# AUSTRIAN GRID

## INVESTIGATIONS ON IMPROVING THE SEE-GRID OPTIMIZATION ALGORITHM BY PARALLELIZATION

| | |
|---|---|
| Document Identifier: | **AG-DA-1c-3-2006.pdf** |
| Status: | **Public** |
| Workpackage: | **A1c** |
| Partner(s): | **Research Institute for Symbolic Computation (RISC)**<br>**Upper Austrian Research (UAR)** |
| Lead Partner: | **UAR** |
| WP Leaders: | **Wolfgang Schreiner (RISC), Michael Buchberger (UAR)** |

## Delivery Slip

| | Name | Partner | Date | Signature |
|---|---|---|---|---|
| **From** | Johannes Watzl | UAR | 2006.07.28 | |
| **Verified by** | | | | |
| **Approved by** | | | | |

## Document Log

| Version | Date | Summary of changes | Author |
|---|---|---|---|
| 1.0 | 2006.07.28 | Initial version | Johannes Watzl |
| | | | |
| | | | |
| | | | |

# Investigations on Improving the SEE-GRID Optimization Algorithm by Parallelization

Johannes Watzl

2006

**Abstract**

This report deals with possible improvements of the current implementation of the optimization algorithm in the SEE-GRID project. First the present algorithm is analysed and benchmarked. Then we initiate both sequential and parallel approaches for accelerating the computation. The sequential approach is done by the *Broyden update method*; the parallel strategies work on the one hand with parallel *Delaunay triangulation* for interpolating the function to minimize and on the other hand with decoupling optimization from triangulation. The interpolation is chosen because the function we have to minimize has to be evaluated thousands of times which takes more than half of the computation time.

## Contents

## List of Figures

# 1 Introduction

The SEE++ software system helps doctors to evaluate certain diseases concerning eye motility disorders [3]. SEE-GRID is an extension of SEE-KID using parallelization and Grid capabilities [2]. The main computation in the SEE-KID/SEE-GRID software system is the optimization of a specific function, called the Torque function. The minimum of this function represents a stable eye position which is needed for simulating the Hess-Lancaster test.

This computation lasts a few seconds for calculating one stable eye position. Every time the user changes some parameters, the software has to compute a new minimum.

The goal of our work is to find strategies for improving and accelerating this computation procedure using the capability of parallel systems. First we look at the existing algorithm and implementation. Then, we discuss the possibilities for improvements especially for (but not restricted to) parallelization. Afterwards, we introduce the *Broyden update method* (or BFGS update formula, see [1]) which can be used to accelerate optimization algorithms working with Jacobian or Hessian matrices.

Finally, we concentrate on the Torque function itself by searching for an appropriate interpolation method. We triangulate the Torque function with the *Delaunay triangulation algorithm* ([4]) and then compute every function value needed for the optimization with a certain interpolation procedure. We describe three strategies how the triangulation can be done in parallel.

In our further work we will focus on different strategies for parallel interpolation where the optimization algorithm is decoupled from the triangulation. This gives us the capability of two concurrent processes doing their work nearly independently with very few communication needed.

In Section 2.1 we describe the minimization problem our work is based on. Then we give a basic overview of optimization techniques in Section 2.2. In Section 2.3, the Levenberg-Marquardt algorithm, which is used for the minimization in the existing implementation, is specified. Section 2.4 deals with benchmarks of the existing algorithm. Section 2.5 describes our approaches for accelerating the existing algorithm and searching for possible parallelization strategies.

The Broyden update is initiated in Section 3.1. A description of our present prototype implementation is given in section 3.2. The benchmarks of this current update.

At last we describe possibilities for parallelization in Section 4. We start with the basic idea (Section 4.1) followed by the description of the Delaunay triangulation (Section 4.2). Finally, we depict the potential of combining the triangulation and optimization for the purpose of parallelization with three possible strategies: bruteforce, realistic and smart strategy (Section 4.3).

# 2 Existing Algorithm

At first we describe the problem and the present solution strategy together with a short overview of nonlinear optimization. Then we give benchmarks of the existing implementation and discuss feasible improvements.

## 2.1 Specification of the Problem

The main task of the computation is to minimize the so called Torque function ($L_T : \mathbb{R}^n \to \mathbb{R}$, $n = 6 + 3$) which determines the eye position. The Torque function gets as input a vector of real numbers. These input values are given as two vectors: $\overrightarrow{I_v}$ has six elements each describing the innervation of an eye muscle. $\overrightarrow{E_p}$ including three elements denotes the eye position based on rotation vectors. For a detailed specification of the Torque function see Chapter 4 in [3], where one can find the whole description of the biomechanical modeling. The minimum of the Torque function determines a *stable eye position.* This position has to be found in order to e.g. simulate the Hess Lancaster test which is the basis for the simulation of eye motility disorders. So we have to solve a nonlinear least-squares minimization (optimization) problem, i.e. to compute the following formula:

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

## 2.2 Nonlinear Optimization

First we introduce the basics of nonlinear optimization. The general structure of an optimization algorithm can be seen in Algorithm 2.1. Basically 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. This 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 our function to find the minimum. The classical method for solving such problems is the *Newton method* [1]. Here the iteration is done by

$$x^{k+1} = x^k - {G^k}^{-1} g^k$$

where

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

```
Algorithm 2.1: OPTIMIZATION($f, startingvalue$)

 Input: $f : \mathbb{R}^n \to \mathbb{R}, x^1$
 Output: minimum $f(x^k)$

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

denotes the gradient of f 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. This means that the number of correct decimal places doubles in every iteration step. An improvement of the Newton method is the *Gauss-Newton method* [1] where the Hessian matrix is approximated by the Jacobian matrix (here denoted as $J$). The Jacobian matrix contains only the first derivatives.

$$G^k = J^{kT} J^k$$

This approximation is also used in the Levenberg-Marquardt algorithm explained in the next section. For further information on the Gauss-Newton method see [1].

## 2.3 Levenberg-Marquardt Algorithm

The Levenberg-Marquardt algorithm 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 [5]. The MatLab-code was transcripted into C++ as part of the SEE-KID project.

The algorithm is a combination of the Gauss-Newton and a Trust-Region method. A Trust-Region is a region around a point with a certain radius which one has to determine. Here we can compute the search direction with a local modelfunction and then determine the global minimum (see [1]). In the Levenberg-Marquardt algorithm the search direction is computed by

$$(J^{k^T} J^k + \lambda^k I)d^k = -J^{k^T} F^k$$

where $\lambda^k$ is a scalar for modifying the search direction and $d^k$ denotes the search direction. If $\lambda^k = 0$, the search direction is the same as in the Gauss-Newton method and if $\lambda^k \to \infty$, $d^k$ is nearly a steepest descent direction.

The Levenberg-Marquardt algorithm is very robust and the convergence is nearly quadratic like in the Newton method.

## 2.4 Timing the Existing Implementation

We benchmarked the implementation of the Levenberg-Marquardt algorithm that is currently used in the SEE-KID/SEE-GRID project. Furthermore, we have measured the execution times of some interesting computations. The benchmarks were performed on a PC with 1.4 GHz P4 processor and 512 MB of RAM. The whole minimization process took 6.5 s to 8 s.

In the benchmarks, we have also computed

- the time for the evaluation of the Torque function and
- the number of evaluations of this function.

The average computation time for one evaluation of the Torque function is 0.598 ms. This evaluation has to be done more than 8000 times, so one can see that this is the main part of computation time.

Finally, we have computed the average computation time of basic matrix and vector operations used in every step of the Levenberg-Marquardt algorithm. These timings can be seen in the following table.

| Operation | ms |
|---|---|
| Matrix multiplication | 0.014 |
| Matrix inversion | 0.015 |
| Matrix-Vector Multiplication | 0.006 |

## 2.5 Discussion of Possible Improvements

One approach to speedup the computation of the sequential algorithm would be some fine grained parallelization of matrix and vector operations. But due to the very small dimensions of the matrices and vectors, the overhead for creating concurrent threads or distributing the data is much too high, as one can see from the timings in the previous section. Thus we have to investigate alternative approaches.

For the acceleration of the sequential algorithm, one can try the Broyden update method (see Section 3.1) where the Jacobian matrix has not to be fully computed in every step.

Because the evaluations of the Torque function take more than half of the computation time, we can think of interpolating the Torque function in a certain way. Here one can look for ways to parallelize this interpolation (see Section 4.2).

# 3 Broyden Update

An approach to accelerate the sequential optimization algorithm without using the benefits of parallelization is the *Broyden update method.* Here we introduce this method and delineate our current work concerning this sequential improvement.

## 3.1 Basic Algorithm

One possible approach improving the optimization algorithm is the so called Broyden update. In the Levenberg-Marquardt optimization algorithm the Jacobian matrix has to be computed in every step and needs a lot of time. To remedy this computation we can use the Broyden update. The Broyden update starts with an initial Jacobian matrix (often the identity matrix is taken as initial matrix) and updates this matrix a certain number of (usually three or four) times. After these updates a "restart" has to be performed, which means that the exact Jacobian has to be computed again and one optimization step with this matrix has to be done. After that we can update the just now computed matrix again three or four 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.

With the Broyden update we have the ability to update the inverse Hessian matrix too using another update formula. In the Levenberg-Marquardt algorithm the Hessian matrix is computed by using the Jacobian matrix. One has to both check if these two methods are suitable for our problem and which one is the better one.

## 3.2 Prototype Implementation

We have developed a prototype implementation of the basic Broyden update. In this implementation we use an update counter initially set so 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.

## 3.3 Benchmarks and Discussion

Our implementation has to be improved because the matrix can be updated only once now. This is not very effective and makes the new implementation slower than the existing algorithm, but due to some not detected implementation problems the reason for this is unclear and needs further investigations.

# 4 Parallelization by Delaunay Triangulation

The most promising approach concerning both timing and parallelization is the *Delaunay triangulation*. We give a detailed insight into the idea of the Delaunay triangulation and describe the strategies for parallelization developed during our work up to now.

## 4.1 Basic Idea

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.

The interpolation we use in our project works by the triangulation of the function in a special domain. As domain of the triangulation we can use the whole domain of the Torque function, but this is not very efficient. Therefore, we have to switch to better strategies, which are discussed later. When triangulating the function, we get a plane 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 [4]. The Delaunay algorithm is used to model 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. Then we have the following possibilities for parallelization:

- We can compute the function values in parallel. First we have to select a certain number of $(x, y)$ sample points and determine the corresponding $z$ value by evaluating the Torque function in this points.
- The triangulation of the set of points can be done in parallel.
- During the optimization we can compute the triangulation of subsets of the domain of the Torque function in parallel.

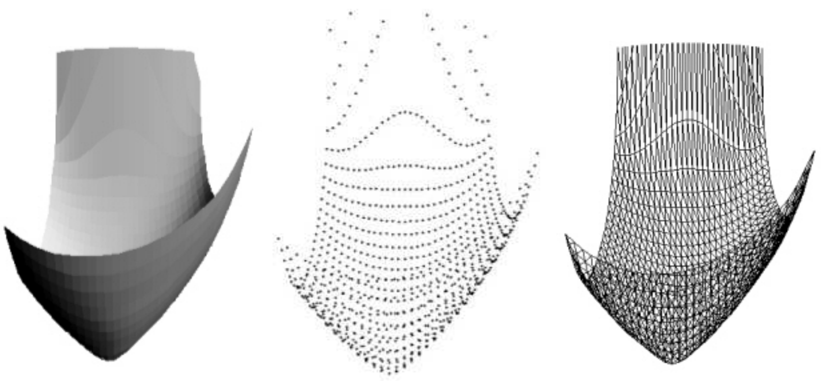

Figure 1: The Torque Function (original - set of points - triangulated)

Figure 1 represents the Torque function for a certain pathology. In this example 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 Delaunay Triangulation

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 computation of this set can be done by multiple processes. The output of the Delaunay algorithm delivers us a triangulation $T$ of the points given before. The first step doing the Delaunay triangulation is to create a triangle containing all points of the set $P$.

The next steps 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 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 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). This is done by

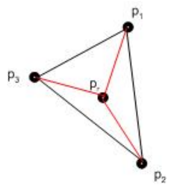

Figure 2: Inserting a point located in a triangle

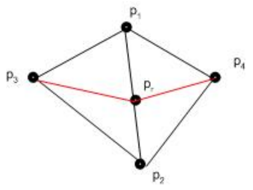

Figure 3: Inserting a point located on an edge of a triangle

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 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. The pseudo-code of the algorithm is depicted as Algorithm 4.1.

This algorithm can be done in parallel by multiple processes or on the Grid. Every process gets a subset of $P$ and does a Delaunay triangulation. Problems can occur on the borders of each subset because the processors do not know their neighbors' border values. So one has to look for a smart overlapping strategy.

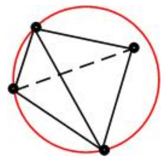

Figure 4: Edge flip

```
Algorithm 4.1: DELAUNAY($P$)

Input: Set of points $P$
Output: Triangulation $T$

generate a triangle surrounding the whole set $P$
generate a random permutation of $P$
for $r \leftarrow 1$ 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)
```

## 4.3 Combining Triangulation and Optimization

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.

## 4.4 A Brute Force Strategy

The first and easiest way to do the triangulation is a brute force strategy (see Algorithm 4.2). If we have enough computing power like in a Grid system we can triangulate the function over the whole function domain in a very short period of time (see Figure 5). The brute force method does not need any information from the optimization algorithm like the search direction.

We have to pay attention to borders between the processes. When combining the triangulations computed in parallel, one has to check the circumcircle condition for the border points. If this condition does not hold, edge flipping has to be performed.

The problem with such an implementation is that the bigger part of the computed values and triangulation 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.

## 4.5 A Realistic Strategy

The next strategy is more realistic because it restricts the computed triangulations to certain areas of interest (see Algorithm 4.3, Figure 6). In the beginning, we compute the triangulation of a subdomain around the starting value. If this is done we can start the optimization. Every time evaluating the Torque function we have to check if the requested point is in our subdomain. If this is the case, we return the interpolated value, otherwise we have to choose a *new* sub domain and compute the *new* triangulation. Each can compute the triangulation in parallel in the corresponding subdomain.

The problem with this solution is that for each new subdomain the optimization algorithm hast to wait for the Delaunay triangulation until it can continue. One idea to improve the solution is to decouple the triangulation from the optimization algorithm and let it work with the new domain, while the optimization algorithm still operates in the old domain. This leads us to the last — the smart strategy.

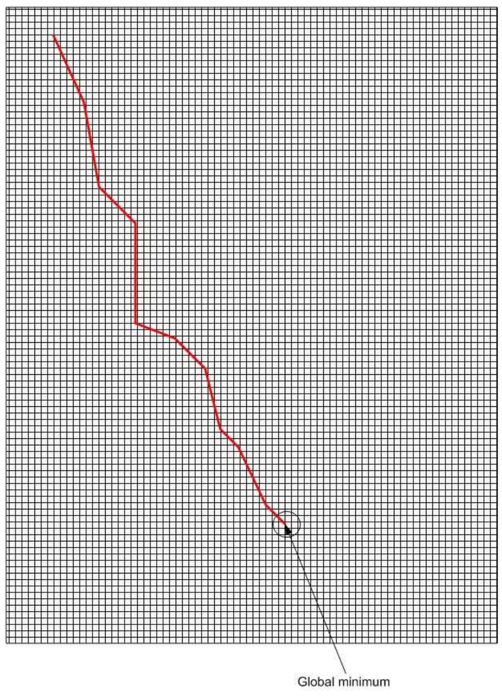

Figure 5: The Brute Force Strategy

**Algorithm 4.2:** PARALLELDELAUNAY($P$)

```
Input: Set of points $P$
Output: Triangulation $T$

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

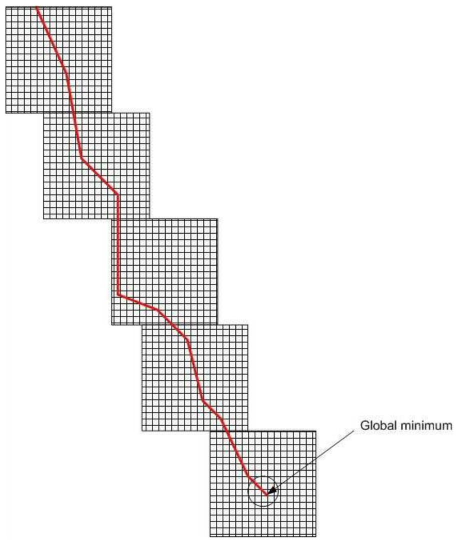

Figure 6: The realistic Strategy

```
Algorithm 4.3: REALISTIC($P, startingvalue$)

Input: Set of points $P$, starting value
Output: Minimum of the Torque function

$i \leftarrow 0$
determine an initial subdomain $\overline{P_i}$ around the starting value
PARALLELDELAUNAY($\overline{P_i}$)
while minimum not found
  do
    while optimization does not leave subdomain
      do
        Levenberg-Marquardt optimization step
        if minimum is found
          then {break
    determine a new subdomain $\overline{P_i}$
    PARALLELDELAUNAY($\overline{P_i}$)
    $i \leftarrow i + 1$
return ($minimum$)
```

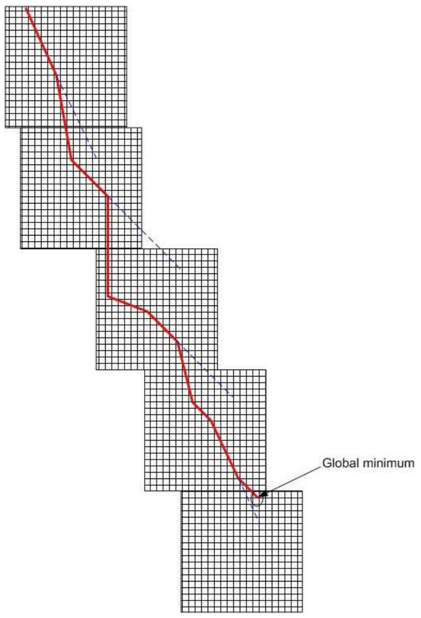

Figure 7: The Smart Strategy

## 4.6 A Smart Strategy

This strategy (see Algorithm 4.4, Figure 7) combines optimization and triangulation best, because both can work nearly independent. For this purpose we need some information about the search directions. Out of this information we try to get the position of the new subdomain. This cannot be done exactly but we predict the new position by extending one of the last search directions in the last subdomain and compute the new triangulation at a certain position adjoined to the old sub domain.

An implementation of this algorithm needs two threads: one for the optimization algorithm and one for the Delaunay triangulation. There has to be some communication between the threads for providing the search directions to the Delaunay thread. The Delaunay computation itself can be done on a Grid system. So the computation time can be minimized and the optimization does not have to wait for the new triangulation.

**Algorithm 4.4:** SMART($P$, *startingvalue*)

```
Input: Set of points $P$, starting value
Output: Minimum of the Torque function

$i \leftarrow 0$
determine an initial subdomain $\overline{P_i}$ around the starting value
PARALLELDELAUNAY($\overline{P_i}$)
compute Optimization and Triangulation in parallel
Optimization {
    while minimum not found
    do {
        Levenberg-Marquardt optimization step
        if minimum is found
          then {break
        Send current position of optimization and
         search direction to Triangulation
Triangulation {
    Receive current position of optimization and
     search direction from Optimization
    if position is near the border
      then {
        determine a new subdomain $\overline{P_i}$
         (predict with search direction)
        PARALLELDELAUNAY($\overline{P_i}$)
        $i \leftarrow i + 1$
return (minimum)
```

# 5 Conclusions and Outlook

In our present work we analysed the current implementation of SEE-GRID and did theoretical investigations on improving this implementation. First we searched for possible acceleration techniques for the sequential algorithm. Afterwards, we developed strategies using parallelization. The fine grained approach had to be canceled because of the small dimensions of the matrices and vectors. The most promising future directions are:

- Broyden update
- Prototype implementation of the parallel triangulation

We have to check our current implementation of the Broyden method to get usable results performing the update more than one time and moreover obtain better timings from our implementation.

Concerning the triangulation we have to implement the three strategies top-down, because we need the parallel Delaunay algorithm both in the realistic and the smart strategy. Furthermore, the determination of the subdomains in the realistic strategy is needed also in the smart strategy.

## References

[1] Walter Alt. *Nichtlinare Optimierung (Nonlinear Optimization, in German)*. vieweg, Wiesbaden, 2002.

[2] 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.

[3] 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.

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

[5] 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.

# A Source Code

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 3.1.

⋮

```
    pk=OLDJ.i()*(-1)*FOLD;
    XOUT=XOUT+pk;

    //compute fk+1
    for(int gcnt = 0; gcnt < nvars; gcnt++) {
        x = XOUT;
        f = MatrixToColumnVector (funcObj->Evaluate(x));
        if (funcObj->EvaluationError()){
            return Matrix(0,0);
        }
    }
    //Broyden
    yk = f - FOLD;
    sk = XOUT - OLDX;
    wk = OLDJ * sk;
    vk = sk.t() * sk;
    uk = (1 / vk.element(0,0)) * (yk-wk);

    //update matrix
    GRAD = OLDJ + (uk * sk.t());
    OLDJ = GRAD;

    FOLD << f;
    OLDX << XOUT;
}
//update counter
broydenc++;
if(broydenc==1){
    broydenc=0;
}
```

⋮