Elastic Non-Rigid Image Registration with Diffpack

Georg R. W. Steidl

Master Thesis
Elastic Non-Rigid Image Registration with Diffpack

Georg R. W. Steidl

Master Thesis

Aufgabensteller: Prof. Dr. U. Rüde
Betreuer: Dipl. Inf. Dipl. Kfm. H. Köstler
Erklärung:

Ich versichere, daß ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und daß die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Abstract

This thesis arises from a cooperation with the innTech company that distributes an engineering software package called Diffpack. Diffpack is a modern software-environment for solving partial differential equations. For a programmer Diffpack just acts as a numerical library consisting of C++ classes. This enables the computational scientist to develop his own application significantly faster and more securely. Still, the flexibility granted by a standard C++ program is almost entirely maintained. Diffpack shall be applied to image registration, in particular elastic non-rigid image registration formulated as a variational problem. The resulting non-linear system of partial differential equations is to be solved by various non-linear solution techniques (Fixed-Point iteration, Time-Marching scheme and Newton-type methods). These methods are thoroughly analyzed with varying parameters and finally compared. At the same time, the functionality of Diffpack is pictured and explained by comprehensible examples. In addition its computational efficiency is illuminated. The aim is to provide the conception for an accurate valuation of both the software package Diffpack and the different non-linear solution techniques.
Zusammenfassung

Contents

1 Introduction 1

2 Elasticity Model - Navier-Lamé equations 4
   2.1 Distance measure - Force 4
   2.2 Smoother - The Physical Model 5
   2.3 FEM Formulation 7
   2.4 Non-Linear Solution Techniques 8
      2.4.1 Fixed-Point Iteration 9
      2.4.2 Time-Marching Scheme 9
      2.4.3 Newton Type Method 9

3 Diffpack 12
   3.1 Introduction 12
   3.2 Functionality 14
      3.2.1 FEM Numerics 14
      3.2.2 Writing a FEM-program in Diffpack 20
      3.2.3 Program interfaces to user and other programs 25
   3.3 Application on Elasticity Problem 27

4 Registration Results 34
   4.1 2D Case 34
      4.1.1 Memory Requirements 34
      4.1.2 Fixed-Point Iteration 35
      4.1.3 Time-Marching Scheme 40
      4.1.4 Newton Type Method with Jacobian approximated by weighted Identity matrix 46
      4.1.5 Newton Type Method with linearly approximated Jacobian 53
   4.2 3D Case 55
      4.2.1 Time-Marching Scheme 56
      4.2.2 Newton Type Method I 59
      4.2.3 Newton Type Method II 61
4.3 Summarizing the results of the non-linear solution techniques ............................. 63
4.4 Diffpack Performance ......................................................................................... 64

5 Conclusions ........................................................................................................... 66

6 Acknowledgements .............................................................................................. 68

List of Tables ........................................................................................................... 68

List of Figures ......................................................................................................... 69

List of Listings ......................................................................................................... 71

Bibliography ............................................................................................................ 73
Chapter 1

Introduction

Image registration is an important task of image processing. Especially in the field of medical imaging its present impact is huge. In fact it can be stated: whenever two sets of data are acquired by sampling one object at different times, or from different perspectives, those sets of data are available only in different coordinate systems. In order to compare or integrate such sets of data they need to be aligned. The process of transforming the different sets of data into one coordinate system is called image registration. In that context the image registration problem can be phrased in brief: given a reference and a template image, find a suitable transformation such that the transformed template becomes similar to the reference image.

This shall be motivated by an example from medicine: Two datasets with different modalities are taken from the same patient. The left image of Fig. 1.1 shows the CT image which visualizes varying densities inside the body. By this, structures like bones with a lower density can be distinguished from organic structures like muscles or organs with a higher density.

The middle image depicts a PET scan. Its special ability is to visualize metabolic processes. Therewith, diagnoses about e.g. tumors and the search for metastases are possible.

However, in order to gain a comprehensive view of the patient these two datasets have to be registered (see Fig. 1.1).

Mathematically our framework reads as follows:

Given two images \( T, R : \Omega \rightarrow \mathbb{R} \), where \( \Omega \subset \mathbb{R}^d \), we are looking for a transformation \( \phi : \Omega \rightarrow \mathbb{R}^d \) which maps the image \( T \) to the image \( R \) such that corresponding structures are mapped onto each other. In the following the image \( T \) is called the template and \( R \) the reference.

\[
T \circ \phi \approx R. \tag{1.1}
\]

For our purpose, however, it is more convenient to split the transformation \( \phi \) into the trivial identity part and the deformation or displacement \( u \), i.e.

\[
\phi(x) = x - u(x), \quad u : \mathbb{R}^d \rightarrow \mathbb{R}^d. \tag{1.2}
\]

In order to measure the similarity between two images \( R \) and \( T \) the distance measure \( D \) is
introduced. The most intuitive approach to solve the registration problem is to find a $u$ such that the distance between $R$ and $T_u$ is minimal:

$$D[R, T; u] := D[R, T_u] \rightarrow \min,$$

(1.3)

where $T_u$ is the deformed template

$$T_u(x) := T(x - u(x)).$$

(1.4)

Various choices of distance measures are available. For multi-modal registration the so-called mutual-information is appropriate. Yet, we concentrate on mono-modal image registration. For this purpose the most common distance measure is the sum of squared distances $D = D^{\text{SSD}}$:

$$D^{\text{SSD}}[R, T] := \frac{1}{2} \|T - R\|_{L_2}^2 = \frac{1}{2} \int_{\mathbb{R}^d} (T(x) - R(x))^2 dx.$$

(1.5)

However, direct minimization of the distance measure yields some major difficulties: First of all the problem is ill-posed since small changes of the input data may result into large changes of the output data. Further the solution is not necessarily unique since the problem is not convex. This can be easily exemplified by Fig. 1.2 where by rotation an infinite number of solutions exists. Moreover the deformation may be even not continuous [CDH+05].

Therefore we introduce an additional regularizing term or smoother $\mathcal{S}$. This smoother has to guarantee on the one hand solutions for the above-mentioned mathematical difficulties and on the other hand guarantee physically reasonable deformations.

Our new optimization problem reads as

$$\mathcal{E}[u] := D[R, T; u] + \alpha \mathcal{S}[u] \rightarrow \min.$$

(1.6)
We have introduced a positive weighting factor $\alpha \in \mathbb{R}^+$ which controls the influence of the smoother $S$.

It is usually convenient to express the functional $\mathcal{E}$ in bilinear form:

$$\mathcal{E}[u] := \alpha \frac{1}{2} a[u, u] + b[u],$$

where $D = b[u]$ and $S = \frac{1}{2} a[u, u]$.

Chapter 2 explains the elasticity model serving as our smoother $S$ and the consequential Navier-Lamé equations, our base equations.
Chapter 2

Elasticity Model - Navier-Lamé equations

The main idea was presented by Broit (1981) [Bro81] and states that the image is considered as an elastic body which can be deformed by outer forces. In contrast to rigid transformations, where only translations and rotations are allowed, distances and angles between the particles of an elastic body may vary.

A necessary condition for a minimizer $u$ of Problem 1.6 is that the Gateaux derivative $dE$ of $E$ vanishes for all possible perturbations $v$. This derivative is also known as the first variation of $E$ in the direction of $v$.

At first we study the distance term $D[R, T; u]$ and its derivative.

2.1 Distance measure - Force

We shall see that the Gateaux derivative $dD[R, T; u; v]$ of our SSD distance measure is given by

$$dD[R, T; u; v] = -\int_{\mathbb{R}^d} (f(x, u(x)), v(x))_{\mathbb{R}^d} \, dx,$$

(2.1)

where $f: \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}^d$,

$$f(x, u(x)) := (R(x) - T_u(x))\nabla T_u(x).$$

(2.2)

In the following this result is proofed in some detail [Mod04]. For this purpose the Taylor expansion of $T(x - u(x) + hv(x))$ with respect to $h$ at the expanding point $x - u(x)$ is required,

$$T(x - u(x) + hv(x)) = T_u(x) + h (\nabla T_u(x), v(x))_{\mathbb{R}^d} + O(h^2).$$

(2.3)
Thus,
\[ d\mathcal{D}[R; T; u; v] =\]
\[ = \lim_{\delta h \to 0} \frac{1}{\delta h} (D[R; T; u + \delta h] - D[R; T; u]) =\]
\[ = \lim_{\delta h \to 0} \frac{1}{2\delta h} \int_{\Omega} (T_u(x) + h \langle \nabla T_u(x), v(x) \rangle_{\mathbb{R}^d} + \mathcal{O}(h^2) - R(x))^2 -\]
\[ \quad - (T_u(x) - R(x))^2 dx =\]
\[ = \lim_{\delta h \to 0} \frac{1}{2\delta h} \int_{\Omega} (T_u(x) - R(x))^2 +\]
\[ \quad + \left( T_u(x) - R(x) \right) h \langle \nabla T_u(x), v(x) \rangle_{\mathbb{R}^d} + h^2 ((\nabla T_u(x), v(x))_{\mathbb{R}^d})^2 + \mathcal{O}(h^2) -\]
\[ \quad - (T_u(x) - R(x))^2 dx =\]
\[ = - \int_{\Omega} ((R(x) - T_u(x)) \nabla T_u(x), v(x))_{\mathbb{R}^d} dx. \quad \square \]

Note that \( \nabla T_u(x) \) means
\[ \nabla T_u(x) = \begin{pmatrix} \frac{\partial x_1}{\partial x} \\ \vdots \\ \frac{\partial x_d}{\partial x} \end{pmatrix} T(x - u(x)), \quad (2.5) \]
and since the argument \( u(x) \) in eq.\((2.5)\) depends on \( x \) the chain rule has to be applied. For simplicity we now assume \( x, u \in \mathbb{R}^1 \). Then the finite difference form reads as
\[ \frac{\partial}{\partial x} T_u(x) \approx \frac{T_u(x + h) - T_u(x - h)}{2h} = \frac{T(x + h - u(x + h)) - T(x - h - u(x - h))}{2h}. \quad (2.6) \]

Fortunately it is recommended in most cases to compute \( T_u(x) \) anyway and then it is convenient to compute \( \nabla T_u(x) \) directly within the deformed image \( T_u(x) \), see eq.\((2.6)\). For the computation of \( T_u(x) \) an interpolation scheme has to be applied since \( T(x - u(x)) \) is naturally defined pixel-wise and \( x - u(x) \) usually does not ‘hit’ the center of the pixel.

Recalling eq. 1.7 it is obvious that
\[ d\mathcal{D}[R; T; u; v] = \lim_{\delta h \to 0} \frac{1}{\delta h} (b[u + \delta h v] - b[u]) = b[v]. \quad (2.7) \]

### 2.2 Smoother - The Physical Model

We recall Broit’s concept that the image is treated as an elastic body. Taking now the force \( f \) arising from the derivation of the distance measure (eq. 2.1) and applying it on the image leads to a deformation of the image.

We need a principle stating in which way these spatial deformations are driven by external forces. The continuum mechanics provides thereto formulas taking the material properties into account.
Given an elastic medium $\Omega$ the deformation of this body is determined by Newton’s first law, the balance of forces [TG51]:

$$ f = -\text{div}\Sigma $$

where $\Sigma(x,t) := (\sigma_{r,s})_{r,s=1,\ldots,d} \in \mathbb{R}^{d\times d}$ is the so-called stress tensor. The term div$\Sigma$ on the right-hand side reflects the internal forces in the medium due to stresses and $f$ reflects the outer forces. Looking only at the $r$-th component of $f$, Newton’s first law can be rewritten as

$$ f_r = -\sigma_{r,s}.$$  

(2.9)

Note, that here and in the following repeated indices imply a summation.

The stresses in elastic materials are related to deformation gradients $u_{r,s}$, represented through the strain tensor

$$ \varepsilon_{rs} = \frac{1}{2}(u_{r,s} + u_{s,r}). $$

(2.10)

Recapitulating we know how deformation is related to strain and how outer forces are related to stress. The missing connection between strain and stress is provided by Hooke’s law. With the assumption of isotropic, linear elastic properties Hooke’s law is defined by

$$ \sigma_{rs} = \lambda \varepsilon_{qq} \delta_{rs} + 2\mu \varepsilon_{rs}. $$

(2.11)

Here $\lambda$ and $\mu$ are the so-called Lamé constants, which may vary in space throughout heterogeneous materials but are pure constants if the elastic material is homogeneous.

Originally Hooke’s law is defined with the material properties Young’s modulus of Elasticity $E$ and Poisson’s contraction ratio $\nu$. In order to write Hooke’s law in the compact form of eq.(2.11) the Lamé constants have been introduced:

$$ \mu := \frac{1}{2} \frac{E}{1 + \nu}, \quad \lambda := \frac{E\nu}{(1 + \nu)(1 + \nu(1 - d))}. $$

(2.12)

To get a feeling for the material constants they are briefly characterized:

**Poisson ratio.** When a sample of material is stretched in one direction, it tends to get thinner in the other two directions. This tendency is described by the Poisson’s ratio $\nu$. For a perfectly incompressible material deformed elastically (only valid at small strains since we have a linear elasticity model), the Poisson’s ratio is exactly 0.5. Most materials have $\nu$ between 0.0 and 0.5. Cork is close to 0.0 and rubber is almost 0.5.

**Young’s modulus.** Young’s modulus $E$ is a measure of the stiffness of a given material. It is defined as the ratio of the rate of change of stress with strain, i.e. $E = \frac{d\sigma}{d\varepsilon}$. The higher $E$ is, the stronger the stress has to be in order to enforce a certain strain. So the body becomes more rigid with increasing $E$. The range of values reaches from 0.01 to 1000 GPa. Typical values used for image registration are $\nu = 0.25$ and $E = 100$. With eq.(2.12) this results into $\lambda = 40$ and $\mu = 40$. 

Physical considerations show that the stress tensor is symmetric, i.e. $\sigma_{r,s} = \sigma_{s,r}$.

Combining equations (2.11) and (2.10) yields

$$\sigma_{rs} = \lambda u_{q,q} \delta_{rs} + \mu (u_{r,s} + u_{s,r}).$$

(2.13)

Inserting eq. (2.13) into eq. (2.9) returns $d$ equations for the $d$ unknown components of the vector deformation field $u_r$:

$$((\lambda + \mu)u_{q,q})_r + (\mu u_{r,q})_q = -f_r$$

(2.14)

In vector notation eq. (2.14) takes the form

$$\nabla[(\lambda + \mu)\nabla \cdot u] + \nabla \cdot [\mu \nabla u] = -f.$$  

(2.15)

Usually the elasticity parameters $\lambda$ and $\mu$ are constant and we obtain the well-known Navier-Lamé equations

$$\mu \Delta u + (\lambda + \mu)\nabla \text{div} u = f.$$  

(2.16)

Defining the Navier-Lamé operator $\mathcal{A}$,

$$\mathcal{A}[u] = \mu \Delta u + (\lambda + \mu)\nabla \text{div} u$$

(2.17)

we review our optimization problem (1.7) and can state:

$$dS[u; v] = \lim_{h \to 0} \frac{1}{2h} (a[u+hv, u+h] - a[u, u]) = a[u, v] =$$

$$= \int_{\mathbb{R}^d} \langle \mathcal{A}[u](x), v(x) \rangle_{\mathbb{R}^d}.$$  

(2.18)

Hence our problem is formulated by

$$\mathcal{A}[u] = f.$$  

(2.19)

Summarizing one can say that the term $\mathcal{A}[u]$ serves as a regularizer. It is modeling the material properties and the inner forces. The term $f$ reflects the outer forces arising from the difference measure.

### 2.3 FEM Formulation

Our solution variable $u_r$ is now approximated by

$$\hat{u}_r = \sum_{j=1}^{n} w_j^r N_j(x_1, \ldots, x_d), \quad r = 1, \ldots, d,$$

(2.20)

where $n$ is the number of all nodes, $N_j$ are prescribed finite element basis functions and $w_j^r$ are $n \cdot d$ coefficients which are to be found. Application of the Galerkin method on our problem (2.9) yields the weak formulation:

$$\int_{\Omega} \sigma_{rs,s} N_i d\Omega = - \int_{\Omega} f_r N_i d\Omega$$

(2.21)
Integrating the left-hand side by parts

\[
\int_{\Omega} \sigma_{rs,s} N_i \, d\Omega = - \int_{\Omega} \sigma_{rs} N_{i,s} \, d\Omega + \int_{\partial\Omega} \sigma_{rs} N_i n_s \, d\Gamma
\]  

leads to

\[
\int_{\Omega} \sigma_{rs} N_{i,s} + f_r N_i \, d\Omega = \int_{\partial\Omega} \sigma_{rs} N_i n_s \, d\Gamma
\]  

with

\[
\sigma_{rs} = \lambda u_{q,q} \delta_{rs} + \mu (u_{r,s} + u_{s,r})
\]  

and c.f. eq.(2.2)

\[
f_r = (R - T_u) \frac{\partial}{\partial(x_r)} T_u.
\]  

Fortunately boundary conditions for images which are pre-registered and embedded into a uniform background are of minor importance. Hence we may omit boundary integrals like \( \int_{\partial\Omega} \sigma_{rs} N_i n_s \, d\Gamma \) in general. This is equivalent with assuming suitable boundary conditions like \( \sigma_{rs} N_i n_s = 0 \) (traction-free) on the boundary. Our main equation is then given by

\[
\int_{\Omega} \sigma_{rs} N_{i,s} \, d\Omega = - \int_{\Omega} f_r N_i \, d\Omega
\]  

or respectively just

\[
Au = f.
\]  

\section{2.4 Non-Linear Solution Techniques}

We are faced with a semi-linear problem of the form

\[
\mathcal{A}[u](x) = f(x, u(x)), \quad \forall x \in \Omega,
\]  

since the right-hand side depends on \( u \). The region of interest (commonly the entire image) is treated as a \( d \)-dimensional unit image, i.e. \( \Omega = [0, 1]^d \).

There are different approaches to solve eq.(2.28). Yet all approaches have in common to linearize the highly non-linear problem around a current approximation \( u^k \). This linearization defines the so-called outer iteration and \( u^k \) denotes the solution in the \( k \)-th outer iteration. In every step of the outer iteration a linear subproblem arises which can efficiently be solved by usual iterative linear equation solvers (Conjugated Gradients method, Multigrid method etc.). The iteration steps of the iterative equation solvers are called inner iterations.

The following sections explain the different approaches to tackle the non-linearity.
2.4.1 Fixed-Point Iteration

A convenient and easy way to solve the Navier-Lamé equations (2.28) is to exploit a fixed-point iteration scheme. Starting with an initial guess $u^{(0)}$ we define $u^{(k+1)}$ implicitly by

$$A[u^{k+1}](x) = f(x, u^k(x)), \quad x \in \Omega, \quad k \in \mathbb{N}_0. \quad (2.29)$$

However, this scheme is not very stable and may lead to oscillations.

2.4.2 Time-Marching Scheme

Mostly the divergence in the fixed-point iteration scheme results from too high changes of $u$ in each outer iteration step. So it may happen that an initial guess is too far away from the solution and lies outside of the domain of attraction. In order to stabilize the scheme we want to have influence on the changes of $u$ and introduce an artificial time $t$:

$$\partial_t u(x, t) + A[u](x, t) = f(x, u^k(x)), \quad (2.30)$$

making $u$ time dependent, i.e. $u(x,t)$.

Note that a steady-state solution of eq. (2.30) fulfills identically the Navier-Lamé equations (2.28).

Corresponding to eq. (2.29) we apply a semi-implicit discretization, i.e. we treat $A[u]$ implicitly and $f(u)$ explicitly:

$$\frac{u^{k+1}(x) - u^k(x)}{\tau} + A[u^{k+1}](x) = f(x, u^k(x)), \quad (2.31)$$

with the fixed time-step $\tau$ and $u^{(k)}(x) = u(x, k\tau)$. Eq. (2.31) may also be written as

$$(I + \alpha \tau A)u^{k+1} = u^k - \tau f(u^k). \quad (2.32)$$

2.4.3 Newton Type Method

A higher order approach is provided by the Newton Method. We first consider a general function $F(u) = 0$. In order to solve this multi-dimensional problem Newton’s method shall be applied:

$$u^{k+1} = u^k - \tau \left( J(u^k) \right)^{-1} F(u^k), \quad (2.33)$$

where $J \equiv \nabla F$ is the Jacobian of $F$ and $\tau \in \mathbb{R}^+$ denotes the so-called step size.

If $F = (F_1, \ldots, F_n)^T$ and $u = (u_1, \ldots, u_n)$, the entry $(i,j)$ of $J$ equals $\partial F_i / \partial u_j$. Instead of solving for

$$b := \left( J(u^k) \right)^{-1} F(u^k) \quad (2.34)$$

by computing the inverse of a matrix and subsequent multiplication with $F(u^k)$ it is numerically preferable to solve the following system of linear equations for $b$:

$$J(u^k) b = F(u^k). \quad (2.35)$$
The new solution \( u^{k+1} \) is then obtained by
\[
 u^{k+1} = u^k - \tau \mathbf{b}. \tag{2.36}
\]

We recall our original problem
\[
 \int_\Omega \sigma_{rs,s} N_i \, d\Omega = - \int_\Omega f_r N_i \, d\Omega, \quad r, s = 1, \ldots, d \quad i = 1, \ldots, n, \tag{2.37}
\]
where \( n \) is the number of all nodes and \( d \) is the space dimension. With \( \mathbf{F} = (F_1^1, \ldots, F_1^n, F_n^1, \ldots, F_n^n)^T \) we set
\[
 F_i^r = \int_\Omega \sigma_{rs,s} N_i \, d\Omega + f_r N_i \, d\Omega = 0. \tag{2.38}
\]

There exist \( dn \times dn \) entries in \( \mathbf{J} \) since each of the \( dn \) functions \( F_i^r \) has to be derived with respect to the \( dn \) variables \( u_j^s \).

Once we have established a linear system of equations for the problem (2.37) (demonstrated in section 3.3) the left-hand side \( \int_\Omega \sigma_{rs,s} N_i \, d\Omega \) of eq.(2.37) can be comprised by a matrix-vector multiplication \( \mathbf{A} \mathbf{u} \). For sake of simplicity we split \( F_i^r \) up and define:
\[
 \hat{F}_i^r = \int_\Omega \sigma_{rs,s} N_i \, d\Omega \quad \quad \hat{F}_i^r = \int_\Omega f_r N_i \, d\Omega \tag{2.39}
\]

Thus we can easily calculate the Jacobian of \( \hat{F}_i^r \) since the derivative of \( \mathbf{A} \mathbf{u} \) is simply \( \mathbf{J} := \mathbf{A} \).

Only the remaining term \( \hat{F}_i^r \) and its Jacobian \( \hat{\mathbf{J}} \) have to be considered in more detail then.
\[
 \hat{F}_i^r = \int_\Omega \left( (R - T_u) \frac{\partial T_u}{\partial x_r} \right) N_i \, d\Omega. \tag{2.40}
\]

Its Jacobian entries are given by
\[
 \frac{\partial \hat{F}_i^r}{\partial u_j^s} = \int_\Omega \frac{\partial}{\partial u_j^s} \left( (R - T_u) \frac{\partial T_u}{\partial x_r} \right) N_i \, d\Omega = \int_\Omega \frac{\partial T_u}{\partial x_r} \frac{\partial}{\partial u_j^s} \left( (R - T_u) N_i \right) + (R - T_u) N_i \frac{\partial}{\partial u_j^s} \frac{\partial T_u}{\partial x_r} \, d\Omega. \tag{2.41}
\]

Indeed the term \( S \) contains second derivatives of the image \( T \) which potentially leads to difficulties on noisy images (in particular higher derivatives tend to get manipulated when noise occurs). Because of that and in order to save computing time the non-linear term \( S \) is canceled [Höm06].

So we speak of a newton type method in contrast to the classical newton method. With canceled \( S \) and \( \hat{u}(x) = \sum_{k=1}^n u_k N_k \) the calculation can be continued:
\[
 \int_\Omega \frac{\partial T_u}{\partial x_r} \frac{\partial}{\partial u_j^s} (R - T_u) N_i \, d\Omega = \int_\Omega \frac{\partial T_u}{\partial x_r} N_i \frac{\partial}{\partial u_j^s} T_u \, d\Omega = \int_\Omega - \frac{\partial T_u}{\partial x_r} N_i \frac{\partial T_u}{\partial u_j^s} \frac{\partial u_j^s}{\partial u_j^s} \, d\Omega \tag{2.42}
\]
and since \( \frac{\partial \sum_{k=1}^n u_k N_k}{\partial u_j^s} = N_j \) we finally obtain:
\[
 \frac{\partial \hat{F}_i^r}{\partial u_j^s} = \int_\Omega - \frac{\partial T_u}{\partial x_r} N_i \frac{\partial T_u}{\partial u_j^s} N_j \, d\Omega. \tag{2.43}
\]

As an option we may approximate \( \frac{\partial T_u}{\partial x_r} \approx \frac{\partial T_u}{\partial u_j^s} \) because \( \frac{\partial T_u}{\partial x_r} \) has to be computed anyway.

To draw a conclusion, various techniques are employed in practice to treat the Jacobian \( \hat{\mathbf{J}} \):
• Approximation of $\hat{J}$ by a scaled identity matrix $\epsilon I$
• Approximation of $\hat{J}$ by eq. (2.43) with avoiding the non-linear term $S$
• Exact Newton’s method by incorporating $S$ and its second derivatives

After computing the contributions of $\hat{F}_r$ to $J$ the matrix $A$ can just be added to realize the contributions from $\hat{F}_r$ (eq. (2.39)), i.e.

$$J = \hat{J} + \hat{J} = \hat{J} + A.$$ (2.44)

The Newton type method can then be executed as described in the beginning of this section by solving the linear system

$$\left(\hat{J} + A\right) b^{k+1} = Au^k - f(u^k).$$ (2.45)
Chapter 3

Diffpack

3.1 Introduction

Diffpack is a sophisticated tool for developing numerical software, with main emphasis on numerical solution of partial differential equations. For a programmer Diffpack acts as a numerical library consisting of C++ classes. By this chapter it shall become evident that programming with classes is more comfortable and more flexible than shuffling code structures in and out of subroutines in the traditional Fortran or C way. The C++ classes can be combined into application codes to solve problems in diverse fields, including engineering, natural sciences, economics, and medicine. Since the application codes make extensive use of well-tested libraries and high-level abstractions, the time spent on writing and debugging code is significantly reduced compared with traditional software development in plain Fortran or C. This enables the computational scientist or engineer to concentrate more on modeling, algorithms, and numerical experimentation. The customer activities span from simple prototype applications to projects involving several man-years of simulator development.

Unlike most products for PDEs, Diffpack is not a ready-to-use application, it is rather a development framework, designed to allow easy modification and combination of all numerical building blocks making up a potentially new application. It has strong focus on the flexible construction, modeling power and computational efficiency of the numerical kernel, rather than the specific nomenclature and graphical user interfaces targeting a particular engineering area.

As a result, Diffpack has no inherent restrictions on the types of PDEs that can be solved. Diffpack may of course very well be used to solve traditional engineering PDE problems. However, it is typical for current Diffpack users to solve problems that can be characterized as ‘non-standard’ and that lie beyond the mainstream engineering areas. Such problems often occur in pure research contexts or in the context of problems involving a variety of mutually dependent physical effects.

Thus Diffpack is meant to complement the functionality of advanced industry-standard FEM software systems like ANSYS, CFX, FLUENT, NASTRAN, ABAQUS, etc.
The Diffpack libraries contain a substantial collection of data structures and numerical algorithms. Its object oriented design is maintained by distinct use of inheritance and virtual functions.

These libraries can be accessed anywhere in your program code. Therefore interaction with other programs is very facile. Diffpack integrates easily with existing software as e.g. existing FORTRAN code which is to be re-utilized.

Another advantage is granted by the platform independency, so no porting problems may arise.

The functionality includes vectors, matrices, general multi-index arrays, strings, enhanced I/O, menu system for input data handling, simulation result database system, systems for automatic report generation, execution statistics, GUIs, preprocessors, interfaces to third party pre- and post-processing tools, representation of linear systems (in particular large sparse systems), a large number of state-of-the-art iterative methods for sparse linear systems, solution routines for non-linear systems, finite element and finite difference meshes, scalar and vector fields over meshes, a collection of finite elements, various finite element algorithms and associated data structures, numerical integration schemes, high level finite difference support, stencils (difference molecules), probability distributions, real and complex arithmetic, adaptive meshes, error estimation, multigrid methods, generalized (mixed) finite element methods, parallel computing tools and a large collection of source code examples for problems in a variety of application areas. The list of functionality is comprehensive.

However, the major advantage for the user is the freedom with which different parts of this...
functionality can be combined. In the design of Diffpack, great care has been taken to make all objects and functions highly interoperable. For the user this means increased ability to construct and maintain applications and numerical algorithms.

By this means an elasticity model, which is in Diffpack already implemented as a complex example from the field of solid mechanics, has been adducted to our purpose of image registration and serves as the basis for all the implementations regarding this Master Thesis.

The Diffpack product line is organized as a kernel product, referred to as the Diffpack Kernel, and a set of toolboxes providing specialized functionality. Available toolboxes are the following:

**Adaptivity Toolbox** Dynamic refinement of structured and unstructured meshes in 1, 2 and 3 dimensions.

**Multilevel Toolbox** A complete and flexible lab for constructing fast multigrid linear solvers.

**Datafilter Toolbox** A tool for import of industry standard FEM meshes into Diffpack.

**Parallel Toolbox** Automated parallelization of any PDE based sequential Diffpack application.

These four add-on toolboxes offer access to advanced functionality for any application with minimal development efforts. Typically, only a handful of additional code lines are necessary to bring them into operation.

Regarding the required user skills it can be stated that usage of Diffpack requires at least basic programming skills in C++.

Its fully functionality Diffpack releases in the field of Finite Elements. The mentioned features of Diffpack are presented step-by-step in the next section.

### 3.2 Functionality

#### 3.2.1 FEM Numerics

The functionality of Diffpack and how to use it can be best explained by an example. We take thereto the most common and elementary Poisson equation:

\[-u''(x) = f(x), \quad x \in \Omega \tag{3.1}\]

with homogeneous Dirichlet boundary conditions \(u = 0\) on \(d\Omega\). We want to apply a finite element method and are searching for an approximation

\[\hat{u} = \sum_{j=1}^{M} u_j N_j(x) \tag{3.2}\]

of the unknown function \(u(x)\), where \(M\) is the number of nodes (or respectively the number of basis functions \(N_j\)). Rewriting the Poisson equation in the residual form \(R = u''(x) + f(x) = 0\), the discrete weak formulation, also called *Weighted Residual Method*, is given by

\[\int_{\Omega} RW_i \, dx = 0. \tag{3.3}\]
with the weighting functions $W_i$. Inserting eq.(3.2) into the differential equation $R$ leads to

$$R = f(x) + \sum_{j=1}^{M} u_j N_j''(x). \quad (3.4)$$

The most common choice for the weighting function $W_i$ is to set $W_i = N_i$. This choice is referred to as Galerkin’s method:

$$- \sum_{j=1}^{M} \left( \int_{\Omega} N_i(x) N_j''(x) \, dx \right) u_j = \int_{\Omega} f N_i(x) \, dx, \quad i = 1, \ldots, M. \quad (3.5)$$

Integrating the left-hand side by parts, the integral over the boundary vanishes due to the homogeneous boundary conditions and we finally obtain:

$$\sum_{j=1}^{M} \left( \int_{\Omega} N_i'(x) N_j'(x) \, dx \right) u_j = \int_{\Omega} f N_i(x) \, dx, \quad i = 1, \ldots, M. \quad (3.6)$$

These equations can be comprised by a linear system $Au = b$ with matrix entries $A_{i,j} = \int_{\Omega} N_i'(x) N_j'(x) \, dx$ and $b_i = \int_{\Omega} f N_i(x) \, dx$.

In most cases the basis functions are chosen to be piecewise linear functions (hence the name 'finite elements'). Fig. 3.2 shows a sketch of piecewise linear function $N_i$ in 1D for a few elements.

![Figure 3.2: The index $i$ refers to the nodes in the grid and corresponds to the x axis. Here the graph illustrates the basis functions $N_3, N_4, N_5$ and $N_6$.](image)

On that account the product $N_i N_j$ as well as the product $N_i' N_j'$ are only different from zero if and only if $i$ and $j$ are nodes belonging to the same element. In other words, $A_{i,j} \neq 0$ if and only if $i$ and $j$ are nodes of the same element no. $e$. The fundamental idea is to construct the linear system arising in the finite element method in an element-by-element fashion since we may write

$$A_{i,j} = \int_{\Omega} N_i' N_j' \, dx = \sum_{e=1}^{m} A_{i,j}^{(e)} \quad \text{where} \quad A_{i,j}^{(e)} = \int_{\Omega_e} N_i' N_j' \, dx, \quad (3.7)$$
\[ b_i = \int_{\Omega} f N_i \, dx = \sum_{e=1}^{m} b_i^{(e)} \quad \text{where} \quad b_i^{(e)} = \int_{\Omega_e} f N_i \, dx. \quad (3.8) \]

It is therefore possible to consider only a certain element \( e \) and its local nodes \( i \) and \( j \) to compute \( A_{i,j}^{(e)} \). This motivates the concept of a local node numbering in each element. Since the element-wise construction (assembly) of the linear system is an underlying conception in Diffpack we step into the mathematics in more detail and examine a 1D linear element. If \( \Omega_e = [x^{[e]}, x^{[e+1]}] \), we refer to \( x^{[e]} \) as local node no. 1 and \( x^{[e+1]} \) as local node no. 2. In general for each local node no. \( r \) in element no. \( e \) a unique mapping exists to its global node number \( i \). This mapping to global nodes is granted by a function \( q \), i.e. \( i = q(e, r) \). For the present linear elements we have \( q(e, r) = e - 1 + r, r = 1, 2 \). For multi-dimension problems with intricate domain geometry, the function \( q \) has no simple analytical expression and is only known in form of a table.

**Local coordinates**

Moreover these local coordinates allow the concept of reference elements. The idea is to map elements being part of a non-trivial multi-dimensional domain onto a reference element of a fixed simple shape. Thus only the basis functions for the reference element are to be defined.

To begin with, we collect the nonzero contributions to \( A_{i,j}^{(e)} \) in a \( 2 \times 2 \) element matrix \( \tilde{A}_{r,s}^{(e)} \), where \( r, s = 1, 2 \) are local node numbers. An element vector \( \tilde{b}_r^{(e)} \) is also introduced. The element matrix and vector only involve integrals over element no. \( e \).

Now the mapping of the (actual) physical element \( \Omega_e = [x^{[e]}, x^{[e+1]}] \) onto the reference element \([-1, 1]\] has to be accomplished. The new coordinates \( \xi \in [-1, 1] \) in the reference element are related to the physical coordinates \( x \) by \( x = x^{(e)}(\xi) \). In our case this relationship becomes

\[ x^{(e)}(\xi) = \frac{1}{2} \left( x^{[e]} + x^{[e+1]} \right) + \xi \frac{1}{2} \left( x^{[e]} - x^{[e+1]} \right). \quad (3.9) \]

We need now the expressions for the basis functions in local coordinates \( \tilde{N}_r(\xi) \) in order to compute the integrals. It is easy to construct these functions for reference elements, because we know that for our example \( \tilde{N}_1(-1) = \tilde{N}_2(1) = 1, \tilde{N}_1(1) = \tilde{N}_2(-1) = 0 \), and that the functions must be linear. The unique result is then

\[ \tilde{N}_1(\xi) = \frac{1}{2} (1 - \xi), \quad \tilde{N}_2(\xi) = \frac{1}{2} (1 + \xi). \quad (3.10) \]

In general it must hold \( \tilde{N}_r(\xi) = \tilde{N}_i(x^{(e)}(\xi)), i = q(e, r) \), when \( x \in \Omega_e \).

The mapping between local and global (physical) coordinates is taken as

\[ x^{(e)}(\xi) = \sum_{r=1}^{n_e} \tilde{N}_r(\xi) x^{[q(e,r)]}, \quad (3.11) \]

where \( n_e \) is the number of nodes in the element \( (n_e = 2 \text{ for linear elements}) \). This formula coincides with the relation (3.9) for the special case of linear elements. When the \( \tilde{N}_r \) functions are used both for interpolation the unknown function and for mapping the reference element to global coordinates,
we refer to this element as an isoparametric element. The associated mapping (3.11) is called isoparametric.

### Coordinate Transformation of Derivatives and Integrals

Changing coordinates from $x$ to $\xi$ affects the derivatives:

$$
\frac{dN_i}{dx} = \frac{d\tilde{N}_i}{d\xi} = J^{-1} \frac{d\tilde{N}_i}{d\xi}, \quad i = q(e,r).
$$

(3.12)

Carrying the formula over to a $d$-dimensional problem, i.e. $x^{(e)}(\xi)$, $J$ becomes a $d \times d$ matrix with $J_{i,j} = \partial x_j/\partial \xi_i$. Thus the gradient operator is transformed according to the formula $\nabla N_i = J^{-1} \cdot \nabla_\xi \tilde{N}_r$, where $\nabla_\xi = (\partial/\partial \xi_1, \ldots, \partial/\partial \xi_d)^T$.

This shall be illustrated by transforming the integral of $\nabla N_i \cdot \nabla N_j$ over a 2D element $\Omega_e$ to the corresponding reference element $\tilde{\Omega}$, which is taken to be the square $[-1,1] \times [-1,1]$:

$$
\int_{\Omega_e} \nabla N_i \cdot \nabla N_j \, dx_1 dx_2 = \int_{-1}^{1} \int_{-1}^{1} J^{-1} \nabla_\xi \tilde{N}_r J^{-1} \nabla_\xi \tilde{N}_s \det J \, d\xi_1 d\xi_2.
$$

(3.13)

Notice that $J^{-1} \nabla_\xi \tilde{N}_j$ is a matrix-vector product.

Yet, for the implementation in Diffpack the transformation from the physical coordinates to the reference element $\tilde{\Omega}$ is done implicitly and automatically and so we omit the explicit transformation of derivatives and the explicit shape of the reference element. This means that we may write

$$
\int_{\Omega_e} \nabla N_i \cdot \nabla N_j \, d\Omega = \int_{\tilde{\Omega}} \nabla \tilde{N}_r \nabla \tilde{N}_s \det J \, d\xi_1 \ldots d\xi_d.
$$

(3.14)

Diffpack provides functions for evaluating $\tilde{N}_r, \nabla \tilde{N}_s, \text{ and } \det J$ at a point in $\tilde{\Omega}$. The Laplace term of a PDE is then reflected in the program code through the integrand

$$
\nabla \tilde{N}_r \nabla \tilde{N}_s \det J.
$$

(3.15)

Times a weight in the numerical integration rule used for evaluating integrals.

Revisiting our 1D model problem (see eq.(3.7) the element matrix is given by

$$
\tilde{A}^{(e)}_{r,s} = \int_{-1}^{1} J^{-1} \tilde{N}_r(\xi) J^{-1} \tilde{N}_s(\xi) \det J \, d\xi, \quad r, s = 1, 2
$$

(3.16)

and the element vector by

$$
\tilde{b}^{(e)}_r = \int_{-1}^{1} f^{(e)}(\xi) \tilde{N}_r(\xi) \det J \, d\xi.
$$

(3.17)

Now only the numerical integration is left to be done.

The general numerical integration rule of an integrand $g(\xi)$ in 1D has the form

$$
\int_{-1}^{1} g(\xi) \, d\xi \approx \sum_{k=1}^{n_I} g(\xi_k) w_k.
$$

(3.18)

Here $\xi_k$ is a numerical integration point and the integrand $g(\xi_k)$ is evaluated at each of the $n_I$ integration points and weighted by $w_k$. The accuracy of the integration rule is determined by the number of integration points $n_I$ and their location $\xi_k$. Most of the rules used in finite element computations are constructed such that they integrate a polynomial of a given order $p$ exactly. Various integration rules can be chosen in run-time.
Assembly of Element Matrices and Vectors

We review the splitting of the global coefficient matrix in a sum of element matrices as shown in eq. (3.7):

\[
A_{i,j} = \int_{\Omega} N_i' N_j' \, dx = \sum_{e=1}^{m} A^{(e)}_{i,j} \text{ where } A^{(e)}_{i,j} = \int_{\Omega_e} N_i' N_j' \, dx.
\]

But how do we actually add \(A^{(e)}\) to \(A_{i,j}\)? The entry \(A^{(e)}_{r,s}\) in the element matrix reflects the coupling between the local nodes \(r\) and \(s\) in the element \(e\). Because of that this entry \(A^{(e)}_{r,s}\) is equivalently a contribution to the coupling between the global nodes \(i\) and \(j\), where \(i = q(e, r)\) and \(j = q(e, s)\).

Hence we can formulate the algorithm for updating \(A_{i,j}\) with the entries in \(A^{(e)}_{r,s}\):

\[
A_{i,j} = A_{q(e, r), q(e, s)} \leftarrow A_{q(e, r), q(e, s)} + A^{(e)}_{r,s}.
\] (3.19)

The right-hand side is updated in a similar way:

\[
b_{q(e, r)} \leftarrow b_{q(e, r)} + b^{(e)}_{r}.
\] (3.20)

As usual, \(r\) and \(s\) run from 1 to the number of nodes in the element, \(e\). The process of adding element contributions to the global linear equation system is called assembly of element matrices and vectors. This assembly process is illustrated in Fig. 3.3. Algorithm 1 summarizes the element-wise construction of the global linear system. All evaluations of the basis functions are performed in local coordinates. That formulation of the finite element assembly is general enough to suffice a wide range of boundary value problems (various Dirichlet or Neumann boundary conditions can be fulfilled by setting the so-called essential boundary conditions).

Please note that Diffpack always starts to count with 1, not with 0 as it is common in C-style.

Having now an explicit impression about the numerical routines which have to be implemented we dedicate ourselves to the creation of a new Diffpack program from the very beginning.

Figure 3.3: Each element matrix is added to the according place in the global matrix
Algorithm 1 assembly algorithm

INITIALIZE GLOBAL LINEAR SYSTEM:

set $A_{i,j} = 0$ for $i, j = 1, \ldots, n$
set $b_i = 0$ for $i = 1, \ldots, n$

LOOP OVER ALL ELEMENTS:

for $e = 1, \ldots, m$ do

set $\tilde{A}^{(e)}_{r,s} = 0$; $r, s = 1, \ldots, n_e$
set $\tilde{b}^{(e)}_r = 0$; $r = 1, \ldots, n_e$

LOOP OVER NUMERICAL INTEGRATION POINTS:

for $k = 1, \ldots, n_I$ do

evaluate $\tilde{N}_r, d\tilde{N}_r/d\xi, d\tilde{N}_r/d\xi$ and $J$ at $\xi = \xi_k$

ADD CONTRIBUTION TO ELEMENT MATRIX AND VECTOR FROM THE CURRENT INTEGRATION POINT:

for $r = 1, \ldots, n_e$ do

for $s = 1, \ldots, n_e$ do

$\tilde{A}^{(e)}_{r,s} \leftarrow \tilde{A}^{(e)}_{r,s} + \frac{d\tilde{N}_r}{d\xi} \frac{d\tilde{N}_s}{d\xi} \det J w_k$

$\tilde{b}^{(e)}_r \leftarrow \tilde{b}^{(e)}_r + f(x^{(e)}(\xi_k)) N_r \det J w_k$

INCORPORATE ESSENTIAL BOUNDARY CONDITIONS

for $r = 1, \ldots, n_e$ do

if node $r$ has an essential b.c. then

modify $\tilde{A}^{(e)}_{r,s}$ and $\tilde{b}^{(e)}_r$ according to this condition

ASSEMBLE ELEMENT MATRIX AND VECTOR

for $r = 1, \ldots, n_e$ do

for $s = 1, \ldots, n_e$ do

$A_{q(r,e),q(s,e)} \leftarrow A_{q(r,e),q(s,e)} + \tilde{A}^{(e)}_{r,s}$

$b_{q(r,e)} \leftarrow b_{q(r,e)} + \tilde{b}^{(e)}_r$. 
3.2.2 Writing a FEM-program in Diffpack

In Diffpack exists a fundamental library class for finite elements, called FEM. We derive our own solver (or often called simulator) class Poisson from this class and gain the major framework, routines and capabilities for solving our PDE.

It turned out that the general FEM algorithm is essentially a loop over all elements in the grid, where for each element the local contribution to the discretized problem is computed by numerical integration. The integration method itself usually takes the form of an inner loop over a collection of integration points, where the integrand needs to be evaluated for each point picked. Observing that the PDE enters the calculation only in terms of the point samples of the integrand, this mathematical/numerical view of the solution process is mirrored in the Diffpack software. As an application programmer, the user supplies the necessary expressions to evaluate the integrand of the particular PDE for a given integration point. The rest of the solution process is automatically handled by the Diffpack library components, all the way from element definitions via numerical integration methods and matrix assembly, to the solution of the resulting algebraic problems. In addition, the user might have to supply the necessary input data concerning boundary conditions and initial conditions.

The most important class members and their mechanisms are illustrated in Fig. 3.4. Note that

![Diffpack classes and functions](image)

Figure 3.4: Diffpack classes and functions.
the functions on the right side of Fig. 3.4 are pre-defined and automatically called whilst only the yellow-marked functions, especially the integrands function, have to be user programmed.

These main functions have the following tasks:

- The routine **define** sets up the so-called menu system (described in greater detail later).
- The routine **scan** reads information necessary for setting up the problem, such as reading input files, reading the domain/grid, dynamic allocation of fields (e.g. images) etc.
- The routine **fillEssBC**, if essential boundary conditions like \( u = g \) on \( \partial \Omega \) must be fulfilled.
- The function **integrands** represents the numerical heart of solver. It samples the integrands in the element matrix and vector at a numerical integration point. The assembly process and the numerical integration are administered automatically by a function **makeSystem** in class **FEM**.
- The main routine **solveProblem** which calls the routines **scan**, **fillEssBC**, **makeSystem** (in **FEM**) and **LineEqAdmFE::solve()** (a function for solving linear systems).
- The routine **resultReport** for automated reporting of selected results, e.g. for writing the solution in each outer iteration into a file.

Typical data members are:

- a finite element grid object of type **GridFE**,
- a finite element field object of type **FieldFE** which holds the values of the scalar field \( u \) over the specified grid
- an object of type **LineEqAdmFE** containing the linear system \( Ax = b \) in the current problem and possesses various solvers for linear systems
- a degree of freedom handler object of type **DegFreeFE** that transforms the field values of \( u \) into a vector of unknowns \( x \) in the linear system (or vice versa).

For a comprehensive understanding of Diffpack programming we take a closer look at the Diffpack’s finite element toolbox and the with it implemented class **Poisson0** which concerns our slightly extended Poisson problem:

\[
-\nabla \cdot [k(x)\nabla u(x)] = f(x), \quad x \in \Omega \subset \mathbb{R}^2
\]

\[
u(x) = g(x), \quad x \in \partial \Omega,
\]

where \( f(x) \), \( g(x) \), \( k(x) \) are prescribed functions and \( u(x) \) is the primary unknown.
The element matrix and the right-hand side contributions are maintained in local coordinates, denoted in by $\tilde{A}_{i,j}^{(e)}$ and $\tilde{b}_{i,j}^{(e)}$:

$$
\tilde{A}_{i,j}^{(e)} = \int_{\Omega} k \nabla N_i \cdot \nabla N_j \det J d\xi_1 \ldots d\xi_d, \quad (3.23)
$$

$$
\tilde{b}_{i,j}^{(e)} = \int_{\Omega} f N_i \det J d\xi_1 \ldots d\xi_d. \quad (3.24)
$$

The definition of class Poisson0 derived from the basic class FEM, placed in a header file Poisson.h, is listed below:

```cpp
// Simple FEM solver for the 2D Poisson equation
#ifndef Poisson0_h_IS_INCLUDED
#define Poisson0_h_IS_INCLUDED

#include <FEM.h> // FEM algorithms, class FieldFE, GridFE
#include <DegFreeFE.h> // degree of freedom book-keeping
#include <LinEqAdmFE.h> // linear systems: storage and solution

class Poisson0 : public FEM {
  protected:
    // general data:
    Handle(GridFE) grid; // pointer to a finite element grid
    Handle(DegFreeFE) dof; // trivial book-keeping for a scalar PDE
    Handle(FieldFE) u; // finite element field, primary unknown
    Vec(real) linsol; // solution of the linear system
    Handle(LinEqAdmFE) lineq; // linear system: storage and solution

    void fillEssBC (); // set boundary conditions u=g
    virtual void integrands // evaluate weak form in the FEM-equations
      (ElmMatVec& elmat, const FiniteElement& fe);

  public:
    Poisson0 ();
    ~Poisson0 () {}

    void scan (); // read and initialize data
    void solveProblem (); // main driver routine
    void resultReport (); // write comparison with analytical sol.

    real f (real x, real y); // source term in the PDE
    real kf(real x, real y); // coefficient in the PDE
    real g (real x, real y); // essential boundary conditions
};
#endif
```

We first notice that Handles are used instead of primitive C/C++ pointers. Basically, a handle is a normal C++ pointer with some additional "intelligent" features. The main reason for using handles is that they make the management of dynamic memory simple and reliable. They keep track of all the users of an object. Thus, memory is freed when a handle goes out of scope and there are no other handles to the object. The syntax is to create first an empty handle (NULL pointer) and then to rebind it to an object. Assigning two handles $x = y$ yields a warning in debug modus since it is a common error to redirect pointers instead of copying the object itself.
For further syntax of any Diffpack class it is always recommended to consult the detailed and clear Diffpack documentation in html-format.

This basic implementation is written for a 2D problem, but can be easily altered to deal with multi dimensional code. The scan function reads as

```cpp
void Poisson0:: scan ()
{
    // extract input from the command line:
    int nx, ny;  // number of nodes in x- and y-direction
    initFromCommandLineArg ("-nx", nx, 6); // read nx, default: nx=6
    initFromCommandLineArg ("-ny", ny, 6);
    String elm_tp;
    initFromCommandLineArg ("-elm", elm_tp, "ElmB4n2D");

    // the box preprocessor requires input on the form (example):
    // geometry:  d=2 [0,1]x[0,1]
    // partition:  d=2 elm= ElmB4n2D div=[4,4] grading=[1,1]
    String geometry = "d=2 [0,1]x[0,1]";  // 2D specific
    String partition = aform("d=2 elm=%s div:[%d,%d] grading:[1,1]", elm_tp.c_str(),nx-1,ny-1); // 2D specific

    grid.rebind (new GridFE());  // make an empty grid
    PreproBox p;  // preprocessor for box-shaped domains
    p.geometryBox().scan (geometry); // initialize the geometry
    p.partitionBox().scan (partition); // initialize the partition
    p.generateMesh (*grid);  // run the preprocessor

    u.rebind (new FieldFE (*grid,"u"));  // allocate, set name="u"
    dof.rebind (new DegFreeFE (*grid, 1));  // 1 unknown per node
    lineq.rebind (new LinEqAdmFE());  // Ax=b system and solvers
    linsol.redim (grid->getNoNodes());  // redimension linsol
    lineq->attach (linsol);  // use linsol as x in Ax=b
}
```

A significant portion of this function concerns string manipulation due to reading various data. Diffpack offers a pre-processor for creating ordinary grids. The file is commented in detail. The solution variable \( u \) is stored in a field with allocating a new FieldFE. After that a line equation system \( \text{lineq} \) is generated.

If we have to handle boundary conditions we need the \text{fillEssBC} function with the following scheme:

1. Initialize assignment of essential boundary conditions
2. for \( i = 1 \) to the number of nodes in the grid: do
3. if this node is on the boundary then
4. set the essential boundary condition

```cpp
void Poisson0:: fillEssBC ()
{
    dof->initEssBC ();  // init for assignment below
}
```

Listing 3.3: The fillEssBC() function for non-homogenous boundary conditions
const int nno = grid->getNoNodes();  // no of nodes
Ptv(real) x;  // a nodal point
for (int i = 1; i <= nno; i++) {
   // is node i subjected to any boundary indicator?
   if (grid->boNode(i)) {
      x = grid->getCoor(i);  // extract coor. of node i
      dof->fillEssBC(i, g(x(1),x(2)));  // u=g at boundary nodes
   }
}
dof->printEssBC(s_o, 2);  // debug output
}

The Ptv(real) class is a special vector suitable for representing multi-dimension data points, since it is optimized for vectors of length 1, 2, and 3. The syntax of basic operations coincides with the syntax of other vector classes in Diffpack where each vector class has its own purpose.

Finally the integrands function is presented:

1: Evaluate the Jacobian determinant times the integration weight.
2: Find the global coordinates of the current integration point.
3: Evaluate the f and k functions at this global point.
4: for i = 1 to the number of basis functions (element nodes): do
5: for j = 1 to the number of basis functions: do
6: Add the appropriate value to the element matrix.
7: Add the appropriate value to the element vector.

Listing 3.4: The integrands() function
void Poisson0::integrands(ElmMatVec& elmat, const FiniteElement& fe)
{
   const real detJxW = fe.detJxW();  // Jacobian * integr. weight
   // find the global coord. xy of the current integration point:
   Ptv(real) xy = fe.getGlobalEvalPt();
   const real x = xy(1);  const real y = xy(2);  // 2D specific
   const real f_value = f(x,y);  // 2D specific
   const real k_value = kf(x,y);  // 2D specific
   int i,j;
   const int nbf = fe.getNoBasisFunc();  // = no of nodes in element
   for (i = 1; i <= nbf; i++) {
      for (j = 1; j <= nbf; j++)
      {  
         elmat.A(i,j) += k_value*(fe.dN(i,1)*fe.dN(j,1)  // 2D specific
                                   + fe.dN(i,2)*fe.dN(j,2))*detJxW;
         elmat.b(i) += fe.N(i)*f_value*detJxW;
      }
   }
}

This implementation of the integrands perfectly mirrors the mathematical derivation of the integrand in the weak formulation (please compare with eq.(3.24)). Looking at the argument list of integrands it becomes obvious that fe is const and therefore an input parameter, whereas elmat is passed by non-constant reference and hence an output parameter.

However, after a finite element solver has been thoroughly verified, techniques to improve the
computational performance can be exercised. For example it is possible to exploit the symmetry of the element matrix and invariant arithmetic expressions should of course be moved outside the loops.

The `solveProblem` function is the main routine:

1. Call `fillEssBC` to assign boundary conditions.
2. Call `makeSystem` (inherited from `FEM`) to create the linear system.
3. Initialize the values of `linsol`.
4. Call `lineq->solve` to solve the linear system of equations.
5. Load the solution(`linsol`) into the field `u`.

Listing 3.5: The `solveProblem()` function

```c
void Poisson0:: solveProblem () // main routine of class Poisson0
{
    fillEssBC ();               // set essential boundary conditions
    makeSystem (*dof, *lineq); // assembly algorithm from class FEM
    linsol.fill (0.0);          // init start vector (needed for iterative solvers)
    lineq->solve();            // solve linear system
    dof->vec2field (linsol, *u); // load solution (linsol) into u
}
```

The `resultReport` function is at this stage simple and not explicitly presented. In section 3.2.3 it will be explained how complex fields can be displayed.

### 3.2.3 Program interfaces to user and other programs

**Menu System**

One important feature of most Diffpack applications is the so-called menu system. This is a hierarchical data management system used at run time to define initial values of all entities used in a Diffpack program. Such entities may be simple numerical parameters, but may as well be more abstract quantities such as matrix formats, algebraic solvers, convergence criteria, numerical integration schemes, element types, etc. Thus, the menu system gives any application the ability to select, at run time, all program entities, from simple constants to the numerical algorithms that will be used. The configuration is usually defined in input files (*.i files), and may in addition be redefined via graphical or command line selection dialogs. When Diffpack is used as sub-libraries in context of applications with established user communication, the configuration can be loaded directly from such an input stream (without input files or selection dialogs). The menu system allows multiple choices to each program entity. Multiple choices will result in one run per configuration, and are often used to set up consistent numerical experiments, e.g. in a batch process. As an example, the user could define a loop over different preconditioners and iterative solvers, in order to find the optimal set-up for a certain type of linear system.

A plenty of parameters can be issued. A section of a typical input file is shown in Fig. 3.5.
GUIs, Databases, Pre- and Postprocessing

Diffpack defines an internal format (SimRes) for storage of mesh and field structures. This format is used for storage of stand-alone entities (e.g. a simple grid file) as well as complete simulation scenarios (e.g. a SimRes database containing several mutually dependent grid and field objects). For postprocessing, Diffpack provides a suite of filters to convert SimRes data to formats used by third party visualization products, as e.g. Matlab, Gnuplot, IRIS Explorer, AVS and VTK. For preprocessing, filters are provided for mesh formats generated by Abaqus, Ansys and Nastran via the Diffpack Datafilter Toolbox. All these filters can operate either as file-to-file converters or they may be linked to an application for direct data exchange. Some of the postprocessors may also be started directly from a Diffpack application. In addition to filters to third party preprocessors, Diffpack provides internal preprocessors supporting lattice grids, transfinite mappings and super-element techniques as well as interfaces to the public domain grid generators GeomPack and Triangle. On Unix, any application will operate with graphical menu selections provided it is compiled with the appropriate options. On Windows, any application can be equipped with a GUI providing application control, result browsing, curve plotting and state-of-the-art visualization (VTK).

A multi-dimensional field $u$ can be dumped into the SimRes database just by the command

DpMenu -h > help.html.
database->dump (u). Later on the field(s) can be extracted by executing the filter simres2vtk which converts the SimRes-format into vtk-files. Various filter options are hereto available.

**Documentation Tools**

Diffpack provides powerful mechanisms for documentation of numerical experiments. Full project reports containing e.g. problem description, tabulated numerical results, solver statistics, images, videos and, e.g. the source code of the application itself, can be generated in ASCII, LaTeX or HTML formats. The reporting mechanisms interface to the menu system can be used to create reports of multiple runs with overview summaries and links to details for individual runs.

### 3.3 Application on Elasticity Problem

We recall now our FEM-formulation of section 2.3 and construct a discretization as it can easily be implemented with Diffpack, in particular regarding the integrands-function. This discretization procedure follows the description of writing a Diffpack program solving the Poisson equation 3.2.2.

Starting point is the Navier-Lamé equation

$$\int_{\Omega} \sigma_{rs} N_i,s \, d\Omega = - \int_{\Omega} f_r N_i \, d\Omega$$

(3.25)

with

$$\sigma_{rs} = \lambda u_{q,q} \delta_{rs} + \mu (u_{r,s} + u_{s,r}).$$

(3.26)

The unknowns are now multi-dimensional, i.e. $u_{rj}^*, j = 1, \ldots, n$ and $r = 1, \ldots, d$. These unknowns are collected in a vector

$$u = (u_1^1, \ldots, u_1^d, u_2^1, \ldots, u_2^d, \ldots, u_n^1, \ldots, u_n^d)^T.$$  

(3.27)

First we derive the expressions for the matrix $A$. By inserting eq. (3.26) into eq. (3.25) a linear system $Au = f$ arises. The indices $i$ and $r$ indicate the equation number, while $j$ and $s$ are used in the summation over the unknowns $u_{rj}^*$, such that the system can be expressed by

$$\sum_{j=1}^n \sum_{s=1}^d A_{i,j}^{rs} u_j^* = f_i^*, \quad r = 1, \ldots, d, \quad i = 1, \ldots, n.$$  

(3.28)

It is the task now to derive the entries $A_{i,j}^{rs}$ of the matrix. We recognize that (3.26) inserted in (3.25) gives rise to three basic terms which comprise $A_{i,j}^{rs}$. Instead of applying the summation convention we successively replace them by explicit summation symbols.

The integrand of the first term $\lambda u_{q,q} \delta_{rs} N_i,s$ can be written

$$\lambda u_{q,q} \delta_{rs} N_i,s = \sum_s \lambda u_{q,q} \delta_{rs} N_i,s = \sum_s \lambda \sum_k \frac{\partial u_k}{\partial x_k} \delta_{rs} N_i,s =$$

$$= \sum_s \lambda \sum_k \sum_j u_j^k N_{j,k} \delta_{rs} N_i,s =$$

$$= \sum_j \sum_s \lambda N_{i,r} N_{j,s} u_j^*.$$  

(3.29)
A quick comparison shows that this term perfectly fits with eq. (3.28). The second term $\mu u_{r,s}N_{i,s}$ is transformed to

$$\sum_j \sum_s \mu N_{j,s}u_j^r N_{i,s} = \sum_j \sum_s \sum_k \mu N_{j,k}N_{i,s}\delta_{rs}u_j^r =$$

$$= \sum_j \sum_s \left( \sum_k \mu N_{i,k}N_{j,k} \right) \delta_{rs}u_j^r,$$

(3.33)

such that $u_j^r$ is separated. The third term $\mu u_{s,r}N_{i,s}$ is straightforward as it takes the form of eq. (3.28) directly:

$$\sum_j \sum_s \mu N_{i,s}N_{j,r}u_j^s.$$  

(3.34)

Thus,

$$A_{i,j}^{rs} = \int \Omega \left[ \lambda N_{i,r}N_{j,s} + \left( \sum_k \mu N_{i,k}N_{j,k} \right) \delta_{rs} + \mu N_{i,s}N_{j,r} \right] d\Omega.$$  

(3.35)

Due to the $d$ unknowns per node the element matrix compared to the simple Poisson problem has to be extended. We designate the element matrix for our problem by $A_{i,j}$ which is a small block matrix representing the coupling between the nodes $i$ and $j$ of the considered element:

$$A_{i,j} = \begin{pmatrix} A_{i,j}^{11} & \cdots & A_{i,j}^{1d} \\ \vdots & \ddots & \vdots \\ A_{i,j}^{d1} & \cdots & A_{i,j}^{dd} \end{pmatrix}.$$  

(3.36)

Hence $A_{i,j}$ is a $d \times d$ matrix and $A$ can be viewed as a $dn \times dn$ matrix consisting of $n \times n$ blocks $A_{i,j}$.

The right-hand side contribution is simply

$$f_r^i = \int \Omega (R - T_u) \frac{\partial}{\partial x_r} T_u N_i d\Omega.$$  

(3.37)

By constructing the element matrix $A_{i,j}$ and the vector $f_r^i$ in the integrands-function, the basis for our (linear) solver is provided. At that the generality to register images in any dimension $d$ is maintained. List. 3.6 depicts the integrands-function.

```cpp
void Elasticity::integrands(ElmMatVec& elmat, const FiniteElement& fe) {
    const int d = fe.getNoSpaceDim();
    const int nbf = fe.getNoBasisFunc();
    const real detJxW = fe.detJxW();

    // Handle(Field) Tu, lambda, etc must be interpolated at current
    // integration point:
    const real lambda_pt = lambda->valueFEM (fe);
    const real mu_pt = mu->valueFEM (fe);
    const real iR_pt = ImgR->valueFEM (fe); // Image R
    const real iT_pt = ImgT->valueFEM (fe); // Image T
    const real Tu_pt = Tu->valueFEM (fe); // and Image Tu interpolated
    // at current point
```
Listing 3.6: The integrands() function for elastic image registration.

In order to handle the non-linearity our main class Elasticity is, besides from FEM, inherited from NonLinEqSolverUDC. The diagram 3.6 figures out which functions are to implement to accomplish non-linearity. In fact we have to derive our non-linear solver myNLSolver, from our main class Elasticity and inherit all its functionality. Then the solveProblem() and the
Figure 3.6: Extension of the program structure to non-linearity.
makeAndSolveLinearSystem() function may be adapted to the particular non-linear solution technique. Remaining functions as depicted on the right side of Fig. 3.6 usually stay unaltered.

The function \texttt{NL::solve()} starts the loop over the outer iterations. In each outer iteration the function \texttt{myNLSolver::makeAndSolveLinearSystem()} is executed which again starts the inner iterations of the linear iterative solver by calling \texttt{lineq->solve()}.

In order to gain a greater insight into the mechanics we take a look at the implementation of the time-marching scheme 2.4.2 and its resulting equation

\[(I + \alpha \tau A)u^{k+1} = u^k - \tau f(u^k).\] (3.38)

We first notice, that the iteration matrix \(I + \alpha \tau A\) stays constant and must be created only once. In contrast the right-hand side has to be updated in each outer iteration. Since the original matrix \(A\) is needed several times (e.g. for computing the elastic energy \(u^T Au\)), it is generated once by the command \texttt{makeSystem(*dof, *lineq)} and saved as a matrix \texttt{operatorA}. When generating the linear equation system in each outer iteration we actually want to bypass the generation of the iteration matrix since it is constant. Here the command \texttt{makeSystem(*dof, *b)} indicates that only the right-hand side \(b\) is to be modified.

List. 3.7 depicts \texttt{solveProblem()} function of our non-linear solution technique 'time-marching'.

```cpp
void TimeMarching:: solveProblem()
{
    // initialize nonlin_solution
dof->field2vec (*u, nonlin_solution);

    // beginning nonlinear solver:
s_o << "Non-Linear Solver starting..." << 'n';

    assemble_tp = ASSEMBLE_ALL;
    makeSystem(*dof, *lineq);
    assemble_tp = ASSEMBLE_VECONLY;
    lineq->A().makeItSimilar(operatorA);
    *operatorA = lineq->A();

    ExtendA(); // adapt the original EqSystem to the semi-implicit time scheme

    bool converged = nlsolver->solve ();

    // load nonlinear solution found by the solver into the u field:
dof->vec2field (nonlin_solution, *u);

    if (!converged)
    {
        errorFP("Elasticity::solveProblem",
                "The nonlinear solver \"%s\" did not fulfill the convergence\n                criterion in %d iterations (final \"error\" estimate: eps=%g).", 
                getEnumValue(nlsolver->getcurrentState().method).c_str(),
                nlsolver->getcurrentState().iteration_no,
                nlsolver->getcurrentState().eps);
    }

    saveResults();
}
```
The introduced enum-type `assemble_tp` signifies the `integrands`-function whether routines regarding the creation of the element matrix shall be skipped (`ASSEMBLE_ALL`) or must be computed (`ASSEMBLE_ALL`). Further the `makeItSimilar`-function is needed to copy the matrix $A$ of the linear equation system into the new matrix `operatorA` whilst remaining the sparsity pattern $A$.

List. 3.8 shows the `makeAndSolveLinearSystem()` function called automatically in each outer iteration step.

```cpp
void TimeMarching:: makeAndSolveLinearSystem()
{
  dof->vec2field (nonlin_solution, *u); // copy most recent guess to u
  updateTu(); // compute transformed image Tu
  energy_old = energy_new;
  computeEnergy(energy_new);
  relDistError = computeDistError();
  dumpEnergies();

  // normal (default) treatment of essential boundary conditions
  dof->unfillEssBC2zero();

  database->dump (*Tu);

  Vector(real) &b = lineq->b();
  makeSystem(*dof,b);

  // use the most recent nonlinear solution:
  lin_solution = nonlin_solution;
  Extendb(); // extend the right-hand side

  lineq->solve(); // solve the linear system
  // the solution of the linear system is now available in the vector
  lin_solution
}
```

Listing 3.8: The `makeAndSolveLinearSystem()` function for the time marching approach.

Here the function `Extendb()` modifies the right-hand side of the linear equation system according to eq.(3.38), i.e. the preliminary right-hand side $b$ is multiplied by $\tau$ and then added to the current $u^k$ stored as a vector in `lin_solution`:

```cpp
void TimeMarching:: Extendb()
{
  Vector(real) &UminusTauTimesF = lineq->b(); // reference to the right-hand side vector b
  UminusTauTimesF.mult(tau);
  UminusTauTimesF.add(UminusTauTimesF, lin_solution); // UminusTauTimesF += lin_solution
}
```

Listing 3.9: The function `Extendb()` called in each outer iteration step to accomplish the time-marching approach.
We gathered now the important functions employed for solving our registration problem by the time-marching technique. In a similar way the other non-linear solution techniques can be implemented. Remember that a more sophisticated Newton-type method as explained in section 2.4.3 has a non-constant iteration matrix since the Jacobi matrix varies with $u$. Thus it is required to re-compute the complete linear system at each outer iteration step. In addition the integrands function has to be adapted to the corresponding problem.
Chapter 4

Registration Results

We are ready now to run a series of combined test cases.

The test cases are applied on real CT datasets, both in 2D and 3D.

The following non-linear solution techniques (presented in Section 2.4) and some modifications of them are exerted and tested thoroughly:

- Fixed-Point Iteration
- Time-Marching Scheme
- Newton Type Method with Jacobian approximated by a scaled identity matrix $\epsilon I$
- Newton Type Method with Jacobian approximated linearly, i.e. avoiding second derivatives of the intermediate image $Tu$.

Each of these techniques involves its particular parameters which will be analyzed in a logic and structured way.

4.1 2D Case

We first dedicate ourselves to the 2D picture, a $256 \times 256$ image of a human brain.

Fig. 4.1 shows the template image $T$ (a), the reference image $R$ (b) and the initial difference image of these two images (c).

4.1.1 Memory Requirements

The memory requirements of the underlying FEM-solvers only depend on the image size and its dimension but not on the non-linear solver. Due to the sparsity pattern the matrix $A$ of the linear equation system needs the following memory space:

\[
\left( \frac{\text{no. of voxels} \cdot d}{\text{no. of unknowns}} \right) \cdot (\text{stencil size} \cdot d) \cdot 8 \text{ bytes},
\]

(4.1)
where \( d \) is the dimension. The stencil size in 2D amounts 9 and in 3D 27. Hence the memory requirement for the 256 \( \times \) 256 image aggregates to

\[
(256^2 \cdot 2) \cdot (9 \cdot 2) \cdot 8 \text{ bytes} = 18 \text{ Mbyte.}
\]

(4.2)

This calculated value matches the actual measured memory space (up to additive data structures of \texttt{lineq->A()}) very accurate. However, since the matrix \( A \) has to be stored additionally in an \texttt{operatorA} (see p.31) the memory requirements double. Furthermore the \texttt{makeSystem}-function creates some internal data structures which claim roughly 10%-25% of the matrix storage.

### 4.1.2 Fixed-Point Iteration

The fixed point iteration is the simplest variant of the non-linear solution techniques. The sole parameter which can influence the behavior of the solver is the weighting factor \( \alpha \) which controls the influence of the smoother \( S \) (eq.(1.6)). The higher \( \alpha \), the more rigid the image becomes. So, \( \alpha \) is usually to be chosen as small as possible.

In order to measure the quality of a registration two options are available:

**Relative Registration Error.** This is the normalized difference between \( Tu \) and \( R \),

i.e. \( \| Tu - R \| / \| T - R \|. \)

**Energy of the Distance Term.** This is identical with the SSD distance measure \( \frac{1}{2} \| T - R \|^2 \).

It is obvious that these options are equivalent and it is enough to relate to one.

The graphs Fig. 4.2 - Fig. 4.6 display how the relative registration error 'Rel.Reg-Error' and the elastic energy 'Elastic Energy' react on varying \( \alpha \). The Elastic Energy of course starts at zero, that means no deformation is present at the beginning.

It gets apparent that Fixed-Point Iteration is a very unstable technique. For too low values of \( \alpha \) the scheme diverges due to the fact that the stabilizing effect of the Smoother \( S \) narrows and the Data Term \( D \) implying a ill-posed problem predominates. As a consequence the Rel.Reg-Error goes beyond 1 (Fig. 4.2). For some medium value of \( \alpha \), here \( \alpha = 0.1 \), the scheme does not fully
Figure 4.2: Fixed-Point Iteration with $\alpha = 0.01$

Figure 4.3: Fixed-Point Iteration with $\alpha = 0.1$
Figure 4.4: Fixed-Point Iteration with $\alpha = 0.5$; the RegError steadies at 0.9

Figure 4.5: Fixed-Point Iteration with $\alpha = 1.0$; the RegError does not fall below 0.95
diverge but starts an oscillation (compare (Fig. 4.3). Then with increasing $\alpha$ the scheme seems to be stable. Yet the Rel.Reg-Error immediately takes a certain value which is not further decreased. For higher $\alpha$ the image gets more rigid and the outer force deduced from the Distance measure cannot deform the image sufficiently. Thence the RegError (Rel.Reg-Error) tends with increasing $\alpha$ to go up to 1.

In order to compare the different solution schemes we have to consider also the time aspect.

Thereto we introduce a template form of a table 4.1: The first column(s) depicts the underlying parameter(s). The second column \texttt{makeLinSys} returns the average time that is needed to generate the linear system while the third column \texttt{solveLinSys} returns the average time needed to solve this linear system. Further the column \texttt{Time / It} depicts the time per outer iteration which is the sum of the two previous columns and in brackets the inner loops stand needed to fulfill the convergence criterion.

The \texttt{makeLinSys} and \texttt{solveLinSys} function consume most of the computing time, usually about 90 % of the total program running time. Conversely, compelling statements about the total computing time can be made by a form like 4.1.

As iterative solver we employ the CG-method with a RILU pre-conditioner. All tests are run on a 3,4 GHz Intel® Pentium® 4 processor.

At least for this example the Fixed-Point Iteration was not effective since the non-linearity of image registration is very strong. This implicates that an initial guess may lay outside of the domain of attraction such that the scheme diverges.
<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\text{makeLinSys}$</th>
<th>$\text{solveLinSys}$</th>
<th>Time / It</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.01</td>
<td>0.6 sec</td>
<td>27.6 sec</td>
<td>28.2 sec (134)</td>
</tr>
<tr>
<td>0.1</td>
<td>0.8 sec</td>
<td>28.8 sec</td>
<td>29.6 sec (145)</td>
</tr>
<tr>
<td>0.5</td>
<td>0.7 sec</td>
<td>31.6 sec</td>
<td>32.3 sec (136)</td>
</tr>
<tr>
<td>1.0</td>
<td>0.6 sec</td>
<td>26.9 sec</td>
<td>27.5 sec (133)</td>
</tr>
<tr>
<td>5.0</td>
<td>0.6 sec</td>
<td>19.9 sec</td>
<td>20.5 sec (97)</td>
</tr>
</tbody>
</table>

Table 4.1: Fixed-Point Iteration with varying $\alpha$

Just for discussion we pose here a variant of the Fixed-Point Iteration which was encountered by the author of this thesis. It cures a problem of the original Fixed-Point Iteration namely that even for a proper choice of $\alpha$ the equation system takes a certain equilibrium which means the solution is not further improved (see Fig. 4.5 etc.). That is accomplished by adding the previous solution $u^k$ to the actual solution $u^{k+1}$ in each outer iteration step:

\begin{align}
Au^{k+\frac{1}{2}} &= f \\
\begin{aligned}
\quad u^{k+1} &= u^k + u^{k+\frac{1}{2}}.
\end{aligned}
\end{align}

(4.3)  
(4.4)

This pushes the solution forward in the presumed direction what corresponds to a certain over-relaxation. This variant of the Fixed-Point Iteration is presented in Fig. 4.7.

Figure 4.7: Over-relaxation of Fixed-Point Iteration with $\alpha = 0.1$

The over-relation variant for $\alpha = 0.1$ exhibits a surprisingly fast convergence.
Table 4.2: Over-relaxation of the Fixed-Point Iteration

Notice that in case of a stable problem the Elastic Energy behaves contrariwise to the Distance Term Energy or the Rel.Reg-Error, respectively.

To the authors opinion the success of the Over-relaxation of Fixed-Point Iteration is based on the circumstance that the usual Fixed-Point Iteration for $\alpha = 0.1$ only converges to a local minimum which is, globally contemplated, relatively high. The Over-relaxation of Fixed-Point Iteration bypasses this local minimum due to its over-relaxation step by chance and reaches a superior local minimum.

The computation times are collected in Tab. 4.2 with an additional column showing the total computation time to gain a 50 % Rel.Reg-Error.

### 4.1.3 Time-Marching Scheme

The Time-Marching Scheme from section 2.4.2 yields a new parameter $\tau$ which is to be analyzed. First however, we examine the effect of $\alpha$ again.

Fig. 4.8 - Fig. 4.11 illustrate the Time-Marching Scheme for a fixed $\tau = 1$ but with varying $\alpha$.

![Figure 4.8: Time-Marching Scheme with $\alpha = 0.1$ and $\tau = 1$](image)

With an increasing $\alpha$ the Time-Marching Scheme returns as expected lowering convergence
Figure 4.9: Time-Marching Scheme with $\alpha = 0.001$ and $\tau = 1$

Figure 4.10: Time-Marching Scheme with $\alpha = 0.00001$ and $\tau = 1$ yields a good result
Figure 4.11: Time-Marching Scheme with $\alpha = 0.0000001$ and $\tau = 1$ shows no difference to $\alpha = 0.00001$ and $\tau = 1$.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>makeLinSys</th>
<th>solveLinSys</th>
<th>Time / It</th>
<th>Time to 50% Reg. Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.6 sec</td>
<td>2.42 sec</td>
<td>3.02 sec (6)</td>
<td>n.a.</td>
</tr>
<tr>
<td>0.001</td>
<td>0.6 sec</td>
<td>1.73 sec</td>
<td>2.33 sec (2)</td>
<td>n.a.</td>
</tr>
<tr>
<td>0.00001</td>
<td>0.6 sec</td>
<td>1.72 sec</td>
<td>3.32 sec (1)</td>
<td>n.a.</td>
</tr>
<tr>
<td>0.0000001</td>
<td>0.6 sec</td>
<td>1.69 sec</td>
<td>2.29 sec (1)</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

Table 4.3: Time-Marching Scheme with varying $\alpha$ and fixed $\tau = 1$.

rates. However, it becomes apparent that even for a very low $\alpha = 0.0000001$ the Time-Marching Scheme stays stable. Yet, $\alpha = 0.00001$ grants already the optimum and values below do not further increase the convergence rate.

Table 4.3 displays the computational costs. Notice that each outer iteration is very fast and only needs a few inner iterations because a relatively low $\tau = 1$ implies rather small changes from $u^k$ to $u^{k+1}$. Hence the right-hand side (depending on $u^k$) changes little, too, and with $u^k$ as the initial guess for computing $u^{k+1}$ the linear system is solved immediately.

Thus we are ready to test how the time parameter $\tau$ influences the scheme. A priori, it is stated that for an infinite $\tau$ the Time-Marching Scheme is equivalent to the (unstable) Fixed-Point Iteration. For increasing $\tau$ the updates of $u^k$ at each $k$-th step may be larger. All tests were run with $\alpha = 0.00001$ since a smaller value does not further improve the convergence rate. The results are illustrated in Fig. 4.12 - Fig. 4.16.
Figure 4.12: Time-Marching Scheme with $\tau = 2$ and $\alpha = 0.00001$

Figure 4.13: Time-Marching Scheme with $\tau = 5$ and $\alpha = 0.00001$
Figure 4.14: Time-Marching Scheme with $\tau = 10$ and $\alpha = 0.00001$

Figure 4.15: Time-Marching Scheme with $\tau = 20$ and $\alpha = 0.00001$
With increasing $\tau$ the Time-Marching Scheme becomes faster. However, at $\tau = 10$ the curve of the RegError turns to be slightly contorted compared to the curve at $\tau = 5$. At $\tau = 50$ the Time-Marching Scheme lastly diverges.

Table 4.4 displays the computational costs. Notice that the time to obtain 50% RegError is nearly inversely proportional to the time step size $\tau$.

It is typical for the Time-Marching Scheme that the RegError decreases on costs of rising Smoother energy in order to find an equilibrium of the two energies. The scheme works very well. Yet, it does not perfectly mirror or reveal the process of minimizing the total energy like we will experience this behavior in the Newton Type Method.

It gets evident that the Time-Marching Scheme is very fast due to the convenient linear system to be solved in every time step. Moreover, the convergence rate is fine.
The difference images before and after the Time-Marching Scheme with $\tau = 20$ and $\alpha = 0.00001$ as illustrated graphically in Fig. 4.15 are presented in Fig. 4.17.

![Initial difference image](image1) ![Final difference image](image2)

Figure 4.17: Difference images before and after application of Time-Marching Scheme

The solution $u$ for this example is figured in Fig. 4.18.

![Solution in x-direction](image3) ![Solution in y-direction](image4)

Figure 4.18: The solution $u$ obtained from the Time-Marching Scheme

### 4.1.4 Newton Type Method with Jacobian approximated by weighted Identity matrix

We consider now a Newton Type Method where its Jacobian is replaced by an identity matrix for sake of simpleness and stability (see 2.4.3). This identity matrix is weighted by a factor $\epsilon$. Thus
Table 4.5: Newton Type Method I with varying initial step sizes and fixed $\epsilon = 1$

The system of linear equations of the Newton-Type method (2.45) can be rewritten:

\[(\epsilon I + A) b^k = Au^k - f(u^k).\]  \hfill (4.5)

\[u^{k+1} = u^k - \tau b^k.\]  \hfill (4.6)

We also call this method in the following Newton Type Method I. Hence we have got three parameters to control:

- The new parameter $\tau$ as the step size of the Newton Method (not to confound with the time parameter $\tau$ of the Time-Marching Scheme).
- The new parameter $\epsilon$ which weights the Jacobian.
- The already known parameter $\alpha$.

Fortunately experiments show that the influence of $\alpha$ is negligible for values low enough, e.g. for $\alpha = 0.00001$ as shown in the Time-Marching Scheme.

The step size control is implemented by a naive but reasonable and effective mechanism:

We keep track of the energy of the Distance Term of the current solution $u$ and of the previous solution $u_{old}$. If the energy for a certain step size has increased we re-roll the solution, i.e. we restore $u_{old}$ and halve the current step size. Then the next outer iteration is re-entered with the smaller step size. If the energy has not increased the current step size is enlarged by 10%.

Indeed an initial value has to be determined which has in particular impact on the convergence rate at an early stage. Fig. 4.19 - Fig. 4.21 illustrate the Newton Type Method for different initial step sizes.

In Fig. 4.21 the re-roll of the solution, due to the increased energy at iteration step 18, gets visible.

The computational costs are depicted in Table 4.5. However, if nothing contrary is mentioned we use 1 as initial choice for the step size $\tau$ in the following.

The factor $\epsilon$ effects the stability of the Newton Type Method I since $\epsilon$ is in fact added to each entry of the diagonal of the iteration matrix. Therefore we may expect higher convergence rates when we lower $\epsilon$.

The impact of $\epsilon$ in decreasing order is demonstrated in Fig. 4.22 - Fig. 4.26.

When $\epsilon$ is rather large (Fig. 4.22 - Fig. 4.24) the specified 20 outer iterations are not enough to achieve a minimum of the total energy since the Elastic Energy is still raising. Thus the RegError would further decrease if the computation is continued beyond the 20 iterations.
Figure 4.19: Newton Type Method I with initial step size $\tau = 1$ and fixed $\epsilon = 1$

Figure 4.20: Newton Type Method I with initial step size $\tau = 4$ and fixed $\epsilon = 1
Figure 4.21: Newton Type Method I with initial step size $\tau = 10$ and fixed $\epsilon = 1$; to avoid an increase of the RegError the re-roll mechanism takes effect

Figure 4.22: Newton Type Method I with $\epsilon = 1$, $\tau = 1$ and $\alpha = 0.00001$
Figure 4.23: Newton Type Method I with $\epsilon = 0.4$, $\tau = 1$ and $\alpha = 0.00001$

Figure 4.24: Newton Type Method I with $\epsilon = 0.1$, $\tau = 1$ and $\alpha = 0.00001$
Figure 4.25: Newton Type Method I with $\epsilon = 0.04$, $\tau = 1$ and $\alpha = 0.00001$

Figure 4.26: Newton Type Method I with $\epsilon = 0.01$, $\tau = 1$ and $\alpha = 0.00001$
<table>
<thead>
<tr>
<th>$\epsilon$</th>
<th>makeLinSys</th>
<th>solveLinSys</th>
<th>Time / It</th>
<th>Time to 50% RegError</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.6 sec</td>
<td>2.34 sec</td>
<td>2.94 sec (1)</td>
<td>n.a.</td>
</tr>
<tr>
<td>0.4</td>
<td>0.6 sec</td>
<td>1.78 sec</td>
<td>2.38 sec (1)</td>
<td>33.3 sec</td>
</tr>
<tr>
<td>0.1</td>
<td>0.6 sec</td>
<td>1.86 sec</td>
<td>2.46 sec (1)</td>
<td>13.0 sec</td>
</tr>
<tr>
<td>0.04</td>
<td>0.6 sec</td>
<td>2.03 sec</td>
<td>2.63 sec (2)</td>
<td>7.1 sec</td>
</tr>
<tr>
<td>0.01</td>
<td>0.6 sec</td>
<td>1.94 sec</td>
<td>2.54 sec (2)</td>
<td>n.a.</td>
</tr>
</tbody>
</table>

Table 4.6: Newton Type Method I with varying $\epsilon$ for fixed $\tau = 1$ and $\alpha = 0.00001$

For higher $\epsilon$ (compare Fig. 4.25 and Fig. 4.26) we recognize that the Elastic Energy reaches a certain value and then decreases together with the RegError towards a total energy minimum. We will see that this behavior gets even more evident in the 3D case.

For $\epsilon = 0.01$ the scheme seems to step into a local minimum where, even in spite of the re-roll mechanism and re-trying with a lower step size, the solution is stuck. The effect of the step size on the scheme with rather optimal parameters ($\epsilon = 0.04$) is minor. On the other hand, a good step size control compensates non-optimized parameters satisfactorily.

The computational costs for the Newton Type Method I and its efficiency are collected in Table 4.6. In order to affirm the good results for $\epsilon = 0.04$ the re-roll mechanism on increasing energy has been disabled. In fact the scheme did not diverge and nearly the same results were obtained. Fig. 4.27 depicts the behavior for the prone choice of $\epsilon = 0.01$ without re-rolling. No serious differences

![Figure 4.27: Newton Type Method I for $\epsilon = 0.01$ without re-roll mechanism](image-url)
in comparison to Fig. 4.26 can be detected.

4.1.5 Newton Type Method with linearly approximated Jacobian

We consider now the Newton Type Method II where the Jacobian is approximated by eq. (2.43) which has been derived in section 2.4.3. This approximation $\hat{J}$ has the characteristic feature that it discards the terms containing second derivatives of the image $Tu$. Still, the iteration matrix depends on $u$ and has to be recreated in every outer iteration.

Yet, the pure approach with approximating the Jacobi matrix by $\hat{J}$ turned out to be instable. Thus we reclaim the Jacobian approximated by the identity matrix for reasons of stability and introduce a mixed scheme between Newton Type Method I and the approximation by $\hat{J}$:

$$
(\delta \hat{J} + \epsilon I + A) \ b^k = Au^k - f(u^k),
$$

(4.7)

where $\delta$ weights the influence of the Jacobi approximation $\hat{J}$.

The implementation of that Newton Type Method II is validated by setting $\epsilon = 1$ and $\delta = 0$. This yields the same results like the Newton Type Method I with the only difference that the makeLinSys time increases from 0.6 sec to ca. 4 seconds.

We compare the Newton Type Method I with $\epsilon = 0.1$ and the Newton Type Method II with $\epsilon = 0.1, \delta = 0.9$ in order to clarify the influence of $\hat{J}$. Fig. 4.28 and Fig. 4.29 illustrate both cases.

![Figure 4.28: Newton Type Method I with $\epsilon = 0.1$, $\alpha = 0.00001$](image)

Yet, the convergence rate decreases with the impact of this choice of $\delta$. 

53
Figure 4.29: Newton Type Method II with $\epsilon = 0.1$ and $\delta = 0.9$, $\alpha = 0.00001$

<table>
<thead>
<tr>
<th>$\epsilon$</th>
<th>$\delta$</th>
<th>makeLinSys</th>
<th>solveLinSys</th>
<th>Time / It</th>
<th>Time to 50% RegError</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0</td>
<td>0.6 sec</td>
<td>1.86 sec</td>
<td>2.46 sec (1)</td>
<td>13.0 sec</td>
</tr>
<tr>
<td>0.1</td>
<td>0.9</td>
<td>3.6 sec</td>
<td>1.81 sec</td>
<td>5.41 sec (2)</td>
<td>33.6 sec</td>
</tr>
<tr>
<td>0.1</td>
<td>0.2</td>
<td>3.6 sec</td>
<td>1.98 sec</td>
<td>5.58 sec (2)</td>
<td>29.6 sec</td>
</tr>
</tbody>
</table>

Table 4.7: Confronting Newton Type Method I with Newton Type Method II

In fact it turned out that only a choice of $\delta = 0.2$ could yield a hardly noticeable enhancement compared to the pure Newton Type Method I with $\epsilon = 0.1$. This is depicted in Fig. 4.30. The Newton Type Method II reacts rather sensitive to the choice of $\delta$. With $\delta = 2$ the scheme already diverges.

The computing times for the three cases illustrated in Fig. 4.24 - Fig. 4.30 are shown in Table 4.7. It gets evident that at least for this example the very small enhancement compared to the Newton Type Method I is negated by the heightened computational costs. We shall see that the improvements of the Newton Type Method II are slightly better for the 3D image but after all we have to conclude that the Newton Type Method II is to handle with great care since it is very sensitive to $\delta$. Moreover it is computationally inefficient because the iteration matrix has to be adopted every step.
Figure 4.30: Newton Type Method II with $\epsilon = 0.1$ and $\delta = 0.2$, $\alpha = 0.00001$; compared with Fig. 4.28 only a very small improvement is achieved.

4.2 3D Case

In this section we want to approve the gained results from the 2D image.

The original 3D image we are facing has a size of $256 \times 256 \times 160$. In a finite element method, however, the mass matrix to handle this size of problem would amount to (see eq. (4.1))

$$(256 \times 256 \times 160 \cdot 3) \cdot (27 \cdot 3) \cdot 8 \text{ bytes} = 19440 \text{ Mbyte.}$$  

In order to keep memory consumption and computation time reasonable, we break the image down to a $32 \times 32 \times 20$ image which is content with 38 Mbyte. Fig. 4.31 depicts sections of the high-resolution 3D reference and template image.

Actually a finite element method is not really practicable for registration of 3D images due to its 27-stencil. A finite difference method is preferable. But the different non-linear approaches can be compared properly anyway.

The Fixed-Point Iteration is, similar as in the 2D case, not suited to cope with the strong non-linearity. Because of that, the over-relation scheme presented in section 4.1.2 neither yields adequate results. Hence we skip that section and consider directly the Time-Marching Scheme.
4.2.1 Time-Marching Scheme

It is inspected whether the advantageous results from Time-Marching Scheme in 2D expand to the three dimensional case.

Fig. 4.32 - Fig. 4.37 show the results of the Time-Marching Scheme for some choices of $\alpha$ and in particular for varying $\tau$.

We begin with a guess $\alpha = 1$ and $\tau = 1$ and realize there is room to decrease $\alpha$. With $\alpha = 0.0001$ we have found an optimal value for $\alpha$. Lower values will not further influence the performance.
Figure 4.33: Time-Marching Scheme applied on the 3D image with $\alpha = 0.0001$ and $\tau = 1$

Figure 4.34: Time-Marching Scheme applied on the 3D image with $\tau = 10$ and $\alpha = 0.0001$
Figure 4.35: Time-Marching Scheme applied on the 3D image with $\tau = 100$ and $\alpha = 0.0001$

Figure 4.36: Time-Marching Scheme applied on the 3D image with $\tau = 200$ and $\alpha = 0.0001$
Figure 4.37: Time-Marching Scheme applied on the 3D image with $\tau = 500$ and $\alpha = 0.0001$

Then we start to control the time parameter $\tau$ with a fixed $\alpha = 0.0001$ and we successively increase $\tau$. It gets evident again that, as long as the scheme stays stable, the convergence rate is directly proportional to $\tau$. Or in other words, the computation time to achieve a certain RegError is inversely proportional to $\tau$. Lastly, at $\tau = 500$ the scheme starts to oscillate.

The corresponding computing times are presented in Table 4.8.

### 4.2.2 Newton Type Method I

For sake of completeness the results for the Newton Type Method I are depicted, too. In principle Fig. 4.38 - Fig. 4.40 simply attest the results from the 2D case.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\tau$</th>
<th>makeLinSys</th>
<th>solveLinSys</th>
<th>Time / It</th>
<th>Time to 50% RegError</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.0</td>
<td>0.6 sec</td>
<td>15.5 sec</td>
<td>16.1 sec (7)</td>
<td>n.a.</td>
</tr>
<tr>
<td>0.0001</td>
<td>1.0</td>
<td>0.6 sec</td>
<td>13.1 sec</td>
<td>13.7 sec (1)</td>
<td>n.a.</td>
</tr>
<tr>
<td>0.0001</td>
<td>10.0</td>
<td>0.6 sec</td>
<td>13.7 sec</td>
<td>14.3 sec (1)</td>
<td>n.a.</td>
</tr>
<tr>
<td>0.0001</td>
<td>100.0</td>
<td>0.6 sec</td>
<td>13.5 sec</td>
<td>14.1 sec (2)</td>
<td>56.4 sec</td>
</tr>
<tr>
<td>0.0001</td>
<td>200.0</td>
<td>0.6 sec</td>
<td>13.7 sec</td>
<td>14.3 sec (2)</td>
<td>27.2 sec</td>
</tr>
<tr>
<td>0.0001</td>
<td>500.0</td>
<td>0.6 sec</td>
<td>13.5 sec</td>
<td>14.1 sec (2)</td>
<td>29.6 sec</td>
</tr>
</tbody>
</table>

Table 4.8: Time-Marching Scheme applied on the $32 \times 32 \times 20$ 3D image
Figure 4.38: Newton Type Method I applied on the 3D image with $\epsilon = 0.1$ and fixed $\alpha = 0.0001$

Figure 4.39: Newton Type Method I applied on the 3D image with $\epsilon = 0.01$ and fixed $\alpha = 0.0001$
<table>
<thead>
<tr>
<th>$\epsilon$</th>
<th>makeLinSys</th>
<th>solveLinSys</th>
<th>Time / It</th>
<th>Time to 50% RegError</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>0.6 sec</td>
<td>13.8 sec</td>
<td>14.4 sec</td>
<td>n.a.</td>
</tr>
<tr>
<td>0.01</td>
<td>0.6 sec</td>
<td>13.8 sec</td>
<td>14.4 sec</td>
<td>44.6 sec</td>
</tr>
<tr>
<td>0.001</td>
<td>0.6 sec</td>
<td>14.2 sec</td>
<td>14.8 sec</td>
<td>51.8 sec</td>
</tr>
</tbody>
</table>

Table 4.9: Newton Type Method I applied on the 3D image with fixed $\alpha = 0.0001$

Fig. 4.40 nicely demonstrates a typical Newton Method effect: At first the Elastic Energy rises rapidly. But then the Elastic Energy together with the RegError gets minimized as a total.

Table 4.9 comprises the computational costs of the 3D Newton Type Method I for varying $\epsilon$.

4.2.3 Newton Type Method II

The Newton Type Method II applied on the 3D image is still difficile. In comparison to the 2D case however, the Newton Type Method II improves the Newton Type Method I a little bit more significantly. For the case $\epsilon = 0.01$ Fig. 4.41 and Fig. 4.42 oppose the Newton Type Method I with the Newton Type Method II.
Figure 4.41: Newton Type Method I with $\epsilon = 0.01$ and fixed $\alpha = 0.0001$

Indeed $\delta = 0.6$ is well-chosen. For higher values the convergence rate starts to decrease again as
Table 4.10: Newton Type Method I applied on the 3D image with fixed $\alpha = 0.0001$

\[
\begin{array}{|c|c|c|c|c|c|}
\hline
\epsilon & \delta & \text{makeLinSys} & \text{solveLinSys} & \text{Time / It} & \text{Time to 50\% RegError} \\
\hline
0.01 & 0 & 0.6 sec & 13.8 sec & 14.4 sec (2) & 44.6 sec \\
0.01 & 0.6 & 5.7 sec & 15.5 sec & 21.2 sec (2) & 57.2 sec \\
0.01 & 0.99 & 5.7 sec & 15.8 sec & 21.5 sec (8) & n.a. \\
\hline
\end{array}
\]

it is illustrated in Fig. 4.43.

Figure 4.43: Newton Type Method II with $\delta = 0.99$ and $\epsilon = 0.01$; the influence of $\delta$ affects the convergence rate negatively.

The computational costs for the comparison of the Newton Type Method I with the Newton Type Method II for varying $\delta$ are gathered in Table 4.10. In the 3D case the Newton Type Method II is slightly closer to the computational efficiency of the Newton Type Method I.

4.3 Summarizing the results of the non-linear solution techniques

It turned out that the Fixed Point Iteration is not practicable. It mostly diverges because the initial guess lies outside of the domain of attraction.

The Time Marching scheme cures this problem by introducing an artificial time which controls the change of $u$ at each step. Thus the initial guess $u^k$ for each outer iteration is much more likely
in the domain of attraction since the right-hand side depends on $u^k$, too. The convergence rate of the Time Marching scheme depends inversely proportional on $\tau$. By this way it is possible to achieve excellent convergence rates. Its computational costs correspond to that of the Newton Type Method I. But in fact, due to its convergence rate, it demonstrated the best overall performance.

The Newton Type Method I presented a good performance, too. Yet with well-chosen $\epsilon$ its convergence rate could not fully compete with that of the Time Marching scheme.

The Newton Type Method II turned out to be crucial. Only with very carefully chosen $\delta$ a slightly better convergence rate compared to the Newton Type Method I could be attained. Moreover, its computational costs are extensive because the iteration matrix has to be adapted in each outer iteration.

In general it may be annotated that the parameters do not depend severely on the input data but more on the utilized non-linear solution technique. Hence, as soon as we have decided in favor of a certain technique, the parameters can be assigned a priori and uses for a wide range of input data.

4.4 Diffpack Performance

An important question of course is: how fast is Diffpack?

To start with, we recall some of the features Diffpack provides. For example it supports sophisticated, unstructured geometries. Thereto the FEM solver maps each finite element on a reference element.

Furthermore, in order to cope with all the necessary mathematical abstractions Diffpack others a vast number of inherited classes. Often, there originates a class hierarchy consisting of several generations. For example, the top-level class \texttt{FieldsFE} grants methods to act on this multi-dimensional field, like computing gradients or automatic interpolation when a non-node point is accessed. This class is successively derived from less complex classes until the C-array level is reached.

This flexibility is accomplished by extensive use of object-oriented functionality, in particular virtual functions.

However, on the lowest levels of the Diffpack libraries this object-oriented design is sacrificed for the sake of computational efficiency. Here for example \texttt{inline} functions for its particular goal (e.g. 2D-array access) are implemented. Yet, these optimizations are not visible to the end-user who accesses this functionality through a clean interface. In this manner several C++ language features, such as dynamic memory allocation, dynamic binding, operator overloading, and further, vector operations, matrix-vector products for dense and sparse matrices etc. have been thoroughly considered [Str].

By this means we benefit for example greatly from the uniform Laplace grid as used for image registration.

Experiments of H.P.Langtangen [ABC+99] show that a full-scale C++ program with Diffpack
takes about twice the computing time compared to a highly-tuned FORTRAN code.

Measurements of the elastic image registration implemented with Diffpack compared to a pure, non-optimized C++ code [SKR07] testify the factor 2 revealed by Langtangen. Thus, it can be stated:

Diffpack requires about 200 % the computational time a common C++ code takes.

But in fact, we have to take into account that the mentioned C++ is not optimized. By considering the underlying hardware, cache optimization etc. may be added. Thus, an additional performance loss of about 200 % by using Diffpack has to be accepted:

Diffpack requires about 400 % the computational time a hardware-optimized C++ code takes.

Indeed, state-of-the-art image registration is done by finite differences using a fast full multigrid solver. Therewith a Time Marching step for the 256 $\times$ 256 $\times$ 160 3D image is accomplished in 1.4 sec [SKR07].
Chapter 5

Conclusions

The evaluation of the implemented non-linear solution techniques has been performed in section 4.3. Yet, the Newton-type methods bear a high potential. Accordingly, ongoing research is seeking for better approximations of the Jakobi matrix. Here the Broyden-Fletcher-Goldfarb-Shanno algorithm (BFGS) is to mention. Stability and performance are the fundamental criterions. Of course implementation and analysis of such methods could be a challenging task for Diffpack.

After all, we are ready to draw conclusions from Diffpack programming. Diffpack proofed to be a solid programming environment for solving PDEs.

In fact, after some training period, the hours worked for implementing a new simulator decreases drastically. A huge advantage is granted by the well-tested libraries and high-level abstractions. The user written code reduces to a fraction and remains clear. Therefore debugging has never been that easy.

Regarding the training period, an experienced C++ programmer still will need two months training period on Diffpack. After this time he is fully equipped to develop his own simulators fast and securely. For many application areas a simulator example is already implemented. So it is often possible to adopt such an example for the own simulator. After some shortened training period this simulator can be extended for the own more specialized purpose.

On the side of the pros I like to mention further the tutorials for beginners and the more advanced tutorials concerning a particular topic, the good manual page, Diffpack support directly from the developers [Dif07] and last but not least the very nice book from the founder of Diffpack, H.P. Langtangen [Lan03].

Nevertheless, there exist drawbacks, too:

- Diffpack is a commercial program that has to be licensed for each working station.
- A training period to start with Diffpack is required.
- Some of the scripts (e.g. for visualization in gnuplot/wgnuplot) only work with Linux.
- Diffpack is still a niche program and not very wide-spread. And finally,
• the performance is about 400% worse than an optimized C++ program. Hence, Diffpack is not suitable for producing and publishing state-of-the-art computational research results.

Summarizing I want to state that Diffpack is an excellent program for solving PDEs:

1. if the developer is interested only in relative computational results, like comparing various solution techniques amongst each other;

2. unless the developer possesses his own comprehensive C++ libraries after years of research.
Acknowledgements

List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Fixed-Point Iteration with varying $\alpha$</td>
<td>39</td>
</tr>
<tr>
<td>4.2</td>
<td>Over-relaxation of the Fixed-Point Iteration</td>
<td>40</td>
</tr>
<tr>
<td>4.3</td>
<td>Time-Marching Scheme with varying $\alpha$ and fixed $\tau = 1$</td>
<td>42</td>
</tr>
<tr>
<td>4.4</td>
<td>Time-Marching Scheme with varying $\tau$ and fixed $\alpha = 0.00001$</td>
<td>45</td>
</tr>
<tr>
<td>4.5</td>
<td>Newton Type Method I with varying initial step sizes and fixed $\epsilon = 1$</td>
<td>47</td>
</tr>
<tr>
<td>4.6</td>
<td>Newton Type Method I with varying $\epsilon$ for fixed $\tau = 1$ and $\alpha = 0.00001$</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Confronting Newton Type Method I with Newton Type Method II</td>
<td>54</td>
</tr>
<tr>
<td>4.8</td>
<td>Time-Marching Scheme applied on the $32 \times 32 \times 20$ 3D image</td>
<td>59</td>
</tr>
<tr>
<td>4.9</td>
<td>Newton Type Method I applied on the 3D image with fixed $\alpha = 0.0001$</td>
<td>61</td>
</tr>
<tr>
<td>4.10</td>
<td>Newton Type Method I applied on the 3D image with fixed $\alpha = 0.0001$</td>
<td>63</td>
</tr>
</tbody>
</table>
List of Figures

1.1 Multi-modal image registration: left CT, middle PET, and right aligned image [Wik07] 2
1.2 The registration of two squares by rotation and translation is not unique ........ 3

3.1 Various mathematical abstractions like GridFE, FieldFE, Degree-Of-Freedom han-
der, LinEqSys and other classes simplify developers daily life ...................... 13
3.2 The index $i$ refers to the nodes in the grid and corresponds to the $x$ axis. Here the
graph illustrates the basis functions $N_3,N_4,N_5$ and $N_6$. .......................... 15
3.3 Each element matrix is added to the according place in the global matrix ...... 18
3.4 Diffpack classes and functions. .............................................................. 20
3.5 Example of a more extensive input file. .................................................. 26
3.6 Extension of the program structure to non-linearity. ............................... 30

4.1 The two images to be registered and its difference image ....................... 35
4.2 Fixed-Point Iteration with $\alpha = 0.01$ .................................................. 36
4.3 Fixed-Point Iteration with $\alpha = 0.1$ ..................................................... 36
4.4 Fixed-Point Iteration with $\alpha = 0.5$; the RegError steadies at 0.9 ............. 37
4.5 Fixed-Point Iteration with $\alpha = 1.0$; the RegError does not fall below 0.95 .... 37
4.6 Fixed-Point Iteration with $\alpha = 5.0$ ..................................................... 38
4.7 Over-relaxation of Fixed-Point Iteration with $\alpha = 0.1$ ......................... 39
4.8 Time-Marching Scheme with $\alpha = 0.1$ and $\tau = 1$ ............................. 40
4.9 Time-Marching Scheme with $\alpha = 0.001$ and $\tau = 1$ ............................ 41
4.10 Time-Marching Scheme with $\alpha = 0.00001$ and $\tau = 1$ yields a good result .......... 41
4.11 Time-Marching Scheme with $\alpha = 0.0000001$ and $\tau = 1$ shows no difference to
    $\alpha = 0.00001$ and $\tau = 1$ .................................................................... 42
4.12 Time-Marching Scheme with $\tau = 2$ and $\alpha = 0.00001$ ......................... 43
4.13 Time-Marching Scheme with $\tau = 5$ and $\alpha = 0.00001$ ......................... 43
4.14 Time-Marching Scheme with $\tau = 10$ and $\alpha = 0.00001$ ....................... 44
4.15 Time-Marching Scheme with $\tau = 20$ and $\alpha = 0.00001$ ....................... 44
4.16 Time-Marching Scheme with $\tau = 50$ and $\alpha = 0.00001$ ....................... 45
4.17 Difference images before and after application of Time-Marching Scheme ...... 46
4.18 The solution \( u \) obtained from the Time-Marching Scheme ............... 46
4.19 Newton Type Method I with initial step size \( \tau = 1 \) and fixed \( \epsilon = 1 \) ............... 48
4.20 Newton Type Method I with initial step size \( \tau = 4 \) and fixed \( \epsilon = 1 \) ............... 48
4.21 Newton Type Method I with initial step size \( \tau = 10 \) and fixed \( \epsilon = 1 \); to avoid an increase of the RegError the re-roll mechanism takes effect ............... 49
4.22 Newton Type Method I with \( \epsilon = 1, \tau = 1 \) and \( \alpha = 0.00001 \) ............... 49
4.23 Newton Type Method I with \( \epsilon = 0.4, \tau = 1 \) and \( \alpha = 0.00001 \) ....................... 50
4.24 Newton Type Method I with \( \epsilon = 0.1, \tau = 1 \) and \( \alpha = 0.00001 \) ....................... 50
4.25 Newton Type Method I with \( \epsilon = 0.04, \tau = 1 \) and \( \alpha = 0.00001 \) ....................... 51
4.26 Newton Type Method I with \( \epsilon = 0.01, \tau = 1 \) and \( \alpha = 0.00001 \) ....................... 51
4.27 Newton Type Method I for \( \epsilon = 0.01 \) without re-roll mechanism ....................... 52
4.28 Newton Type Method I with \( \epsilon = 0.1, \alpha = 0.00001 \) ....................... 53
4.29 Newton Type Method II with \( \epsilon = 0.1 \) and \( \delta = 0.9 \), \( \alpha = 0.00001 \) ....................... 54
4.30 Newton Type Method II with \( \epsilon = 0.1 \) and \( \delta = 0.2 \), \( \alpha = 0.00001 \); compared with Fig. 4.28 only a very small improvement is achieved ....................... 55
4.31 \( 256 \times 256 \)-slices of the 3D image ....................... 56
4.32 Time-Marching Scheme applied on the 3D image with \( \alpha = 1 \) and \( \tau = 1 \) ....................... 56
4.33 Time-Marching Scheme applied on the 3D image with \( \alpha = 0.00001 \) and \( \tau = 1 \) ............... 57
4.34 Time-Marching Scheme applied on the 3D image with \( \tau = 10 \) and \( \alpha = 0.0001 \) .......... 57
4.35 Time-Marching Scheme applied on the 3D image with \( \tau = 100 \) and \( \alpha = 0.0001 \) .......... 58
4.36 Time-Marching Scheme applied on the 3D image with \( \tau = 200 \) and \( \alpha = 0.0001 \) .......... 58
4.37 Time-Marching Scheme applied on the 3D image with \( \tau = 500 \) and \( \alpha = 0.0001 \) .......... 59
4.38 Newton Type Method I applied on the 3D image with \( \epsilon = 0.1 \) and fixed \( \alpha = 0.0001 \) .......... 60
4.39 Newton Type Method I applied on the 3D image with \( \epsilon = 0.01 \) and fixed \( \alpha = 0.0001 \) .......... 60
4.40 Newton Type Method I applied on the 3D image with \( \epsilon = 0.001 \) and fixed \( \alpha = 0.0001 \) .......... 61
4.41 Newton Type Method I with \( \epsilon = 0.01 \) and fixed \( \alpha = 0.0001 \) ....................... 62
4.42 Newton Type Method II with \( \delta = 0.6 \) and \( \epsilon = 0.01 \) and fixed \( \alpha = 0.0001 \) ....................... 62
4.43 Newton Type Method II with \( \delta = 0.99 \) and \( \epsilon = 0.01 \); the influence of \( \delta \) affects the convergence rate negatively ....................... 63
Listings

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>The Poisson0 header file</td>
<td>22</td>
</tr>
<tr>
<td>3.2</td>
<td>The scan() function</td>
<td>23</td>
</tr>
<tr>
<td>3.3</td>
<td>The fillEssBC() function for non-homogenous boundary conditions</td>
<td>23</td>
</tr>
<tr>
<td>3.4</td>
<td>The integrands() function</td>
<td>24</td>
</tr>
<tr>
<td>3.5</td>
<td>The solveProblem() function</td>
<td>25</td>
</tr>
<tr>
<td>3.6</td>
<td>The integrands() function for elastic image registration</td>
<td>28</td>
</tr>
<tr>
<td>3.7</td>
<td>The solveProblem() function for the time marching approach</td>
<td>31</td>
</tr>
<tr>
<td>3.8</td>
<td>The makeAndSolveLinearSystem() function for the time marching approach</td>
<td>32</td>
</tr>
<tr>
<td>3.9</td>
<td>The function Extendb() called in each outer iteration step to accomplish the time-marching approach</td>
<td>32</td>
</tr>
</tbody>
</table>
Bibliography


