UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 361

Visual tracking of soft tissue targets in sequences of 3D ultrasound images

Petar Palašek

Rennes/Zagreb, July 2012.

I would like to thank both of my supervisors: Prof. Siniša Šegvić, Ph.D., for guiding me throughout all of my projects and for showing me what all a computer connected to a camera can do; and Alexandre Krupa, Ph.D., for helping me deal with high ultrasound noise with his useful comments and suggestions, all in a friendly working atmosphere. I would also like to thank François Chaumette, Ph.D., for letting me stay for four months with the Lagadic team at INRIA, Rennes, without me knowing any French. To all other Lagadicians I met during my internship: thank you for being friends not just colleagues, for driving me to the restaurant every day and, of course, for Cloclo. Now for the Croatian part. I would like to thank my colleagues for being the crème de la crème, and my friends for understanding that this hasn't been an easy job. I would like to thank my parents for being supportive, for educating me and doing everything they did for me, even though I won't get over the fact that I didn't have a trampoline when I was a kid. I would also like to thank my older brother for being an older brother and letting me graduate first. Last but not least, I would like to thank Katarina for the last two years of our life, especially for the four months when I wasn't there.

Thank you, people - hvala, ljudi!

Contents

1. Introduction

There is a number of advantages for using ultrasound for acquiring medical images compared to, for example, magnetic resonance imaging (MRI) or computed tomography (CT). Some of the advantages are the low price and portability of the device, the high speed of image acquisition, its safety for the human health and the fact that it doesn't interact with ferromagnetic materials [9].

In this work we are interested in tracking non-rigid deformations of soft tissue structures in sequences of three-dimensional ultrasound images, caused by physical motions such as breathing, beating of the heart or some external motion. If we were able to track a selected part of a deforming tissue in an ultrasound volume throughout time, we could, for example, keep this part of the tissue in place during some kind of medical treatment.

The main drawback of ultrasound images is their low signal to noise ratio. In order to track the deformations of soft tissues in such conditions, Lee and Krupa suggest the usage of an intensity-based tracking method with the thin-plate spline warp as the motion model, as described in [9]. The idea is to put a number of control points in the ultrasound volume, track them through the sequence and use their locations to find the deformations of the tracked tissue. The problem with their method is that the motion model explodes when there is too much of noise present in the images. The goal of this work is to reimplement the procedures described in their paper and to improve the tracking where it previously failed.

The methods tried in this work in order to improve the previous tracking procedure were adding regularization to the thin-plate spline warp motion model used in the tracker and physically constraining the movement of control points using a mass-spring system.

The work is structured as follows. First we describe the thin-plate spline warps in Chapter 2, which is followed by the explanation of intensity-based tracking in Chapter 3. In order to test our tracker implementation we had to generate some ground-truth data, which is done as explained in Chapter 4. The mass-spring

system is described in Chapter 5, followed by the discussion of the experimental results in Chapter 6. The implementation details are given in Appendix A.

2. Thin-plate spline warps

To explain the idea and the purpose of thin-plate splines warps, we first have to see what a warp and a thin-plate spline are. To be able to do so, we will first define radial basis functions and explain how they can be used for interpolating values over scattered points of data. After that, we will introduce thin-plate splines, and finally the thin-plate spline warps. All of this will be briefly described in the following sections. Let's start!

2.1. Radial basis functions

We say that a function $\phi(\boldsymbol{x}, \boldsymbol{c}) : \mathbb{R}_+ \to \mathbb{R}$ is a radial basis function (RBF) if it has the form

$$\phi(\boldsymbol{x}, \boldsymbol{c}) = \phi\left(||\boldsymbol{x} - \boldsymbol{c}||\right), \tag{2.1}$$

where the norm $||\cdot||$ usually represents Euclidean distance. In other words, a function is a radial basis function if its value depends only on the distance of $\boldsymbol{x} \in \mathbb{R}^d$ from the point $\boldsymbol{c} \in \mathbb{R}^d$, which is usually refered to as the *center*.

2.1.1. Radial basis function interpolation

Given an unknown function $f(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}$ and a set of n_c points, $\boldsymbol{c}_i$, at which the values of the function f are known, we can define the interpolation problem as the problem of finding a function $s(\boldsymbol{x}) : \mathbb{R}^d \to \mathbb{R}$ which satisfies the interpolation conditions

$$s(\boldsymbol{c}_i) = f(\boldsymbol{c}_i) \quad \forall i \in \{1, 2, \ldots, n\}. \tag{2.2}$$

The function $s(\boldsymbol{x})$ is called the *interpolant* and it is known that it can be found by summing up n_c weighted radial basis functions [3], as shown in the following equation

$$f(\boldsymbol{x}) \approx s(\boldsymbol{x}) = \sum_{i=1}^{n_c} w_i \phi(\boldsymbol{x}, \boldsymbol{c}_i). \tag{2.3}$$

The w_i in the equation represent the weight coefficients and $\boldsymbol{c}_i$ represent the centers of the radial basis functions. The centers are chosen at the points where the values of the function f are known and the weight coefficients can be calculated by putting the interpolation conditions (2.2) into equation (2.3) [3], and then solving the system

$$\begin{pmatrix} f(\boldsymbol{c}_1) \\ \vdots \\ f(\boldsymbol{c}_{n_c}) \end{pmatrix} = \begin{pmatrix} \Phi_{1,1} & \dots & \Phi_{1,n_c} \\ \vdots & \ddots & \vdots \\ \Phi_{n_c,1} & \dots & \Phi_{n_c,n_c} \end{pmatrix} \begin{pmatrix} w_1 \\ \vdots \\ w_{n_c} \end{pmatrix}, \tag{2.4}$$

where

$$\Phi_{i,j} = \phi(\boldsymbol{c}_i, \boldsymbol{c}_j). \tag{2.5}$$

Without going into proofs and details, we will just state that the matrix $\boldsymbol{\Phi}$ is always nonsingular for *some choices* of the radial basis function type, if the centers are distinct, resulting in a uniquely defined coefficients vector, given by

$$\boldsymbol{w} = \boldsymbol{\Phi}^{-1}\boldsymbol{f}. \tag{2.6}$$

For a detailed discussion about the nonsingularity of the matrix $\boldsymbol{\Phi}$ and the existance of a unique interpolant of form (2.3), refer to [15], [1] and [11].

Again without any proof and details, we will introduce a more general form of the interpolant equation

$$f(\boldsymbol{x}) \approx s(\boldsymbol{x}) = \sum_{i=1}^{n_c} w_i \phi(\boldsymbol{x}, \boldsymbol{c}_i) + P(\boldsymbol{x}), \quad P \in \Pi_m(\mathbb{R}^d), \tag{2.7}$$

where $\Pi_m(\mathbb{R}^d)$ represents the vector space of polynomials in d real variables of total degree m [1]. The choice of the value of m depends on the choice of the radial basis function type ϕ and will be mentioned in the following subsection. Notice that the equation (2.7) comes to the form of (2.3) in cases when there is no polynomial term, that is, when $m = 0$. For $m > 1$ the polynomial term in (2.7) adds $q = \frac{(m-1+d)!}{d!(m-1)!}$ degrees of freedom [11] which are eliminated by satisfying q conditions [1]:

$$\sum_{i=1}^{n_c} w_i p(\boldsymbol{c}_i) = 0, \tag{2.8}$$

for all the terms p in the polynomial P. By combining (2.7) with the conditions (2.8), we can write a linear system

$$\begin{pmatrix} \boldsymbol{\Phi} & \boldsymbol{C} \\ \boldsymbol{C}^T & \boldsymbol{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{w} \\ \boldsymbol{a} \end{pmatrix} = \begin{pmatrix} \boldsymbol{f} \\ \boldsymbol{0} \end{pmatrix}, \tag{2.9}$$

where $\boldsymbol{\Phi}$ represents the matrix defined in (2.5), the i-th row of $\boldsymbol{C}$ is defined as $\boldsymbol{C}_i = \begin{pmatrix} 1 & \boldsymbol{c}_i \end{pmatrix}$, w_i are the *RBF* weight coefficients and a_i are the coefficients of the polynomial term in (2.7). Using this system we can find the coefficient vectors $\boldsymbol{w}$ and $\boldsymbol{a}$ which are needed to define the interpolant (2.7). The sizes of matrices used in (2.9) are going to be discussed later in Section 2.2, when we choose a concrete *RBF* type to be used in (2.7).

2.1.2. Radial basis function types

There exists a number of different radial basis functions and some of the most used are listed in Table 2.1. The rightmost column in the table shows the values of m, the total degree of the polynomial term used in (2.7), depending on the choice of the *RBF* type ϕ. We can see that $m = 0$ when we use the Gaussian or Inverse multiquadratic *RBF*, so there is no polynomial term added in the interpolant and (2.7) comes to the form of (2.3). These are the cases when $\boldsymbol{\Phi}$ is invertible and the weight vector w can be found using equation (2.6). When

Table 2.1: Some of the different types of radial basis functions [11]

RBF type	$\phi(r)$, $r = \|\|\boldsymbol{x} - \boldsymbol{c}\|\|$	β conditions	$m(\phi)$
Gaussian	e^{-r^2}	—	0
Multiquadratic	$(-1)^{\lceil \beta/2 \rceil}(1 + r^2)^{\beta/2}$	$\beta > 0, \beta \notin 2\mathbb{N}$	$\lceil \beta/2 \rceil$
Inverse multiquadratic	$(1 + r^2)^{\beta/2}$	$\beta < 0$	0
Polyharmonic spline	$(-1)^{\lceil \beta/2 \rceil} r^\beta$	$\beta > 0, \beta \notin 2\mathbb{N}$	$\lceil \beta/2 \rceil$
	$(-1)^{1+\beta/2} r^\beta \log r$	$\beta > 0, \beta \in 2\mathbb{N}$	$1 + \beta/2$

using the Multiquadratic or Polyharmonic spline *RBF*, m is greater than zero and a polynomial term exists in (2.7). The *RBF* that we are going to use in this work is of the polyharmonic spline type, with β chosen to be 2. It has the form

$$\phi(r) = r^2 \log r, \quad r = ||\boldsymbol{x} - \boldsymbol{c}|| \tag{2.10}$$

and it is called the *thin-plate spline*. It should be noted that (2.10) is defined this way in $\mathbb{R}^2$. The $\mathbb{R}^3$ form will be introduced later.

2.2. Thin-plate splines

As defined in the previous subsection, the thin-plate spline (*TPS*) is a special type of polyharmonic splines, with $\beta = 2$. Thin-plate splines were first presented

by Harder and Desmarais in [8] and were later mathematically formalized by Duchon [5] and Meinguet [12]. Thin-plate splines are known for their property of minimizing bending energy of a thin metal plate which will be briefly described in the following subsection.

2.2.1. Physical analogy

Imagine an infinite thin metal plate and four points lying on it, forming a square. Now take four hooks and put one of them into one of the four defined points, so that they can be pulled in directions perpendicular to the plate. Pull the two hooks on one diagonal of the square in the direction of the plate's normal and pull the other two hooks in the opposite direction. The plate will bend and take a shape similar to the one shown in Figure 2.1, which is the shape that minimizes the plate's bending energy.

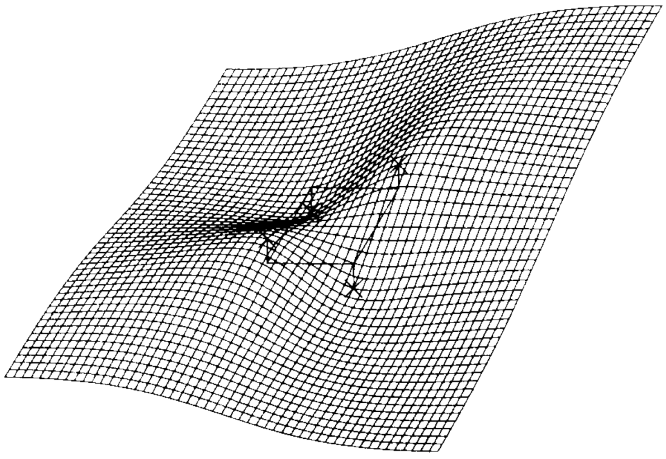


Figure 2.1: An infinite thin metal plate constrained by four hooks. Two of the hooks on one diagonal are pulling the plate in one direction and the other two are pulling the plate into the other direction. The shape that the plate takes minimizes its bending energy. This image was borrowed from [2].

2.2.2. Two dimensions

We can look at the thin metal plate shown in Figure 2.1 as a function $f : \mathbb{R}^2 \to \mathbb{R}$ which defines the height of the plate at every location $\boldsymbol{x} \in \mathbb{R}^2$ lying on the plate. In our example, the heights are constrained at four locations $\boldsymbol{c}_i$ shown as hooks in the figure. We want to find an interpolant s which will find the heights of the plate at unknown locations given the heights at points $\boldsymbol{c}_i$, minimizing the

bending energy at the same time. The energy which we want to minimize can be expressed by the equation

$$I_s = \iint_{\mathbb{R}^2} \left(\frac{\partial^2 s}{\partial x^2}\right)^2 + 2\left(\frac{\partial^2 s}{\partial x \partial y}\right)^2 + \left(\frac{\partial^2 s}{\partial y^2}\right)^2 dxdy. \tag{2.11}$$

It has been proven by Duchon [5] that the function s which minimizes this energy has the form

$$s(x,y) = a_1 + a_2 x + a_3 y + \sum_{i=1}^{n_c} w_i \phi_{2D}(||(x,y) - (x_{c_i}, y_{c_i})||), \tag{2.12}$$

where

$$\phi_{2D}(r) = r^2 \log r, \quad r = ||\boldsymbol{x} - \boldsymbol{c}||. \tag{2.13}$$

We can see that s is actually the interpolant defined in (2.7) with the *RBF* (2.13) chosen to be the thin-plate spline *RBF* defined in (2.10). In order for s to have square integrable second derivatives [4], [2] we add the following constraints

$$\sum_{i=1}^{n_c} w_i = \sum_{i=1}^{n_c} w_i x_i = \sum_{i=1}^{n_c} w_i y_i = 0, \tag{2.14}$$

which are the conditions mentioned previously in (2.8). If we define a vector $\boldsymbol{v}$ containing the known values of the function f at points $\boldsymbol{c}_i$, $v_i = f(\boldsymbol{c}_i)$, we can combine it with (2.12) and the constraints (2.14) to get the system

$$\begin{pmatrix} \boldsymbol{\Phi}_{n_c \times n_c} & \boldsymbol{C}_{n_c \times 3} \\ \boldsymbol{C}^T_{3 \times n_c} & \boldsymbol{0}_{3\times 3} \end{pmatrix} \begin{pmatrix} \boldsymbol{w}_{n_c \times 1} \\ \boldsymbol{a}_{3\times 1} \end{pmatrix} = \begin{pmatrix} \boldsymbol{v}_{n_c \times 1} \\ \boldsymbol{0}_{3\times 1} \end{pmatrix}. \tag{2.15}$$

The matrices in (2.15) are defined analogous to the ones defined in (2.9). Solving this system gives us the vectors $\boldsymbol{w}$ and $\boldsymbol{a}$, which then uniquely define the interpolant (2.12). We can see that the number of parameters needed to define the *TPS* interpolant depends on the number of chosen centers n_c. In two dimensions the interpolant is determined by $n_c + 3$ parameters.

2.2.3. Taking it into three dimensions

As we have to deal with three-dimensional data in our work, we will have to extend the interpolation described in the previous subsection by adding one more dimension. Unfortunately, the interpolant in three dimensions cannot be shown as nicely as the two-dimensional one pictured by a metal plate in Figure 2.1. As Bookstein says in [2], the equivalent of the metal plate for the three-dimensional case is a slightly bent hyperplane in Euclidean four-space and will thus require

considerable imagination, because it is not really clear how to draw it. However, extending the two-dimensional interpolant into three dimensions can be done without any major changes to what we have previously defined.

The bending energy which our three-dimensional interpolant has to minimize is the following [16]:

$$I_s = \iiint_{\mathbb{R}^3} \left(\frac{\partial^2 s}{\partial x^2}\right)^2 + \left(\frac{\partial^2 s}{\partial y^2}\right)^2 + \left(\frac{\partial^2 s}{\partial z^2}\right)^2 + 2\left[\left(\frac{\partial^2 s}{\partial x \partial y}\right)^2 + \left(\frac{\partial^2 s}{\partial x \partial z}\right)^2 + \left(\frac{\partial^2 s}{\partial y \partial z}\right)^2\right] dxdydz, \tag{2.16}$$

which is achieved by a function of the form

$$s(x, y, z) = a_1 + a_2 x + a_3 y + a_4 z + \sum_{i=1}^{n_c} w_i \phi_{3D}(||(x, y, z) - (x_{c_i}, y_{c_i}, z_{c_i})||). \tag{2.17}$$

According to [2], the *RBF* used in (2.17) is now

$$\phi_{3D}(r) = r, \quad r = ||\boldsymbol{x} - \boldsymbol{c}||. \tag{2.18}$$

One more constraint is added to the ones defined in (2.14):

$$\sum_{i=1}^{n_c} w_i z_i = 0 \tag{2.19}$$

and again a system similar to (2.15) can be written:

$$\begin{pmatrix} \boldsymbol{\Phi}_{n_c \times n_c} & \boldsymbol{C}_{n_c \times 4} \\ \boldsymbol{C}^T_{4 \times n_c} & \boldsymbol{0}_{4 \times 4} \end{pmatrix} \begin{pmatrix} \boldsymbol{w}_{n_c \times 1} \\ \boldsymbol{a}_{4 \times 1} \end{pmatrix} = \begin{pmatrix} \boldsymbol{v}_{n_c \times 1} \\ \boldsymbol{0}_{4 \times 1} \end{pmatrix}. \tag{2.20}$$

Solving this system gives us the vectors $\boldsymbol{w}$ and $\boldsymbol{a}$, needed to determine the interpolant (2.17). In the three-dimensional case the interpolant depends on $n_c + 4$ parameters.

2.3. The thin-plate spline warp

We are given a set of n_c points lying on a plane (in $\mathbb{R}^2$), $A = \{\boldsymbol{a}_1, \boldsymbol{a}_2, \ldots, \boldsymbol{a}_{n_c}\}$. Let us take these points and move them around the plane a bit to get a new set of points, $B = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_{n_c}\}$. As we know the point correspondences from set A to set B (we know which point from A has moved to which location in B), we can define a function $m(\boldsymbol{x}) : \mathbb{R}^2 \to \mathbb{R}^2$ which will map the points from A into points from B. This mapping function is called a *warp* and once found, it can be

used to move all the points on a plane according to the movement of just a set of control points.

The usage of thin-plate spline warps for modelling deformations of biological images has been described in detail by Bookstein in [2]. The main idea of defining a warp is to observe the changes of a point's location separately in every dimension. Following this principle, the *thin-plane spline warp* in two dimensions if written as

$$m_{2D}(\boldsymbol{x}) = \begin{pmatrix} s^x(\boldsymbol{x}) \\ s^y(\boldsymbol{x}) \end{pmatrix} = \begin{pmatrix} a_1^x & a_2^x & a_3^x \\ a_1^y & a_2^y & a_3^y \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \end{pmatrix} + \sum_{i=1}^{n_c} \begin{pmatrix} w_i^x \\ w_i^y \end{pmatrix} \phi_{2D}(||\boldsymbol{x} - \boldsymbol{c}_i||), \quad (2.21)$$

where $s^x(\boldsymbol{x})$ represents the two-dimensional *TPS* interpolation function (2.12) describing the displacements in the x dimension, and similarly, $s^y(\boldsymbol{x})$ describing the displacements in the y dimension. Notice that the centers $\boldsymbol{c}_i$ are the same for both splines. As the *TPS* warp defined in (2.21) consists of two *TPS* interpolants (2.12), each of which depends on $n_c + 3$ parameters, we can see that the two-dimensional *TPS* warp depends on $2 \cdot (n_c + 3)$ parameters. These parameters can be calculated by solving a system similar to (2.15):

$$\begin{pmatrix} \boldsymbol{\Phi}_{n_c \times n_c} & \boldsymbol{C}_{n_c \times 3} \\ \boldsymbol{C}^T_{3 \times n_c} & \boldsymbol{0}_{3\times 3} \end{pmatrix} \begin{pmatrix} \boldsymbol{w}^x_{n_c \times 1} & \boldsymbol{w}^y_{n_c \times 1} \\ \boldsymbol{a}^x_{3\times 1} & \boldsymbol{a}^y_{3\times 1} \end{pmatrix} = \begin{pmatrix} \boldsymbol{v}^x_{n_c \times 1} & \boldsymbol{v}^y_{n_c \times 1} \\ \boldsymbol{0}_{3\times 1} & \boldsymbol{0}_{3\times 1} \end{pmatrix}. \quad (2.22)$$

The matrices and vectors used in this system have been defined in previous sections.

2.3.1. The warp in three dimensions

Following the same ideas which were used to define the two-dimensional *TPS* warp, we can define the warp in three dimensions as

$$m_{3D}(\boldsymbol{x}) = \begin{pmatrix} s^x(\boldsymbol{x}) \\ s^y(\boldsymbol{x}) \\ s^z(\boldsymbol{x}) \end{pmatrix} = \begin{pmatrix} a_1^x & a_2^x & a_3^x & a_4^x \\ a_1^y & a_2^y & a_3^y & a_4^y \\ a_1^z & a_2^z & a_3^z & a_4^z \end{pmatrix} \begin{pmatrix} 1 \\ x \\ y \\ z \end{pmatrix} + \sum_{i=1}^{n_c} \begin{pmatrix} w_i^x \\ w_i^y \\ w_i^z \end{pmatrix} \phi_{3D}(||\boldsymbol{x} - \boldsymbol{c}_i||), \quad (2.23)$$

where the functions s^x, s^y, s^z refer to three three-dimensional *TPS* interpolants defined in (2.17). As each of the interpolants needs n_c+4 parameters to be defined, the three-dimensional *TPS* warp will be defined with $3 \cdot (n_c + 4)$ parameters. The

system used to find the parameters is then written as:

$$\begin{pmatrix} \boldsymbol{\Phi}_{n_c \times n_c} & \boldsymbol{C}_{n_c \times 4} \\ \boldsymbol{C}^T_{4 \times n_c} & \boldsymbol{0}_{4\times 4} \end{pmatrix} \begin{pmatrix} \boldsymbol{w}^x_{n_c\times 1} & \boldsymbol{w}^y_{n_c\times 1} & \boldsymbol{w}^z_{n_c\times 1} \\ \boldsymbol{a}^x_{4\times 1} & \boldsymbol{a}^y_{4\times 1} & \boldsymbol{a}^z_{4\times 1} \end{pmatrix} = \begin{pmatrix} \boldsymbol{v}^x_{n_c\times 1} & \boldsymbol{v}^y_{n_c\times 1} & \boldsymbol{v}^z_{n_c\times 1} \\ \boldsymbol{0}_{4\times 1} & \boldsymbol{0}_{4\times 1} & \boldsymbol{0}_{4\times 1} \end{pmatrix}. \quad (2.24)$$

2.3.2. Using the warp to find new point locations

Let us examine a simple example of how the 3D *TPS* warp can be used. Imagine a cube defined in a 3D discrete grid, consisting of n points. Now we will add n_c more points inside the cube and call them control points. The i-th point in the cube is refered to as $\boldsymbol{p}_i$ and the i-th control point is $\boldsymbol{c}_i$. We know the locations of both all of the n cube points, $\boldsymbol{p}_i = (x_{p_i}, y_{p_i}, z_{p_i})$, and all of the n_c control points, $\boldsymbol{c}_i = (x_{c_i}, y_{c_i}, z_{c_i})$. Imagine now that the control points move a bit, each of them in a random direction. We want to find out where we should move the n cube points, so that the points follow the movement of the control points. If we know the new locations of each of the control points, $\boldsymbol{c}'_i$, we can find the new cube point locations as follows.

First we will write the warp function (2.23) in matrix form as

$$\begin{pmatrix} \boldsymbol{x}^{p'}_{n\times 1} & \boldsymbol{y}^{p'}_{n\times 1} & \boldsymbol{z}^{p'}_{n\times 1} \end{pmatrix} = \begin{pmatrix} \boldsymbol{B}_{n\times n_c} & \boldsymbol{Q}_{n\times 4} \end{pmatrix} \begin{pmatrix} \boldsymbol{w}^x_{n_c\times 1} & \boldsymbol{w}^y_{n_c\times 1} & \boldsymbol{w}^z_{n_c\times 1} \\ \boldsymbol{a}^x_{4\times 1} & \boldsymbol{a}^y_{4\times 1} & \boldsymbol{a}^z_{4\times 1}, \end{pmatrix} \quad (2.25)$$

where we define the matrix $\boldsymbol{B}$ as $B_{i,j} = \phi_{3D}\left(||\boldsymbol{p}_j - \boldsymbol{c}_i||\right)$, the j-th row in matrix $\boldsymbol{Q}$ as $\boldsymbol{Q}_j = \begin{pmatrix} 1 & x_{p_j} & y_{p_j} & z_{p_j} \end{pmatrix}$ and the vectors $\boldsymbol{x}^{p'}$, $\boldsymbol{y}^{p'}$ and $\boldsymbol{z}^{p'}$ contain the new x, y and z coordinates of point $\boldsymbol{p}_i$ in the i-th row. Now, if we denote the leftmost matrix in (2.24) as $\boldsymbol{K}$, and the $(n_c + 4) \times n_c$ submatrix of the inverse $\boldsymbol{K}^{-1}$ as $\boldsymbol{K}_*$, then we can find the parameters matrix as

$$\begin{pmatrix} \boldsymbol{w}^x_{n_c\times 1} & \boldsymbol{w}^y_{n_c\times 1} & \boldsymbol{w}^z_{n_c\times 1} \\ \boldsymbol{a}^x_{4\times 1} & \boldsymbol{a}^y_{4\times 1} & \boldsymbol{a}^z_{4\times 1}, \end{pmatrix} = \boldsymbol{K}_* \begin{pmatrix} \boldsymbol{x}^{c'}_{n_c\times 1} & \boldsymbol{y}^{c'}_{n_c\times 1} & \boldsymbol{z}^{c'}_{n_c\times 1} \end{pmatrix}, \quad (2.26)$$

where the vectors $\boldsymbol{x}^{c'}$, $\boldsymbol{y}^{c'}$ and $\boldsymbol{z}^{c'}$ contain the new x, y and z coordinates of the control point $\boldsymbol{c}'_i$ in the i-th row. At last, if we denote the $\begin{pmatrix} \boldsymbol{B} & \boldsymbol{Q} \end{pmatrix}$ matrix in (2.25) as $\boldsymbol{M}$, we can calculate the new positions of all the n cube points using the equation

$$\begin{pmatrix} \boldsymbol{x}^{p'}_{n\times 1} & \boldsymbol{y}^{p'}_{n\times 1} & \boldsymbol{z}^{p'}_{n\times 1} \end{pmatrix} = \boldsymbol{M}\boldsymbol{K}_* \begin{pmatrix} \boldsymbol{x}^{c'}_{n_c\times 1} & \boldsymbol{y}^{c'}_{n_c\times 1} & \boldsymbol{z}^{c'}_{n_c\times 1} \end{pmatrix}. \quad (2.27)$$

If we denote the leftmost matrix in (2.27) as $\boldsymbol{P}'$ and the rightmost matrix as $\boldsymbol{C}'$, we will be able to write the equation in a simpler way, as $\boldsymbol{P}' = \boldsymbol{M}\boldsymbol{K}_*\boldsymbol{C}'$.

The only problem left is what to do if we don't know the new control point locations. This will be discussed in the following chapter.

3. Intensity-based tracking

As it was described in the example at the end of the previous chapter (2.3.2), by knowing the initial control point positions and the positions of control points in the volume at the current time, we can calculate the $3 \cdot (n_c + 4)$ parameters needed by the 3D *TPS* warp defined in (2.23) and then use the warp to find new locations of other points in the volume. This means that we can select a region of interest in the initial volume and estimate its deformation through time only by knowing the positions of the control points. The remaining problem is to find the control point locations in the current volume in the sequence. To be able to do this, we must use some kind of visual tracking. Because the signal to noise ratio in ultrasound images is very low, it is hard to extract trackable features, so direct pixel intensity values are used instead.

In this work we are using the same method as described in [9], which is based on the 2D region tracking method presented in [7]. The used tracking method will be described in the following sections.

3.1. The motion model and the tracking region

First of all, we will define the position of a point $\boldsymbol{p}$ by using a parametric *motion model*

$$\boldsymbol{p} = f(\boldsymbol{p}_0, \boldsymbol{\mu}(t)), \tag{3.1}$$

where $\boldsymbol{p}_0$ represents the initial position of the point, and $\boldsymbol{\mu}(t)$ represents a vector of motion parameters at given time t. In our case, the motion model is the 3D *TPS* warp defined in (2.23), where the motion parameters are actually the control point positions that we are trying to estimate. We can write the motion parameter vector as

$$\boldsymbol{\mu}(t) = \begin{pmatrix} x_{c_1}(t) & y_{c_1}(t) & z_{c_1}(t) & \dots & x_{c_{n_c}}(t) & y_{c_{n_c}}(t) & z_{c_{n_c}}(t) \end{pmatrix}^T, \tag{3.2}$$

where $x_{c_i}(t)$, $y_{c_i}(t)$ and $z_{c_i}(t)$ are respectively the x, y and z coordinates of the control point $\boldsymbol{c}_i$ at time t, and n_c is the number of control points. As mentioned a few times before, by knowing the control point positions, *i.e.*, the motion parameters, we can find new positions of the other points that we want to track.

If we say that $I(\boldsymbol{p})$ gives us the intensity value at point $\boldsymbol{p}$, and if we mark the set of n points in the region that we want to track as $\boldsymbol{P} = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \ldots, \boldsymbol{p}_n\}$, we can define the tracking region $\boldsymbol{I}$ as a function of $\boldsymbol{P}$ that gives us a vector of intensities

$$\boldsymbol{I}(\boldsymbol{P}) = \begin{pmatrix} I(\boldsymbol{p}_1) & I(\boldsymbol{p}_2) & \ldots & I(\boldsymbol{p}_n) \end{pmatrix}^T. \tag{3.3}$$

If we combine (3.3) with our motion model (3.1) and mark the initial set of the tracked points as $\boldsymbol{P}_0 = \{\boldsymbol{p}_1^0, \boldsymbol{p}_2^0, \ldots, \boldsymbol{p}_n^0\}$, then the *tracking region* $\boldsymbol{I}$ can be defined as a function of $\boldsymbol{P}_0$ and the motion parameter vector $\boldsymbol{\mu}$ at time t, and we can write

$$\boldsymbol{I}(\boldsymbol{P}_0, \boldsymbol{\mu}(t)) = \begin{pmatrix} I(f(\boldsymbol{p}_1^0, \boldsymbol{\mu}(t))) \\ I(f(\boldsymbol{p}_2^0, \boldsymbol{\mu}(t))) \\ \vdots \\ I(f(\boldsymbol{p}_n^0, \boldsymbol{\mu}(t))) \end{pmatrix}. \tag{3.4}$$

The initial tracking region (set at time $t = t_0$) will be denoted as

$$\boldsymbol{I}^* = \boldsymbol{I}(\boldsymbol{P}_0, \boldsymbol{\mu}(t_0)) \tag{3.5}$$

and will be referred to as the *reference tracking region*.

3.2. Estimating the motion parameters

The goal of the intensity-based region tracking is to minimize the difference between the current tracking region $\boldsymbol{I}$ and the initial, reference tracking region $\boldsymbol{I}_*$. This can be described by an objective function

$$O(\boldsymbol{\mu}) = ||\boldsymbol{I}(\boldsymbol{P}_0, \boldsymbol{\mu}) - \boldsymbol{I}^*||^2, \tag{3.6}$$

where we are using the tracking regions as defined in (3.4) and (3.5).

According to [7], our problem of region tracking can be rewritten as a problem of finding a vector of offsets $\delta\boldsymbol{\mu}$ such that

$$\boldsymbol{\mu}(t + \tau) = \boldsymbol{\mu}(t) + \delta\boldsymbol{\mu} \tag{3.7}$$

can be written for an image at time $t + \tau$. Using this equation, we can redefine the objective function (3.6) as a function of $\delta\boldsymbol{\mu}$

$$O(\delta\boldsymbol{\mu}) = ||\boldsymbol{I}(\boldsymbol{P}_0, \boldsymbol{\mu}(t) + \delta\boldsymbol{\mu}) - \boldsymbol{I}^*||^2. \tag{3.8}$$

By expanding $\boldsymbol{I}(\boldsymbol{P}_0, \boldsymbol{\mu}(t) + \delta\boldsymbol{\mu})$ in a Taylor series about $\boldsymbol{\mu}$ and t we will get

$$\boldsymbol{I}(\boldsymbol{P}_0, \boldsymbol{\mu}(t) + \delta\boldsymbol{\mu}) = \boldsymbol{I}(\boldsymbol{P}_0, \boldsymbol{\mu}(t)) + \boldsymbol{J}_{\boldsymbol{\mu}}\delta\boldsymbol{\mu} + \tau\frac{\partial \boldsymbol{I}}{\partial t} + h.o.t., \tag{3.9}$$

where $\boldsymbol{J}_{\boldsymbol{\mu}}$ is the Jacobian matrix of $\boldsymbol{I}$ with respect to $\boldsymbol{\mu}$, which will be discussed more in detail in the following subsection.

By ignoring the higher order terms (marked as $h.o.t$) and including (3.9) in (3.8) we get

$$O(\delta\boldsymbol{\mu}) \approx ||\boldsymbol{I}(\boldsymbol{P}_0, \boldsymbol{\mu}(t)) + \boldsymbol{J}_{\boldsymbol{\mu}}\delta\boldsymbol{\mu} + \tau\frac{\partial \boldsymbol{I}}{\partial t} - \boldsymbol{I}^*||^2. \tag{3.10}$$

If we now approximate $\tau\frac{\partial \boldsymbol{I}}{\partial t}$ as

$$\tau\frac{\partial \boldsymbol{I}}{\partial t} \approx \boldsymbol{I}(\boldsymbol{P_0}, \boldsymbol{\mu}(t + \tau)) - \boldsymbol{I}(\boldsymbol{P_0}, \boldsymbol{\mu}(t)), \tag{3.11}$$

the equation (3.8) then becomes

$$O(\delta\boldsymbol{\mu}) \approx ||\boldsymbol{I}(\boldsymbol{P}_0, \boldsymbol{\mu}(t + \tau)) + \boldsymbol{J}_{\boldsymbol{\mu}}\delta\boldsymbol{\mu} - \boldsymbol{I}^*||^2. \tag{3.12}$$

Finally, by solving the equations $\nabla O = \boldsymbol{0}$, we get

$$\delta\boldsymbol{\mu} = -\boldsymbol{J}_{\boldsymbol{\mu}}^{+}(\boldsymbol{I}(\boldsymbol{P_0}, \boldsymbol{\mu}(t + \tau)) - \boldsymbol{I}^*), \tag{3.13}$$

where $\boldsymbol{J}_{\boldsymbol{\mu}}^{+}$ denotes the Moore-Penrose pseudo-inverse of $\boldsymbol{J}_{\boldsymbol{\mu}}$, defined as

$$\boldsymbol{J}_{\boldsymbol{\mu}}^{+} = (\boldsymbol{J}_{\boldsymbol{\mu}}^{T}\boldsymbol{J}_{\boldsymbol{\mu}})^{-1}\boldsymbol{J}_{\boldsymbol{\mu}}^{T}. \tag{3.14}$$

If we define an error vector as

$$\boldsymbol{e}(t + \tau) = \boldsymbol{I}(\boldsymbol{P}_0, \boldsymbol{\mu}(t + \tau)) - \boldsymbol{I}^*, \tag{3.15}$$

we can write the motion vector offset as

$$\delta\boldsymbol{\mu} = -\boldsymbol{J}_{\boldsymbol{\mu}}^{+}\boldsymbol{e}(t + \tau). \tag{3.16}$$

In the work described in [9] the motion vector offset is additionally scaled by a positive constant λ in order to ensure an exponential decrease of the error (3.15). This is then reffered to as the *velocity vector of motion parameters* and is written as

$$\boldsymbol{v}_{\mu} = -\lambda\boldsymbol{J}_{\boldsymbol{\mu}}^{+}\boldsymbol{e}(t + \tau). \tag{3.17}$$

3.2.1. Construction of the Jacobian matrix

Following the work described in [9], the Jacobian matrix can be calculated from our motion model f (3.1) and the 3D image gradient $\nabla_{\boldsymbol{P}_0}\boldsymbol{I}$ at time t using the equation

$$\boldsymbol{J}_{\boldsymbol{\mu}} = \frac{\partial \boldsymbol{I}}{\partial \boldsymbol{\mu}} = \frac{\partial \boldsymbol{I}}{\partial \boldsymbol{P}_0} \times \frac{\partial \boldsymbol{P}_0}{\partial \boldsymbol{P}} \times \frac{\partial \boldsymbol{P}}{\partial \boldsymbol{\mu}} = \nabla_{\boldsymbol{P}_0}\boldsymbol{I} \times f_{\boldsymbol{P}_0}^{-1} \times f_{\boldsymbol{\mu}}. \tag{3.18}$$

This matrix is a $n \times 3n_c$ matrix and here it is shown how to construct it row by row.

Calculating $\nabla_{\boldsymbol{P}_0}\boldsymbol{I}$

The first term in (3.18) is the 3D image gradient $\nabla_{\boldsymbol{P}_0}\boldsymbol{I}$. It is calculated at all of the points in the previously defined set $\boldsymbol{P}_0$ by using a $3 \times 3 \times 3$ Sobel operator at each point. For example, the gradient at point $\boldsymbol{p}_i^0$ that we will denote as $\nabla_{\boldsymbol{p}_i^0}\boldsymbol{I}$, will be a 1×3 row vector, containing the values of the gradient in x, y and z directions.

Calculating $f_{\boldsymbol{P}_0}^{-1}$

The second term, $f_{\boldsymbol{P}_0}^{-1}$, represents the inverse of the 3×3 Jacobian matrix of $f(\boldsymbol{p}_0, \boldsymbol{\mu}(t))$ regarded as a function of $\boldsymbol{p}_0$. This matrix is calculated at every point $\boldsymbol{p}_i^0$ in $\boldsymbol{P}_0$. If we denote the matrix $f_{\boldsymbol{P}_0}$ at point $\boldsymbol{p}_i^0$ as $f_{\boldsymbol{p}_i^0}$, we can define it as

$$f_{\boldsymbol{p}_i^0} = \left(\begin{array}{c|c|c} \frac{\partial f(\boldsymbol{p}_i^0, \boldsymbol{\mu}(t))}{\partial x_{p_i^0}} & \frac{\partial f(\boldsymbol{p}_i^0, \boldsymbol{\mu}(t))}{\partial y_{p_i^0}} & \frac{\partial f(\boldsymbol{p}_i^0, \boldsymbol{\mu}(t))}{\partial z_{p_i^0}} \end{array} \right) = \begin{pmatrix} \frac{\partial x_{p_i}}{\partial x_{p_i^0}} & \frac{\partial x_{p_i}}{\partial y_{p_i^0}} & \frac{\partial x_{p_i}}{\partial z_{p_i^0}} \\ \frac{\partial y_{p_i}}{\partial x_{p_i^0}} & \frac{\partial y_{p_i}}{\partial y_{p_i^0}} & \frac{\partial y_{p_i}}{\partial z_{p_i^0}} \\ \frac{\partial z_{p_i}}{\partial x_{p_i^0}} & \frac{\partial z_{p_i}}{\partial y_{p_i^0}} & \frac{\partial z_{p_i}}{\partial z_{p_i^0}} \end{pmatrix}, \tag{3.19}$$

where we have used $x_{p_i^0}$, $y_{p_i^0}$ and $z_{p_i^0}$ to denote respectively the x, y and z coordinates of the point $\boldsymbol{p}_i^0$, and x_{p_i}, y_{p_i} and z_{p_i} to denote the x, y and z coordinates of the point $\boldsymbol{p}$, calculated from the motion model $\boldsymbol{p} = f(\boldsymbol{p}_0, \boldsymbol{\mu}(t))$. If we now use the equation defined in (2.27), denoting the rightmost matrix in it as $\boldsymbol{C}'$, we can rewrite the equation (3.19) as

$$f_{\boldsymbol{p}_i^0} = \left[\begin{pmatrix} \frac{\partial \boldsymbol{M}_i}{\partial x_{p_i^0}} \\ \frac{\partial \boldsymbol{M}_i}{\partial y_{p_i^0}} \\ \frac{\partial \boldsymbol{M}_i}{\partial z_{p_i^0}} \end{pmatrix} \boldsymbol{K}_* \boldsymbol{C}' \right]^T = \boldsymbol{C}'^T \boldsymbol{K}_*^T \begin{pmatrix} \frac{\partial \boldsymbol{M}_i}{\partial x_{p_i^0}} \\ \frac{\partial \boldsymbol{M}_i}{\partial y_{p_i^0}} \\ \frac{\partial \boldsymbol{M}_i}{\partial z_{p_i^0}} \end{pmatrix}^T, \tag{3.20}$$

where we have denoted the i-th row of $\boldsymbol{M}$ as $\boldsymbol{M}_i$. To make it more clearer, we can expand the equation further to get

$$f_{\boldsymbol{p}_i^0} = \boldsymbol{C}'^T \boldsymbol{K}_*^T \begin{pmatrix} \frac{\partial \phi_{3D}(\boldsymbol{p}_i^0, \boldsymbol{c}_1)}{\partial x_{p_i^0}} & \frac{\partial \phi_{3D}(\boldsymbol{p}_i^0, \boldsymbol{c}_2)}{\partial x_{p_i^0}} & \dots & \frac{\partial \phi_{3D}(\boldsymbol{p}_i^0, \boldsymbol{c}_{n_c})}{\partial x_{p_i^0}} & 0 & 1 & 0 & 0 \\ \frac{\partial \phi_{3D}(\boldsymbol{p}_i^0, \boldsymbol{c}_1)}{\partial y_{p_i^0}} & \frac{\partial \phi_{3D}(\boldsymbol{p}_i^0, \boldsymbol{c}_2)}{\partial y_{p_i^0}} & \dots & \frac{\partial \phi_{3D}(\boldsymbol{p}_i^0, \boldsymbol{c}_{n_c})}{\partial y_{p_i^0}} & 0 & 0 & 1 & 0 \\ \frac{\partial \phi_{3D}(\boldsymbol{p}_i^0, \boldsymbol{c}_1)}{\partial z_{p_i^0}} & \frac{\partial \phi_{3D}(\boldsymbol{p}_i^0, \boldsymbol{c}_2)}{\partial z_{p_i^0}} & \dots & \frac{\partial \phi_{3D}(\boldsymbol{p}_i^0, \boldsymbol{c}_{n_c})}{\partial z_{p_i^0}} & 0 & 0 & 0 & 1 \end{pmatrix}^T, \quad (3.21)$$

where $\boldsymbol{\phi}_{3D}$ denotes the 3D *TPS RBF* defined in (2.18) and $\boldsymbol{c}_i$ is the i-th control point. The sizes of matrices in (3.21) are as follows. The first matrix $\boldsymbol{C}'^T$ is a $3 \times n_c$ matrix, $\boldsymbol{K}_*^T$ is a $n_c \times (n_c + 4)$ matrix, and the rightmost matrix which we can call $d\boldsymbol{M}_i^T$ is a $(n_c + 4) \times 3$ matrix.

Calculating $f_{\boldsymbol{\mu}}$

The last term in (3.18), $f_{\boldsymbol{\mu}}$, represents the $3 \times n_c$ Jacobian matrix of $f(\boldsymbol{p}_0, \boldsymbol{\mu}(t))$ regarded as a function of $\boldsymbol{\mu}(t)$. Like the matrix $f_{\boldsymbol{P}_0}$, this matrix is also calculated at every point $\boldsymbol{p}_i^0$ in $\boldsymbol{P}_0$. If we denote the matrix $f_{\boldsymbol{\mu}}$ at point $\boldsymbol{p}_i^0$ as $f_{\boldsymbol{\mu}}^i$, we can define it as

$$f_{\boldsymbol{\mu}}^i = \left(\frac{\partial f(\boldsymbol{p}_i^0, \boldsymbol{\mu}(t))}{\partial x_{c_1}} \middle| \frac{\partial f(\boldsymbol{p}_i^0, \boldsymbol{\mu}(t))}{\partial y_{c_1}} \middle| \frac{\partial f(\boldsymbol{p}_i^0, \boldsymbol{\mu}(t))}{\partial z_{c_1}} \middle| \dots \middle| \frac{\partial f(\boldsymbol{p}_i^0, \boldsymbol{\mu}(t))}{\partial x_{c_{n_c}}} \middle| \frac{\partial f(\boldsymbol{p}_i^0, \boldsymbol{\mu}(t))}{\partial y_{c_{n_c}}} \middle| \frac{\partial f(\boldsymbol{p}_i^0, \boldsymbol{\mu}(t))}{\partial z_{c_{n_c}}} \right). \quad (3.22)$$

By expanding this equation, we will get

$$f_{\boldsymbol{\mu}}^i = \begin{pmatrix} \frac{\partial x_{p_i}}{\partial x_{c_1}} & \frac{\partial x_{p_i}}{\partial y_{c_1}} & \frac{\partial x_{p_i}}{\partial z_{c_1}} & \dots & \frac{\partial x_{p_i}}{\partial x_{c_{n_c}}} & \frac{\partial x_{p_i}}{\partial y_{c_{n_c}}} & \frac{\partial x_{p_i}}{\partial z_{c_{n_c}}} \\ \frac{\partial y_{p_i}}{\partial x_{c_1}} & \frac{\partial y_{p_i}}{\partial y_{c_1}} & \frac{\partial y_{p_i}}{\partial z_{c_1}} & \dots & \frac{\partial y_{p_i}}{\partial x_{c_{n_c}}} & \frac{\partial y_{p_i}}{\partial y_{c_{n_c}}} & \frac{\partial y_{p_i}}{\partial z_{c_{n_c}}} \\ \frac{\partial z_{p_i}}{\partial x_{c_1}} & \frac{\partial z_{p_i}}{\partial y_{c_1}} & \frac{\partial z_{p_i}}{\partial z_{c_1}} & \dots & \frac{\partial z_{p_i}}{\partial x_{c_{n_c}}} & \frac{\partial z_{p_i}}{\partial y_{c_{n_c}}} & \frac{\partial z_{p_i}}{\partial z_{c_{n_c}}} \end{pmatrix}, \quad (3.23)$$

where we have used the same notations as before, with x_{c_i}, y_{c_i} and z_{c_i} being respectively the x, y and z coordinates of the i-th control point $\boldsymbol{c}_i$. If we denote the i-th element of the vector $\boldsymbol{K}_*^T \boldsymbol{M}_i^T$ as v_i, where $\boldsymbol{M}_i$ represents the i-th row of the matrix $\boldsymbol{M}$, then we can write the matrix $f_{\boldsymbol{\mu}}^i$ as

$$f_{\boldsymbol{\mu}}^i = \begin{pmatrix} v_1 & 0 & 0 & v_2 & 0 & 0 & \dots & v_{n_c} & 0 & 0 \\ 0 & v_1 & 0 & 0 & v_2 & 0 & \dots & 0 & v_{n_c} & 0 \\ 0 & 0 & v_1 & 0 & 0 & v_2 & \dots & 0 & 0 & v_{n_c} \end{pmatrix}. \quad (3.24)$$

Putting it all together

After we have shown how to calculate all of the terms in (3.18), we can construct the Jacobian matrix in the following way:

$$\boldsymbol{J}_{\boldsymbol{\mu}} = \begin{pmatrix} \nabla_{\boldsymbol{p}_1^0} \boldsymbol{I} \cdot f_{\boldsymbol{p}_1^0}^{-1} \cdot f_{\boldsymbol{\mu}}^1 \\ \nabla_{\boldsymbol{p}_2^0} \boldsymbol{I} \cdot f_{\boldsymbol{p}_2^0}^{-1} \cdot f_{\boldsymbol{\mu}}^2 \\ \vdots \\ \nabla_{\boldsymbol{p}_n^0} \boldsymbol{I} \cdot f_{\boldsymbol{p}_n^0}^{-1} \cdot f_{\boldsymbol{\mu}}^n \end{pmatrix}. \tag{3.25}$$

This matrix will be a $n \times 3n_c$ matrix, where n is the number of points $\boldsymbol{p}_i$ in $\boldsymbol{P}_0$ and n_c is the number of the control points $\boldsymbol{c}_i$.

3.2.2. The final tracking algorithm

After we have defined the velocity vector (3.17), the final equation used for estimating the motion parameter vector (3.2) can be written as

$$\boldsymbol{\mu}(t+\tau) = \boldsymbol{\mu}(t) + \boldsymbol{v}_{\boldsymbol{\mu}}\tau, \tag{3.26}$$

where τ denotes the sampling period of the tracking process.

Due to the fact that the Jacobian $\boldsymbol{J}_{\boldsymbol{\mu}}$ is a $n \times 3n_c$ matrix, which is usually quite large, it is very time-consuming to calculate it and its pseudo-inverse at every step of the tracker. An alternative is to use an approximated pseudo-inverse $\widehat{\boldsymbol{J}_{\boldsymbol{\mu}}^+}$ which is calculated only once at the initialization, when $t = t_0$, and is used instead of $\boldsymbol{J}_{\boldsymbol{\mu}}^+$ in (3.17).

3.3. Testing the tracker

After implementing the tracker described above (see Appendix A for implementation details) we have to somehow test the tracker. The basic idea used for tracker testing will be explained in this section.

As mentioned before, the goal of the tracker is to minimize the error defined in (3.15). Unfortunately, we can't use the final value of this error as the only measure of correctness of the tracking, because a low error doesn't mean that the estimated control point positons, *i.e.*, the values of the motion parameter vector (3.2), are close to their real values. This is due to the fact that the movement of the control points cannot be entirely explained by the motion model (3.1) alone and because there is high noise present in ultrasound images. A better way would

be to compare the estimated control point positions directly to their real values. This, of course, cannot be done if we don't know the real control points positions. But there is a solution! We can generate our own deformed volumes using our motion model (3.1) in which the real control positions will be known and use them as *ground truth data* to test the tracker.

The methods used in the process of generating our own sequences of deforming ultrasound volumes will be explained in the following Chapter 4.

3.4. Improving the tracker

There are two methods that we tried in order to improve the accuracy of the tracker. The first one was to add simple regularization to our motion model (3.1), *i.e.*, to the three-dimensional *TPS* warp (2.23), and the second method was to add some physical constraints to the motion of the control points.

3.4.1. Adding regularization to the TPS warp

As it is mentioned in [4] and [10], adding regularization to *TPS* relaxes the interpolation requirement that the interpolant has to pass directly through the values at the centers, which is useful when dealing with very noisy data, such as ultrasound images.

According to [4], the relaxation of the interpolation condition is done by minimizing

$$H(s) = \sum_{i=1}^{n_c} (v_i - s(x_{c_i}, y_{c_i}, z_{c_i}))^2 + \lambda_{reg} I_s, \tag{3.27}$$

where v_i represents the value of the function we are interpolating at the point $\boldsymbol{c}_i$, $v_i = f(x_{c_i}, y_{c_i}, z_{c_i})$, s is the three-dimensional *TPS* interpolant as defined in (2.17), I_s is the three-dimensional bending energy (2.16) and λ_{reg} is the *regularization parameter*, a positive scalar. As stated in [6], to include the regularization into the *TPS* warp, we must replace the matrix $\boldsymbol{\Phi}$ in the equation (2.24) by $\boldsymbol{\Phi} + \lambda_{reg}\boldsymbol{I}$, where $\boldsymbol{I}$ represents an identity matrix. This change will affect the matrix $\boldsymbol{K}_*$ used in the equation (3.20) in the tracker. Notice that the regularized warp is used only in the tracker to estimate the motion parameter vector. When finding the new locations of points in the tracked region as described in Subsection 2.3.2, a warp without regularization is used.

3.4.2. Constraining the motion of the control points

The second method used for improving the tracker was to physically constrain the movement of the control points in order to prevent them from going into unnatural arrangements. This is done by combining the tracker with a mass-spring system, which is explained in detail in Chapter 5.

4. Generating ground truth data

The ground truth data in this work are the control point positions in our generated sequence of deformed ultrasound volumes. This chapter will describe the idea used for deforming an ultrasound volume and the methods used in the process.

4.1. The basic idea of volume deforming

The idea used for deforming an ultrasound volume used in this work is very simple. We will take one ultrasound volume, define a three-dimensional grid of control points inside of it, move the control points a bit at each step and use our motion model (3.1) to find new positions of all of the other points contained in the initial volume. This idea has already been described in detail by an example that can be found in Subsection 2.3.2.

There are only two things remaining to explain the process of volume deformation completely. The first one is how to define the movement of the control points during the volume deformation, and the second is the problem of missing intensities in the generated volumes. Just to be more clearer, imagine a volume that we are expanding. The dimensions of the expanded volume will be bigger than the dimensions of the initial volume, which means that the deformed volume will contain empty space between the voxels with intensities from the initial volume. The problem of these missing intensities will be solved by interpolation.

The following two sections explain the problems stated above.

4.2. Defining the motion of the control points

In this section we are going to explain how the movement of control points from their initial positions was defined in this work.

The idea is as follows. For each control point $\boldsymbol{c}_i$ we will define a vector $\boldsymbol{v}_i$ pointing into the direction in which we want to move the point. Also, for each

of these vectors we will add an another parameter l_i which will define its length, *i.e.*, the maximal distance the control point can move from its initial position. The last parameter will be a real number $\alpha_i \in [0, 100]$ describing the percentage that the control point has traveled from its initial position along the given vector. An example of a vector with different values of α is shown in Figure 4.1.

Notice that when we define the control point positions and their vectors pointing in the direction we want to move them, not all of the values of parameter α have to be the same for all of the vectors. A two-dimensional example with four control points is shown in Figure 4.2, where we have initialized the vectors with $\alpha_1 = 70$, $\alpha_2 = 25$, $\alpha_3 =$ and $\alpha_4 = 50$. By using the directions of the vectors $\boldsymbol{v}_i$, their defined lengths l_i and their α_i parameters, the starting point $\boldsymbol{v}_i^s$ and ending point $\boldsymbol{v}_i^e$ of each vector the control points will move on can be calculated as

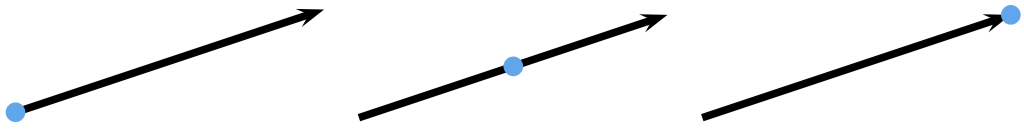
$$\boldsymbol{v}_i^s = \boldsymbol{c}_i - ||\boldsymbol{v}_i|| \cdot l_i \cdot \frac{\alpha_i}{100}, \tag{4.1}$$

$$\boldsymbol{v}_i^e = \boldsymbol{v}_i^s + ||\boldsymbol{v}_i|| \cdot l_i. \tag{4.2}$$

After that, we can change the position of the control point $\boldsymbol{c}_i$ by increasing the parameter α_i and using it in the equation

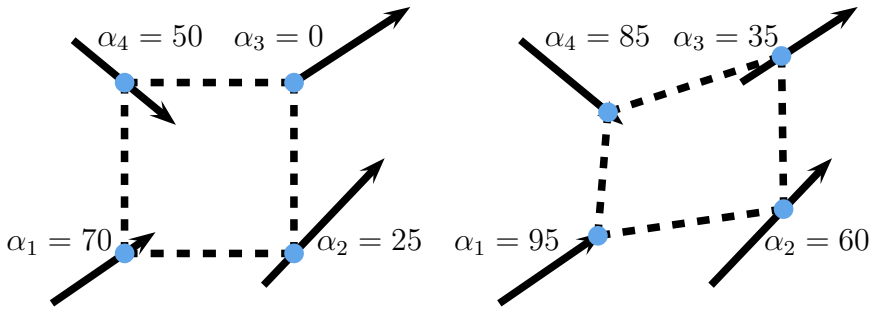
$$\boldsymbol{c}_i' = \boldsymbol{v}_i^s + ||\boldsymbol{v}_i|| \cdot l_i \cdot \frac{\alpha_i}{100}. \tag{4.3}$$

To make sure that the control points move only between the points $\boldsymbol{v}_i^s$ and $\boldsymbol{v}_i^e$, the value of the parameter α_i has to be between 0 and 100. Because of that, we will increase the value of α_i only until it reaches 100, then start decreasing it to start increasing it again when it reaches 0. An example of how the control point positions changed when we increased the α_i values by $\Delta\alpha = 35$ is also shown in Figure 4.2.



(a) Vector with $\alpha = 0$. (b) Vector with $\alpha = 50$. (c) Vector with $\alpha = 100$.

Figure 4.1: An example of a vector pointing in the direction in which we want to move the control point, with different values of the parameter α. When $\alpha = 0$ the control point is at the start of the vector, with $\alpha = 50$ the control point moved half way from the start towards the end, and with $\alpha = 100$ the control point is at the end of the vector.



(a) Control points with $\Delta\alpha = 0$. (b) Control points with $\Delta\alpha = 35$.

Figure 4.2: A two-dimensional example with control points moving along defined vectors. The subfigure on the left shows the control point positons as they were initialized, and the right subfigure shows their new positions after increasing the parameters α_i of each vector by $\Delta\alpha = 35$. The values of the parameter α_i increase until they reach 100, after what they start decreasing. Also, after reaching 0, the value of α_i starts increasing again.

Finally, to generate a deformed volume from the initial one, we will increase all of the α_i parameters by a chosen $\Delta\alpha$, find new control point locations using the equation (4.3) and use the new control point locations to find new positions of all of the other points in the initial volume as described in Subsection 2.3.2. The solution to the remaining problem of unknown intensities at some points in the deformed volume will be explained in the following section.

4.3. Interpolating the missing intensities

As it was mentioned before, if we expand a volume using the methods described in the previous two sections, we will end up with a volume that will contain some empty space, *i.e.*, points at which no intensity value is known. To calculate the intensities at these points, we will use a method called *tetrahedral interpolation* which is going to be explained in the following subsection.

4.3.1. Tetrahedral interpolation

We are given a function $f(\boldsymbol{x}) : \mathbb{R}^3 \to \mathbb{R}$ and four points $\boldsymbol{v}_i \in \mathbb{R}^3$ forming a tetrahedron, as shown in Figure 4.3. The values of the function f are known at all of the given points $\boldsymbol{v}_i$ and we are interested in finding the value of the function at the point $\boldsymbol{v}$ that is lying inside of the tetrahedron. To be able to find

the unknown value, we will first have to find this point's *barycentric coordinates*.

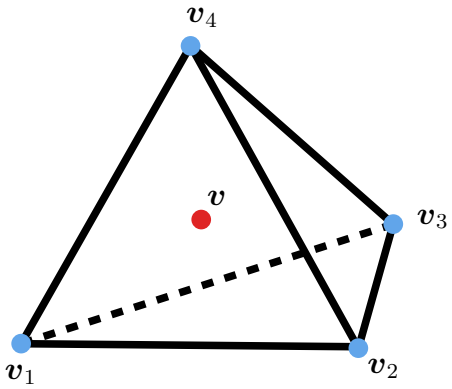


Figure 4.3: Four points forming a tetrahedron. The values of the observed function f are known at all of the tetrahedron's vertices. We are interested in finding the value of the function at the point $\boldsymbol{v}$ lying inside of the tetrahedron, marked here in red color.

Finding the barycentric coordinates of a point

Barycentric coordinates allow us to calculate the location of a point $\boldsymbol{v}$ lying inside of the tetrahedron by summing up the locations of points forming the tetrahedron, each multiplied by a corresponding weight. This is written as

$$\boldsymbol{v} = \lambda_1 \boldsymbol{v}_1 + \lambda_2 \boldsymbol{v}_2 + \lambda_3 \boldsymbol{v}_3 + \lambda_4 \boldsymbol{v}_4. \tag{4.4}$$

There is an additional constraint for the weigths

$$\lambda_1 + \lambda_2 + \lambda_3 + \lambda_4 = 1, \tag{4.5}$$

from which we can express

$$\lambda_4 = 1 - \lambda_1 - \lambda_2 - \lambda_3. \tag{4.6}$$

If we rewrite the equation (4.4) for each component of the point $\boldsymbol{v}$, we will have the equations

$$x_v = \lambda_1 x_{v_1} + \lambda_2 x_{v_2} + \lambda_3 x_{v_3} + \lambda_4 x_{v_4}, \tag{4.7}$$

$$y_v = \lambda_1 y_{v_1} + \lambda_2 y_{v_2} + \lambda_3 y_{v_3} + \lambda_4 y_{v_4}, \tag{4.8}$$

$$z_v = \lambda_1 z_{v_1} + \lambda_2 z_{v_2} + \lambda_3 z_{v_3} + \lambda_4 z_{v_4}. \tag{4.9}$$

By substituting λ_4 in the equations above with the equation (4.6) and rearranging the equations a bit, we can write

$$\lambda_1(x_{v_1} - x_{v_4}) + \lambda_2(x_{v_2} - x_{v_4}) + \lambda_3(x_{v_3} - x_{v_4}) = x_v - x_{v_4}, \tag{4.10}$$

$$\lambda_1(y_{v_1} - y_{v_4}) + \lambda_2(y_{v_2} - y_{v_4}) + \lambda_3(y_{v_3} - y_{v_4}) = y_v - y_{v_4}, \tag{4.11}$$

$$\lambda_1(z_{v_1} - z_{v_4}) + \lambda_2(z_{v_2} - z_{v_4}) + \lambda_3(z_{v_3} - z_{v_4}) = z_v - z_{v_4}, \tag{4.12}$$

or in matrix form

$$\boldsymbol{T}\boldsymbol{\lambda} = \boldsymbol{v} - \boldsymbol{v}_4, \tag{4.13}$$

where the matrix $\boldsymbol{T}$ is defined as

$$\boldsymbol{T} = \begin{pmatrix} x_{v_1} - x_{v_4} & x_{v_2} - x_{v_4} & x_{v_3} - x_{v_4} \\ y_{v_1} - y_{v_4} & y_{v_2} - y_{v_4} & y_{v_3} - y_{v_4} \\ z_{v_1} - z_{v_4} & z_{v_2} - z_{v_4} & z_{v_3} - z_{v_4} \end{pmatrix}. \tag{4.14}$$

In the end, the barycentric coordinates of the point $\boldsymbol{v}$, *i.e.*, λ_1, λ_2, and λ_3, are found using the equation

$$\boldsymbol{\lambda} = \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \lambda_3 \end{pmatrix} = \boldsymbol{T}^{-1}(\boldsymbol{v} - \boldsymbol{v_4}). \tag{4.15}$$

The last barycentric coordinate, λ_4, is found from the equation (4.6). The matrix $\boldsymbol{T}$ in (4.15) is invertible because $\boldsymbol{v}_1 - \boldsymbol{v}_4$, $\boldsymbol{v}_2 - \boldsymbol{v}_4$ and $\boldsymbol{v}_3 - \boldsymbol{v}_4$ are linearly independent. Also, if the point $\boldsymbol{v}$ lies inside of the tetrahedron, all of its barycentric coordinates will be from the interval $[0, 1]$. If any of the coordinates is less than 0 or greater than 1, the point lies outside of the tetrahedron.

Using the barycentric coordinates to find the unknown value

After finding the barycentric coordinates of the point $\boldsymbol{v}$ at which we want to calculate the unknown value, the value can be estimated using the following expression

$$f(\boldsymbol{v}) = \lambda_1 f(\boldsymbol{v}_1) + \lambda_2 f(\boldsymbol{v}_2) + \lambda_3 f(\boldsymbol{v}_3) + \lambda_4 f(\boldsymbol{v}_4). \tag{4.16}$$

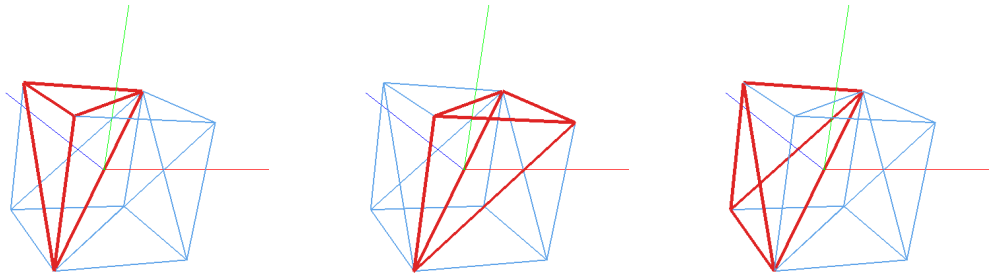
In our case, the function we are interpolating are the intensities in the volume at a given point, which can also be regarded as a function $I(\boldsymbol{v}) : \mathbb{R}^3 \to \mathbb{R}$. Using this notation, the missing intensity at the point $\boldsymbol{v}$ is found using the equation

$$I(\boldsymbol{v}) = \lambda_1 I(\boldsymbol{v}_1) + \lambda_2 I(\boldsymbol{v}_2) + \lambda_3 I(\boldsymbol{v}_3) + \lambda_4 I(\boldsymbol{v}_4), \tag{4.17}$$

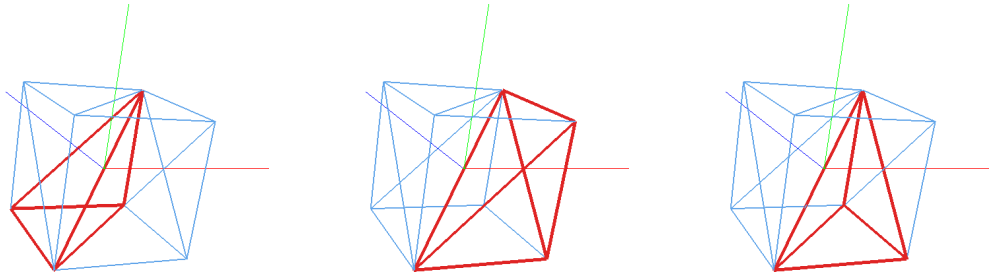
which concludes the tetrahedral interpolation subsection.

4.3.2. Dividing the volume into tetrahedra

In order to use the described tetrahedral interpolation method in our problem of volume deformation, we must first somehow divide the volume we want to deform into a set of tetrahedra. To explain how to do this, we will first show how a cube can be divided into tetrahedra. The process of dividing an object into a set of tetrahedra is called *three-dimensional triangulation* and it is shown in Figure 4.4.



(a) First tetrahedron. (b) Second tetrahedron. (c) Third tetrahedron.



(d) Fourth tetrahedron. (e) Fifth tetrahedron. (f) Sixth tetrahedron.

Figure 4.4: Triangulating a cube into a set of six tetrahedra.

Now that we know how to triangulate one cube, let us explain how we can triangulate a whole volume. For example, let's say that dimensions of the initial volume that we want to deform are $10 \times 20 \times 30$ voxels. If we consider each of the voxels as vertices, we can divide the volume into a set of $9 \times 19 \times 29$ cubes. After that, the rest of the process is pretty straightforward; we just divide each of the cubes in the volume into tetrahedra, as it was shown in Figure 4.4.

4.4. The final volume deformation algorithm

The final volume deformation algorithm can be described by the following few steps. First we choose a volume that we want to deform and place a three-dimensional grid of control points somewhere inside the volume. After that, we triangulate the volume to get a set of tetrahedra, as it was shown in Subsection 4.3.2. For each control point $\boldsymbol{c}_i$ we define a vector $\boldsymbol{v}_i$ pointing in the direction in which we want to move the point, along with its parameters l_i and α_i, as it was described in Section 4.2. To deform the volume, we increase the α_i parameters of each of the vectors defined above by a value $\Delta\alpha$ and find the new locations of control points using equation (4.3). Using the new positions of control points and our motion model (3.1) we find the new locations of all of the other points in the volume, *i.e.*, all of the voxels, as described in the example in Subsection 2.3.2. After that, we can use the previously generated set of tetrahedra and tetrahedral interpolation described in Subsection 4.3.1 to find the missing intensities in the deformed volume. For example, this can be done by iterating through each of the tetrahedra in the set and then calculating the barycentric coordinates of each of the points lying inside of the bounding box of the current tetrahedron. If the current point lies inside of the current tetrahedron, its intensity is calculated using the equation (4.17). If we want to generate the next deformed volume in the sequence, we just repeat the whole process from the step where we increase the α_i parameters by $\Delta\alpha$.

5. Mass-spring system

In this chapter we will describe the main method that we have tried using in order to prevent unnatural movement of control points during tracking, caused by high noise present in ultrasound images. The idea is to connect the control points with a set of springs which will hopefully help keep the control points in a more natural arrangement, by pushing the points away if they come too close to each other, and by pulling them together if they move too far from each other. Our mass-spring system will consist of particles with some mass and springs connecting pairs of particles. In the next sections we will explain the physics needed to simulate this kind of a system.

5.1. A single particle

In order to build a mass-spring system, we must first explain what information is known about a single particle and how this information is used to simulate our system.

Each particle at the time t is defined by its position $\boldsymbol{x}(t)$, velocity $\boldsymbol{v}(t)$, acceleration $\boldsymbol{a}(t)$ and, of course, its mass m, which is considered to be constant. The velocity of a particle can be linked with its position by writing the equation

$$\boldsymbol{v}(t) = \frac{d\boldsymbol{x}(t)}{dt}, \tag{5.1}$$

and similarly, the acceleration can be linked with both the particle's velocity and its position by writing

$$\boldsymbol{a}(t) = \frac{d\boldsymbol{v}(t)}{dt} = \frac{d^2\boldsymbol{x}(t)}{dt^2}. \tag{5.2}$$

We can rewrite the equations above as

$$\boldsymbol{v} = \dot{\boldsymbol{x}}, \tag{5.3}$$

and

$$\boldsymbol{a} = \dot{\boldsymbol{v}} = \ddot{\boldsymbol{x}}. \tag{5.4}$$

By knowing the particle's acceleration $\boldsymbol{a}(t)$, we can find its velocity by integrating

$$\boldsymbol{v}(t) = \int \boldsymbol{a}(t)dt, \tag{5.5}$$

and similarly, its position by integrating

$$\boldsymbol{x}(t) = \int \boldsymbol{v}(t)dt. \tag{5.6}$$

The next question is how to find the particle's acceleration? Well, if we know the total sum of forces that are applied at one particle, we can use the *Newton's second law of motion* stated as

$$\boldsymbol{F} = m\boldsymbol{a} \tag{5.7}$$

to find the particle's acceleration.

5.1.1. Simulating the dynamics of a single particle

Here we consider a single particle changing its position through time due to the forces applied to the particle. If we know the particle's mass m, its position $\boldsymbol{x}(t)$, velocity $\boldsymbol{v}(t)$ and the values of the forces $\boldsymbol{F}_i$ applied to it at time t, we can use the following procedure to find its position and velocity at time $t + \Delta t$.

First, we will find the total force $\boldsymbol{F}_{total}$ applied at the particle by simply summing all of the forces $\boldsymbol{F}_i$ acting on the particle

$$\boldsymbol{F}_{total} = \sum_i \boldsymbol{F}_i. \tag{5.8}$$

After that we can use the Newton's second law of motion (5.7) to get the particle's acceleration

$$\boldsymbol{a}(t) = \frac{\boldsymbol{F}_{total}}{m}. \tag{5.9}$$

To find the current particle velocity and its location, we just have to solve the integrals defined in (5.5) and (5.6). In our work we chose the *Euler's method* [14] as a method of numerical integration, which approximates the particle's velocity and location at time $t + \Delta t$ as

$$\boldsymbol{v}(t + \Delta t) = \boldsymbol{v}(t) + \boldsymbol{a}(t)\Delta t \tag{5.10}$$

and

$$\boldsymbol{x}(t + \Delta t) = \boldsymbol{x}(t) + \boldsymbol{v}(t + \Delta t)\Delta t. \tag{5.11}$$

The same steps are repeated in the future everytime when t increases by Δt. The Euler's method was chosen as a method for numerical integration because of its simplicity. Other methods such as the Runge Kutta method or the Verlet integration method could also be used instead [13].

5.2. The physics behind a spring

Let us consider a spring with a particle on one end, connected to a wall on the other end as shown in Figure 5.1. If we now apply some force to the particle to change the spring's length from the initial l_0 to $l_0 + \Delta l$, the spring will react by pulling the particle back towards its starting location with a force $\boldsymbol{F}_{spring}$. This

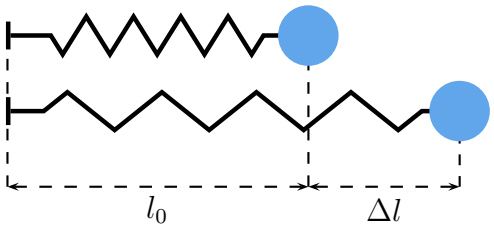


Figure 5.1: A spring in its resting position with a length of l_0 and the same spring after expanding its length to $l_0 + \Delta l$.

force can be found by using the *Hooke's law*, which can be stated as

$$\boldsymbol{F}_{spring} = -k_{spring}\Delta l, \tag{5.12}$$

where Δl represents the difference between the current spring's length and its initial length, and k_{spring} is the spring's constant. We can see that the force $\boldsymbol{F}_{spring}$ tries to restore its initial length by acting in the opposite direction of the force that caused the change of the length. If we tried to contract the spring by pushing the particle in Figure 5.1 to the left, the force described by the Hook's law would push the particle back to the right.

If we tried to simulate a spring described only by (5.12) using the methods described in the Subsection 5.1.1, the particle would undergo a simple harmonic motion, oscillating infinitely around its starting position. To change this behaviour, we must add an another force that will add some friction to our system. The frictional force that we will add is called the *damping* force, and it is described by the following equation

$$\boldsymbol{F}_{damping} = -b\boldsymbol{v}', \tag{5.13}$$

where $\boldsymbol{v}'$ represents the velocity of the particle $\boldsymbol{v}$ projected onto the vector pointing into the direction of the spring, and b is the damping ratio.

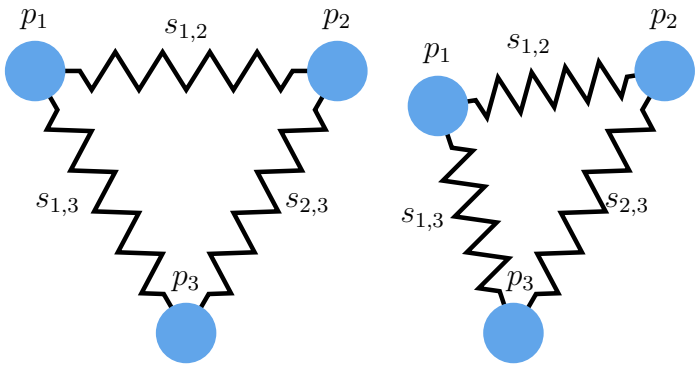
Combining the two forces described in (5.12) and (5.13), we can see that the total force acting on a particle connected to a spring can be stated as

$$\boldsymbol{F}_{total} = -k_{spring}\Delta l - b\boldsymbol{v}' + \boldsymbol{F}_{external}, \tag{5.14}$$

where we have used $\boldsymbol{F}_{external}$ to mark the external force acting on the particle.

5.3. A system of masses and springs

Now let us consider a system of multiple particles connected by springs, as shown in Figure 5.2. The particles are labeled as p_1, p_2 and p_3 and the springs as $s_{i,j}$, where i and j represent the indices of the particles connected by this spring. We



(a) Initial system.

(b) After moving the first particle.

Figure 5.2: A system of three particles connected by springs. The first subfigure shows the initial state of the system and the second subfigure shows the state of the system after we moved the first particle a bit.

are interested in finding the forces that are acting on the particle p_1. If we mark the position of the particle p_i as $\boldsymbol{x}_{p_i}$, then, we can write the vector pointing into the direction of the spring $s_{1,2}$ as

$$\boldsymbol{e}_{1,2} = \frac{\boldsymbol{x}_{p_2} - \boldsymbol{x}_{p_1}}{||\boldsymbol{x}_{p_2} - \boldsymbol{x}_{p_1}||}. \tag{5.15}$$

The force $\boldsymbol{F}_{s_{1,2}}$ acting on the particle p_1, caused by the spring $s_{1,2}$, can be then written as

$$\boldsymbol{F}_{s_{1,2}} = -k_{s_{1,2}} \Delta l_1 = -k_{s_{1,2}} (l^0_{1,2} - l_{1,2}) \boldsymbol{e}_{1,2}, \tag{5.16}$$

where we used $l^0_{1,2}$ to mark the initial length of the spring $s_{1,2}$, $l_{1,2}$ to mark its current length and $k_{s_{1,2}}$ to mark its spring constant. The damping force acting on the particle p_1 caused by the spring $s_{1,2}$ can be calculated as

$$\boldsymbol{F}_{d_{1,2}} = -b_{1,2}(\boldsymbol{v}'_1 + \boldsymbol{v}'_2), \tag{5.17}$$

where $\boldsymbol{v}'_1$ and $\boldsymbol{v}'_2$ represent the speeds of particles p_1 and p_2 projected onto the vector $\boldsymbol{e}_{1,2}$, defined in (5.15). The above equation can be rewritten as

$$\boldsymbol{F}_{d_{1,2}} = -b_{1,2}(\boldsymbol{v}_1 \cdot \boldsymbol{e}_{1,2} + \boldsymbol{v}_2 \cdot \boldsymbol{e}_{1,2}) \boldsymbol{e}_{1,2}, \tag{5.18}$$

where $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are the speed vectors of the particles p_1 and p_2 and $\cdot$ marks the vector dot product. If we use the same principles as described above, we can find the forces $\boldsymbol{F}_{s_{1,3}}$ and $\boldsymbol{F}_{d_{1,3}}$ also acting on the particle p_1, but caused by the spring $s_{1,3}$. The total sum of forces acting on the particle p_1 can then be calculated as

$$\boldsymbol{F}_{total_1} = \boldsymbol{F}_{s_{1,2}} + \boldsymbol{F}_{d_{1,2}} + \boldsymbol{F}_{s_{1,3}} + \boldsymbol{F}_{d_{1,3}} + \boldsymbol{F}_{external_1}. \tag{5.19}$$

The total force acting on the i-th particle can more generally be written as

$$\boldsymbol{F}_{total_i} = \sum_j \boldsymbol{F}_{s_{i,j}} + \sum_j \boldsymbol{F}_{d_{i,j}} + \boldsymbol{F}_{external_i}, \tag{5.20}$$

where the forces $\boldsymbol{F}_{s_{i,j}}$ and $\boldsymbol{F}_{d_{i,j}}$ are calculated for every spring $s_{i,j}$ connected to the particle p_i.

5.3.1. Simulating the dynamics of the whole system

The simulation of the dynamics of the whole mass-spring system is simple. All we have to do is calculate the total forces acting on each of the particles in the system, calculate their accelerations using the Newton's second law of motion (5.7) and then use the methods described in the Subsection 5.1.1 to find their velocities and new positions.

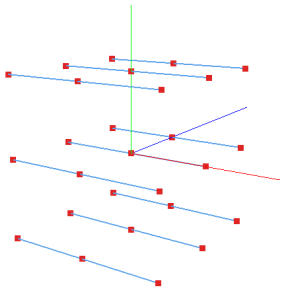
5.4. Integrating the mass-spring system with the tracker

In order to integrate the mass-spring system defined in the previous sections with our intensity-based tracker, we must first define its particles and how they will be connected with springs. As the idea is to constrain the movement of control points to prevent their unnatural arrangements, we will use the control points as the particles in our mass-spring system. The control points in the tracker in our work are initialized in a three-dimensional grid, so some of the possible ways to connect them with springs are shown in Figure 5.3. By combining these nine types of connections, we can get 512 different mass-spring system structures, each of which will cause different behaviour of the system when forces are applied to its particles.

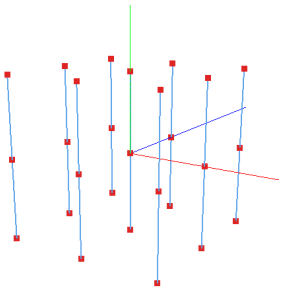
The intensity-based tracker was described in detail in Chapter 3 and is summarized in the Subsection 3.2.2. As it can be seen from the equation (3.26), the

motion parameter vector is increased at every iteration of the tracker by the vector $\boldsymbol{v_\mu}\tau$. The changes we have to make in the tracker to include our mass-spring system constrains are simple. Let's repeat that the motion parameter vector is a vector that contains the locations of control points, as defined in (3.2). So, adding the vector $\boldsymbol{v_\mu}\tau$ to the vector $\boldsymbol{\mu}$ means that we move each of the control points $\boldsymbol{c}_i$ by some offset contained in $\boldsymbol{v_\mu}\tau$. Let us denote the offset of each of the control points $\boldsymbol{c}_i$ as $\Delta\boldsymbol{c}_i$.

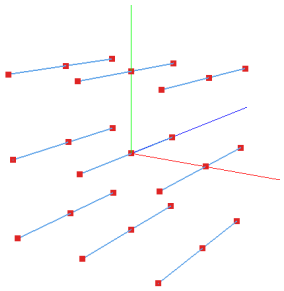
Instead of just moving each of the control points $\boldsymbol{c}_i$ by $\Delta\boldsymbol{c}_i$ as it is done in the original tracker, we will think about the offset $\Delta\boldsymbol{c}_i$ as a force and apply it to the particle $\boldsymbol{p}_i$ in our mass spring system. To make the system more configurable, the force $\Delta\boldsymbol{c}_i$ can be scaled by a *force scale constant* before applying it to the particle. After applying the forces, we will simulate the dynamics of the mass-spring system for a number of iterations, as it is described in Subsection 5.3.1. The positions of the particles at which the simulation has ended will be considered as the new control point locations. This process is repeated at each step of the tracker.



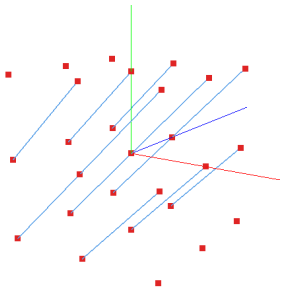
(a) Only in x direction.



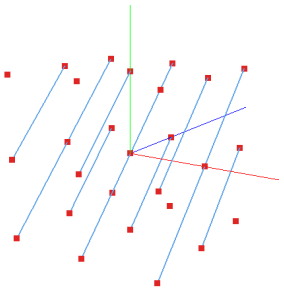
(b) Only in y direction.



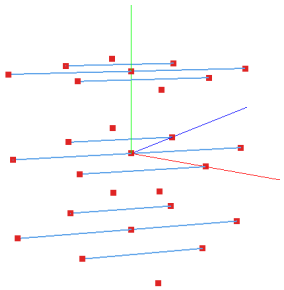
(c) Only in z direction..



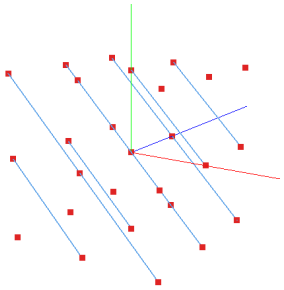
(d) Only first diagonals in x direction.



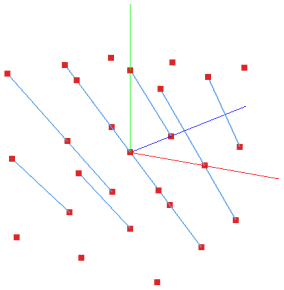
(e) Only first diagonals in y direction.



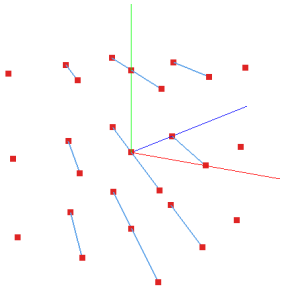
(f) Only first diagonals in z direction.



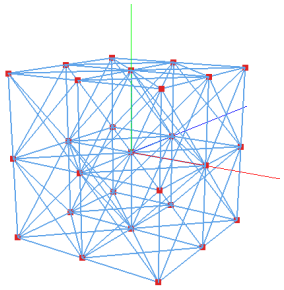
(g) Only second diagonals in x direction.



(h) Only second diagonals in y direction.



(i) Only second diagonals in z direction.



(j) All directions at once.

Figure 5.3: The nine basic options of connecting springs in a three-dimensional grid are shown in subfigures a) to i). By combining these options we can get different kinds of mass-spring system structures. The last subfigure shows the spring connections in a system where all nine types of connections are used at once.

6. Experimental results

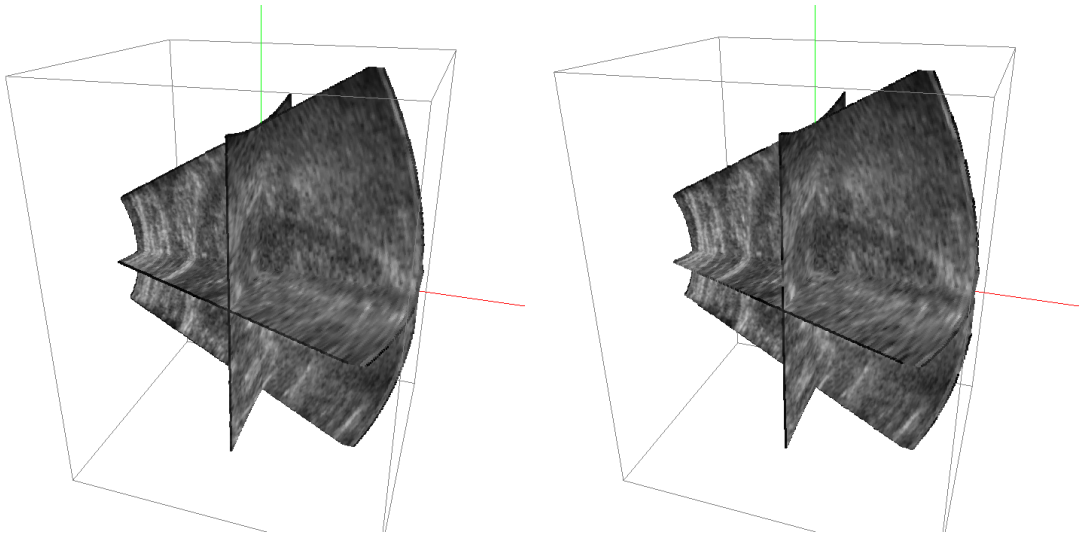
In order to test the developed tracking methods, we have generated a small sequence of deforming ultrasound volumes using the methods described in Chapter 4. As the sequence was generated artificially, we know the exact positons the control points should take in every volume in the sequence, which is used as ground truth data in our experiments. The accuracy of our tracking method is measured by calculating the average of the Euclidean distances of each of the control points from their real locations. We have added different amounts of noise to the generated sequence of volumes to see how the tracker behaves in different conditions. The used sequences are briefly described in the following section.

6.1. The used volume sequences

The first volume sequence is a generated sequence of 10 ultrasound volumes, without any added noise. Some of the volumes from this sequence are shown in Figure 6.1.

The other used sequences are the same as the sequence mentioned above, but with some noise added. The values of noise added to each voxel in a volume are taken from the normal distribution with the parameters μ and σ, where μ denotes the mean and σ denotes the standard deviation of the distribution. Each of the voxels in the volume can have an intensity value from the interval $[0, 255]$. If some values are less than 0 or greater than 255 after adding the noise, we just set them to 0 and 255 respectively.

Four additional sequences with noise have been generated. The noise parameters used were $\mu_1 = 20$, $\sigma_1 = 10$, $\mu_2 = 20$, $\sigma_2 = 20$, $\mu_3 = 20$, $\sigma_3 = 30$ and $\mu_4 = 20$, $\sigma_4 = 40$. Some of the volumes from these sequences are shown in Figure 6.2.



(a) Volume number 1.

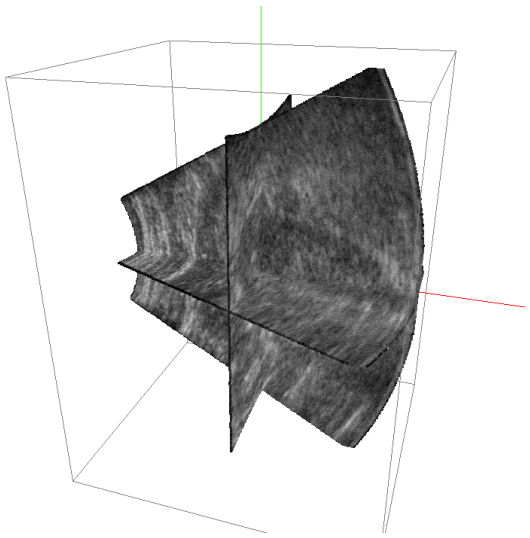
(b) Volume number 9.

Figure 6.1: Two volumes from the sequence without any added noise.

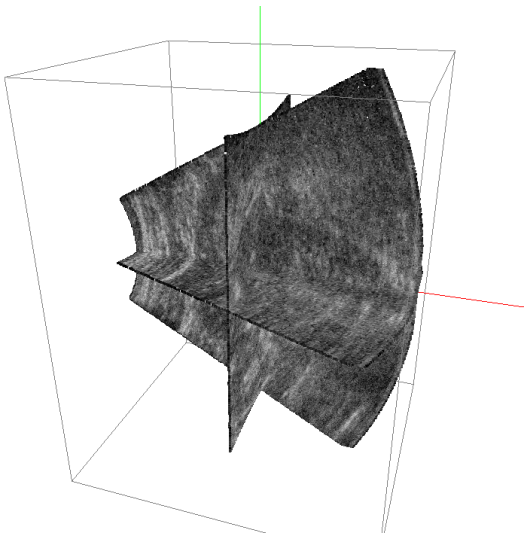
6.2. The used tracker parameters

The tracker depends on the following parameters. First we have to specify the number of control points in x, y and z directions, the size and location of the three-dimensional grid formed by this number of control points and the dimensions of the chosen tracking region. Next we have to choose the number of tracker iterations, τ used in (3.26) and λ used in (3.17). If we want to add some regularization we have to specify the regularization parameter λ_{reg} used in (3.27). We have the choice to decide if the mass-spring system is used or not and choose the nine different parameters describing the structure of the mass-spring system as shown in Figure 5.3. The other parameters of the mass-spring system are the spring stiffness constant, the spring damping constant, the spring system simulation time step, the particle mass, the force scale constant mentioned in Section 5.4 and the number of iterations used when simulating the behaviour of the mass-spring system.

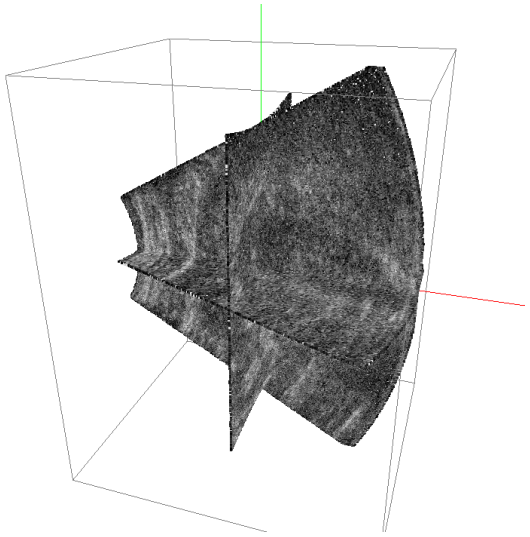
As the number of parameters that the tracker depends on is very big and every one of them has an impact on the tracker accuracy, it is hard to find the best configuration. Some combinations of the tracker parameters that have been tested are shown in the following section.



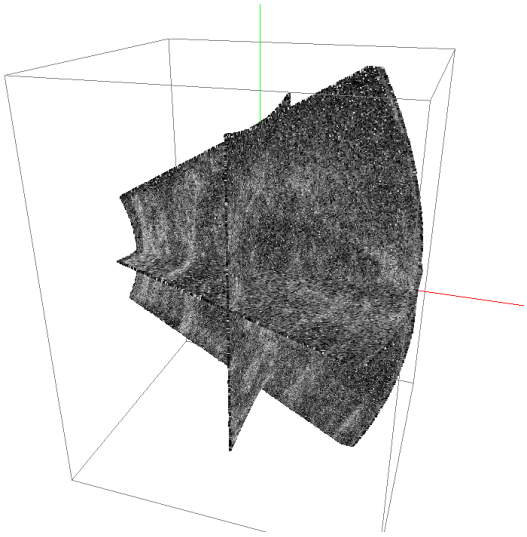
(a) Volume with added noise, $\mu = 20$, $\sigma = 10$.



(b) Volume with added noise, $\mu = 20$, $\sigma = 20$.

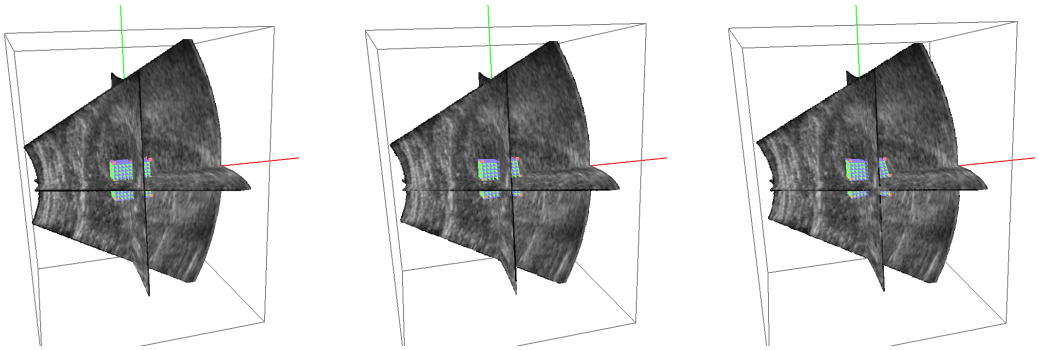


(c) Volume with added noise, $\mu = 20$, $\sigma = 30$.



(d) Volume with added noise, $\mu = 20$, $\sigma = 40$.

Figure 6.2: The same volume with four different quantities of noise added.



(a) Initial volume. (b) Iteration number 10. (c) Iteration number 20.

Figure 6.3: An example of tracking of an region in a generated sequence of 20 volumes, without any added noise. The points marked in red color are the control points and the lines are used to show the deformation of the tracked region. When this sequence was generated the control points on the left were fixed in place, which is why they aren't moving during the tracking.

6.3. The results

It this section we will show some of the tracker testing results. The accuracy of the tracker was measured as the average Euclidean distance of control points from ground truth control point locations. As the differences between these values are hard to visualise graphically, here they will be shown in table format. In order to show how the tracking looks like, an example of tracking in a sequence of 20 deformed ultrasound volumes can be seen in Figure 6.3. If not stated differently, the tracker parameters τ used in (3.26), λ used in (3.17), and the number of tracker iterations are set as $\tau = 0.7$, $\lambda = 0.5$ and $n_{it} = 15$ in all of the tests described here.

The first five tables (6.1, 6.2, 6.3, 6.4 and 6.5) will show the results of tracker testing on the volume sequences described in Section 6.1 with different values of the *TPS* regularization parameter λ_{reg}, described in Subsection 3.4.1. The tested values of the regularization parameter were 0, 2, 5, 10 and 15. As it was expected, the tracking accuracy increased with the regularization parameter in cases when sequences with high noise were used. In the tests with low noise, setting λ_{reg} to lower values caused better results.

The following two tables (6.6 and 6.7) show the results of testing the tracker with the mass-spring system constraints turned on. The tracker parameters τ, λ and n_{it} remain the same as before and the other parameters related to the

Table 6.1: Tracker error in the volume sequence without added noise, with different values of the regularization parameter λ_{reg}.

λ_{reg}	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0	2.36802	1.73567	1.70777	2.90666	2.44047	2.93724	2.96127	2.43656	2.93394
2	2.36738	**1.73547**	**1.70764**	2.90631	**2.44026**	**2.93707**	**2.96112**	**2.43639**	**2.93387**
5	2.36662	1.73581	1.70782	**2.90626**	2.4404	2.93725	2.96113	2.43644	2.93407
10	2.36555	1.73632	1.70803	2.9067	2.44101	2.93791	2.96128	2.43701	2.93484
15	**2.36544**	1.74224	1.72158	2.93412	2.51744	3.10657	3.36753	3.40863	4.71218

Table 6.2: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 10$, with different values of the regularization parameter λ_{reg}.

λ_{reg}	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0	2.35134	1.75782	1.72794	2.85068	2.4504	2.92386	2.95976	**2.46321**	2.933
2	2.35064	1.75744	**1.72781**	**2.85048**	**2.45036**	**2.92364**	**2.95969**	2.46322	**2.93287**
5	2.34971	1.75681	1.72794	2.85066	2.45073	2.92393	2.96026	2.46371	2.93316
10	2.34857	1.75568	1.72844	2.852	2.45291	2.92607	2.96325	2.46719	2.93563
15	**2.34822**	**1.7545**	1.73082	2.85908	2.47118	2.95095	3.00284	2.53393	3.0161

Table 6.3: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 20$, with different values of the regularization parameter λ_{reg}.

λ_{reg}	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0	2.2746	1.79773	1.73104	2.66682	**2.43624**	**2.86136**	**2.95435**	**2.49595**	**2.87106**
2	2.27322	1.79687	1.72948	2.66656	2.43687	2.86196	2.9554	2.49798	2.872
5	2.27161	1.79484	1.72696	2.66599	2.43827	2.86413	2.95817	2.50207	2.87507
10	**2.26942**	1.79088	1.72308	2.66446	2.44125	2.86918	2.9669	2.51665	2.88857
15	2.27	**1.78859**	**1.71977**	**2.66316**	2.44457	2.87881	2.98472	2.55306	2.93087

Table 6.4: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 30$, with different values of the regularization parameter λ_{reg}.

λ_{reg}	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0	2.10299	1.8431	1.78313	2.30884	2.26711	**2.65693**	**2.84579**	**2.57209**	**2.83097**
2	2.10007	1.83968	1.7806	2.30808	2.26632	2.65724	2.84723	2.57418	2.83467
5	2.0966	1.83435	1.77647	2.30708	**2.26544**	2.65946	2.85168	2.5782	2.84098
10	2.09275	1.8266	1.76968	**2.30622**	2.26686	2.66664	2.86446	2.59281	2.85902
15	**2.09233**	**1.8246**	**1.76806**	2.3112	2.27724	2.68375	2.89362	2.62971	2.89933

Table 6.5: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 40$, with different values of the regularization parameter λ_{reg}.

λ_{reg}	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0	1.97111	1.89677	1.89267	2.23442	2.25727	2.49449	**2.64669**	2.56163	**2.73404**
2	1.96609	1.8927	1.88885	2.22581	2.25127	2.49173	2.64731	2.56127	2.73507
5	1.95971	1.8886	1.88411	2.21649	2.24385	**2.48825**	2.64789	**2.56042**	2.73702
10	1.95293	**1.88546**	**1.88001**	**2.2094**	**2.24147**	2.48979	2.65371	2.56776	2.75032
15	**1.9525**	1.88873	1.88549	2.21826	2.2594	2.5109	2.67959	2.60481	2.79653

mass-spring system are set as follows. The spring damping constant is set to 0.2, the simulation time step is set to 0.3 and the structure of the mass-spring system is as the last one shown in Figure 5.3, with all the connections in use. In these two tests we wanted to see how the spring stiffness and particle mass affect the accuracy of the tracker. As it can be seen from the tables, the results were better as the spring stiffness constant was lower.

Table 6.6: Tracker error in the volume sequence without added noise, with the mass-spring system turned on, force scale constant=1, particle mass=1 and different values of the spring stiffness.

stiff.	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0.01	**2.51485**	**1.66474**	**1.71794**	**3.17036**	2.35339	**3.02459**	2.97796	**2.34677**	**3.09712**
0.05	2.52134	1.66726	1.72824	3.17228	2.3651	3.05382	**2.97535**	2.39056	3.1013
0.1	2.53446	1.66551	1.73247	3.1943	2.38197	3.06715	3.00399	2.43077	3.12276
0.2	2.57009	1.65558	1.74268	3.25147	2.39633	3.08636	3.06153	2.48576	3.16865
0.5	2.7139	1.60016	1.7763	3.47168	**2.34806**	3.13813	3.17863	2.53755	3.27049

Table 6.7: Tracker error in the volume sequence without added noise, with the mass-spring system turned on, force scale constant=1, particle mass=0.5 and different values of the spring stiffness.

stiff.	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0.01	2.4457	1.67885	**1.71168**	**3.05238**	**2.38944**	3.01294	**2.96078**	**2.39881**	**3.01952**
0.05	**2.44506**	1.68896	1.71344	3.05419	2.41641	**3.0101**	2.9886	2.43364	3.03461
0.1	2.45352	1.69463	1.71944	3.06525	2.43534	3.02697	3.01359	2.47558	3.06791
0.2	2.48094	1.69047	1.73388	3.09765	2.45135	3.06747	3.04678	2.53738	3.13173
0.5	2.57032	**1.64903**	1.75845	3.21817	2.45069	3.17303	3.0959	2.60925	3.29262

The following five tables (6.8, 6.9, 6.10, 6.11 and 6.12) show how lowering the spring stiffness affected the accuracy of the tracker, tested in all five of the available sequences. First two rows of each of these tables show the results when particle mass is set to 1 and the last two rows show the results with particle mass set to 0.5. As it can be seen from the tables, the results were almost always better when spring stiffness was set to 0.01 than when it was set to 0.05. Setting the particle mass to 1 had slightly better results than when it was set to 0.5.

The last five tables (6.13, 6.14, 6.15, 6.16 and 6.17) show the results of tests where we tested the effects of different values of the force scale constant combined with two different values of the regularization term λ_{reg}. Other tracker parameters were fixed with the following values: particle mass=1, spring stiffness=0.01, spring damping=0.2 and the simulation time step=0.3. As it can be seen from the tables, in all of the tests the best results were acquired when the force scale

Table 6.8: Tracker error in the volume sequence without added noise, with the mass-spring system turned on, force scale constant=0.5, particle mass=1 (first two rows) or 0.5 (last two rows) and different values of the spring stiffness.

m	stiff.	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
1	0.01	**2.17152**	1.88037	**1.69286**	**2.5975**	2.6133	**2.75246**	3.00807	2.56255	**2.76617**
	0.05	2.17801	1.88214	1.70874	2.62019	2.61697	2.80143	3.04624	2.62643	2.85401
0.5	0.01	2.22595	**1.79159**	1.73451	2.72945	**2.45184**	2.8672	**2.94581**	**2.5091**	2.89739
	0.05	2.22796	1.80209	1.74892	2.73466	2.48011	2.90192	2.98289	2.58543	2.95506

Table 6.9: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 10$, with the mass-spring system turned on, force scale constant=0.5, particle mass=1 (first two rows) or 0.5 (last two rows) and different values of the spring stiffness.

m	stiff.	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
1	0.01	**2.12454**	1.92117	**1.71584**	**2.45342**	2.5867	**2.75238**	2.98249	2.62956	**2.7771**
	0.05	2.13076	1.92023	1.73603	2.48996	2.59797	2.79765	3.04734	2.72346	2.89268
0.5	0.01	2.18921	**1.81924**	1.75383	2.60644	**2.45369**	2.81712	**2.95189**	**2.56581**	2.88675
	0.05	2.19161	1.82677	1.77571	2.61717	2.48193	2.87029	3.00222	2.66711	2.98586

Table 6.10: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 20$, with the mass-spring system turned on, force scale constant=0.5, particle mass=1 (first two rows) or 0.5 (last two rows) and different values of the spring stiffness.

m	stiff.	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
1	0.01	**2.00883**	1.94708	1.78397	**2.24327**	2.44745	**2.68381**	2.91723	2.7209	**2.80214**
	0.05	2.01353	1.94707	1.7909	2.26777	2.46715	2.72055	2.98853	2.84151	2.96168
0.5	0.01	2.07295	**1.87108**	**1.7784**	2.37054	**2.39712**	2.70187	**2.90087**	**2.65249**	2.86006
	0.05	2.0757	1.87403	1.78999	2.3854	2.41453	2.75303	2.9689	2.77631	3.01232

Table 6.11: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 30$, with the mass-spring system turned on, force scale constant=0.5, particle mass=1 (first two rows) or 0.5 (last two rows) and different values of the spring stiffness.

m	stiff.	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
1	0.01	**1.87957**	1.89878	1.8317	**2.04689**	2.19574	**2.43517**	**2.70141**	2.68911	**2.78239**
	0.05	1.88222	1.89956	1.84032	2.07626	2.24273	2.49306	2.77755	2.80292	2.94484
0.5	0.01	1.92412	**1.86007**	**1.81451**	2.10467	**2.19198**	2.46722	2.70853	**2.63332**	2.79307
	0.05	1.9254	1.86213	1.82943	2.13446	2.23478	2.52528	2.78357	2.75762	2.96535

constant was set to 0.05. The results were usually better when no regularization was used, except in some cases with sequences with high values of noise. For example, see Table 6.17, where the tracking was the best with the regularization parameter set to 5. After four iterations the tracking without regularization had better results. This could perhaps be explained by the low number of tracking iterations preventing the control points to converge in the volume sequence with

Table 6.12: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 40$, with the mass-spring system turned on, force scale constant=0.5, particle mass=1 (first two rows) or 0.5 (last two rows) and different values of the spring stiffness.

m	stiff.	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
1	0.01	**1.80556**	1.87698	1.90019	**2.04571**	2.15808	**2.30673**	**2.48279**	2.55133	**2.6688**
	0.05	1.80635	1.8734	1.89556	2.04717	2.1745	2.34364	2.54296	2.64682	2.80263
0.5	0.01	1.83844	1.87163	1.8861	2.06804	**2.15522**	2.31648	2.49008	**2.53025**	2.67837
	0.05	1.83709	**1.86486**	**1.88416**	2.07545	2.1738	2.35804	2.55368	2.63478	2.8151

high noise present.

Finally, if we compare the results shown in the first five tables (6.1, 6.2, 6.3, 6.4 and 6.5) where the mass-spring system was not used with the results shown in the last five tables (6.13, 6.14, 6.15, 6.16 and 6.17) where the tracking was tested with the mass-spring system turned on, we can see that the constrained tracking almost always resulted in better accuracy. By testing other combinations of tracker parameters, even better results should probably be obtained.

Table 6.13: Tracker error in the volume sequence without added noise, with the mass-spring system turned on, particle mass=1, spring stiffness=0.01, $\lambda_{reg} = 0$ (first four rows), $\lambda_{reg} = 2$ (second four rows) or $\lambda_{reg} = 5$ (last four rows), with different values of the force scale constant.

λ_{reg}	force	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0	0.5	2.17152	1.88037	1.69286	2.5975	2.6133	2.75246	3.00807	2.56255	2.76617
	0.2	1.88881	1.89782	1.80872	2.16523	2.40556	2.61605	2.80744	2.71477	2.76037
	0.1	1.79111	1.82941	1.82068	2.02266	2.20793	2.40267	2.59209	2.64357	2.73043
	0.05	**1.74366**	**1.77945**	**1.80453**	**1.93848**	**2.07949**	**2.23851**	**2.40538**	**2.51277**	**2.63818**
2	0.5	2.17094	1.88077	1.69189	2.59566	2.61581	2.75284	3.00679	2.56429	2.76676
	0.2	1.8888	1.89836	1.80961	2.16604	2.40767	2.61971	2.81157	2.72039	2.76657
	0.1	1.79117	1.82991	1.82183	2.02455	2.21116	2.40747	2.5987	2.65286	2.74189
	0.05	1.74371	1.77978	1.80537	1.94009	2.08235	2.24284	2.41145	2.52159	2.6494
5	0.5	2.17034	1.88188	1.69165	2.59344	2.62098	2.75498	3.00575	2.5684	2.76971
	0.2	1.8889	1.89934	1.81163	2.16848	2.41236	2.62716	2.82029	2.73212	2.77958
	0.1	1.79132	1.83076	1.82394	2.02808	2.21711	2.41611	2.61047	2.66947	2.76251
	0.05	1.74381	1.78031	1.80683	1.94278	2.0872	2.24999	2.42131	2.53588	2.66773

Table 6.14: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 10$, with the mass-spring system turned on, particle mass=1, spring stiffness=0.01, $\lambda_{reg} = 0$ (first four rows), $\lambda_{reg} = 2$ (second four rows) or $\lambda_{reg} = 5$ (last four rows), with different values of the force scale constant.

λ_{reg}	force	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0	0.5	2.12454	1.92117	1.71584	2.45342	2.5867	2.75238	2.98249	2.62956	2.7771
	0.2	1.86912	1.89953	1.82984	2.11978	2.33474	2.55317	2.76481	2.73308	2.79745
	0.1	1.78196	1.8266	1.83018	2.00455	2.16959	2.35355	2.54387	2.62588	2.73754
	0.05	**1.73949**	**1.77738**	**1.80818**	**1.93006**	**2.06129**	**2.21285**	**2.37722**	**2.49726**	**2.63567**
2	0.5	2.12396	1.92199	1.71614	2.4516	2.58952	2.75536	2.98275	2.63202	2.78025
	0.2	1.86905	1.89998	1.83124	2.12144	2.3375	2.55761	2.77052	2.74103	2.80692
	0.1	1.78199	1.82692	1.83132	2.00641	2.17268	2.35825	2.55037	2.63506	2.74916
	0.05	1.73953	1.77758	1.8089	1.93133	2.06358	2.21634	2.38206	2.50421	2.64462
5	0.5	2.1234	1.9235	1.71723	2.44989	2.59489	2.76175	2.98517	2.63813	2.7875
	0.2	1.86904	1.90087	1.83377	2.12483	2.34303	2.56606	2.78143	2.75581	2.82466
	0.1	1.78206	1.82746	1.83309	2.00945	2.1779	2.36616	2.56133	2.65035	2.7686
	0.05	1.73958	1.7779	1.80997	1.93332	2.06724	2.22192	2.3898	2.51528	2.65881

Table 6.15: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 20$, with the mass-spring system turned on, particle mass=1, spring stiffness=0.01, $\lambda_{reg} = 0$ (first four rows), $\lambda_{reg} = 2$ (second four rows) or $\lambda_{reg} = 5$ (last four rows), with different values of the force scale constant.

λ_{reg}	force	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0	0.5	2.00883	1.94708	1.78397	2.24327	2.44745	2.68381	2.91723	2.7209	2.80214
	0.2	1.81944	1.86109	1.83689	2.04129	2.21283	2.4157	2.62907	2.6795	2.78328
	0.1	1.75747	1.79744	1.81781	1.954	2.09088	2.25458	2.43383	2.54474	2.68141
	0.05	**1.7275**	**1.76059**	**1.79684**	**1.89994**	**2.01549**	**2.15407**	**2.30915**	**2.43844**	**2.58615**
2	0.5	2.00771	1.94776	1.78382	2.24205	2.44898	2.68732	2.92139	2.72772	2.80943
	0.2	1.81921	1.86124	1.83759	2.04221	2.21477	2.41895	2.63404	2.68767	2.79385
	0.1	1.75743	1.79756	1.8183	1.95488	2.09265	2.25729	2.43768	2.55094	2.68971
	0.05	**1.7275**	1.7607	1.79716	1.90054	2.01671	2.15588	2.31159	2.4423	2.59125
5	0.5	2.00688	1.94843	1.78346	2.24131	2.45232	2.69444	2.92989	2.74034	2.8231
	0.2	1.81917	1.8617	1.8388	2.04414	2.21852	2.42511	2.64316	2.70173	2.8119
	0.1	1.7575	1.79789	1.8192	1.95652	2.09585	2.26214	2.44444	2.56126	2.70335
	0.05	1.72757	1.76094	1.79774	1.9016	2.01878	2.15893	2.31567	2.44851	2.59933

Table 6.16: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 30$, with the mass-spring system turned on, particle mass=1, spring stiffness=0.01, $\lambda_{reg} = 0$ (first four rows), $\lambda_{reg} = 2$ (second four rows) or $\lambda_{reg} = 5$ (last four rows), with different values of the force scale constant.

λ_{reg}	force	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0	0.5	1.87957	1.89878	1.8317	2.04689	2.19574	2.43517	2.70141	2.68911	2.78239
	0.2	1.76817	1.81234	1.8286	1.94925	2.06965	2.23708	2.42956	2.53666	2.66591
	0.1	1.73255	1.76936	1.80565	1.90056	2.00777	2.14708	2.30676	2.43478	2.57769
	0.05	1.71531	1.74611	**1.78911**	**1.87138**	**1.97071**	**2.09499**	**2.2374**	**2.37171**	**2.51901**
2	0.5	1.87856	1.89811	1.83177	2.04814	2.19868	2.43893	2.70586	2.69682	2.79239
	0.2	1.76793	1.81201	1.82894	1.9506	2.07233	2.24062	2.43395	2.54311	2.67481
	0.1	1.73248	1.76921	1.80581	1.90128	2.00938	2.14928	2.30951	2.4388	2.58313
	0.05	**1.71529**	**1.74606**	1.78921	1.87179	1.97162	2.09624	2.23894	2.37392	2.52195
5	0.5	1.87769	1.89778	1.83209	2.05055	2.20364	2.44564	2.71394	2.70946	2.80898
	0.2	1.76782	1.81192	1.8296	1.95277	2.07651	2.24627	2.44102	2.55328	2.68867
	0.1	1.73249	1.76922	1.80621	1.90259	2.01202	2.15285	2.31395	2.44507	2.59145
	0.05	1.71531	1.74609	1.78944	1.87251	1.97309	2.09824	2.24141	2.37735	2.52639

Table 6.17: Tracker error in the volume sequence with noise parameters $\mu = 20$, $\sigma = 40$, with the mass-spring system turned on, particle mass=1, spring stiffness=0.01, $\lambda_{reg} = 0$ (first four rows), $\lambda_{reg} = 2$ (second four rows) or $\lambda_{reg} = 5$ (last four rows), with different values of the force scale constant.

λ_{reg}	force	iter. 1	iter. 2	iter. 3	iter. 4	iter. 5	iter. 6	iter. 7	iter. 8	iter. 9
0	0.5	1.80556	1.87698	1.90019	2.04571	2.15808	2.30673	2.48279	2.55133	2.6688
	0.2	1.73891	1.78942	1.83886	1.94114	2.04465	2.17109	2.31712	2.43671	2.57742
	0.1	1.71821	1.75623	1.80694	1.89371	1.99198	2.11046	2.24566	2.37518	2.52106
	0.05	1.70824	1.73926	1.78885	1.86728	**1.96195**	**2.07575**	**2.20545**	**2.33919**	**2.48695**
2	0.5	1.8041	1.87505	1.89888	2.04464	2.15874	2.30958	2.48674	2.55799	2.67749
	0.2	1.73861	1.78877	1.83816	1.94064	2.04514	2.1727	2.31939	2.44042	2.58258
	0.1	1.71812	1.75598	1.8066	1.89345	1.99228	2.11137	2.24701	2.37738	2.52398
	0.05	1.70822	1.73916	1.78869	1.86715	1.96209	2.07618	2.20613	2.3403	2.48841
5	0.5	1.80201	1.87225	1.89699	2.04413	2.16107	2.31459	2.49343	2.56872	2.69166
	0.2	1.73816	1.78788	1.83714	1.94031	2.04646	2.17565	2.32346	2.44654	2.59068
	0.1	1.71798	1.7556	1.80605	1.89322	1.9929	2.11281	2.24915	2.3807	2.52832
	0.05	**1.70816**	**1.73901**	**1.78844**	**1.86705**	1.96242	2.07691	2.20724	2.34203	2.49063

7. Conclusion

In this work we have reimplemented the methods for intensity-based tracking of soft tissues in sequences of three-dimensional ultrasound images, as described in [9]. Two new methods were added to the previous work in order to improve the robustness of the tracking. The first method is regularization of the thin-plate spline warp used as a motion model in the tracker, and the second method is adding of a mass-spring system used for physically constraining the movement of the control points.

The performed tests have shown that the tracker performed better when the developed mass-spring system was used, compared to when no physical constraints were added to the control points at all. The use of regularization of the *TPS* warp has shown slight improvements in tests performed on volume sequences with high noise present.

As the new developed methods for tracking depend on a large number of parameters, only a few of their combinations have been tested. Further testing and optimization of the used parameters could result in significant improvements of the tracking accuracy.

In the future, the described tracking method could be reimplemented for the *GPU* in order to achieve real-time performance. The described method for deformed volume generation should also be reimplemented, maybe with the use of external libraries such as *CGAL* [17] in order to speed it up. Other future work could contain a simpler method for selecting the tracked area in the initial volume in the sequence and a simpler way of choosing the positions of the initial control points.

Bibliography

[1] B. J. C. Baxter. *The interpolation theory of radial basis functions.* PhD thesis, University of Cambridge, Trinity College, 1992.

[2] F. L. Bookstein. Principal Warps: Thin-Plate Splines and the Decomposition of Deformations. *IEEE Trans. Pattern Anal. Mach. Intell.*, 11(6):567–585, June 1989.

[3] D. S. Broomhead and D. Lowe. Multivariable Functional Interpolation and Adaptive Networks. *Complex Systems 2*, pages 321–355, 1988.

[4] G. Donato and S. Belongie. Approximation methods for thin plate spline mappings and principal warps. *7th European Conference on Computer Vision*, pages 13–31, 2002.

[5] J. Duchon. Splines minimizing rotation-invariant semi-norms in Sobolev spaces Constructive Theory of Functions of Several Variables. In Walter Schempp and Karl Zeller, editors, *Constructive Theory of Functions of Several Variables*, volume 571 of *Lecture Notes in Mathematics*, chapter 7, pages 85–100. Springer Berlin / Heidelberg, 1977.

[6] F. Girosi, M. Jones, and T. Poggio. Regularization theory and neural networks architectures. *Neural Computation*, 7:219–269, 1995.

[7] G. D. Hager and P. N. Belhumeur. Efficient region tracking with parametric models of geometry and illumination. *IEEE Trans. Pattern Anal. Mach. Intell.*, 20(10):1025–1039, 1998.

[8] R. L. Harder and R. N. Desmarais. Interpolation Using Surface Splines. *Journal of Aircraft*, pages 189–191, 1972.

[9] D. Lee and A. Krupa. Intensity-based visual servoing for non-rigid motion

compensation of soft tissue structures due to physiological motion using 4d ultrasound. In *IROS*, pages 2831–2836. IEEE, 2011.

[10] J. Lim and M. Yang. A direct method for modeling non-rigid motion with thin plate spline. *CVPR*, 1:1196–1202, 2005.

[11] D. Matijašević. *Hermitian radial basis function interpolation in finite volume method*. PhD thesis, University of Zagreb, Faculty of Mechanical Engineering and Naval Architecture, 2011.

[12] J. Meinguet. Multivariate interpolation at arbitrary points made simple. *Zeitschrift für Angewandte Mathematik und Physik (ZAMP)*, 30(2):292–304, 1979.

[13] J. Mosegaard. *Cardiac Surgery Simulation*. PhD thesis, University of Aarhus, Faculty of Science, Department of Computer Science, 2006.

[14] A. Nealen, M. Mueller, R. Keiser, E. Boxerman, and M. Carlson. Physically Based Deformable Models in Computer Graphics. *Computer Graphics Forum*, 25(4):809–836, 2006.

[15] M. J. D. Powell. Radial basis function methods for interpolation to functions of many variables. *Fifth Hellenic-European conference on Computer Mathematics and its applications*, 2001.

[16] D. Rueckert and P. Burger. Shape-based segmentation and tracking in 4D cardiac MR images. In *In Conference on Computer Vision, Virtual Reality and Robotics in Medicine (CVRMed'97)*, pages 43–52. Springer, 1997.

[17] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org.

Appendix A
Implementation details

In this chapter we will describe the most important functionalities of the classes developed in this work. All the implementation was done using the *C++* programming language. For more details refer to the source codes.

A.1. ImageVolume class

The ImageVolume class is used to store all the voxel intensities of one volume. The intensities are internally stored in an `volumeSizeX` × `volumeSizeY` × `volumeSizeZ` array of `unsigned chars`, where the voxel at the coordinates (x, y, z) is placed at index `z*volumeSizeX*volumeSizeY+y*volumeSizeX+x` in the array. An object of type `ImageVolume` can be created by using the default constructor which creates an empty volume, by using an another constructor which creates a volume of given dimensions, or by using the constructor shown below. This last constructor prepares the volume to be loaded from a given directory which contains the volume slices stored in image files.

```
ImageVolume(const std::string& folderLocation_,
            const std::string& sliceName_,
            const unsigned int idLength_,
            const std::string& extension_);
```

If the slices are stored in a directory with the path `/path/to/folder/` and the names of the slice images are `slice0000.png`, `slice0001.png`, ..., then the arguments of the constructor would be: `folderLocation="/path/to/folder"`, `sliceName="slice"`, `idLength=4` and `extension=".png"`. When this constructor is used, the dimensions of the volume in given folder are found automagically, by examining the number of slice images and their dimensions. After creating the object using this constructor, we have to call the following function in order to load the intensities

from the images into our array.

```
void loadVolume();
```

We can get a reference to the intensity stored at the coordinates (x, y, z) in our volume by using the following operator. Notice that this operator accepts only integer values of the coordinates as it returns the intensity data as it is stored internally in the object. If an voxel outside of the volume is accessed, a dummy with the value of 0 is returned.

```
inline unsigned char& operator()(const unsigned int x,
                                 const unsigned int y,
                                 const unsigned int z) const;
```

If we are interested in finding an intensity at a subvoxel location, *i.e.*, at some real valued coordinates (x, y, z), the following function can be used. This function returns the interpolated value calculated using *trilinear interpolation* from the eight known surrounding intensity values.

```
unsigned char getInterpolatedIntensityAtPoint(const double x,
                                              const double y,
                                              const double z);
```

The following three functions are used to obtain the slice images we would get if we cut the volume at the specified position. For example, `getImageX` called with `index=50` would cut the volume along the plane $x = 50$ and return the slice image in `imageX`. These functions are useful when we want to display a volume.

```
void getImageX(const unsigned int index,
               vpImage<unsigned char>& imageX) const;
void getImageY(const unsigned int index,
               vpImage<unsigned char>& imageY) const;
void getImageZ(const unsigned int index,
               vpImage<unsigned char>& imageZ) const;
```

To get the gradient values at a given vector of points, we use the following function. The values of the gradient in x direction at the given points will be stored in the array `gX`, and similarly, the gradients in y direction will be stored in `gY` and the gradients in z direction will be stored in `gZ`. Interpolated values of intensities are used when calculating the gradients. The `Point` class used here is just a simple `struct` containing the x, y and z coordinates of a single point.

```
void getGradientsAtPoints(const std::vector<Point>& roiPoints,
                          int* gX, int* gY, int* gZ) const;
```

To get an array of intensities at a given vector of points we use the next function. Similarly to the previous function, the intensities will be stored in the array `intensities`. Also, for the points that are located at real valued coordinates, trilinear interpolation is used.

```
void getIntensitiesAtPoints(const std::vector<Point>& roiPoints,
                            unsigned char* intensities) const;
```

In order to save a volume to the disk as a set of slice images we must first set the output file information, which is done using the following function. The parameters of this function have the same format as the ones in the previously described constructor.

```
void setOutputFileInfo(const std::string& folderLocation_,
                       const std::string& sliceName_,
                       const unsigned int idLength_,
                       const std::string& extension_);
```

After the output file information are set, we save the volume by calling the function `saveToDisk()`.

```
void saveToDisk();
```

If we want to add some noise to a volume, we can use the following function. The parameters of the function are `sigma` and `mean`; the standard deviation and the mean of the normal distribution from which the noise values added to every voxel are chosen. If the value of a voxel's intensity is bigger than 255 or lower than 0 after adding the noise, we set those values to 255 and 0 respectively.

```
void addNoise(const double sigma, const double mean);
```

A.2. Tps class

The `Tps` class implements the thin-plate spline warp described in Chapter 2. The object of `Tps` type is initialized by calling the following function. The first parameter is an `enum TpsType` that can take the values `TPS_2D` or `TPS_3D` allowing us to chose whether we want to use the warp in the two-dimensional or three-dimensional case. The next parameter is a vector of points whose locations we want to change later according to the changes of control point locations. The control points vector is the third parameter, and the last parameter is the regu-

larization term λ_{reg} described in Subsection 3.4.1. By default, the regularization term is set to 0, *i.e.*, no regularization is used.

```
void initTPS(TpsType type,
             const std::vector<Point>& roiPoints,
             const std::vector<Point>& controlPoints,
             const double regularizationLambda=0.);
```

The `initTPS` function explained above initializes the matrices $\boldsymbol{M}$, and $\boldsymbol{K}_*$ used by the *TPS* warp. These matrices are accessible by their getter functions, `getMatrixM` and `getMatrixsubKreginv`. If the matrix $\boldsymbol{K}_*$ without the added regularization term is needed, it can be access by the function `getMatrixsubKinv`. When the new control point locations are known and we want to update the positions of points we used to initialize the `Tps` object, we can use the following function. The first parameter is a vector of points that we are updating, and the second parameter is a vector of control points at new locations.

```
void updatePositions(std::vector<Point>& roiPoints,
                     const std::vector<Point>& newControlPoints);
```

A.3. Deformer class

The `Deformer` class is used to generate deformed volumes from a given initial volume, as it was described in Chapter 4. An object of `Deformer` type is initialized by using the following function. The first parameter is a vector of control points that we will move in order to deform the volume. The second parameter is a pointer to the initial volume that we want to deform and the last parameter is a pointer to an `ImageVolume` object in which the deformed volume will be stored.

```
void init(const std::vector<Point>& controlPoints,
          ImageVolume* volume_,
          ImageVolume* deformedVolume_);
```

The following function is used to load the deformer vector information described in Section 4.2.

```
void loadDeformerVectors();
```

The format of the file containing the vector information is simple. For each of the control points used in the deformer we store the x, y and z components of the vector the point will move on, its length l and its parameter α, described in the

section mentioned above. All these parameters are written in one line for each control point, separated by spaces.

The following function increases the α value of each of the control points, divides the given volume into smaller volumes of dimensions `sizeX`×`sizeY`×`sizeZ`, deforms each of them by using the methods described in Chapter 4 and joins them to get the final deformed volume.

```
void nextStepDivided(std::vector<Point>& controlPoints,
                     int sizeX, int sizeY, int sizeZ);
```

The current implementation used for generating deformed volumes is rather slow, taking about 20 minutes to generate one volume, so reimplementing it should be considered.

A.4. MassSpringSystem class

The `MassSpringSystem` class implements the mass-spring system described in Chapter 5. The parameters of the mass-spring system can be set by using the function `setParameters`. The parameters we can change are `springStiffness`, `springDamping`, `springSimulationTimeStep` and `particleMass`. In the current implementation all the parameters will be set the same for the whole system, *i.e.*, all the particles will have the same mass, all the springs will have the same stiffness constants, *etc.*

```
void setParameters(const double springStiffness_,
                   const double springDamping_,
                   const double springSimulationTimeStep_,
                   const double particleMass_);
```

To add particles to our mass-spring system we can use the following function in which we pass a vector of control points.

```
void addParticles(std::vector<Point>& points);
```

The particles can be connected by springs by using the function `connectParticlesAt`, where the argumens are the indices of particles we want to connect.

```
void connectParticlesAt(const unsigned int index1,
                        const unsigned int index2);
```

Another way of connecting the particles is by using the following function. The areguments of this function are the number of control points in the x, y and z direction and nine boolean variables. The nine boolean variables are explained

best in the Figure 5.3. By combining these boolean variables we can get 512 different system configurations.

```
void connectParticlesWithSprings(const unsigned int numControlX,
  const unsigned int numControlY, const unsigned int numControlZ,
  const bool connectSpringsInXDirection,
  const bool connectSpringsInYDirection,
  const bool connectSpringsInZDirection,
  const bool connectSpringsInXDirectionDiagonal1,
  const bool connectSpringsInYDirectionDiagonal1,
  const bool connectSpringsInZDirectionDiagonal1,
  const bool connectSpringsInXDirectionDiagonal2,
  const bool connectSpringsInYDirectionDiagonal2,
  const bool connectSpringsInZDirectionDiagonal2);
```

Applying a force at a particle can be done by the function `applyForceAtParticle` where the only argument is the index of the particle the force is applied on.

```
void applyForceAtParticle(const unsigned int index,
                          vpColVector& force);
```

The last important function in the `massSpringSystem` class is used for simulating the behaviour of the system for the given number of iterations. If no number is given, the behaviour is simulated only for one iteration.

```
void simulatePhysicsForNIterations
                      (const unsigned int numIterations=1);
```

A.5. Tracker class

The `Tracker` class implements the intensity-based tracking method described in Chapter 3. The initialization of the tracker is done using the following function. The first argument is the initial volume at which the tracking is started. The second argument is the enum that can take the values `TPS_2D` or `TPS_3D`, depending on if we are tracking in two or three dimensions. The next two arguments are a vector of points forming the tracking region, and a vector containing the control points. The last parameter is the value of the regularization parameter λ_{reg}.

```
void initTracker(const ImageVolume& volume_, TpsType type,
                 const std::vector<Point>& roiPoints,
                 const std::vector<Point>& controlPoints,
                 const double tpsRegularizationLambda);
```

To set include the mass-spring system into the tracker, we have to pass an object of `MassSpringSystem` type using the following function.

```
void setMassSpringSystem(MassSpringSystem& massSpringSystem_);
```

When the mass-spring system is included into the tracker, it can be switched on and off by the function `toggleIgnoreMassSpringSystem`. The return value of this function is `true` if the mass-spring system is currently used and `false` otherwise.

```
bool toggleIgnoreMassSpringSystem();
```

The last and the most important function of the `Tracker` is the `track` function. The first argument of this function is the current volume in the sequence. The next two arguments are a vector of the points forming the tracking region and a vector of control points. The rest of the arguments are the tracker λ and τ and the number of tracking iterations.

```
void track(ImageVolume& newVolume, std::vector<Point>& roiPoints,
           std::vector<Point>& controlPoints, const double lambda,
           const double tau, const int numIterations=1);
```

A.6. Compiling and running the examples

This work comes with a *DVD* containing a few example projects showing how the implemented classes described above are used.

Two external libraries are needed in order for the example projects to be compiled successfully. The first one is the *VISP* library[1], and the second one is the *OpenGL* library[2] used for displaying purposes.

Each of the projects comes with an intructions file describing how to compile it using the *CMake* tool[3] and a file containing the instructions on how the built projects are run.

[1] VISP stands for the Visual Servoing Platform, a library developed by the Lagadic team at INRIA, Rennes. http://www.irisa.fr/lagadic/visp/

[2] Open Graphics Library. http://www.opengl.org/

[3] CMake. http://www.cmake.org/

Visual tracking of soft tissue targets in sequences of 3D ultrasound images

Abstract

This work considers tracking of deforming soft tissues in sequences of three-dimensional ultrasound images. The chosen approach to the tracking is based on modeling the deformations using the thin-plate spline warp. In practice, the tracking is reduced to the estimation of the changes in control point locations by examining the intensity changes of neighbouring three-dimensional images in the sequence. The problem this approach faces is its low robustness to ultrasound noise, preventing the correct evolution of the deformation model.

In this work, the basic method for tracking of the deforming soft tissues was implemented, along with two new methods used for improving of the basic method's robustness. The two new methods used were the regularization of the thin-plate spline warp used as the motion model, and the adding of a mass-spring system used for physically constraining the movement of the control points.

The new used methods are described and the results acquired by performing tests on simulated sequences of deforming three-dimensional ultrasound images are discussed.

Keywords: ultrasound, soft tissue tracking, thin-plate spline, TPS, regularization, mass-spring system, constrained motion

Vizualno praćenje mekanog tkiva u slijedu 3D ultrazvučnih slika

Sažetak

Rad razmatra praćenje deformirajućeg mekanog tkiva u slijedu trodimenzionalnih ultrazvučnih slika. Odabrani pristup praćenju temelji se na modeliranju deformacija poliharmoničnim interpolacijskim plohama thin-plate spline. U praksi se praćenje svodi na procjenjivanje promjena lokacija kontrolnih točaka analizom promjene intenziteta susjednih slika iz 3D ultrazvučnog slijeda. Glavni problem ovog pristupa je nedovoljna robustnost na ultrazvučni šum, što sprječava korektnu evoluciju modela deformacija.

U okviru rada, implementiran je osnovni postupak praćenja mekanog tkiva, uz dvije nove dodatne metode korištene za poboljšavanje robustnosti osnovnog postupka. Te dvije nove korištene metode su dodavanje regularizacije modelu deformacija i dodavanje sustava masa i opruga u svrhu fizičkog ograničavanja kretanja kontrolnih točaka.

Razvijene metode su opisane zajedno s rezultatima dobivenim testiranjem razvijenih metoda na simuliranim sljedovima deformirajućih trodimenzionalnih ultrazvučnih slika.

Ključne riječi: ultrazvuk, praćenje mekanog tkiva, thin-plate spline, TPS, regularizacija, sustav masa i opruga, ograničeno kretanje