {
      kill((pid_t)pid,SIGKILL);
      wait( &status );
    }
}
double * Subcube::myEvaluate(double x1, double x2, double x3){

    double *back =new double[3];
    double *tmp =new double[3];

    // This is the parent process
    if( pid != 0 ) {
      sem_init(&shared_msg->semm, 1, 0);
      shared_msg->m1=x1;
      shared_msg->m2=x2;
      shared_msg->m3=x3;

      sem_post(&shared_msg->sem);
      sem_wait(&shared_msg->semm);

      back[0]=shared_msg->m1;
      back[1]=shared_msg->m2;
      back[2]=shared_msg->m3;
      return(back);
    }
    return(back);

}
```

Bibliography

[1] Austrian Grid. Website: http://www.austriangrid.at/, 2008.

[2] M. Al-Baali. Variational Quasi-Newton Methods for Unconstrained Optimization. *J. Optim. Theory Appl.*, 77(1):127–143, 1993.

[3] Walter Alt. *Nichtlinare Optimierung.* vieweg, Wiesbaden, 2002.

[4] Blaise Barney. OpenMP. Website: https://computing.llnl.gov/tutorials/openMP/, 2008.

[5] Blaise Barney. POSIX Threads Programming. Website: https://computing.llnl.gov/tutorials/pthreads/, 2008.

[6] Karoly Bosa, Wolfgang Schreiner, Michael Buchberger, and Thomas Kaltofen. SEE-GRID, A Grid-Based Medical Decision Support System for Eye Muscle Surgery. In *Proceedings of 1st Austrian Grid Symposium*, 2005.

[7] Michael Buchberger. *Biomechanical Modelling of the Human Eye.* PhD thesis, Johannes Kepler Universität Linz, March 2004. http://www.see-kid.at/download/Dissertation_MB.pdf.

[8] Michael Buchberger and Thomas Kaltofen. SEE-KID Homepage. Website: http://www.see-kid.at, 2008.

[9] Michael Buchberger and Sigfried Priglinger. *Augenmotilitätsstörungen. Computerunterstütze Diagnose und Therapie.* Springer, Wien, December 2004.

[10] Michael Buchberger and Sigfried Priglinger. *Eye Motility Disorders.* Springer, Wien, To appear. September 2008.

[11] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: Portable -Shared Memory Parallel Programming (Scientific*

and Engineering Computation). MIT Press, Cambridge, MA, USA, October 2007.

[12] C.T.Kelley. *Iterative Methods for Linear and Nonlinear Equations*. Society for Industrial and Applied Mathematics, 1995.

[13] C.T.Kelley. *Iterative Methods for Optimization*. Society for Industrial and Applied Mathematics, 1999.

[14] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, 1998.

[15] Mirela Damian. Synchronizing Threads with POSIX Semaphores. Website: http://www.csc.villanova.edu/ mdamian/threads/posixsem.html, 2006.

[16] Robert Davies. Newmat C++ matrix library Version 10. Website: http://www.robertnz.net/nm_intro.htm, April 2006.

[17] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer, Berlin, 2000.

[18] Josh Simons et al. The OpenMP API specification for parallel programming. Website: http://www.openmp.org, 2008.

[19] Porrill J., Warren PA, and Dean P. A Simple Control Law Generates Listing's Positions in a Detailed Model of the Extraocular Muscle System. *Vision Research, 40(27):3743-58*, 2000.

[20] Werner Platzer. *Pernkopf: Atlas der topographischen und angewandten Anatomie des Menschen, 3. Auflage*. Band 1. Urban & Schwarzenberg, München-Wien-Baltimore, 1987.

[21] Johannes Plötner and Steffen Wendzel. *Linux, Das distributionsunabhängige Handbuch*. Galileo Computing, 2006.

[22] *System Application Program Interface (API) [C Language]*. Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1990.

[23] *Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities (Volume 1)*. Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1993.

[24] *Portable Operating System Interface (POSIX)—Part 2: Shell and Utilities (Volume 2).* Information technology—Portable Operating System Interface (POSIX). IEEE Computer Society, 345 E. 47th St, New York, NY 10017, USA, 1993.

[25] Sjaak Priester. Delaunay Triangles, July 2005. Website: http://www.codeguru.com/Cpp/data/mfc_database/misc/article.php/c8901/.

[26] Wolfgang Schreiner and Karoly Bosa. The Porting of a Medical Grid Application from Globus 4 to the gLite Middleware. In *DAPSYS 2008, 7th International Conference on Distributed and Parallel Systems*, Hungary, September 2008. Lecture Notes in Computer Science, Springer, 11 pages.

[27] Wolfgang Schreiner, Karoly Bosa, Michael Buchberger, and Thomas Kaltofen. A Grid Software for Virtual Eye Surgery Based on Globus and gLite. In *ISPDC 2007, 6th International Symposium on Parallel and Distributed Computing*, pages 151–158, Austria, July 2007. IEEE Computer Society, Los Alamitos, CA.

[28] SGI. SGI Altix 4700, Deliverin New Levels of Performance and Flexibility. Website: http://www.sgi.com/products/servers/altix/4000/index.html, 2008.

[29] w. Richard Stevens. *UNIX Network Programming, Second Edition: Interprocess Communications*, volume 2. Prentice Hall, 1999. http://www.kohala.com/start/unpv22e/unpv22e.html.

[30] Johannes Watzl. Investigations on Improving the SEE-GRID Optimization Algorithm by Parallelization. Technical report, UAR, RISC, 2006.

[31] Johannes Watzl. Investigations on Improving the SEE-GRID Optimization Algorithm by Parallelization, Status update. Technical report, UAR, RISC, 2006.

[32] Peter L. Williams. *Gray's Anatomy, The Anatomical Basis of Medicine and Surgery.* 38th Edition. Churchill Livingstone, Copyright Elsevier, 38th edition, 1995.