Bartosz Fabianowski *

Efficient GPU-Based Multi-Modal Medical Volume Registration

Diploma Thesis

18th August, 2006

*bartosz@fabianowski.eu

Chair VII: Computer Graphics
Department of Computer Science
University of Dortmund

Supervisors:
Prof. Dr. Heinrich Müller
Dipl.-Inform. Pedram Hadjian

Contents

1. Introduction **1**

1.1. Tomography . . . 1
1.1.1. Physiological Imaging . . . 2
1.1.2. Anatomical Imaging . . . 3
1.1.3. Modality Fusion . . . 3
1.2. Registration . . . 4
1.2.1. Application Areas . . . 5
1.2.1.1. Remote Sensing . . . 5
1.2.1.2. Computer Vision . . . 5
1.2.1.3. Medical Image Computation . . . 5
1.2.2. General Problem Statement . . . 6
1.2.3. Problem Specification . . . 6
1.2.4. Classification of Techniques . . . 8
1.3. Multi-Modal Medical Registration . . . 9
1.3.1. Problem Specification . . . 9
1.3.2. Proposed Technique . . . 10
1.4. Overview of the Thesis . . . 12
1.5. Notations . . . 12

2. Similarity Metric **13**

2.1. Survey of Metrics . . . 14
2.1.1. Feature Based Approaches . . . 14
2.1.1.1. Shortcomings . . . 15
2.1.2. Image Based Approaches . . . 15
2.1.2.1. Shortcomings . . . 17
2.2. Mutual Information . . . 17
2.3. Derivation . . . 18
2.3.1. Discrete Random Variables . . . 18
2.3.2. Pairs of Discrete Random Variables . . . 19
2.3.3. Entropy . . . 20
2.3.4. Joint and Conditional Entropy . . . 21
2.3.5. Mutual Information . . . 22
2.4. Application to Image Registration . . . 23
2.5. Density Estimation . . . 24
2.5.1. Gaussian Distribution . . . 24
2.5.2. Parametric Estimation . . . 26

2.5.3. Histogram . . . 27
2.5.4. Parzen Window Technique . . . 28
2.6. Similarity Metric . . . 29
2.6.1. Alignment Quality . . . 30
2.6.2. Derivative of Alignment Quality . . . 31
2.6.3. Parzen (Co-)Variances . . . 32
2.7. Normalized Mutual Information . . . 34

3. Search Strategy **35**
3.1. Survey of Strategies . . . 35
3.1.1. Specialized Techniques . . . 35
3.1.2. Naïve Search . . . 36
3.1.3. Gradient Ascent . . . 36
3.1.4. Conjugate Gradients . . . 37
3.2. Stochastic Gradient Ascent . . . 38
3.3. Multiresolution . . . 38

4. Transformation Type **41**
4.1. Global . . . 41
4.1.1. Rigid . . . 42
4.1.2. Intensity Derivative . . . 44
4.1.2.1. Indirect Calculation . . . 45
4.1.2.2. Direct Calculation . . . 47
4.1.3. Search Space . . . 48
4.1.4. Other Tranformation Types . . . 48
4.2. Local . . . 49
4.2.1. Scattered Data Interpolation . . . 49
4.2.1.1. Radial Basis Functions . . . 50
4.2.1.2. Radial Basis Functions with Compact Support . . . 52
4.2.2. Transformation Function . . . 53
4.2.3. Placement of Nodes . . . 54
4.2.4. Intensity Derivative . . . 56
4.2.5. Optimizations . . . 59
4.2.5.1. Matrix $\mathbf{C}^{-1}$ Generation . . . 59
4.2.5.2. Local Evaluation of $MI^{\star}$ . . . 60
4.2.5.3. Local Evaluation of $\frac{d}{d\mathbf{q}_i} MI^{\star}$ . . . 60
4.2.5.4. Calculation of T and $\frac{d}{d\mathbf{q}_i} T$ . . . 60
4.2.6. Search Space . . . 63
4.2.7. Other Transformation Types . . . 63

5. Registration System **65**
5.1. Preprocessing . . . 65
5.1.1. Loading: The DICOM Standard . . . 66
5.1.2. Regions of Interest . . . 67

5.1.3. Saving: The UNDO File Format . . . 67
5.2. Registration . . . 68
5.2.1. Class Structure . . . 70
5.2.2. Implementation of Classes . . . 72
5.2.2.1. Registration . . . 72
5.2.2.2. Data . . . 74
5.2.2.3. TransformationRigid . . . 74
5.2.2.4. RigidMIDerivativeCPU . . . 76
5.2.2.5. RigidMICPU . . . 78
5.2.2.6. TransformationNonRigid . . . 79
5.2.2.7. NonRigidMIDerivativeCPU . . . 81
5.2.2.8. NonRigidMICPU . . . 82

6. GPU Acceleration **83**
6.1. GPU Programming . . . 83
6.1.1. Rendering Pipeline . . . 84
6.1.2. Texture Maps . . . 85
6.1.3. Framebuffer Objects and Multiple Render Targets . . . 87
6.1.4. Shaders . . . 88
6.1.4.1. Fragment Shaders . . . 88
6.1.4.2. Vertex Shaders . . . 90
6.2. GPGPU Programming . . . 90
6.2.1. Paradigms . . . 91
6.2.1.1. Scatter versus Gather Operations . . . 92
6.2.1.2. Multipass Rendering . . . 93
6.2.1.3. Reduction . . . 93
6.2.2. Optimizations . . . 94
6.3. Registration . . . 95
6.3.1. Notations . . . 96
6.3.2. Rigid Sampling . . . 97
6.3.3. Rigid Mutual Information Derivative . . . 100
6.3.4. Rigid Mutual Information . . . 103
6.3.5. Non-Rigid Sampling . . . 105
6.3.5.1. Monolithic Shader . . . 105
6.3.5.2. Multiple Shaders . . . 111
6.3.6. Non-Rigid Mutual Information Derivative . . . 113
6.3.7. Non-Rigid Mutual Information . . . 115

7. Results **117**
7.1. Mutual Information . . . 117
7.2. Registration System . . . 120
7.2.1. Quality . . . 121
7.2.1.1. Rigid . . . 121
7.2.1.2. Non-Rigid . . . 126

7.2.2. Speed 128

8. Discussion **133**
8.1. Future Work 137

A. Shaders **139**
A.1. Rigid Sampling 139
A.2. Rigid Mutual Information Derivative 140
A.3. Rigid Mutual Information 141
A.4. Non-Rigid Sampling, Monolithic Shader 143
A.5. Non-Rigid Sampling, Multiple Shaders 151
A.6. Non-Rigid Mutual Information Derivative 152
A.7. Non-Rigid Mutual Information 153

Bibliography **155**

List of Figures

1.1. Tomographic slices stacked for the reconstruction of volumetric data . . . 1
1.2. Transaxial cut through a patient's chest in multiple modalities 2
(a). Physiological PET slice 2
(b). Anatomical CT scan 2
1.3. Fusion of the slices from figure 1.2 4

2.1. Entropy of a biased coin toss as a function of the bias 21
2.2. Scalar Gaussian density functions for different combinations of μ and σ^2 . 25
2.3. Two-dimensional Gaussian density function for $\boldsymbol{\mu} = \mathbf{0}$ and $\boldsymbol{\Sigma} = \mathbf{I}$ 26
2.4. Histogram and Parzen window based estimates of a density function . . . 29

3.1. Two-dimensional cuts through Gaussian pyramids for different datasets . 40
(a). PET scan 40
(b). CT scan 40

4.1. Two-dimensional cut through the grid of nodes 56
4.2. Automatically generated node arrangements 57
4.3. Neighboring grid points of node $\mathbf{p}_i$ 59
4.4. One-dimensional view of grid points relevant to $\mathbf{x}'_u$ 62
(a). $\mathbf{x}'_u$ coinciding with a grid point 62
(b). $\mathbf{x}'_u$ located at arbitrary position between grid points 62
(c). $\mathbf{x}'_u$ located midway between grid points 62
4.5. Topology of the registered image corrupted by a too large displacement . 63
(a). No displacements 63
(b). Small displacement at center node, preserving topology 63
(c). Large displacement at center node, corrupting topology 63

5.1. Screenshot of the preprocessing application 66
5.2. Regions of interest for two CT slices 68
(a). First slice 68
(b). First slice with specified region of interest and bounding box 68
(c). First slice after filling with background intensity and clipping 68
(d). Second slice 68
(e). Second slice with specified region of interest and bounding box . . . 68
(f). Second slice after filling with background intensity and clipping . . . 68
5.3. Screenshot of the registration application 69
5.4. Class structure of the registration application 71

6.1. Simplified OpenGL rendering pipeline . . . 84
6.2. Simplified programmable OpenGL rendering pipeline . . . 85
6.3. Addressing of elements . . . 86
(a). Array . . . 86
(b). OpenGL 1D / 2D / 3D texture . . . 86
(c). OpenGL `Rect` texture . . . 86
6.4. Control and data flow: Rigid sampling . . . 98
6.5. Control and data flow: Rigid MI derivative calculation . . . 101
6.6. Control and data flow: Rigid MI calculation . . . 104
6.7. Control and data flow: Non-rigid sampling, monolithic shader . . . 106
6.8. Control and data flow: Non-rigid sampling, monolithic shader (continued) 109
6.9. Control and data flow: Non-rigid sampling, multiple shaders . . . 112
6.10. Control and data flow: Non-rigid MI derivative calculation . . . 114
6.11. Control and data flow: Non-rigid MI calculation . . . 116

7.1. $MI^\star$ for different PET-CT alignments . . . 118
7.2. $\frac{d}{dt_x} MI^\star$ for different PET-CT alignments . . . 119
7.3. $MI^\star$ for different PET-CT alignments and parameter values . . . 120
(a). $N = 1024$, $\sigma^2 = 5$ . . . 120
(b). $N = 1024$, $\sigma^2 = 60$ . . . 120
(c). $N = 2048$, $\sigma^2 = 5$ . . . 120
(d). $N = 2048$, $\sigma^2 = 60$ . . . 120
7.4. Cuts through PET-CT for automatic initial alignment . . . 123
(a). Reference slice 32 before registration . . . 123
(b). Reference slice 32 after registration . . . 123
(c). Reference slice 64 before registration . . . 123
(d). Reference slice 64 after registration . . . 123
7.5. Cuts through PET-CT for artificial initial misalignment . . . 124
(a). Reference slice 32 before registration . . . 124
(b). Reference slice 32 after registration . . . 124
(c). Reference slice 64 before registration . . . 124
(d). Reference slice 64 after registration . . . 124
7.6. Cuts through CT-CT for artificial initial misalignments . . . 125
(a). Experiment 1 before registration . . . 125
(b). Experiment 2 after registration . . . 125
(c). Experiment 2 before registration . . . 125
(d). Experiment 2 after registration . . . 125
7.7. Local $MI^\star$ for different PET-CT nodes and alignments . . . 127
(a). Node 11 . . . 127
(b). Node 43 . . . 127
(c). Node 65 . . . 127
(d). Node 200 . . . 127
7.8. MSE of $(\mathbf{q}_i)_k$ for CT-CT with different initial misalignments . . . 128

1. Introduction

Ever since the discovery of X-rays by physicist Wilhelm Conrad Röntgen in 1895, medical imaging has played an important role in modern diagnostics. Advancements in imaging technology led to more precise diagnoses and new possibilities of assessing the conditions inside a human body without the need for invasive procedures. A great step forward has been made with the inception of devices that provide a three-dimensional view of a patient's insides. The processes used to acquire such volumetric data are jointly referred to as tomography.

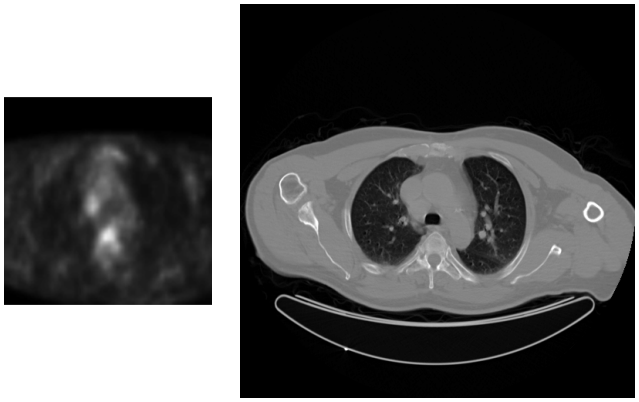
1.1. Tomography

The patient is placed on a table that is then slid into a tomograph. The tomograph generates a two-dimensional image of the patient's cross-section currently in the device, known as a *slice*. A complete dataset consists of a series of slices, each obtained after the table has been advanced slightly. Volumetric information is reconstructed by stacking the slices (figure 1.1[1]). Each intensity value recorded is that of a *voxel*, or volume element, extending half-way to the location of the next intensity value in all directions. The resolution of a tomographic scan is therefore determined by both the spacing of points inside a slice and the distance the table has been advanced between the acquisition of consecutive slices.

Several types of tomographs have been developed for the noninvasive or nearly-non-invasive three-dimensional internal imaging of a patient's body [Zai97] [NGL+98] [CB94] [BBR89]. Common to all is that the data produced falls into one of two basic categories – it is either *physiological* or *anatomical*.

[1] All datasets obtained from the OsiriX website at `http://homepage.mac.com/rossetantoine/osirix/`

Figure 1.1: Tomographic slices stacked for the reconstruction of volumetric data



(a) (b)

Figure 1.2: Transaxial cut through a patient's chest in multiple modalities: (a) physiological PET scan showing metabolic activity; (b) anatomical CT scan showing bone and tissue structure

1.1.1. Physiological Imaging

Physiological data provides functional information about the metabolic processes in a living organism (figure 1.2a). It is usually acquired by methods of nuclear medicine. The patient is injected with a tracer consisting of a radioactive isotope attached to a biologically active molecule. The tracer is given time to travel through the patient's body and concentrate in regions of high metabolic activity. Rotating gamma cameras are then used to measure tracer concentration by detecting the radiation it emits. The types of scanners used are:

PET The tracer used in Positron Emission Tomography emits a positron which in turn releases two gamma rays in exactly opposing directions. By measuring the difference in time between the two rays reaching the scanner's cameras, the exact location of the tracer molecule along the line between the cameras can be calculated. The radionuclides used in PET scanners have very short half-life times necessitating the use of expensive on-site cyclotrons for their production.

SPECT Single Photon Emission Computed Tomography employs a tracer that directly emits gamma rays. The radionuclides have longer half-life times and the scanner is less expensive. The drawback is that the calculation of a tracer molecule's position from a single ray is less precise and leads to lower quality results.

fMRI Functional Magnetic Resonance Imaging is the most recently developed method for obtaining functional diagnostic information. MRI measures the concentration of hydrogen throughout a patient's body by applying a strong magnetic field and detecting the reaction of hydrogen nuclei to excitation by a second pulsed field. MRI images taken in rapid succession can be used to visualize blood flow and changes in oxygenation levels, which coincide with neuronal activity. More general functional information may be obtained through chemical shift imaging. By

recording the spectrum of the responses to the pulsed field, it allows different nuclei to be distinguished and their concentrations to be measured.

The most important advantages of fMRI are that it produces images of very high resolution and is completely noninvasive. PET and SPECT generate lower quality images and require the injection of a radioactive tracer into the patient's body. However, these techniques are also more flexible. By choosing different types of tracer molecules, a variety of metabolic processes can be visualized. While chemical shift imaging can produce similar data, it suffers from very long acquisition times so that the regions scanned must be limited to small areas or single two-dimensional slices.

1.1.2. Anatomical Imaging

Anatomical data provides structural information about a patient's body, showing the shapes of bones and internal organs (figure 1.2b). The types of scanners used to acquire such information are:

CT In Computed Tomography, an X-ray beam is directed at the patient and its intensity measured after it has passed through the patient's body. As in traditional X-ray imaging, a stronger attenuation indicates more solid tissue. Readings obtained at different angles by rotating the X-ray source around the patient are combined to compute an image of the patient's anatomy.

MRI Magnetic Resonance Imaging uses the same technique as fMRI described in the previous section to measure the concentration of hydrogen in the patient's body. As such, it produces images accurately depicting soft tissue and individual organs.

CT and MRI generally produce images of much higher resolution and sharpness than PET and SPECT. Typical examples are shown in figure 1.2 with the PET slice having a resolution of 128×128 pixels and the CT, 512×512. This translates into a voxel spacing within the slice plane of several millimeters for PET/SPECT and one millimeter or less for CT/MRI. The spacing between slices is determined by the distance the patient table is advanced each time. Although it can be set almost arbitrarily, a natural lower limit is given by the coarseness of the imaging device and an upper, by the dose of radiation the patient is exposed to. Spacings of several millimeters are the norm for all tomographs.

1.1.3. Modality Fusion

An important observation to be made about figure 1.2 is that the physiological and anatomical images provide complementary information which would benefit from being combined into a single dataset. Numerous pathological conditions ranging from tumors to defects in the nervous system may be diagnosed by identifying abnormal metabolic activity. In order to distinguish normal from abnormal, each center of activity needs to be associated with its position in the patient's anatomy. A physiological tomographic scan indicates metabolic activity but fails to provide the necessary frame of reference. An anatomical scan truthfully represents the patient's anatomy but contains no information

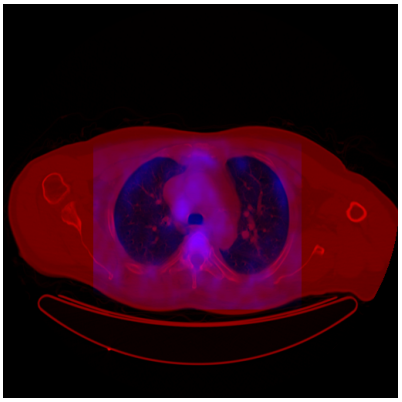


Figure 1.3: Fusion of the slices from figure 1.2; physiological activity shown in blue, anatomical reference in red

about metabolism. Only when fused into one image would the two scans allow for a fast and precise localization of pathological conditions (figure 1.3).

The slices from two scans cannot be simply overlaid because they differ in more than just their modality. They have been taken at different resolutions, may be spaced differently, often show overlapping but not identical regions of the body and capture the patient in similar but never completely equal poses. Instead, the two datasets must be treated as volumetric images that are to be properly aligned in all three dimensions before they may be fused.

1.2. Registration

The process of geometrically aligning a pair of images with each other is known as registration. It is an important technique used throughout image computation whenever the difference between two images is to be assessed or their information contents combined. Surveys of registration methods such as [Bro92] and [ZF03] identify four types of registration problem:

Multi-View The images show either the same scene from different points of view or different regions of a larger scene.

Multi-Temporal The images show the same scene but have been taken at different points in time.

Multi-Modal The images have been acquired using different types of sensors and contain complementary information.

Scene-to-Model One of the images shows an actual view of the scene while the other is a model of what should be visible.

1.2.1. Application Areas

Each problem type is encountered in a wide range of applications. Some of the most prominent examples from three major areas are:

1.2.1.1. Remote Sensing

Satellite and aerial photography provide images of the earth's surface taken from high above. They may be evaluated to conveniently monitor a number of factors, ranging from land use to pollution levels, cattle migration or weather conditions. *Multi-temporal* registration is used to align images taken at different times, often in regular intervals, allowing for the assessment of changes such as increases in pollution levels or the movement of clouds. *Multi-modal* registration is used to fuse the readings from different types of sensors, such as visible light and infra-red cameras, and provide a more complete representation of atmospheric conditions. *Scene-to-model* registration is employed to align camera images with maps of the region, providing geographic information and allowing for the precise localization of areas of interest. *Multi-view* registration is performed to stitch a larger view from several images of smaller regions.

1.2.1.2. Computer Vision

Multi-view registration is used in stereoscopic vision, employed for example by autonomous robots or face recognition systems. The images from two cameras located a small distance apart are fused to obtain depth information, analogous to the mechanisms at work in human and animal vision. *Multi-temporal* registration can be used in camera based security and intrusion detection systems. Changes in the image indicate motion and, if not explained by a known process in the scene, the presence of an intruder. *Scene-to-model* registration is employed in object recognition, most notably for biometric access control. An image of a person's face or fingerprint is aligned with template images stored in a database and the degree of correspondence with each template is evaluated. A correspondence above a certain threshold constitutes a match and access is granted.

1.2.1.3. Medical Image Computation

All four types of registration problem are encountered within this application area. *Multi-view* registration is used when multiple X-ray images taken from different directions are aligned to get a three-dimensional impression. Another scenario is in the matching of a two-dimensional X-ray image to a three-dimensional body scan. *Multi-temporal* registration serves to assess various changes in the human body over time. It can be used to monitor tumor growth or the progression of metabolic processes and to evaluate the results of an operation by aligning pre- and postoperative images. *Scene-to-model* registration is employed when matching a view of the patient's body to one obtained from an anatomic atlas. A frequent application of this technique is in neurology, where a three-dimensional scan of the patient's brain is aligned with a brain

model to identify the location of brain centers. This allows for a better assessment of damages to the brain or the planning of operations. *Multi-modal* registration is used to align datasets from different tomographic scanners. This is the problem that needs to be solved when fusing an anatomical with a physiological view of the patient's body. It is described in more detail in section 1.3.

1.2.2. General Problem Statement

In all registration scenarios, the two images given are referred to as the *reference* and *registered* image. Each is expressed as an array of intensity values over an associated coordinate frame. The coordinate system has its origin at a corner of the image and axes aligned with its sides. The intensity of the reference image at a point $\mathbf{x}_u$ in reference coordinates is $u(\mathbf{x}_u)$. Similarly, the intensity of the registered image at a point $\mathbf{x}_v$ in registered coordinates is $v(\mathbf{x}_v)$. For registration, a transformation T from reference to registered coordinate frame is used:

$$\mathbf{x}_v = T(\mathbf{x}_u) \tag{1.1}$$

This transformation maps a point $\mathbf{x}_v$ in the registered image to each point $\mathbf{x}_u$ in the reference image. It must be an injective function so that the registered image is distorted but not folded over itself. The reference image is unaffected by the transformation and always retains its original shape. A formal definition of the registration problem is given as:

Definition (Registration Problem)**.** Determine an injective transformation T that will assign the corresponding point $\mathbf{x}_v$ in the registered image to each point $\mathbf{x}_u$ in the reference image.

After the correct mapping has been determined, the information contents of the two images may be fused. For each point $\mathbf{x}_u$ in reference coordinates, $u(\mathbf{x}_u)$ is the intensity of the reference image and $v(T(\mathbf{x}_u))$ that of the registered image.

1.2.3. Problem Specification

While the registration problem is easily stated, it is in most cases very difficult to solve. Due to the diversity of imaging techniques and the various scenarios in which registration is encountered, it is not possible to provide a single algorithm that will determine the optimal alignment for any pair of images. Instead, a wide range of techniques have been developed, focusing on particular application areas or attempting to be as general as possible. To date, more than two thousand papers have been written on image registration, often suggesting small changes and tweaks to previously published algorithms. This has lead to a complex and hard to navigate collection of approaches.

Which of the many techniques may be used depends on the exact nature of the problem that is to be solved. The most important criteria by which a registration problem is specified are:

Type of Registration Problem As described in section 1.2, there are four basic types of registration problem. The type of problem has great influence on the registration process. For example, multi-temporal registration must be robust to changes is lighting conditions while multi-modal and model-to-scene registration cannot even rely on the same object having a similar intensity in both images.

Dimensionality of the Input Data Dimensionality influences the type of transformation as well as the overall complexity of the problem. Three combinations of image dimensionalities are commonly encountered:

2D In the simplest and best studied case, two planar images are aligned with each other. $T : \mathbb{R}^2 \rightarrow \mathbb{R}^2$

2D – 3D This is a special case where a planar image is positioned within a volumetric image. $T : \mathbb{R}^3 \rightarrow \mathbb{R}^2$

3D In the last case, two volumetric images are aligned. $T : \mathbb{R}^3 \rightarrow \mathbb{R}^3$

The registration of volumetric images is far more challenging than that of their planar counterparts. There is an order of magnitude more data to be processed and misalignment may be present in all three dimensions.

Source, Kind and Magnitude of Misalignment Many types of misalignments can be undone by registration. They range from the most simple one-dimensional shift to complex deformations that vary throughout the image. For a misalignment to be rectified, a transformation T needs to be chosen that can express the required mapping in its entire complexity.

Expected Results It is not always desirable or necessary to perfectly align the two images. In scene-to-model registration applied to face recognition, the features of the model should not be warped to coincide precisely with those of the face in the camera image. The faces should only be overlaid to highlight any differences which are then used to determine whether there is a match or not.

Computation Time Another reason to accept less than perfect registration is the number of computations required. The calculation of a transformation T that reverses the effects of a complex misalignment can be very time-consuming. It can be accelerated by determining a rough estimate instead of the exact solution. This is frequently achieved by choosing a simpler kind of transformation which accounts only for the most important sources of misregistration, leaving minor distortions uncorrected. In some settings, such as the matching of visual terrain data with maps for missile guidance control, real-time performance is required. Computational efficiency varies greatly between registration techniques.

Permissibility of Manual Intervention Computer-implemented registration is meant to replace the tedious and error-prone manual alignment of images. Still, the difficulty of determining what constitutes a correct registration makes human input necessary in some cases. Semi-automatic registration techniques can achieve highly

qualitative results in a short time when directed by a skilled operator. Fully automatic methods have to solve a harder problem which may take more time or lead to less precise results, but ultimately they are the goal, allowing a computer to perform registration without help from a human.

Contextual Information Additional information available about the particular application domain may be beneficial to the registration process. For example, when aligning two satellite images it is possible to match the positions of roads and buildings instead of working with raw intensities. Multi-modal registration can benefit from prior knowledge about the correspondence of intensities in the two images. Techniques meant to be used across a wide range of problems must be carefully constructed not to make any assumptions about the data which may only hold in some scenarios.

1.2.4. Classification of Techniques

Despite the large number of registration methods developed and their sometimes high degree of specialization, most approaches follow the simple iterative scheme shown in algorithm 1.1.

Algorithm 1.1 Registration process

$T \leftarrow$ initial guess
repeat
  calculate *similarity metric*
  adjust T according to *search strategy*
until similarity above threshold

A particular registration technique is characterized by the choices made for the three major components of this algorithm:

Transformation Type The transformation type determines the family of functions from which T can be selected. It affects the registration process in several ways. First, it decides what kind of misalignment can be reversed as adjustments which the transformation type is unable to express cannot be performed. Second, it also directly affects computation time because a more complex type has more degrees of freedom, or parameters, for which adjustments need to be calculated during each loop iteration. When choosing a transformation type, the opposing goals of undoing complex misalignments and achieving fast registration must be weighted and a compromise found.

As each setting of the parameters directly corresponds to a particular transformation function, both the set of parameters and the function they select are referred to by the same letter T in this thesis.

The choice of an initial transformation has great influence on the registration process, determining how quickly the transformation converges and whether it

converges at all. Unfortunately, when no information about the misalignment of the two images is available, automatic generation of a good first guess may be impossible. Since only a raw estimate of the correct alignment is required and humans can quickly assess the general relation of two images, this initial step can be efficiently performed by a human operator. Another possibility is to begin with an identity transform, that is to initially overlay the images as if no misalignment was present.

Similarity Metric The similarity metric is used to assess the quality of alignment that is achieved by T and to calculate the information required to improve it. If the existing misalignment can be precisely determined, a transformation that corrects it may be computed directly and there is no need for an iterative process. In most cases, only an estimate of the alignment quality is available. A reliable estimate is important for the registration process as it is the only means by which the progress of the registration can be evaluated and a direction chosen for the next loop iteration. The metric needs to provide sufficient information for the adjustment of each parameter offered by the transformation type. This limits the possible choices of similarity metric depending on the type of transformation selected and the kind of information required by the search strategy.

Search Strategy A search strategy uses the information made available by the similarity metric to modify T in an attempt to improve the alignment. A more complex strategy may be able to achieve registration in fewer loop iterations or to better cope with inconsistencies in the data provided by the similarity metric.

1.3. Multi-Modal Medical Registration

This section addresses the specific registration problem that arises when aligning physiological with anatomical tomographic scans. The problem is first expressed in terms of the criteria identified in section 1.2.3. A registration technique is then presented which has been developed to efficiently solve this particular kind of problem.

1.3.1. Problem Specification

The problem type and dimensionality of the data are clearly those of multi-modal 3D registration. For every $\mathbf{x}_u \in \mathbb{R}^3$, $u(\mathbf{x}_u)$ is the intensity of the reference scan at $\mathbf{x}_u$ and $v(T(\mathbf{x}_u))$ that of the registered scan at the associated position $T(\mathbf{x}_u)$. How the intensity is estimated if a position falls between voxel centers or outside the image differs between registration techniques. Another choice that needs to be made individually for each technique is which of the two scans is to be treated as the reference and which as the registered image. Depending on the properties of the techniques, this may or may not have an influence on the result.

The kind and magnitude of misalignment to be corrected are not well defined. Clinical staff may have taken care to position the patient in very similar poses for the two scans

or there may be significant deviations. Inevitable metabolic processes could have led to changes in the exact shape and position of internal organs. If a large amount of time has passed between the scans being taken, the patient's condition may have progressed and their anatomy changed more radically.

The expected results are again easier to state as the primary goal is as precise a registration as possible. Anatomical references are only useful in the assessment of a physiological scan when there is a high degree of confidence in their correctness.

Short computation time also is of high priority. In a clinical setting, long waiting times while the data is being prepared are not acceptable. Considering that volumetric datasets are to be aligned and the transformation should be able to undo some degree of changes in the shapes of internal organs, a very efficient registration method is required. A transformation type should be chosen that is able to express the necessary mapping while offering the least possible number of parameters to be adjusted. The similarity metric must perform its calculations quickly despite the large amount of data in the two images. If the metric is made faster at the expense of precision, the search strategy must be able to cope with this imprecision and still complete the registration in a small number of iterations.

A need for manual intervention is not desired but can be tolerated at a moderate level. If only small regions within the two scans are relevant to the diagnosis, the registration process may be accelerated by a human operator specifying the regions of interest. A human could also be asked to provide a rough initial alignment of the two datasets.

The amount of contextual information available depends on the desired degree of specialization of the registration technique. As shown in sections 1.1.1 and 1.1.2, there are several methods for the acquisition of tomographic data and the images they produce vary in their appearance. If only a specific combination such as PET to CT registration is to be considered, the particular characteristics of those two modalities may be used. For a more general approach, it should only be assumed that one of the images will provide physiological and the other anatomical data.

1.3.2. Proposed Technique

Using the metadata provided by tomographic scanners, it is possible to automatically match the scales of the two datasets. However, rescaling one of the two images to match the size of the other may lead to a loss of information. A different approach is therefore proposed. $\mathbf{x}_u$ and $\mathbf{x}_v$ are both expressed in millimeters throughout the registration process and only when accessing image intensities via the functions u and v, they are implicitly converted to voxel coordinates. In addition to a scaling, this involves a translation because the origin in an array of intensities refers to the center of the first voxel, not its corner.

To transparently handle a wide range of modalities, intensities are normalized to an integer scale of $[0, 255]$ for each image. Voxels that correspond to positions outside the patient's body attain intensities of zero. This value is therefore also substituted when $\mathbf{x}_u$ or $T(\mathbf{x}_u)$ point to locations outside the volumetric datasets. For positions that fall between voxel centers, trilinear interpolation is used. It estimates the intensity from

those of up to eight surrounding voxels, weighted by the distances of their centers to the desired position.

The registration technique is based on the general iterative scheme of algorithm 1.1. It can therefore be characterized by the choices made for the algorithm's three main components. Brief overviews are provided here and each component is addressed in more detail in its own dedicated chapter later in this thesis:

Transformation Type To narrow down the range of potential misalignments, the assumption is made that the two scans have been taken in close succession and it has been attempted to reach similar poses. This avoids the need to cope with large anatomical differences but cannot eliminate the possibility of small changes in the shapes of internal organs due to metabolic processes.

The images are initially positioned so that their centers coincide. A transformation is then applied which treats the registered image as completely rigid. The image can be translated and rotated freely in three-dimensional space by adjusting six parameters. This allows the general body shapes to be aligned.

Once the parameter values have been found that maximize the similarity metric, a non-rigid transformation type is used. A pattern of control nodes is constructed. Each node, when moved, leads to a deformation of the registered image that is strongest at the node's position and decreases with distance. By adjusting the node positions, complex deformations can be applied to the registered image, undoing local changes in the patient's anatomy. The effects of each node can be considered approximatively local and independent of the other nodes. To assess the alignment achieved, the similarity metric therefore only needs to be evaluated in small regions around each node.

Similarity Metric The goal is the construction of a registration method that is able to align all types of anatomical and physiological tomographic scans. No assumption about a specific relationship between the intensities in the two images is made. The similarity metric chosen is the information-theoretic measure of mutual information. It is able to assess the quality of alignment between the images without needing any further contextual information.

The mutual information is calculated using not the entire images but a randomly selected small subset of points. This results in a metric that can be very efficiently evaluated even for large volumetric datasets but due to its stochastic nature provides only noisy estimates of alignment quality. In addition to the mutual information itself, derivatives with respect to the parameters T of the transformation type can be obtained. They are determined using the same stochastic approach and are also noisy.

Search Strategy The search strategy is stochastic gradient ascent. Each transformation parameter is adjusted by adding a number proportional to the derivative of the similarity metric with respect to its value. This simple technique is not normally

suitable for a randomized metric as it gets stuck in local extrema. However, because the derivatives are also noisy, they do not point directly toward the nearest extremum and the search is able to overcome maxima introduced by noise if their magnitude is sufficiently small. To improve efficiency, a multiresolution approach is used whereby reduced versions of the images are aligned first and their sizes increased as registration progresses.

1.4. Overview of the Thesis

The problem of multi-modal medical registration and a technique for its solution were introduced in this chapter. Next, the three main components of the technique are addressed in more detail. For each component, the method chosen in this thesis is developed and compared to other established approaches. The similarity metric is considered in chapter 2, the search strategy in 3 and the two transformation types in 4.

Chapter 5 describes the implementation of an actual registration system that employs the proposed technique to align tomographic datasets. In chapter 6, GPU based implementations of the most time-consuming computation steps are presented. An overview of the paradigms employed by a graphics card is given, their use for general purpose programming explained and the implementation of the registration process described.

Chapter 7 provides experimental results indicating the registration quality and speed achieved by the CPU and GPU based implementations. Chapter 8 is a discussion of the registration system, analyzing the degree to which the desired goals have been met and suggesting directions of future work.

1.5. Notations

The notations used in this thesis for vectors, matrices and their components are summarized in table 1.1.

Example Expressions	Description
$\mathbf{x}$, $\mathbf{q}_i$	Column vector
$(\mathbf{x})_k$, $(\mathbf{q}_i)_k$	kth component of vector
x_k	Short hand notation for the above used only when the vector's name contains no subscripts
$(\mathbf{x})_x$, $(\mathbf{x})_y$, $(\mathbf{x})_z$, $(\mathbf{x})_w$	Alternative notations for the up to four components of a coordinate vector
$\mathbf{A}$, $\mathbf{M}_T$	Matrix
$(\mathbf{A})_{ij}$, $(\mathbf{M}_T)_{ij}$	Matrix element in ith row and jth column
A_{ij}	Short hand notation for the above used only when the matrix name contains no subscripts
$\operatorname{grad} f(\mathbf{x})$	Gradient of f, defined as a row vector

Table 1.1: Notations

2. Similarity Metric

The similarity metric serves two purposes. It estimates the quality of alignment achieved by the current transformation T and, more importantly, generates the information required to further improve the alignment. The choice of similarity metric is crucial in the design of a registration technique. If the metric provides unreliable results, it is not possible to adjust the transformation and correct the misalignment between the two images. Occasional inconsistencies may be compensated by the search strategy at the expense of a slower registration process that requires more iterations. Systematic failure of the similarity metric makes registration impossible as it leads to either a search that never converges or one that arrives at an incorrect final alignment.

The primary factor affecting the difficulty of assessing the current alignment is the nature of the data to be registered. [Bro92] divides the possible variations between the two images into three categories, each of which has a different influence on the similarity metric:

Corrected Distortions These are the differences between the two images that are to be undone by registration. The similarity metric needs to determine the level of corrected distortions still present in the registered image and the changes to the transformation required to correct them.

Uncorrected Distortions Uncorrected distortions are those which are not desired but will not be corrected by the registration process. Examples are differences in lighting between the two images or geometric deformations too complex for the chosen transformation type to address. The similarity metric must be robust to this type of distortions and not take them into account when assessing the current alignment.

Variations of Interest This category encompasses the differences that are to be highlighted and not removed by registration. In multi-modal registration, the variations of interest are the intensity differences of corresponding voxels. The metric must also be robust to this class of distortions so that they do not affect its evaluation of the alignment.

An important aspect in 3D registration is computational efficiency. A similarity metric whose calculation time scales linearly with the size of the image may work well for 2D registration but will take a very long time to compute when used on volumetric data. Metrics have been developed that address this concern. While efficient to compute, they are also robust to a wide range of uncorrected distortions and variations of interest, achieving fast and accurate assessment even with large volumes of data.

2.1. Survey of Metrics

The next two sections explore the similarity metrics used by existing registration techniques, focusing on their applicability to multi-modal medical 3D registration. As with the techniques themselves, a great number of different metrics have been proposed many of which differ only in small details. Such similar metrics are grouped and evaluated as common families. For more comprehensive treatments that list more members of each family, see [Bro92, ZF03].

2.1.1. Feature Based Approaches

Similarity metrics fall into two large classes. The first is that of feature based approaches. They introduce a preprocessing step in which salient features are identified in both images. When assessing the alignment of the two images later on, only the positions of these features are considered. The feature extraction may be computationally expensive as it has to be performed only once at the beginning of the registration process. This allows for a very careful selection of features identifying those that precisely represent corrected distortions while being insensitive to uncorrected distortions and variations of interest.

In order to achieve the desired level of robustness and performance, a featureset needs to be chosen that is suitable for the nature of the images being registered. Three families of features are identified in [ZF03]:

Lines

Many algorithms exist for the fast and precise extraction of edges and contours, from the simple intensity gradient based Canny edge detector [Can86] to sophisticated Snake based approaches [KWT88]. Line features are invariant to many types of uncorrected distortions. They are also invariant to the variations of interest in multi-modal image registration as long as the contours of the objects in the scene are discernable in both images. The disadvantage of using lines as features is that it is generally not possible to determine which lines in the two images correspond to each other. The search strategy may thus align incorrect pairs of lines leading to misregistration.

Regions

Closed-boundary regions in an image may be found by segmentation [PP93]. They are frequently represented by points whose locations are determined by the properties of the regions. A prominent example are centers of gravity. While region features may take longer to identify, their smaller number makes the assessment of alignment between two images more efficient. Depending on the nature of the data, additional properties such as region shapes and sizes may be invariant to uncorrected distortions and variations of interest. Such additional information makes it possible to directly identify feature pairs and avoid misregistration.

Points

Points selected as features are either specific locations on the edges of an image such as line intersections or points of high curvature or positions where highly distinguishable changes in intensity occur. The kind of point feature to be used can be adapted very precisely to the nature of the data. If the point type is chosen so that only a few highly distinct features are found, fast registration with the potential of directly identifying feature pairs is again possible.

2.1.1.1. Shortcomings

All three families of features share a serious problem when applied to multi-modal medical 3D registration. They depend on image properties which are not fulfilled by the two tomographic scans being registered. As seen in figure 1.2a, the functional data returned by a PET scan is of low sharpness and resolution. The lack of sharp edges impedes the performance of line based similarity metrics. Regions, even if they can be identified despite their blurred borders, often do not correspond directly to regions in the anatomical scan as evidenced by the fusion of the slices in figure 1.3. Point based metrics working with locations on edges suffer from the inability to find edges in the first place. Point based metrics indicating distinctive changes in intensity also fail as changes occur gradually throughout the image in the physiological scan and are concentrated around the edges of homogeneously colored areas in the anatomical scan.

An additional shortcoming of many feature based metrics is their strong reliance on contextual information. Better performance is achieved if more distinctive features can be identified. This leads to approaches based on high-level features suitable to a single application only. For example, [Rou96] details a technique for the alignment of satellite images which matches urban areas and crossroads instead of generic areas and line crossings. This tendency is not desirable in multi-modal medical registration. A general method is sought that is able to register PET, SPECT and fMRI images with CT and PET scans despite the differences between these modalities.

2.1.2. Image Based Approaches

The second class of similarity metrics are image based approaches. Instead of identifying a set of features, they work directly with raw image data. To assess the quality of alignment achieved by a transformation T, pairs of coinciding intensities $u(\mathbf{x}_u)$ from the reference and $v(T(\mathbf{x}_u))$ from the registered image are compared. There are two different ways to evaluate the alignment. The first possibility is to compare intensities from the entire images. The result is a single numerical quantity expressing the degree to which the registered and reference image agree. By maximizing this quantity, the correct alignment can be found.

The other option is the use of smaller windows arranged within the images in a predefined pattern. Each window in the reference image is compared to all windows in the registered image in order to determine which pairs constitute the closest matches. The distance between matching windows then serves as a measure of alignment quality and

the transformation is adjusted to minimize it. This metric should not be confused with techniques that use image regions as features. The two important differences are that there is no preprocessing step as windows are positioned irrespective of actual image contents and that to determine matching pairs of windows, the intensities of the points they contain are compared.

Whether windows are employed or not, image based approaches work by assessing the level of agreement between two sets of intensities, one obtained from the reference image and the other from the corresponding points of the registered image. An overview of classical techniques is given in [Lew95]. They can be divided into two families:

Cross-Correlation

While cross-correlation is often directly defined [GW06], it can also be derived from another similarity metric. For S a set of points in the coordinate frame of the reference image, an intuitive measure of the similarity between the two images is the squared Euclidean distance of their intensities at these points:

$$d^2_{u,v}(T) = \sum_{\mathbf{x}_u \in S} \left[u(\mathbf{x}_u) - v(T(\mathbf{x}_u))\right]^2 \tag{2.1}$$

Under the assumption of constant energy throughout the registered image, the only term varying under T is the cross-correlation:

$$C_{u,v}(T) = \sum_{\mathbf{x}_u \in S} u(\mathbf{x}_u)\, v(T(\mathbf{x}_u)) \tag{2.2}$$

For most images, the energy distribution is not constant. Cross-correlation can be normalized not to favor bright areas in such cases:

$$NC_{u,v}(T) = \frac{\sum_{\mathbf{x}_u \in S} u(\mathbf{x}_u)\, v(T(\mathbf{x}_u))}{\sqrt{\sum_{\mathbf{x}_u \in S} v^2(T(\mathbf{x}_u))}} \tag{2.3}$$

Even when normalized, cross-correlation is not an absolute measure. A related quantity which expresses image similarity on the absolute scale of $[-1, 1]$ is the normalized correlation coefficient. With $\overline{u}$ the mean of $u(\mathbf{x}_u)$ and $\overline{v}$ the mean of $v(T(\mathbf{x}_u))$ over all $\mathbf{x}_u \in S$, it is given as:

$$NCC_{u,v}(T) = \frac{\sum_{\mathbf{x}_u \in S} \left[u(\mathbf{x}_u) - \overline{u}\right]\left[v(T(\mathbf{x}_u)) - \overline{v}\right]}{\sqrt{\sum_{\mathbf{x}_u \in S} \left[u(\mathbf{x}_u) - \overline{u}\right]^2} \sqrt{\sum_{\mathbf{x}_u \in S} \left[v(T(\mathbf{x}_u)) - \overline{v}\right]^2}} \tag{2.4}$$

All of these techniques require a large number of multiplications to be performed for each pair of images or windows being compared. [Lew95] lists several ways in which the calculations can be accelerated to somewhat reduce the problem. Prefiltering can be

used to decrease variations in image energy making normalization unnecessary. After rewriting the equations, some values can be tabulated. Finally, the Correlation Theorem allows a calculation in the frequency domain after the images have been transformed via FFT.

Sequential Detection

Sequential similarity detection algorithms were first introduced by [BS72] to improve computational efficiency over cross-correlation based metrics. Instead of the squared Euclidean distance between image intensities, they use absolute differences, eliminating the need for multiplications. With $\overline{u}$ and $\overline{v}$ mean intensities as defined above, the plain and normalized similarity metrics are:

$$E_{u,v}(T) = \sum_{\mathbf{x}_u \in S} |u(\mathbf{x}_u) - v(T(\mathbf{x}_u))| \tag{2.5}$$

$$NE_{u,v}(T) = \sum_{\mathbf{x}_u \in S} |u(\mathbf{x}_u) - \overline{u} - v(T(\mathbf{x}_u)) + \overline{v}| \tag{2.6}$$

A sum is not evaluated fully but only until its value exceeds a certain threshold. The faster this threshold is reached, the better the correspondence is deemed to be. The result of this faster calculation is correct only if the points that were looked at are a representative sample of all points in S. By traversing the members of S in random order and setting the threshold high enough for a significant number of points to be evaluated each time, a very high probability of correct assessment can be achieved.

2.1.2.1. Shortcomings

Numerous variations of the metrics described above exist, many of which are listed in [Bro92] and [ZF03]. A fundamental weakness shared by all is the assumption of a linear relationship between the intensities of corresponding points in the two images. Bright points in the reference image are matched to bright points in the registered image and similarly with dark points. This assumption is violated in multi-modal registration, where, as seen in figure 1.3, the relationship between the two images can be more complex. Bright areas in one image often correspond to dark areas in the other.

2.2. Mutual Information

Mutual information is a recently introduced similarity metric that addresses many of the shortcomings found in traditional image based approaches. By also working on raw image intensities, it does not introduce any of the additional requirements of a feature based solution. There is no need for a preprocessing step and no dependency on detailed contextual information. As is shown in the remainder of this chapter, mutual information can be calculated efficiently so as to be suitable for 3D registration and is able to align images of different modalities.

2.3. Derivation

The concept of mutual information was first suggested by Claude Shannon in [Sha48] more than fifty years ago. This paper started the field of information technology and mutual information was one of its first tools. The definition of mutual information is given not in the context of image registration but in that of random variables. In the next two sections, the required concepts from probability theory are introduced. They are described and their notations presented, but no formal definitions are given. For a more complete and formal treatment of probability theory, see [Bil95].

2.3.1. Discrete Random Variables

A *random variable* X is a function whose numerical value is a priori unknown and depends on the outcome of a random experiment. The possible outcomes are given by a set Δ_X, referred to as the *sample space*. If the set contains a finite or countable number of elements, the random variable is *discrete*; otherwise, it is *continuous*.

Only discrete random variables are used throughout this thesis. Some of the variables take on real numbers as their values, others vectors of such numbers. Each of the following concepts is applicable in both cases. The value of a discrete random variable such as $\mathbf{x}$ may therefore stand either for a scalar or a vector depending on the type of variable. Only where a scalar is explicitly assumed the notation x is used.

The number of possible values for X is $n = |\Delta_X|$. The likeliness of X taking on a value $\mathbf{x}_i \in \Delta_X$ is proportional to its *probability* $P[X = \mathbf{x}_i]$. Probabilities are assigned to the $\mathbf{x}_i$ by a *density function* f_X so that $P[X = \mathbf{x}_i] = f_X(\mathbf{x}_i)$. The density function satisfies three conditions:

$$\forall\, \mathbf{x} \in \Delta_X : \; 0 \leq f_X(\mathbf{x}) \leq 1 \tag{2.7}$$

$$\forall\, \mathbf{x} \notin \Delta_X : \; f_X(\mathbf{x}) = 0 \tag{2.8}$$

$$\sum_{i=1}^{n} f_X(\mathbf{x}_i) = 1 \tag{2.9}$$

When a discrete random variable is evaluated multiple times and the values recorded, a *sample* is generated. As the sample gets larger, the average value of its elements converges to the random variable's *expected value*, which is defined as:

$$E[X] = \sum_{i=1}^{n} \mathbf{x}_i f_X(\mathbf{x}_i) \tag{2.10}$$

Applying a function g to each outcome creates a new random variable $g(X)$ with an expected value of:

$$E[g(X)] = \sum_{i=1}^{n} g(\mathbf{x}_i) f_X(\mathbf{x}_i) \tag{2.11}$$

2.3.2. Pairs of Discrete Random Variables

A number of measures exist that express various aspects of the relationship between a pair of discrete random variables X and Y. To analyze this relationship, a new random variable $Z = (X, Y)^T$ is constructed. The values it takes on are vectors $\mathbf{z}_k = (\mathbf{x}_i, \mathbf{y}_j)^T$ where $\mathbf{x}_i$ is the outcome of X and $\mathbf{y}_j$ that of Y. The sample space is $\Delta_Z = \Delta_X \times \Delta_Y$, each element of which as assigned a probability by the density function f_Z:

$$\begin{aligned} P\left[Z = \mathbf{z}_k\right] &= f_Z\left(\mathbf{z}_k\right) \\ \Leftrightarrow P\left[X = \mathbf{x}_i \wedge Y = \mathbf{y}_j\right] &= f_Z\left(\mathbf{x}_i, \mathbf{y}_j\right) \end{aligned} \tag{2.12}$$

f_Z is the *joint density function* of X and Y as it represents the probability of the two variables jointly taking on a pair $(\mathbf{x}_i, \mathbf{y}_j)$ of values. Although it is usually named $f_{X,Y}$ with no reference made to Z, each joint density function always is the density function of an auxiliary random variable and all statements made about plain density functions apply to it as well.

The density functions of X and Y may be reconstructed from $f_{X,Y}$ by calculating the *marginal densities*:

$$f_X\left(\mathbf{x}\right) = \sum_{\mathbf{y}_j \in \Delta_Y} f_{X,Y}\left(\mathbf{x}, \mathbf{y}_j\right) \tag{2.13}$$

$$f_Y\left(\mathbf{y}\right) = \sum_{\mathbf{x}_i \in \Delta_X} f_{X,Y}\left(\mathbf{x}_i, \mathbf{y}\right) \tag{2.14}$$

The *conditional density function* $f_{X|Y=\mathbf{y}_j}$ gives the *conditional probability* of X taking on a value $\mathbf{x}_i$ when the value Y is known to be $\mathbf{y}_j$. It is calculated by dividing the joint density function of the two variables by the density function of Y:

$$f_{X|Y=\mathbf{y}_j}\left(\mathbf{x}_i\right) = \frac{f_{X,Y}\left(\mathbf{x}_i, \mathbf{y}_j\right)}{f_Y\left(\mathbf{y}_j\right)} \tag{2.15}$$

By studying the conditional density function, it can be determined whether the two variables are *independent* or not. If there is no dependency, the value of X is not influenced by that of Y in any way. No matter which value $\mathbf{y}_j$ is taken on by Y, X always has the same density function f_X:

$$\forall\ \mathbf{x}_i \in \Delta_X, \mathbf{y}_j \in \Delta_Y :\ f_{X|Y=\mathbf{y}_j}\left(\mathbf{x}_i\right) = f_X\left(\mathbf{x}_i\right) \tag{2.16}$$

After inserting equation (2.15) into (2.16), the formal definition of independence between X and Y is obtained:

$$\forall\ \mathbf{x}_i \in \Delta_X, \mathbf{y}_j \in \Delta_Y :\ f_X\left(\mathbf{x}_i\right) f_Y\left(\mathbf{y}_j\right) = f_{X,Y}\left(\mathbf{x}_i, \mathbf{y}_j\right) \tag{2.17}$$

If equation (2.17) is violated, the two discrete random variables are not independent. Basic probability theory does not provide any way to quantify this dependency. The problem is addressed by mutual information, which is a measure of the level of dependency between X and Y.

2.3.3. Entropy

The first important concept introduced in [Sha48] is that of the *entropy* of a discrete random variable. It expresses the randomness of the variable's outcome. High entropy indicates that the variable can take on many different values with similar probability. The other extreme is an entropy of zero which means that there is only one possible value and no uncertainty exists. Given a set of additional formal criteria, Shannon shows that, save for a scaling constant, the only possible definition of entropy is:

$$H[X] = -\sum_{\mathbf{x}_i \in \Delta_X} f_X(\mathbf{x}_i) \log f_X(\mathbf{x}_i) \tag{2.18}$$

By virtue of equation (2.11), this may be written as:

$$H[X] = -E[\log f_X(X)] \tag{2.19}$$

Because scaling by an arbitrary positive factor is permitted, the base of the logarithm may be chosen freely. If a base two logarithm and no additional scaling factor are used, an alternative interpretation of entropy is possible. It can be understood as a measure of the amount of information generated by X. Given full knowledge of the process used to generate the values of X, an optimal coding scheme is chosen that, on average, is able to encode a sample of values obtained from X in the least number of bits. The entropy $H[X]$ is then the average number of bits required to encode a single outcome using this scheme.

A simple example that illustrates the concept is given in [Sha48]. X is a random variable that indicates the result of a coin toss. The coin is biased so that heads occurs with a probability of p and tails with $q = 1 - p$. The entropy of X, as defined by equation (2.18), is:

$$H[X] = -(p \log p + q \log q) \tag{2.20}$$

The entropy is plotted in figure 2.1 as a function of $0 \leq p \leq 1$. When $p = 0$, the coin always shows heads and no bits are required to encode the result. As p increases, there is a growing chance of the coin falling tails up. Because tails is still much less likely than heads, an optimal code is able to express long streaks of heads with a symbol consisting of a few bits so that on average, only a fraction of a bit is required to encode each outcome. With p approaching 0.5, heads and tails are almost equally likely and the code cannot save as many bits by more efficiently encoding heads. At $p = 0.5$, each toss has exactly a 50% chance of being heads or tails. The completely unpredictable result cannot be stored any more efficiently than with an entire bit per outcome. For $p > 0$, analogous situations occur with the roles of heads and tails exchanged.

Instead of being interpreted as information contents in bits, figure 2.1 can also be understood as a plot of a dimensionless measure of randomness. When there is an equal chance for heads or tails, X is most random. With the bias increasing in either direction, the outcomes become more predictable and X is considered less random.

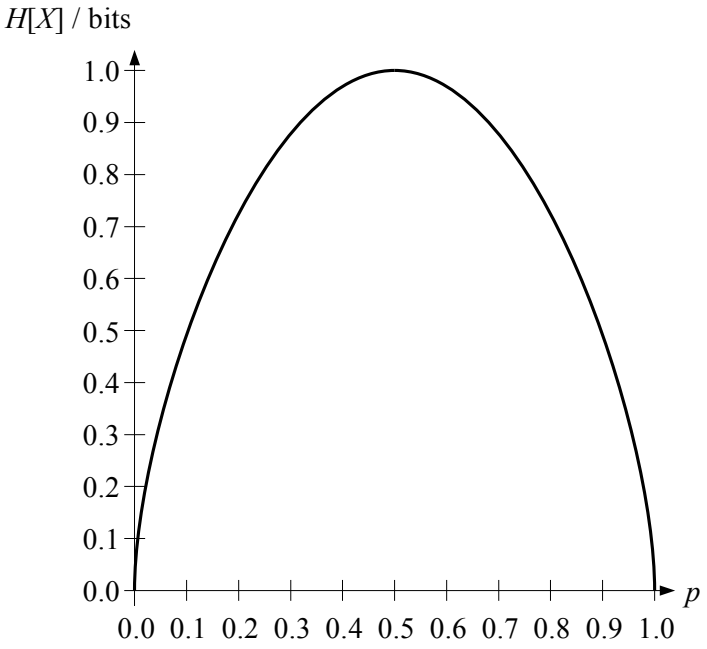


Figure 2.1: Entropy of a biased coin toss as a function of the bias

2.3.4. Joint and Conditional Entropy

As was done with the density function in the previous section, related measures can be derived from the entropy $H[X]$ for pairs of discrete random variables X, Y. The first is the *joint entropy* of X and Y. In analogy to the definition of the joint density function, it is simply the entropy of the random variable $Z = (X, Y)^T$:

$$H[X,Y] = -\sum_{\mathbf{x}_i \in \Delta_X, \mathbf{y}_j \in \Delta_Y} f_{X,Y}(\mathbf{x}_i, \mathbf{y}_j) \log f_{X,Y}(\mathbf{x}_i, \mathbf{y}_j) \tag{2.21}$$

The second measure is *conditional entropy*, which is the remaining entropy of X when the value of Y is known. It is defined under the general assumption that $\mathbf{y}_j$ is known, not that it has a particular value. The entropy of X is therefore averaged over all $\mathbf{y}_j \in \Delta_Y$ weighted by their probabilities:

$$H[X \mid Y] = -\sum_{\mathbf{x}_i \in \Delta_X, \mathbf{y}_j \in \Delta_Y} f_{X,Y}(\mathbf{x}_i, \mathbf{y}_j) \log f_{X|Y=\mathbf{y}_j}(\mathbf{x}_i) \tag{2.22}$$

Joint and conditional entropy have several useful properties. Using the log-sum inequality it can be proven that knowledge of the value of Y may decrease but never increase the entropy of X:

$$H[X \mid Y] \leq H[X] \tag{2.23}$$

Substitution of equations (2.15) and (2.14) into (2.22) leads to the corollary:

$$H[X \mid Y] = H[X,Y] - H[Y] \tag{2.24}$$

The derivations above can also be performed with the roles of X and Y exchanged:

$$H[Y \mid X] \leq H[Y] \tag{2.25}$$
$$H[Y \mid X] = H[X, Y] - H[X] \tag{2.26}$$

2.3.5. Mutual Information

In the absence of any outside knowledge, the entropy of a discrete random variable X is $H[X]$. According to equation (2.23), if the value of another discrete random variable Y is determined, it may provide some information about X and reduce its entropy. The amount by which the entropy decreases is known as *mutual information*:

$$MI(X, Y) = H[X] - H[X \mid Y] \tag{2.27}$$

Using the relationships in equations (2.24) and (2.26), alternative notations for the mutual information of X and Y may be derived:

$$MI(X, Y) = H[X] - H[X \mid Y] \tag{2.28}$$
$$= H[X] + H[Y] - H[X, Y] \tag{2.29}$$
$$= H[Y] - H[Y \mid X] \tag{2.30}$$

Equation (2.30) is identical to (2.27) with X and Y exchanged. This shows that mutual information is symmetrical and knowing the value of either variable decreases the entropy of the other by the same amount:

$$MI(Y, X) = MI(X, Y) \tag{2.31}$$

Mutual information is a significant contribution to probability theory because it finally allows the level of dependency between two discrete random variables X and Y to be quantified. As was seen in section 2.3.3, entropy can be interpreted either as a dimensionless measure of randomness or as information contents in bits. The same applies to mutual information:

Mutual Information as Dimensionless Quantity

When considered dimensionless, mutual information is expressed in terms of entropies. X has an entropy $H[X]$ which indicates how random and hard to guess its outcome is. As the value of Y is determined, it provides additional information about X, decreasing its entropy to $H[X \mid Y]$, making it less random and easier to guess. The reduction in entropy is $MI(X, Y)$. If X and Y are independent, nothing new is learned and $MI(X, Y) = 0$. With increasing level of dependency, more is revealed about X and $MI(X, Y)$ increases. The maximum is reached at $MI(X, Y) = H[X]$ when the two variables are identical and knowledge of the outcome of one instantly reveals the value of the other.

Mutual Information as Information Contents

An example is given. Consider a discrete random variable X that can take on 256 different values, each with an equal probability of $p_i = \frac{1}{256}$. According to equation (2.18), the entropy (calculated using base two logarithms) is $H[X] = 8$. The outcome of X is completely random and 8 bits are required on average to encode it. Now consider a discrete random variable Y that can take on only two values $Y = 0$ if the value of X is odd and $Y = 1$ if it is even. If the value of Y is determined, it cuts the number of possible values of X in half. According to equation (2.22), $H[X \mid Y] = 7$. The amount of new information produced by X is decreased by $MI(X, Y) = 1$ bit. As mutual information is symmetrical, knowledge of the value of X also decreases the entropy of Y by one bit. Since Y has an initial entropy of $H[X] = 1$, it drops to zero. This is logical because if the outcome of X is known, it is immediately apparent whether it is odd or even.

2.4. Application to Image Registration

The use of mutual information as a similarity metric in image registration was first suggested by [Vio95]. The intensities of the two images are interpreted as the outcomes of discrete random variables. For any point $\mathbf{x}_u$ in the reference coordinate frame, the reference image intensity $u(\mathbf{x}_u)$ is considered to be the value taken on by a random variable X. Similarly, the registered image intensity $v(T(\mathbf{x}_u))$ is the outcome a random variable Y.

If the images have been correctly aligned, $\mathbf{x}_u$ and $T(\mathbf{x}_u)$ truly are corresponding points. The assumption is made that this also means there is a relationship between their intensities. Knowing the value of $u(\mathbf{x}_u)$, certain values of $v(T(\mathbf{x}_u))$ are more and others less likely. In terms of random variables, knowledge of the outcome of X provides information about that of Y. The mutual information of the two variables is $MI(X, Y) > 0$. If, on the other hand, the two images are not correctly aligned, $u(\mathbf{x}_u)$ and $v(T(\mathbf{x}_u))$ are the intensities of completely unrelated points. Knowledge of the first provides no hints as to the value of the other. The outcome of X reveals no information about that of Y and $MI(X, Y) = 0$.

In reality, many misalignments also lead to mutual information being greater than zero. This may be because the error in alignment is very small or because completely unrelated structures have been overlaid that happen to show some relationship between their intensities. Still, if the correct alignment is found, there should be correspondence not only in some areas but throughout the entire image, leading to a maximum of $MI(X, Y)$. When using mutual information as a similarity metric, the transformation that leads to this maximum is sought.

There is a certain similarity between mutual information and the cross-correlation based techniques described in section 2.1.2. In both cases, the assumption of a relationship between the intensities $u(\mathbf{x}_u)$ and $v(T(\mathbf{x}_u))$ is made. However, while cross-correlation considers only a linear dependency where intensities are scaled by some positive factor from one image to the other, mutual information accounts for arbitrarily complex relationships. This makes mutual information suitable for multi-modal regis-

tration, where readings from different types of sensors that assign intensities based on different phenomena are registered. It should be noted that the relationship does not have to actually be known as it is never used in an explicit form. Also, the dependency may be, and in most cases is, incomplete, with $u(\mathbf{x}_u)$ providing not the exact value of $v(T(\mathbf{x}_u))$ but only some information about it.

Since both the exact information contents of the two images and the nature of their relationship are unknown, the maximal value of $MI(X,Y)$ can a priori not be determined. The registration process, in its search for a maximum, cannot know whether the present value of $MI(X,Y)$ is globally optimal or a mere local extremum. Like the other similarity metrics introduced in this chapter, mutual information cannot guarantee that the perfect alignment will be found. It is up to the search strategy to perform the registration in such a way that ending up in a local maximum is not likely.

A problem that needs to be solved before mutual information may be used as a similarity metric is that given two images, it is not directly possible to calculate $MI(X,Y)$. All of the formulas developed in section 2.3.5 require knowledge of the random variables' entropies. The calculation of these relies on the availability of density functions. The formula suggested by [Vio95] is that of equation (2.29):

$$MI(X,Y) = H[X] + H[Y] - H[X,Y] \tag{2.32}$$

As shown in sections 2.3.3 and 2.3.4, to evaluate this expression, the density functions f_X of X, f_Y of Y and their joint density function $f_{X,Y}$ are required. Since neither of them is known, they need to be somehow extracted from the image data.

2.5. Density Estimation

A series of intensity values taken from an image represents a sample of the discrete random variable associated with it. Reconstruction of the density function from such a sample is known as density estimation. The term *estimation* is used for three reasons. For one, the sample may not be representative of the exact probabilities throughout the entire image. The density function found will then not precisely fit the random variable. Second, the sample may not provide enough information to unambiguously determine a density function. For example, it can be shown that in the case of continuous random variables, there is an infinite number of functions that fit each sample. Finally, an exact solution may not always be desired. To save computation time or to obtain a simple closed form expression, a density function may calculated that only approximates the actual probabilities in the sample.

2.5.1. Gaussian Distribution

The density function of the *Gaussian*, or *normal*, distribution is an important element of several density estimation techniques. It is also encountered in many other application areas. Although this is a family of functions associated with continuous random variables, it is also very useful when working with their discrete counterparts. The density

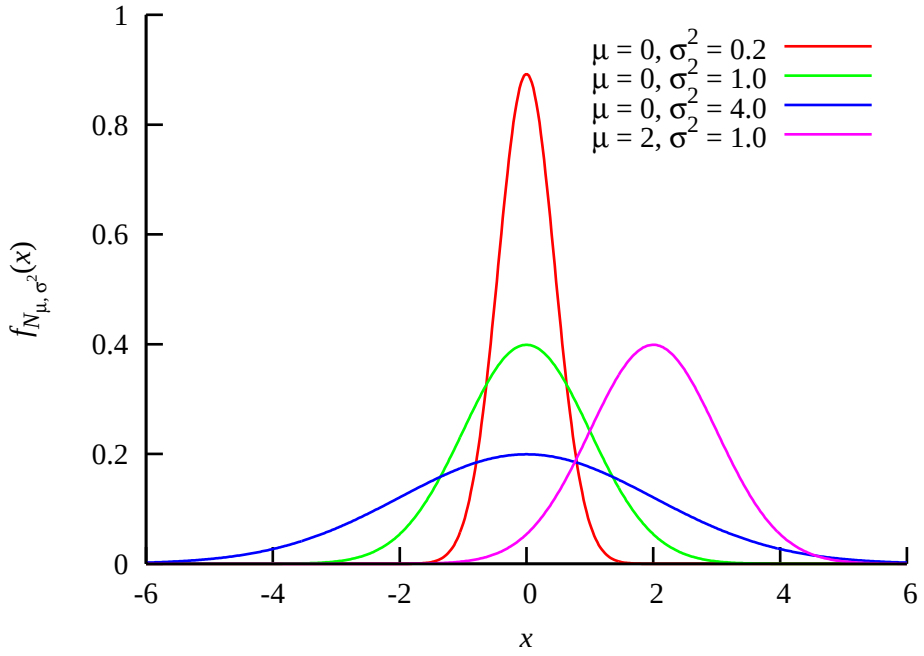


Figure 2.2: Scalar Gaussian density functions for different combinations of μ and σ^2

function of a normally distributed scalar random variable is given as [Bil95]:

$$f_{N_{\mu,\sigma^2}}(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{2.33}$$

A function from this family is selected by specifying the values of its two parameters, the mean μ and variance σ^2. Density functions obtained for several combinations of parameters are plotted in figure 2.2. As is apparent from this plot, the functions always retain the same bell shape. The position of the bell's center is determined by μ while its width depends on the value of σ^2.

The family of Gaussian density functions can also be extended to random variables that take on vectors of real numbers as their values. For an N-dimensional normally distributed random variable, the density function is [Bil95]:

$$f_{N_{\boldsymbol{\mu},\boldsymbol{\Sigma}}}(\mathbf{x}) = \frac{1}{(2\pi)^{\frac{N}{2}}\sqrt{|\boldsymbol{\Sigma}|}} e^{-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})} \tag{2.34}$$

Each function describes an N-dimensional bell shape. The position of the bell's center is given by the N-dimensional vector $\boldsymbol{\mu}$ and its size by the $N \times N$ covariance matrix $\boldsymbol{\Sigma}$; $|\boldsymbol{\Sigma}|$ is the determinant of this matrix. Figure 2.3 shows an example plot of a two-dimensional Gaussian density function. An analysis of the exact influence of each entry in the covariance matrix is not required for this thesis. It is sufficient to note that the eigenvalues determine the scaling of the bell shape along axes whose directions depend

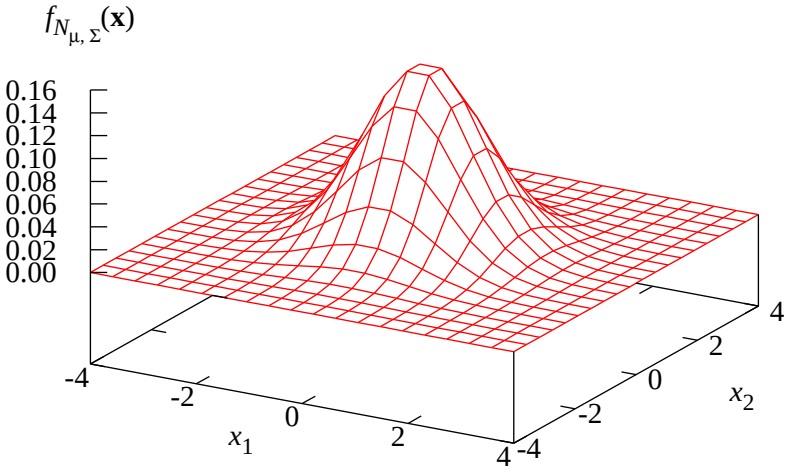


Figure 2.3: Two-dimensional Gaussian density function for $\boldsymbol{\mu} = \mathbf{0}$ and $\boldsymbol{\Sigma} = \mathbf{I}$

on the relative values of the matrix elements. To reduce the number of parameters that need to be specified, the coordinate axes are often used. Scaling factors are then given by the entries on the main diagonal while all other elements are set to zero.

Because Gaussian density functions belong to continuous random variables, they fulfill different criteria than those set forth in equations (2.7) to (2.9). Instead of adding up to one, their values integrate to one. It is apparent from figure 2.2 that, as the variance gets smaller and the bell thinner, it also needs to grow taller if the area underneath it is to remain constant. The condition of equation (2.7) is therefore relaxed from $0 \leq f \leq 1$ to $0 \leq f$. It follows that the values of a continuous density function are not probabilities as they are neither bounded above by 100% nor required to add up to this value.

2.5.2. Parametric Estimation

There are two general approaches to density estimation – parametric and non-parametric. For parametric estimation, a family of density functions is chosen first. The parameters which characterize individual members of this family are then adjusted to find the function that most closely resembles the probability distribution in the sample. Various families of density functions and different techniques for finding the optimal set of parameters can be used.

An example given in [Vio95] highlights popular choices for the two components. Following the central limit theorem [Bil95], a Gaussian density function can be a good approximation of the true density in many cases. The optimization criterion is maximum likelihood. The set of parameter values is sought for which the given sample is most likely to occur. In order to find this parametrization, a likeliness function is constructed and its derivatives calculated. A gradient descent search, as described in chapter 3, is then performed to arrive at the optimum.

Even without stating the precise algorithm used, several of its weaknesses are obvious. One problem is that to determine an approximate density function, an entire optimiza-

tion process is required. This is a costly procedure that, in the case of mutual information used as a similarity metric, would need to be repeated for each iteration of the registration algorithm. Optimization gets more complicated and computationally expensive with the number of parameters to be determined. When estimating a two-dimensional density function, such as $f_{X,Y}$, the Gaussian family already offers six parameters. This is why, as described in the previous section, most of the elements of the covariance matrix are set to zero and never optimized. Also, for many families of functions it cannot be guaranteed that the process will locate the optimal set of parameters. As with mutual information itself, optimization may be constricted by local extrema.

The most important shortcoming of this approach is that a family of density functions needs to be chosen first. An image registration technique should not be limited to a small selection of images that have very similar intensity distributions. All images from an application domain such as physiological tomographic scans should be handled. This makes it impossible to determine beforehand what kind of function will be needed.

An attempt to overcome this problem is found in Gaussian mixture models [MB88]. Instead of assuming a particular family of density functions, the true distribution of the probabilities in the sample is approximated by overlaying several Gaussian bell shapes. Each Gaussian offers two parameters[1], which allow it to be positioned and its width and height to be set. This leads to a more versatile density estimation that, depending on the number of Gaussian bells used, can approximate arbitrarily complex probability distributions with high precision. The flipside is that the number of parameters to be optimized is greatly increased, making it more difficult and time-consuming to find suitable values. The conclusion has to be made that for mutual information based image registration, parametric density estimation is either too inflexible or too computationally expensive.

2.5.3. Histogram

The second possible approach to density estimation is non-parametric. No assumption about the shape of the density function is made. Instead, it is constructed directly from the intensities in the sample. The simplest such technique is the naïve histogram. Each intensity is assigned a probability that is directly proportional to the number of times it appears in the given sample. This leads to a density function that perfectly resembles the sample's probability distribution. As outlined in section 2.5, for the density function to also fit the entire image, the sample needs to be representative. The only way to guarantee this for every image is to use all of its voxel intensities.

For a small image, a histogram can be calculated very quickly. The amount of processing time required per point is minimal as all that needs to be done is to increase the count of points sharing its intensity. However, the computational cost increases linearly with the size of the dataset and makes histograms unsuitable to 3D registration with its large volumetric images. A second weakness is that the histogram produces an empirical representation of the density function, which cannot be differentiated. In consequence,

[1] More parameters are of course present in the case of higher dimensionality.

it is impossible to calculate any derivatives of the mutual information. As will be seen in chapter 3, all but the most basic search strategies rely on derivatives to guide the registration process.

2.5.4. Parzen Window Technique

A technique that improves on the histogram both in terms of computation time and the ability to calculate derivatives was proposed by Parzen in [Par62]. It is referred to as *kernel* or *Parzen window* density estimation. Given a sample S_A that consists of N_A values obtained by evaluating the random variable X, an estimate of the variable's density function is:

$$f_X(\mathbf{x}) \approx \frac{1}{N_A} \sum_{\mathbf{x}_i \in S_A} K(\mathbf{x} - \mathbf{x}_i) \tag{2.35}$$

This estimate is constructed by centering a window function K around each point $\mathbf{x}_i$. Where in a histogram every point in the sample only increases the probability of its own value, K spreads this influence to neighboring values. The window function can be chosen freely and [Par62] suggests a number of possibilities. The only limitation is that if the result is to be a valid density function, so must be K. When each K assumes only values in $[0, 1]$ that add up to one, the sum of all N_A of them weighted by $\frac{1}{N_A}$ has the same properties. Despite this easy way to ensure that a valid density function is constructed, it may be desirable to choose a K that does not meet these requirements. [Vio95] suggests the use of a Gaussian density function with mean zero as K even though it is not a valid density function for discrete random variables.

The Gaussian density function has two properties that make it a good choice for K in image registration. Due to its shape, shown in figures 2.2 and 2.3, each sample point $\mathbf{x}_i$ causes an increase in the probability of intensities that is larger the closer they are to its own. This is in line with the intuitive assumption of a certain smoothness throughout the image. An intensity encountered in the sample may mean that similar intensities are to be found in its vicinity but does not provide any information about intensity values far from its own. The second property is that because the width of the bell shape can be easily set by means of the variance or covariance matrix, different estimators can be constructed simply by adjusting that parameter.

Parzen windows allow for an estimation of the density function with a sample size far smaller than that required by a histogram. If the sample size is small, it may not be precisely representative of the entire image. Some of the intensities that appear infrequently are likely to be completely missing. Using a histogram, the probability of such intensities would be falsely set to zero. With a Parzen window, neighboring intensities contribute increases in probability leading to a smooth density function that assigns nonzero probabilities to all intensities, even those absent from the sample. The smoothing also applies to all other probabilities. If there is a cluster of intensities that all appear with similar frequency, the density function will assume a similar shape regardless of which of them is present in the sample slightly more or less often by pure chance.

Figure 2.4 shows three estimates for the density function of a scalar discrete random variable X with $\Delta_X = \{0, 1, \ldots, 15\}$. They have been constructed from a sample of

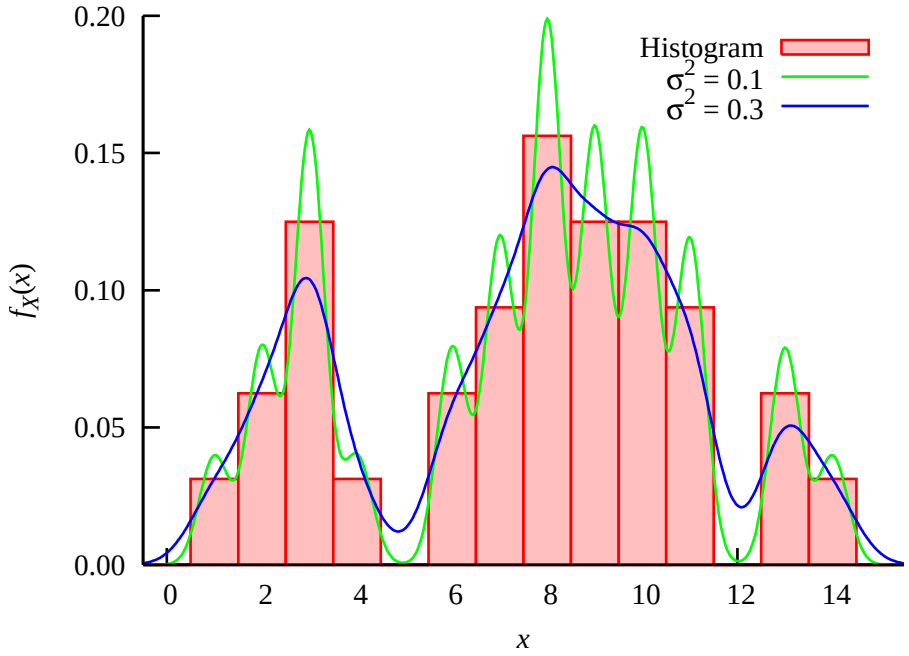


Figure 2.4: Histogram and Parzen window based estimates of a density function

size $N_A = 32$. The first estimate is a histogram that directly represents the probability distribution in the sample. The other two estimates are based on the Parzen window technique with a Gaussian density function as K. With increasing variance of K, the Parzen windows generate a more smooth approximation to the histogram shape. It can also be seen that the approximations are not valid density functions since their values for all $x \in \Delta_X$ do not add up to one. As was explained above, this is a consequence of the Gaussian density function only being valid for continuous random variables.

A Parzen window density estimator employing Gaussian density functions is similar to the Gaussian mixture models mentioned in section 2.5.2. However, the former is non-parametric and can be constructed directly from a sample while the latter require a series of parameters to be laboriously determined.

2.6. Similarity Metric

Using the Parzen window technique to estimate density functions, an estimator for the mutual information of two images may be constructed. Unfortunately, direct calculation of the required entropies following equations (2.18) and (2.21) is computationally very expensive. As seen in section 1.3.2, the images used in this thesis have integer intensities in the range of $[0, 255]$. To estimate the entropy of X, a sum over all of its 256 possible values is calculated. Each addend requires f_X to be approximated for a different intensity, which in turn requires the summation of the effects of N_A parzen windows. The result is a nested sum with $256N_A$ terms to evaluate. The same numbers apply to the

estimation of $H[Y]$. For the joint entropy, $256^2 = 65536$ different intensity combinations are possible leading to a nested sum of $65536 N_A$ terms.

A method for estimating the entropies without having to evaluate these large sums is proposed in [Vio95]. It is based on the concept of the expected value introduced in section 2.3.1. In that section, it was noted that with increasing sample size, the average of a sample converges to the expected value of the variable it was drawn from. According to equation (2.19), the entropy of X is the negated expected value of the discrete random variable $\log f_X(X)$. It can therefore be approximated by the average of a sample $S_{B,X}$ of size N_B obtained from that variable:

$$H[X] \approx -\frac{1}{N_B} \sum_{x_i \in S_{B,X}} \log f_X(x_i) \tag{2.36}$$

If the estimated probability $f_X(x_i)$ of any x_i is very low, it will get aliased to zero due to the limited precision of a computer. To cope with this case, $\log 0 := 0$ is defined, as proposed by [Vio95].

Insertion of the Parzen window based estimate for f_X constructed from a sample $S_{A,X}$ of size N_A using Gaussian density functions with mean zero and variance σ_X^2 as window functions yields:

$$H[X] \approx H^\star[X] = -\frac{1}{N_B} \sum_{x_i \in S_{B,X}} \log \frac{1}{N_A} \sum_{x_j \in S_{A,X}} f_{N_{0,\sigma_X^2}}(x_i - x_j) \tag{2.37}$$

An approximation formula for $H[Y]$ may be analogously obtained:

$$H[Y] \approx H^\star[Y] = -\frac{1}{N_B} \sum_{y_i \in S_{B,Y}} \log \frac{1}{N_A} \sum_{y_j \in S_{A,Y}} f_{N_{0,\sigma_Y^2}}(y_i - y_j) \tag{2.38}$$

Since the joint entropy $H[X,Y]$ is defined as the entropy of a discrete random variable $Z = (X,Y)^T$, the same derivation can be performed for it as well. The samples S_A and S_B contain pairs of values which have been obtained from X and Y. The Gaussian density functions are two-dimensional with mean zero and covariance matrix $\mathbf{\Sigma}$:

$$H[X,Y] \approx H^\star[X,Y] = -\frac{1}{N_B} \sum_{\mathbf{z}_i \in S_B} \log \frac{1}{N_A} \sum_{\mathbf{z}_j \in S_A} f_{N_{\mathbf{0},\mathbf{\Sigma}}}(\mathbf{z}_i - \mathbf{z}_j) \tag{2.39}$$

By inserting the approximations of the three entropies into equation (2.29), an estimation formula for the mutual information is finally constructed:

$$MI(X,Y) \approx MI^\star(X,Y) = H^\star[X] + H^\star[Y] - H^\star[X,Y] \tag{2.40}$$

2.6.1. Alignment Quality

To turn equation (2.40) into a measure that can readily be evaluated given two images, the random variables' values need to be explicitly expressed as image intensities. For

this, points $\mathbf{x}_i$ in the reference coordinate frame are stored in the samples S_A and S_B. The intensity of the reference image at a point in the sample is then $u(\mathbf{x}_i)$ and that of the registered image is $v(T(\mathbf{x}_i))$. The mutual information for a transformation T is now estimated as:

$$\begin{aligned} MI^{\star}(X,Y) = & \\ & -\frac{1}{N_B}\sum_{\mathbf{x}_i\in S_B}\log\frac{1}{N_A}\sum_{\mathbf{x}_j\in S_A} f_{N_{0,\sigma_X^2}}\left(u(\mathbf{x}_i)-u(\mathbf{x}_j)\right) \\ & -\frac{1}{N_B}\sum_{\mathbf{x}_i\in S_B}\log\frac{1}{N_A}\sum_{\mathbf{x}_j\in S_A} f_{N_{0,\sigma_Y^2}}\left(v(T(\mathbf{x}_i))-v(T(\mathbf{x}_j))\right) \\ & +\frac{1}{N_B}\sum_{\mathbf{x}_i\in S_B}\log\frac{1}{N_A}\sum_{\mathbf{x}_j\in S_A} f_{N_{\mathbf{0},\mathbf{\Sigma}}}\left(\begin{pmatrix} u(\mathbf{x}_i) \\ v(T(\mathbf{x}_i)) \end{pmatrix}-\begin{pmatrix} u(\mathbf{x}_j) \\ v(T(\mathbf{x}_j)) \end{pmatrix}\right) \end{aligned} \tag{2.41}$$

While this formula looks rather complex, it can be very efficiently calculated. Each nested sum contains $N_A N_B$ terms so that the computation time is primarily decided by the sizes of the two samples. As will be shown in chapter 7, realistic values range from several hundred to a few thousand entries per sample. In comparison to other image based similarity metrics which iterate over all voxels in the images, this amounts to a very considerable savings. It is also vastly more efficient than the evaluation of the $65536N_A$ terms needed for a direct calculation of $H[X,Y]$. Additionally, the formula lends itself to a series of simplifications. Because only a small number of intensity differences are possible, the values of the Gaussian density functions can be tabulated. If the sampling positions remain constant throughout the entire registration process, $H^{\star}[X]$, corresponding to the first row of the formula, never changes. It can be calculated once at the beginning of the registration and then reused.

Of course, the similarity metric also has weaknesses. It only approximates the mutual information of the two images so that if the sample sizes are chosen too small, the optimal alignment may not be found. Another problem is that the variances and covariances of the Gaussian density functions used in by the Parzen window density estimation must be decided. How this should be done will be addressed in section 2.6.3. At this point, it is only noted that in line with the considerations made in section 2.5.1 when the Gaussian density function was first introduced, the covariance matrix is assumed to be diagonal:

$$\mathbf{\Sigma} = \begin{pmatrix} \sigma_{X,X}^2 & 0 \\ 0 & \sigma_{Y,Y}^2 \end{pmatrix} \tag{2.42}$$

2.6.2. Derivative of Alignment Quality

Besides an assessment of the quality of the current alignment, the similarity metric must provide the data necessary to improve this alignment. As will be seen in chapter 3, the information required by the search strategy is in most cases the derivative of the similarity metric with respect to the transformation parameters T. A few notations are

introduced to make the derivative of $MI^\star[X,Y]$ more legible:

$$v_i = v\left(T\left(\mathbf{x}_i\right)\right) \qquad \mathbf{w}_i = \begin{pmatrix} u\left(\mathbf{x}_i\right) \\ v\left(T\left(\mathbf{x}_i\right)\right) \end{pmatrix} \tag{2.43}$$

$$W_Y\left(v_i, v_j\right) = \frac{f_{N_{0,\sigma_Y^2}}\left(v_i - v_j\right)}{\sum\limits_{\mathbf{x}_k \in S_A} f_{N_{0,\sigma_Y^2}}\left(v_i - v_k\right)} \qquad W_{X,Y}\left(\mathbf{w}_i, \mathbf{w}_j\right) = \frac{f_{N_{\mathbf{0},\mathbf{\Sigma}}}\left(\mathbf{w}_i - \mathbf{w}_j\right)}{\sum\limits_{\mathbf{x}_k \in S_A} f_{N_{\mathbf{0},\mathbf{\Sigma}}}\left(\mathbf{w}_i - \mathbf{w}_k\right)} \tag{2.44}$$

With these notations, the derivative of equation (2.41) is:

$$\begin{aligned} \frac{d}{dT} MI^\star\left(X,Y\right) = & \\ & \frac{1}{N_B} \sum_{\mathbf{x}_i \in S_B} \sum_{\mathbf{x}_j \in S_A} W_Y\left(v_i, v_j\right)\left(v_i - v_j\right) \frac{1}{\sigma_Y^2} \left(\frac{d}{dT} v_i - \frac{d}{dT} v_j \right) \\ & - \frac{1}{N_B} \sum_{\mathbf{x}_i \in S_B} \sum_{\mathbf{x}_j \in S_A} W_{X,Y}\left(\mathbf{w}_i, \mathbf{w}_j\right)\left(\mathbf{w}_i - \mathbf{w}_j\right)^T \mathbf{\Sigma}^{-1} \left(\frac{d}{dT} \mathbf{w}_i - \frac{d}{dT} \mathbf{w}_j \right) \end{aligned} \tag{2.45}$$

Noting that the covariance matrix is diagonal as per equation (2.42) and $\frac{d}{dT} u\left(\mathbf{x}_i\right) = 0$ for all $\mathbf{x}_i$, this further simplifies to:

$$\begin{aligned} \frac{d}{dT} MI^\star\left(X,Y\right) = \frac{1}{N_B} \sum_{\mathbf{x}_i \in S_B} \sum_{\mathbf{x}_j \in S_A} & \left[W_Y\left(v_i, v_j\right) \frac{1}{\sigma_Y^2} - W_{X,Y}\left(\mathbf{w}_i, \mathbf{w}_j\right) \frac{1}{\sigma_{Y,Y}^2} \right] \\ & \left(v_i - v_j\right) \left(\frac{d}{dT} v_i - \frac{d}{dT} v_j \right) \end{aligned} \tag{2.46}$$

The computational complexity of a derivative calculation is similar to that of calculating the mutual information itself. A nested sum of $N_A N_B$ terms needs to be evaluated again. Although W_Y and $W_{X,Y}$ internally contain sums with N_A addends each, these only need to be evaluated once for every point in S_B, leaving the complexity at $N_A N_B$.

To evaluate the terms $\frac{d}{dT} v_i$ and $\frac{d}{dT} v_j$, knowledge of the exact transformation function is needed. This part of the calculation will therefore be addressed in chapter 4. Depending on the transformation type and its number of parameters, the terms may be scalars, vectors or even matrices. Their dimensionality determines the dimensionality of the entire derivative because equation (2.46) is a weighted sum of these terms.

2.6.3. Parzen (Co-)Variances

A method for automatically calculating the optimal values of the Parzen window function variances and covariance matrix is devised by Viola in [Vio95]. Unfortunately, it can easily be demonstrated that this method will not work in practical applications. Following the classical maximum likelihood approach, the value of the (co-)variance parameter is sought which makes the intensity distribution encountered in sample S_B most likely to occur given the estimated density function. Viola shows that this value is always the

one that makes the estimated entropy minimal. Thus, for example, σ_X^2 should be chosen so that $H^\star[X]$ is minimized:

$$\tilde{\sigma}_X^2 = \arg \min_{\sigma_X^2 \in \mathbb{R}^+} -\frac{1}{N_B} \sum_{\mathbf{x}_i \in S_B} \log \frac{1}{N_A} \sum_{\mathbf{x}_j \in S_A} f_{N_{0,\sigma_X^2}} (u(\mathbf{x}_i) - u(\mathbf{x}_j)) \tag{2.47}$$

It is then proposed to begin with an empirical guess for the variances and covariance matrix and optimize them in parallel to the registration process. This could be accomplished for example by using stochastic gradient descent or the EM algorithm of [Bil98]. While the idea is correct in theory, it fails in practice because of the limited precision of a computer. The behavior of the Gaussian density function for decreasing σ_X^2 is:

$$\lim_{\sigma_X^2 \to 0} f_{N_{0,\sigma_X^2}} (u(\mathbf{x}_i) - u(\mathbf{x}_j)) = \begin{cases} 0 & \text{if } u(\mathbf{x}_i) \neq u(\mathbf{x}_j) \\ \infty & \text{if } u(\mathbf{x}_i) = u(\mathbf{x}_j) \end{cases} \tag{2.48}$$

For every $\mathbf{x}_i \in S_B$, if there is a point $\mathbf{x}_j \in S_A$ that has the same intensity, the inner sum in equation (2.47) will tend toward infinity and so will its logarithm. If there is no point $\mathbf{x}_j \in S_A$ with the same intensity as $\mathbf{x}_i$, the inner sum will tend toward zero and its logarithm to negative infinity. Unless every intensity of sample S_B also occurs for sample S_A, there will be at least one inner sum that approaches negative infinity. This effect will be dominant because the decay is much faster than the growth. As a result, the entire expression of equation (2.47) will approach infinity leading the optimization process away from values of σ_X^2 very close to zero.

The limited precision of a computer changes this behavior. Because the Gaussian density function falls exponentially as $\sigma_X^2 \to 0$, its values quickly becomes so small that they are aliased to zero. The result is that each inner sum either approaches positive infinity or is precisely zero. The logarithm in the first case also approaches infinity while in the second case, due of the definition $\log 0 = 0$ made earlier, is zero. Equation (2.47) thus tends toward negative infinity. This leads the optimization process to decrease σ_X^2 giving it a value as close to zero as possible. The same occurs for σ_Y^2 and the elements of the covariance matrix.

This unfortunate behavior is a consequence of the limitations of a computer and cannot be easily prevented. Although it only occurs if at least one intensity appears in both samples, this is a very likely event due to there being only 256 different intensities. A helpful observation made in [Vio95] is that the Parzen window density estimate is relatively insensitive to the precise value of the variance or covariance matrix as long as it is roughly within the correct order of magnitude.

The solution that has been adopted for this thesis therefore is to empirically determine variance and covariance values that lead to correct registration for a variety of images from the application domain. Although the parameters will not be optimal for each set of images, they have been found to be sufficient for the registration of every available pair of scans.

2.7. Normalized Mutual Information

An alternative similarity measure based on mutual information is proposed in [SHH99]. It was found that mutual information may increase when the misalignment is such that there is very little overlap between the two images. To overcome this problem, the measure of *normalized mutual information* was developed, which is insensitive to the amount of overlap:

$$NMI\left(X,Y\right) = \frac{H\left[X\right] + H\left[Y\right]}{H\left[X,Y\right]} \tag{2.49}$$

Normalized mutual information can be estimated as efficiently as plain mutual information using the Parzen window technique. However, the situation is different for the derivative with respect to transformation parameters T. In the case of mutual information, only the derivatives of $H\left[Y\right]$ and $H\left[X,Y\right]$ need to be estimated. For normalized mutual information, it is also necessary to approximate the actual entropies of X and Y and their joint entropy:

$$\frac{d}{dT}NMI\left(X,Y\right) = \frac{\left(\frac{d}{dT}H\left[Y\right]\right)H\left[X,Y\right] - \left(H\left[X\right] + H\left[Y\right]\right)\frac{d}{dT}H\left[X,Y\right]}{\left(H\left[X,Y\right]\right)^2} \tag{2.50}$$

It has been found during the experiments conducted for this thesis that if the registration is started with the two images largely overlapping, there is little risk of the registration process moving them toward very low overlap. Mutual information was therefore chosen over normalized mutual information for its lower computational cost.

3. Search Strategy

The search strategy provides the link that combines a similarity metric and a transformation type into a complete registration technique. Using the information made available by the similarity metric, it adjusts the transformation parameters to improve the alignment and drive the registration process toward completion. The choice of search strategy therefore depends both on the similarity metric and the transformation type being used.

One criterion to be considered is the quality of the data produced by the similarity metric. If it calculates only rough or noisy estimates of the alignment quality, a robust search strategy is required that can arrive at a proper alignment despite these imprecisions. If, however, the similarity metric is highly accurate, there is no need for such robustness. A strategy is then more suitable that relies on the quality of the assessment provided by the similarity metric and trades the robustness for a faster registration process.

A second criterion is the type of data that the similarity metric is able to calculate. Some metrics only produce an assessment of the current alignment quality. Others can also provide derivatives of this quantity or indicate which points exactly in the two images should coincide. Only a search strategy that exploits all the information available makes optimal use of the similarity metric.

Finally, some search strategies are only suitable for particular types of transformations. This may be because due to their simplicity, they cannot handle complex transformations with many parameters. Another possibility are strategies which are tuned to one type of transformation so that they can make assumptions about the transformation function which are then used to provide faster and more robust registration.

3.1. Survey of Strategies

Overviews of many popular search strategies are given in [Bro92] and [ZF03]. Most of these strategies are used in registration techniques based on algorithm 1.1, making adjustments to the transformation parameters in each iteration and gradually improving alignment quality. Some strategies also exist that, given a suitable similarity metric, are able to directly calculate the correct alignment, eliminating the need for an iterative process.

3.1.1. Specialized Techniques

A very large number of specialized search strategies have been developed that provide fast and accurate registration for a limited class of transformation types. They include

decision sequencing, relaxation, dynamic programming, the generalized Hough transform and linear programming. Each makes a series of assumptions about the transformation function, generally precluding the complex transformations necessary to correct complicated misalignments. These techniques can therefore not be used when aligning tomographic scans with possible small changes in the shapes of the patient's inner organs.

Another group of search strategies are those based on point matching. They are used in combination with feature-based similarity metrics where information about the correspondence of the features in the two images is available. From the displacement between corresponding features, a new transformation can be calculated that aligns the features and undoes the misregistration. Such techniques may be able to determine a suitable transformation in a single step but cannot be combined with mutual information as this is an image based similarity metric.

3.1.2. Naïve Search

The simplest general search strategy that can be applied to any transformation type is a naïve search. The parameter space is simply searched exhaustively, trying each combination and locating the one that leads to the best alignment. While this approach is guaranteed to find the optimal alignment if the similarity metric provides correct information, it is also prohibitively expensive for all but the most simple transformation types. As the number of parameters and the range of values they can take on are increased, the cost of a naïve search quickly grows. If any parameter can take on a range of real numbers as values, the number of parameter combinations to be tried becomes infinite and naïve search cannot even be completed in finite time.

Despite its obvious shortcomings, naïve search has the one advantage that the only information required from the similarity metric is an assessment of the alignment quality achieved by the current parameter values. More advanced techniques need additional data to be available.

3.1.3. Gradient Ascent

Gradient ascent is a simple iterative search strategy that uses the derivative of the alignment quality with respect to the transformation parameters T. For each parameter, the derivative expresses how strongly and in which direction the alignment quality is expected to change if the parameter's value is increased. Because this assessment is valid only for the current values of the parameters T, only small changes should be based on it. After these changes, the derivative needs to be reevaluated, which is done in the next iteration.

If the metric is to be maximized, a positive derivative means that the value of a parameter should be increased while a negative derivative indicates that it should be decreased. The magnitude of the derivative is an indication of how large the change should be. If the derivative is close to zero, only a small step should be taken as the derivative may become zero or change sign after this change. When the magnitude of the

derivative is large, a larger step is likely to be permissible before the derivative changes its sign. Each parameter is therefore adjusted by adding a value that is proportional to the derivative of the similarity metric S with respect to it. This can be expressed for the entire set of parameters T as:

$$T \leftarrow T + \lambda \frac{d}{dT} S \tag{3.1}$$

λ is the step size that determines by how much the parameters should change. If it is chosen too large, the search may oscillate about the optimal parameter values, alternatingly increasing and decreasing them, each time jumping over the values at which the derivatives become zero. If the step size is too small, registration takes a long time because the parameters are adjusted only in small increments. This problem may be overcome by beginning with a large step size and reducing it as registration progresses.

Gradient ascent is simple, efficiently calculated and leads to steady improvement of the alignment quality as long as the derivatives calculated by the similarity metric are reliable. It can also readily be adapted to similarity metrics that need to be minimized, not maximized. The step size is simply negated so that when the derivative with respect to a parameter is positive, its value is decreased and vice versa. This is known as gradient descent.

It is also possible to take into account some conditions imposed on the parameters by the transformation type. If the values of a parameter should stay within a predefined interval, it can be checked after each iteration whether the current value is acceptable and if not, to reset it to the closest boundary of the interval. If some parameters should be adjusted more quickly than others, for example favoring a translation over a rotation, different step sizes may be used for each.

A limitation of gradient ascent is that it finds only local extrema. Beginning with the initial values, the parameters are adjusted to continually increase the value of S, tending toward the closest local maximum. If there is a set of parameters that leads to an even better alignment but the value of S would temporarily decrease on the way there, gradient ascent is unable to find this solution. This search strategy should therefore only be used in conjunction with unimodal similarity metrics, where the single local maximum is also globally optimal.

3.1.4. Conjugate Gradients

Like gradient ascent, the conjugate gradient method [She94] is based on the derivatives of the alignment quality with respect to the transformation parameters. However, because the direction in the parameter plane in which an adjustment step is taken is chosen more carefully, a good alignment may be found in fewer iterations. Unfortunately, this method is not able to remove the limitation to unimodal similarity metrics as it also locates only local extrema.

3.2. Stochastic Gradient Ascent

Stochastic gradient ascent is closely related to gradient ascent as described in section 3.1.3. In fact, its mathematical formulation is identical. The difference is that the derivatives of the alignment quality are estimated and not precisely calculated. Their values can therefore deviate from the true derivatives due to the random noise introduced by the estimation process. It is suggested in [Vio95] that gradient ascent using such noisy derivatives not only leads to successful registration but may even produce better results than traditional gradient ascent.

Because the derivatives are noisy, each iterative step does not necessarily bring the parameter values closer to the nearest extremum. The noise superimposes a movement in a random direction in the parameter plane. Registration can only be successful if the noise is small compared to the magnitude of the true derivatives. For the derivative of mutual information calculated using equation (2.46), the amount of noise depends on the sizes N_A and N_B of the two samples and the Parzen window variances. As will be seen in chapter 7, by increasing the sample sizes and variances, the noise can be made arbitrarily small.

A second prerequisite for a finite registration process is that λ be decreased as registration progresses. Otherwise, even if optimal parameter values were found, a random step away would be taken again and registration would never finish.

While noisy derivatives are problematic, they can also serve as an advantage. This occurs in the situation that the similarity metric in itself is also noisy, as is the case for $MI^\star$. Because random noise results in local extrema, traditional gradient ascent is likely to find a local noise maximum and fail to properly align the two images. As will be shown in chapter 7, stochastic gradient ascent may be able to skip over noise maxima and determine the optimal alignment if the magnitude of the noise in the metric is sufficiently small.

With $\frac{d}{dT}MI^\star$ the noisy estimate for the derivative of mutual information obtained from equation (2.46), the iterative step of stochastic gradient ascent as used in this thesis can be expressed as:

$$T \leftarrow T + \lambda \frac{d}{dT} MI^\star \tag{3.2}$$

3.3. Multiresolution

Multiresolution is a technique that can be used to improve the computational efficiency of iterative search strategies [ZF03]. It is based on the observation that when the reference and registered image are scaled down, the similarity metric can be evaluated using fewer operations. If the metric uses the intensities of all voxels, there are fewer voxels to consider. For a metric based on random samples, the samples can be made smaller because the images contain less details and less information. However, when scaled-down versions of the images are aligned, the details eliminated by the scaling process cannot be considered. The alignment may thus not be as precise. Small deformations such as changes in organ shape can also not be corrected if they have been removed from

the images.

In a multiresolution approach, the images are therefore scaled to several different sizes and their alignment determined consecutively, starting with the smallest versions. Once a transformation T has been found that satisfactorily aligns the two images, images of the next higher resolution are registered with T as the initial transformation. The effect is that all aspects of the misalignment are corrected using the smallest possible images, minimizing the computational cost. Only adjustments which require more details are left to the next higher resolution.

A series of images generated by repeatedly decreasing the size of the previous image are known as an image pyramid. The simplest method for generating a smaller image I' from a larger one I is subsampling. For example, when halving the size of the image in each dimension, only the intensity of every second voxel per coordinate is used:

$$I'(x, y, z) = I(2x, 2y, 2z) \tag{3.3}$$

The problem of this approach is that it is prone to aliasing. Because only one voxel in eight is retained, small details are misrepresented. Before subsampling, the image I should therefore be low-pass filtered to remove such details, which correspond to high frequency information. This is most popularly done using Gaussian smoothing, resulting in a Gaussian pyramid [Bur81].

Smoothing is performed by replacing each voxel intensity $I(x, y, z)$ with a weighted sum of neighboring intensities. The weights are determined by a three-dimensional Gaussian density function centered at (x, y, z). As the covariances are set to zero, the density function becomes separable. An equivalent smoothing can then be obtained more efficiently by consecutively weighting the intensities along their three coordinates using a one-dimensional Gaussian density function as defined by equation (2.33) and plotted in figure 2.2.

Because intensities are stored only at discrete voxel locations, the continuous density function must be discretized in such a way that the weights at these locations add up to one, preserving the average intensity level of the original image. This is done by using the density function of the binomial distribution instead, which is discrete and approximates the Gaussian:

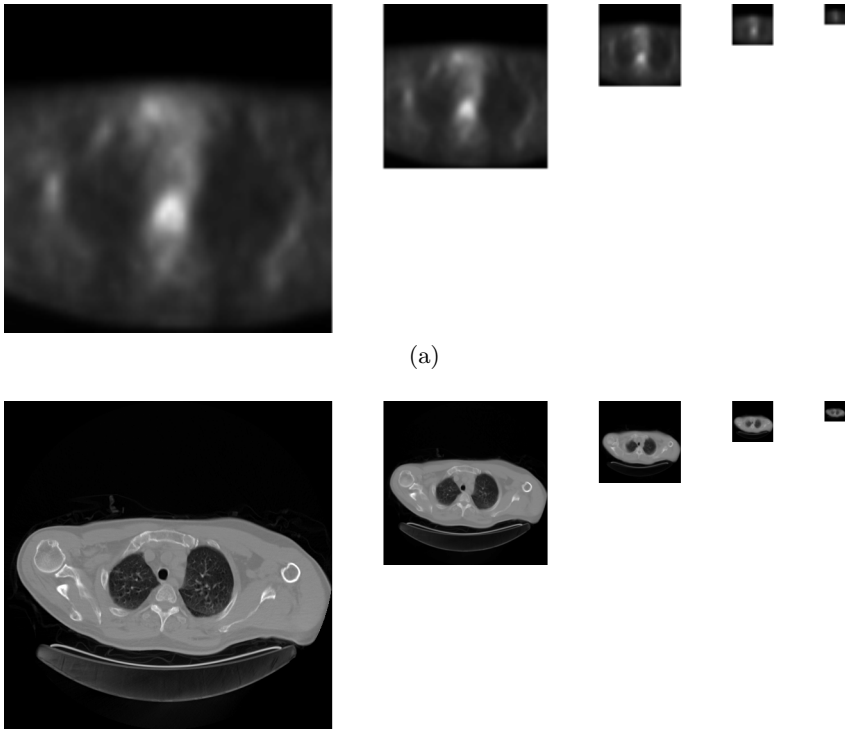
$$f_{B_{n,p}}(x) = \begin{cases} \binom{n}{k} p^x (1-p)^{n-x} & 0 \leq x \leq n \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

p controls the symmetry of the approximated bell shape and is set to $p = 0.5$. The value of n then determines the variance, which is $\sigma = np(1-p)$. A larger n means that a wider Gaussian bell is approximated using more neighboring voxel intensities. This leads to a higher quality smoothing but increases the computational cost. The value used in this thesis is $n = 4$ so that the intensities of five voxels are always weighted. The weights obtained from $f_{B_{4,0.5}}$ are listed in table 3.1.

After the smoothing has been applied, the size of the image is halved in each dimension by subsampling following equation (3.3). Pyramids with a total of five levels are generated for the reference and registered image. Level 0 is the original dataset and each

$\frac{1}{16} \times 1$	$\frac{1}{16} \times 4$	$\frac{1}{16} \times 6$	$\frac{1}{16} \times 4$	$\frac{1}{16} \times 1$

Table 3.1: Weights for neighboring voxels in Gaussian smoothing



(a)

(b)

Figure 3.1: Two-dimensional cuts through Gaussian pyramids generated for different datasets: (a) PET scan; (b) CT scan

consecutive level is the next smaller image. With PET slices having resolutions of only 128×128 and the number of slices usually even less than 128, the images become so small that further pyramid levels are not sensible. Example cuts through the volumetric images generated at different pyramid levels are presented in figure 3.1.

4. Transformation Type

The transformation type determines the set of transformation functions from which the search strategy may choose. It is defined by a family of functions with a parametrization T and a search space that expresses any potential constraints on the values of the parameters in T.

The choice of a transformation type is often a compromise between two conflicting goals. A misalignment can only be corrected if the transformation type is able to express a mapping that undoes its effects. If complex distortions are present, a type with many parameters is required to precisely model this mapping. However, each additional parameter increases the computational cost of the registration process since the similarity metric and search strategy must produce and process information for its adjustment in each iteration.

Transformation types fall into two general categories. A *global* transformation is used when correcting a misalignment that affects the entire registered image, such as a translation or rotation. The mapping from reference to registered coordinates is calculated using the same transformation function and parameter values for each point. When the distortion to be undone varies throughout the registered image, a *local* transformation is employed. Its parameters allow a mapping to be specified that accounts for the distortion in one region of the image without affecting other regions. The influence of each parameter is usually strongest in one location and decreases with distance. The parameters relevant to the transformation may therefore be different for every point in the reference image.

In this thesis, both kinds of transformations are used. First, a global transformation is applied that aligns the general body shapes in the two images. Next, a local transformation is applied to correct small changes in the shapes of the patient's internal organs.

4.1. Global

A global transformation is given by a single equation that is valid for all points in the reference image. When constructing this equation, it is often more intuitive to think not of the transformation function itself but of its inverse. As defined in section 1.2.2, the transformation T provides a mapping from the reference to the registered image coordinate frame:

$$\mathbf{x}_v = T(\mathbf{x}_u) \tag{4.1}$$

If it exists, the inverse transformation specifies a mapping in the reverse direction:

$$T^{-1}(\mathbf{x}_v) = \mathbf{x}_u \tag{4.2}$$

The advantage of considering the inverse transformation is that it directly describes the modifications performed on the registered image. For example, if T^{-1} doubles the value of $(\mathbf{x}_u)_x$, then this corresponds to a stretching of the registered image in x direction by a factor of two.

Another useful concept for the description of global transformations are homogenous coordinates [FvDFH95]. Each coordinate vector is extended by a component that is to be interpreted as a scaling factor. A three-dimensional point is expressed as:

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \triangleq \begin{pmatrix} x/w \\ y/w \\ z/w \end{pmatrix} \tag{4.3}$$

Using homogenous coordinates, a wide range of transformations that combine, scale and offset the coordinates of the original point can be written as simple matrix multiplications:

$$\mathbf{x}_v = \mathbf{M}_T \mathbf{x}_u \tag{4.4}$$

If it exists, the inverse of such a transformation is naturally described by the inverse of this matrix:

$$\mathbf{M}_T^{-1} \mathbf{x}_v = \mathbf{x}_u \tag{4.5}$$

4.1.1. Rigid

The simplest global transformations are those treating the registered image as a rigid object. The image may then only be translated and rotated as a whole. Optionally, scaling by a uniform factor may be allowed as well. The inverse transformation that translates an image by a vector $\mathbf{t}$ is given by the following matrix:

$$\mathbf{T}(\mathbf{t}) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.6}$$

Inverse transformations that rotate the registered image about the x, y and z axes can also be expressed as matrices. They rotate by angles α, β and γ, respectively, in anticlockwise direction when looking toward the origin:

$$\begin{gathered} \mathbf{R}_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad \mathbf{R}_y(\beta) = \begin{pmatrix} \cos\beta & 0 & \sin\beta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\beta & 0 & \cos\beta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ \mathbf{R}_z(\gamma) = \begin{pmatrix} \cos\gamma & -\sin\gamma & 0 & 0 \\ \sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{gathered} \tag{4.7}$$

A more complex inverse transformation may be constructed by performing several of these elementary operations in succession. As each operation is applied to the result of the previous, their matrix representations must be arranged from right to left and multiplied to obtain the final matrix.

According to Euler's rotation theorem, the three rotations about the coordinate axes together are able to express every rotation possible in three-dimensional space. The pivot point of such a rotation is the origin. If a different pivot $\mathbf{p}$ is desired, the image must be translated so that the designated pivot coincide with the origin, rotated, and the translation undone:

$$\mathbf{R}_{x,y,z}(\mathbf{p},\alpha,\beta,\gamma) = \mathbf{T}(\mathbf{p})\,\mathbf{R}_x(\gamma)\,\mathbf{R}_y(\beta)\,\mathbf{R}_z(\alpha)\,\mathbf{T}(-\mathbf{p}) \tag{4.8}$$

The type of rigid transformation used in this thesis allows an arbitrary rotation about the center of the image and a translation by a vector $\mathbf{t}$. With $\mathbf{s}$ the size of the registered image, the pivot therefore is $\frac{1}{2}\mathbf{s}$ and the entire inverse transformation matrix is given by:

$$\mathbf{M}_T^{-1} = \mathbf{T}(\mathbf{t})\,\mathbf{T}\left(\tfrac{1}{2}\mathbf{s}\right)\mathbf{R}_x(\alpha)\,\mathbf{R}_y(\beta)\,\mathbf{R}_z(\gamma)\,\mathbf{T}\left(-\tfrac{1}{2}\mathbf{s}\right) \tag{4.9}$$

The matrix representation of the corresponding transformation T is easiest obtained by inverting the matrices of the elementary operations and multiplying them in reverse order:

$$\mathbf{M}_T = \mathbf{T}^{-1}\left(-\tfrac{1}{2}\mathbf{s}\right)\mathbf{R}_z^{-1}(\gamma)\,\mathbf{R}_y^{-1}(\beta)\,\mathbf{R}_x^{-1}(\alpha)\,\mathbf{T}^{-1}\left(\tfrac{1}{2}\mathbf{s}\right)\mathbf{T}^{-1}(\mathbf{t}) \tag{4.10}$$

For both translation and rotation, the inverse operation is achieved by negating the arguments:

$$\mathbf{M}_T = \mathbf{T}\left(\tfrac{1}{2}\mathbf{s}\right)\mathbf{R}_z(-\gamma)\,\mathbf{R}_y(-\beta)\,\mathbf{R}_x(-\alpha)\,\mathbf{T}\left(-\tfrac{1}{2}\mathbf{s}\right)\mathbf{T}(-\mathbf{t}) \tag{4.11}$$

With $\mathbf{x}_u$ and $\mathbf{x}_v$ expressed in homogenous coordinates, the family of rigid transformations used in this thesis is:

$$\mathbf{x}_v = T(\mathbf{x}_u) = \mathbf{M}_T\mathbf{x}_u = \left[\mathbf{T}\left(\tfrac{1}{2}\mathbf{s}\right)\mathbf{R}_z(-\gamma)\,\mathbf{R}_y(-\beta)\,\mathbf{R}_x(-\alpha)\,\mathbf{T}\left(-\tfrac{1}{2}\mathbf{s}\right)\mathbf{T}(-\mathbf{t})\right]\mathbf{x}_u \tag{4.12}$$

This family offers six parameters. α, β and γ specify the rotation, the three components of $\mathbf{t}$ the translation of the registered image. An explicit expression for $\mathbf{M}_T$ can be obtained by inserting the definitions of the elementary operations as given by equations (4.6) and (4.7) and performing the matrix multiplications:

$$(\mathbf{M}_T)_{11} = \cos\beta\cos\gamma \tag{4.13}$$

$$(\mathbf{M}_T)_{12} = \cos\alpha\sin\gamma + \sin\alpha\sin\beta\cos\gamma \tag{4.14}$$

$$(\mathbf{M}_T)_{13} = \sin\alpha\sin\gamma - \cos\alpha\sin\beta\cos\gamma \tag{4.15}$$

$$\begin{aligned}(\mathbf{M}_T)_{14} = {} & \cos\alpha\left[\left(\tfrac{s_z}{2}+t_z\right)\sin\beta\cos\gamma + \left(-\tfrac{s_y}{2}-t_y\right)\sin\gamma\right] \\ & + \left[\left(-\tfrac{s_y}{2}-t_y\right)\sin\beta\cos\gamma - \left(\tfrac{s_z}{2}+t_z\right)\sin\gamma\right]\sin\alpha \\ & + \left(-t_x-\tfrac{s_x}{2}\right)\cos\beta\cos\gamma + \tfrac{s_x}{2}\end{aligned} \tag{4.16}$$

$$(\mathbf{M}_T)_{21} = -\cos\beta\sin\gamma \tag{4.17}$$

$$(\mathbf{M}_T)_{22} = \cos\alpha\cos\gamma - \sin\alpha\sin\beta\sin\gamma \tag{4.18}$$

$$(\mathbf{M}_T)_{23} = \cos\alpha \sin\beta \sin\gamma + \sin\alpha \cos\gamma \tag{4.19}$$

$$\begin{aligned}(\mathbf{M}_T)_{24} = &\cos\alpha \left[\left(-\tfrac{s_y}{2} - t_y\right)\cos\gamma - \left(\tfrac{s_z}{2} + t_z\right)\sin\beta\sin\gamma\right] \\ &+ \sin\alpha \left[\left(\tfrac{s_y}{2} + t_y\right)\sin\beta\sin\gamma - \left(\tfrac{s_z}{2} + t_z\right)\cos\gamma\right] \\ &+ \left(t_x + \tfrac{s_x}{2}\right)\cos\beta\sin\gamma + \tfrac{s_y}{2}\end{aligned} \tag{4.20}$$

$$(\mathbf{M}_T)_{31} = \sin\beta \tag{4.21}$$

$$(\mathbf{M}_T)_{32} = -\sin\alpha\cos\beta \tag{4.22}$$

$$(\mathbf{M}_T)_{33} = \cos\alpha\cos\beta \tag{4.23}$$

$$\begin{aligned}(\mathbf{M}_T)_{34} = &\left(-\tfrac{s_z}{2} - t_z\right)\cos\alpha\cos\beta + \left(\tfrac{s_y}{2} + t_y\right)\sin\alpha\cos\beta \\ &+ \left(-t_x - \tfrac{s_x}{2}\right)\sin\beta + \tfrac{s_z}{2}\end{aligned} \tag{4.24}$$

$$(\mathbf{M}_T)_{41} = 0 \tag{4.25}$$

$$(\mathbf{M}_T)_{42} = 0 \tag{4.26}$$

$$(\mathbf{M}_T)_{43} = 0 \tag{4.27}$$

$$(\mathbf{M}_T)_{44} = 1 \tag{4.28}$$

An important observation can be made about the last row of the matrix. Because it has a constant value of $(0, 0, 0, 1)$, none of the transformations specified by equation (4.12) alter the last component of $\mathbf{x}_u$. If $(\mathbf{x}_u)_w$ is set to a constant value for every $\mathbf{x}_u$, all $\mathbf{x}_v$ will also have this value in $(\mathbf{x}_v)_w$. The last component of all homogenous coordinate representations is therefore set to $x_w = 1$ in this thesis. This is an obvious choice since it simplifies equation (4.3) so that no divisions or multiplications are necessary when converting between plain Cartesian and homogenous coordinates:

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \triangleq \begin{pmatrix} x \\ y \\ z \end{pmatrix} \tag{4.29}$$

4.1.2. Intensity Derivative

As seen in section 2.6.2, to calculate the derivative of mutual information, terms of the form $\frac{d}{dT} v\left(T\left(\mathbf{x}_u\right)\right)$ must be evaluated. This is the derivative of the registered image intensity at the point $\mathbf{x}_v = T\left(\mathbf{x}_u\right)$ with respect to the transformation parameters T. It can be decomposed using the chain rule:

$$\frac{d}{dT} v\left(T\left(\mathbf{x}_u\right)\right) = \operatorname{grad} v\left(T\left(\mathbf{x}_u\right)\right) \frac{d}{dT} T\left(\mathbf{x}_u\right) \tag{4.30}$$

The first factor is the gradient of the registered image intensity at the point $\mathbf{x}_v$. A fast way of estimating it from the image data is given by forward differences [AS65]. It will be seen that because the fourth component of all homogenous coordinate representations is constant, the derivative of v with respect to it is never needed. The estimation of the last component of the gradient is therefore omitted to accelerate calculations:

$$\operatorname{grad} v\left(\mathbf{x}_v\right) \approx \left(v\left(\mathbf{x}_v + \mathbf{e}_1\right) - v\left(\mathbf{x}_v\right), v\left(\mathbf{x}_v + \mathbf{e}_2\right) - v\left(\mathbf{x}_v\right), v\left(\mathbf{x}_v + \mathbf{e}_3\right) - v\left(\mathbf{x}_v\right), 0\right) \tag{4.31}$$

The second factor in equation (4.30) is the derivative of the transformation function with respect to each of its parameters. Depending on the transformation type, the calculation of this derivative may be very complicated.

4.1.2.1. Indirect Calculation

For transformations that can be expressed as $\mathbf{M}_T\mathbf{x}_u$, the use of an approximation is suggested in [IVA+96]. The derivative is calculated not with respect to the transformation parameters but with respect to the elements of the matrix. The total derivative of $v\left(T\left(\mathbf{x}_u\right)\right)$ can then be written as a matrix $\mathbf{\Delta M}_T$ of the same size as $\mathbf{M}_T$ with each element given by:

$$\left(\mathbf{\Delta M}_T\right)_{ij} = \left(\operatorname{grad} v\left(T\left(\mathbf{x}_u\right)\right)\right)_i \left(\mathbf{x}_u\right)_j \tag{4.32}$$

The entire derivative may be efficiently calculated as:

$$\mathbf{\Delta M}_T = \left(\operatorname{grad} v\left(T\left(\mathbf{x}_u\right)\right)\right)^T \otimes \mathbf{x}_u^T \tag{4.33}$$

If $\mathbf{\Delta M}_T$ is used in place of $\frac{d}{dT}v$, the derivative of the mutual information obtained from equation (2.46) is also expressed with respect to the elements of $\mathbf{M}_T$. Based on this information, the stochastic gradient ascent cannot directly adjust the parameters of the transformation. Instead, the transformation matrix $\mathbf{M}_T$ is modified:

$$\tilde{\mathbf{M}}_T \leftarrow \mathbf{M}_T + \lambda \mathbf{\Delta M}_T \tag{4.34}$$

Because the last row of $\mathbf{M}_T$ is to remain constant, the corresponding elements of $\mathbf{\Delta M}_T$ must be zero. The easiest way to ensure this is by setting the last component of the gradient used in equation (4.33) to zero, as is done in equation (4.31).

Directly changing the elements of the matrix may lead to a transformation that is not valid for the chosen transformation type. For example, when only rotation and translation are permitted, the matrix may additionally express a shearing operation. An additional step is therefore required before registration may continue. Using a set of constraints, a valid transformation must be determined that matches the one expressed by $\tilde{\mathbf{M}}_T$ as closely as possible. There is ambiguity in this calculation because twelve elements of the matrix are independently modified but less degrees of freedom are desired. Depending on how the constraints are enforced, different transformations may be obtained.

One possibility is to use a singular value or polar decomposition [MH94]. This separates the rotations expressed by $\tilde{\mathbf{M}}_T$ from the translations and shears. The operations allowed by the transformation type are then retained and those not desired removed, producing a valid matrix $\mathbf{M}_T$. The approach of [Pau84] is investigated here instead. It is suggested to first equate $\tilde{\mathbf{M}}_T$ to the representation of $\mathbf{M}_T$ given by equation (4.11):

$$\tilde{\mathbf{M}}_T = \mathbf{T}\left(\tfrac{1}{2}\mathbf{s}\right)\mathbf{R}_z\left(-\gamma\right)\mathbf{R}_y\left(-\beta\right)\mathbf{R}_x\left(-\alpha\right)\mathbf{T}\left(-\tfrac{1}{2}\mathbf{s}\right)\mathbf{T}\left(-\mathbf{t}\right) \tag{4.35}$$

Next, both sides of the equation are successively premultiplied by the inverses of the elementary operation matrices until elements that are either zero or constant appear on

the right hand side (ignoring the last row, which trivially always is $(0, 0, 0, 1)$). Two premultiplications are required for the matrix as given above:

$$\mathbf{R}_y(\gamma)\,\mathbf{T}\left(-\tfrac{1}{2}\mathbf{s}\right)\tilde{\mathbf{M}}_T = \mathbf{R}_y(-\beta)\,\mathbf{R}_x(-\alpha)\,\mathbf{T}\left(-\tfrac{1}{2}\mathbf{s}\right)\mathbf{T}(-\mathbf{t}) \tag{4.36}$$

Elementwise comparison of the two sides yields:

$$\left(\tilde{\mathbf{M}}_T\right)_{11}\cos\gamma - \left(\tilde{\mathbf{M}}_T\right)_{21}\sin\gamma = \cos\beta \tag{4.37}$$

$$\left(\tilde{\mathbf{M}}_T\right)_{12}\cos\gamma - \left(\tilde{\mathbf{M}}_T\right)_{22}\sin\gamma = \sin\alpha\sin\beta \tag{4.38}$$

$$\left(\tilde{\mathbf{M}}_T\right)_{13}\cos\gamma - \left(\tilde{\mathbf{M}}_T\right)_{23}\sin\gamma = -\cos\alpha\sin\beta \tag{4.39}$$

$$\begin{aligned}\left(\left(\tilde{\mathbf{M}}_T\right)_{14} - \tfrac{s_x}{2}\right)\cos\gamma - \left(\left(\tilde{\mathbf{M}}_T\right)_{24} - \tfrac{s_y}{2}\right)\sin\gamma = {} & \left(\tfrac{s_z}{2} + t_z\right)\cos\alpha\sin\beta \\ & + \left(-\tfrac{s_y}{2} - t_y\right)\sin\alpha\sin\beta \\ & + \left(-t_x - \tfrac{s_x}{2}\right)\cos\beta\end{aligned} \tag{4.40}$$

$$\left(\tilde{\mathbf{M}}_T\right)_{21}\cos\gamma + \left(\tilde{\mathbf{M}}_T\right)_{11}\sin\gamma = 0 \tag{4.41}$$

$$\left(\tilde{\mathbf{M}}_T\right)_{22}\cos\gamma + \left(\tilde{\mathbf{M}}_T\right)_{12}\sin\gamma = \cos\alpha \tag{4.42}$$

$$\left(\tilde{\mathbf{M}}_T\right)_{23}\cos\gamma + \left(\tilde{\mathbf{M}}_T\right)_{13}\sin\gamma = \sin\alpha \tag{4.43}$$

$$\begin{aligned}\left(\left(\tilde{\mathbf{M}}_T\right)_{24} - \tfrac{s_y}{2}\right)\cos\gamma + \left(\left(\tilde{\mathbf{M}}_T\right)_{14} - \tfrac{s_x}{2}\right)\sin\gamma = {} & \left(-\tfrac{s_y}{2} - t_y\right)\cos\alpha \\ & + \left(-\tfrac{s_z}{2} - t_z\right)\sin\alpha\end{aligned} \tag{4.44}$$

$$\left(\tilde{\mathbf{M}}_T\right)_{31} = \sin\beta \tag{4.45}$$

$$\left(\tilde{\mathbf{M}}_T\right)_{32} = -\sin\alpha\cos\beta \tag{4.46}$$

$$\left(\tilde{\mathbf{M}}_T\right)_{33} = \cos\alpha\cos\beta \tag{4.47}$$

$$\begin{aligned}\left(\tilde{\mathbf{M}}_T\right)_{34} - \tfrac{s_z}{2} = {} & \left(-\tfrac{s_z}{2} - t_z\right)\cos\alpha\cos\beta \\ & + \left(\tfrac{s_y}{2} + t_y\right)\sin\alpha\cos\beta \\ & + \left(-t_x - \tfrac{s_x}{2}\right)\sin\beta\end{aligned} \tag{4.48}$$

An estimate of γ can now be obtained from equation (4.41):

$$\tan\gamma = \frac{\sin\gamma}{\cos\gamma} \approx -\frac{\left(\tilde{\mathbf{M}}_T\right)_{21}}{\left(\tilde{\mathbf{M}}_T\right)_{11}} \tag{4.49}$$

Once γ is determined, all left hand sides of equations (4.37) to (4.48) can be calculated. There are several ways to obtain values for the other parameters. β can be estimated

by dividing equations (4.45) and (4.37), α from equations (4.43) and (4.42):

$$\tan\beta = \frac{\sin\beta}{\cos\beta} \approx \frac{\left(\tilde{\mathbf{M}}_T\right)_{31}}{\left(\tilde{\mathbf{M}}_T\right)_{11}\cos\gamma - \left(\tilde{\mathbf{M}}_T\right)_{21}\sin\gamma} \tag{4.50}$$

$$\tan\alpha = \frac{\sin\alpha}{\cos\alpha} \approx \frac{\left(\tilde{\mathbf{M}}_T\right)_{23}\cos\gamma + \left(\tilde{\mathbf{M}}_T\right)_{13}\sin\gamma}{\left(\tilde{\mathbf{M}}_T\right)_{22}\cos\gamma + \left(\tilde{\mathbf{M}}_T\right)_{12}\sin\gamma} \tag{4.51}$$

Estimates for t_x, t_y and t_z follow from equations (4.40), (4.44) and (4.48). While this scheme is attractive for its computational efficiency, it has several shortcomings. The first is that the estimates in equations (4.49), (4.50) and (4.51) can be undefined. In this case, arbitrary values have to be substituted for the angles. A far more grave problem is a lack of precision. Each estimated parameter is only guaranteed to satisfy the equations it was derived from. When the parameter is subsequently inserted into other equations, imprecisions arise from the fact that the left and right hand side may actually not be equal. The errors increase in the cascade of calculations where the estimated value of γ is used in the calculations of β and α and all three angles are required when estimating the components of $\mathbf{t}$. To obtain reliable values, a different approach is chosen in this thesis.

4.1.2.2. Direct Calculation

The second factor in equation (4.30) is directly calculated for each parameter p of the transformation. It can be rewritten as:

$$\frac{d}{dp}T\left(\mathbf{x}_u\right) = \frac{d}{dp}\mathbf{M}_T\mathbf{x}_u = \left(\frac{d}{dp}\mathbf{M}_T\right)\mathbf{x}_u \tag{4.52}$$

The matrix $\mathbf{M}_{T,p} := \frac{d}{dp}\mathbf{M}_T$ consists of the elements of $\mathbf{M}_T$, each differentiated by p. If such a matrix is calculated for each of the six parameters and equation (4.52) inserted into (4.30), the derivative of $v\left(T\left(\mathbf{x}_u\right)\right)$ with respect each parameter p can be calculated as:

$$\frac{d}{dp}v\left(T\left(\mathbf{x}_u\right)\right) = \operatorname{grad} v\left(T\left(\mathbf{x}_u\right)\right)\mathbf{M}_{T,p}\mathbf{x}_u \tag{4.53}$$

From the elements of $\mathbf{M}_T$ given by equations (4.13) to (4.28), it follows that for each parameter p, the last row of $\mathbf{M}_{T,p}$ contains all zeros. Arithmetic operations can therefore be saved if this row is omitted from the multiplication with the gradient. This also means that the fourth component of the gradient is not required and does not need to be calculated, as proposed in equation (4.31). Because the six matrices $\mathbf{M}_{Tp}$ are straightforward to calculate, only one of them is given here as an example:

$$\mathbf{M}_{T,t_x} = \begin{pmatrix} 0 & 0 & 0 & -\cos\beta\cos\gamma \\ 0 & 0 & 0 & \cos\beta\sin\gamma \\ 0 & 0 & 0 & -\sin\beta \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.54}$$

Since it is a priori known that most elements of the matrix will be zero, even more calculations can be saved by omitting the multiplications and additions that involve such elements.

4.1.3. Search Space

The search space specifies the range of values that should be considered for each of the parameters in T. Mathematically, all six parameters may be set to any real number. In practice, only certain values are sensible. The translation should not be made so large that there is no overlap between the two images. Since rotation is periodic, each angle can be confined to a range of 2π values.

However, in the experiments conducted for this thesis it was found that there is no need to explicitly limit the parameter values. As will be seen in chapter 7, starting from the rough estimate provided by aligning the centers of the two images or even a larger artificial misalignment, the similarity metric and search strategy never produced a misregistration by choosing unreasonable values for the parameters of the rigid transformation.

4.1.4. Other Tranformation Types

Global transformations can be used to correct a wider range of misalignments that those covered by the rigid transformation type used in this thesis. If the elements of $\mathbf{M}_T$ are allowed to be changed freely as long as the last component of $\mathbf{x}_u$ is preserved, the family of affine transformations is obtained. For three-dimensional images, twelve elements of the matrix may be modified:

$$\mathbf{x}_v = T(\mathbf{x}_u) = \begin{pmatrix} (\mathbf{M}_T)_{11} & (\mathbf{M}_T)_{12} & (\mathbf{M}_T)_{13} & (\mathbf{M}_T)_{14} \\ (\mathbf{M}_T)_{21} & (\mathbf{M}_T)_{22} & (\mathbf{M}_T)_{23} & (\mathbf{M}_T)_{24} \\ (\mathbf{M}_T)_{31} & (\mathbf{M}_T)_{32} & (\mathbf{M}_T)_{33} & (\mathbf{M}_T)_{34} \\ 0 & 0 & 0 & 1 \end{pmatrix} \mathbf{x}_u \tag{4.55}$$

Affine transformations include rotation and translation, but also scaling, shearing, reflection and orthogonal projection [FvDFH95]. They all have the property that three points $\mathbf{x}_u$ lying on a line in the reference image are mapped to three points $\mathbf{x}_v$ that also lie on a line in the registered image. If the condition that the last component of $\mathbf{x}_u$ be preserved is waived, all elements of the matrix may be set freely:

$$\mathbf{x}_v = T(\mathbf{x}_u) = \begin{pmatrix} (\mathbf{M}_T)_{11} & (\mathbf{M}_T)_{12} & (\mathbf{M}_T)_{13} & (\mathbf{M}_T)_{14} \\ (\mathbf{M}_T)_{21} & (\mathbf{M}_T)_{22} & (\mathbf{M}_T)_{23} & (\mathbf{M}_T)_{24} \\ (\mathbf{M}_T)_{31} & (\mathbf{M}_T)_{32} & (\mathbf{M}_T)_{33} & (\mathbf{M}_T)_{34} \\ (\mathbf{M}_T)_{41} & (\mathbf{M}_T)_{42} & (\mathbf{M}_T)_{43} & (\mathbf{M}_T)_{44} \end{pmatrix} \mathbf{x}_u \tag{4.56}$$

One important transformation that can be expressed this way is the perspective transformation. It is used in multi-view registration when aligning two images taken from different points of view.

If an even more complex distortion is to be undone, a popular choice are transformations that calculate the components of $\mathbf{x}_v$ as higher order polynomials of those of $\mathbf{x}_u$ [ZF03]. Such transformations are at the fuzzy border between global and local. Arguably, by using a high enough order, an arbitrarily complex transformation can be produced that is able to express any local adjustments required. However, with increasing order of the polynomials not only the number of parameters to be optimized quickly grows, but unwanted warping of the registered image may occur [Bro92]. Polynomials of orders higher than two or there are therefore not generally encountered. Actual local transformations are chosen instead.

4.2. Local

A local transformation is used to specify a mapping which varies throughout the image. This allows deformations to be corrected that only affect certain image areas. Because the transformation functions are usually far more complex than the simple matrix multiplications used in the previous sections, translation and scaling effects can be expressed directly by the functions and there is no need to use homogenous coordinates. The treatment of local transformations therefore assumes plain Cartesian coordinates.

4.2.1. Scattered Data Interpolation

Local transformations are often constructed around the concept of scattered data interpolation. An overview of such approaches is presented in [RM92]. The transformation is parametrized by a series of *control point* or *node* pairs. In each of the n pairs, $\mathbf{p}_i$ is a position in the reference image that remains constant throughout the registration process. $\mathbf{q}_i$ is an associated parameter vector that specifies the transformation of this node. Often, it is the registered image position that $\mathbf{p}_i$ should be mapped to:

$$\forall\ 1 \leq i \leq n:\ T(\mathbf{p}_i) = \mathbf{q}_i \tag{4.57}$$

Another possibility is that $\mathbf{q}_i$ is the displacement that must be added to $\mathbf{p}_i$ in order to obtain the registered image position:

$$\forall\ 1 \leq i \leq n:\ T(\mathbf{p}_i) = \mathbf{p}_i + \mathbf{q}_i \tag{4.58}$$

To extend the mapping to arbitrary points $\mathbf{x}_u$ in the reference image, a function f is used that interpolates between the values $\mathbf{q}_i$ at the nodes $\mathbf{p}_i$. The transformation functions for the two cases described above are:

$$\mathbf{x}_v = T(\mathbf{x}_u) = f(\mathbf{x}_u) \qquad \text{with } \forall\ 1 \leq i \leq n:\ f(\mathbf{p}_i) = \mathbf{q}_i \tag{4.59}$$

$$\mathbf{x}_v = T(\mathbf{x}_u) = \mathbf{x}_u + f(\mathbf{x}_u) \qquad \text{with } \forall\ 1 \leq i \leq n:\ f(\mathbf{p}_i) = \mathbf{q}_i \tag{4.60}$$

It is apparent that irrespectively of whether the $\mathbf{q}_i$ are registered image positions or displacements, the continuity and smoothness of the mapping are dependent only on

those of f. C^0 is always required so that there are no discontinuities in the transformation. To obtain a higher quality interpolation, C^1 or even higher degrees of continuity may be desired.

As noted in [RM92], the interpolation between the $\mathbf{q}_i$ is performed independently for each coordinate. In the case of 3D registration, f can therefore be split into three functions $f_k : \mathbb{R}^3 \to \mathbb{R}$, each of which interpolates between pairs $(\mathbf{p}_i, (\mathbf{q}_i)_k)$ and fulfills:

$$\forall\ 1 \leq i \leq n :\ f_k(\mathbf{p}_i) = (\mathbf{q}_i)_k \tag{4.61}$$

The problem of interpolating between scalar values associated with arbitrary positions $\mathbf{p}_i \in \mathbb{R}^d$ is known as scattered data interpolation and gives this class of local transformations its name. The problem also occurs in many other application areas such as the natural sciences where the distribution of a physical quantity throughout an area is to be estimated from a few discrete measurements [LWS97]. It has therefore been studied extensively and many interpolation techniques have been proposed.

Inverse distance weighted interpolation is a very flexible approach mentioned briefly in [Bro92] and described in more detail in [RM92]. The interpolation function is given by a weighted sum of local interpolants:

$$f_k(\mathbf{x}) = \sum_{i=1}^{n} w_{i,k}(\mathbf{x})\, f_{i,k}(\mathbf{x}) \tag{4.62}$$

Each interpolant $f_{i,k}$ spreads the value $(\mathbf{q}_i)_k$ throughout the image while the weight function $w_{i,k}$ limits its influence with growing distance from $\mathbf{p}_i$. The construction of an interpolation technique based on this approach is made difficult by the wide range of available weight functions, types of local interpolants and methods for determining their parameters.

Another idea is to triangulate the reference image with the $\mathbf{p}_i$ as the corners of the generated triangles. For each point $\mathbf{x}$, interpolation then only occurs between the three values $(\mathbf{q}_i)_k$ associated with the corners of the triangle it falls into. This allows for great computational efficiency but can quickly lead to the image folding over itself.

4.2.1.1. Radial Basis Functions

Another family of flexible interpolation techniques is based on the use of radial basis functions. Given a vector $\mathbf{x}$, the value of a radial basis function R depends only on the length of that vector:

$$R(\mathbf{x}) := R(\|\mathbf{x}\|) \tag{4.63}$$

An interpolation function can be constructed from two terms. The first is a weighted sum of n radial basis functions, each centered at a node $\mathbf{p}_i$ with its value depending on the Euclidean distance between $\mathbf{x}$ and $\mathbf{p}_i$. The second term is a polynomial of degree m, which can be expressed as a weighted sum of M functions ϕ_i forming a basis for all polynomials up to degree m [FRS99]:

$$f_k(\mathbf{x}) = \sum_{i=1}^{n} \alpha_{ik} R(\|\mathbf{x} - \mathbf{p}_i\|) + \sum_{i=1}^{M} \beta_{ik} \phi_i(\mathbf{x}) \tag{4.64}$$

A total of $n + M$ constraints are required to calculate the weights α_{ki} and β_{ki} for the given coordinate k. The first n constraints are provided by equation (4.61) and guarantee that f_k actually is an interpolation function. M additional constraints are obtained by imposing side conditions on the polynomials [FRS99]:

$$\forall\ 1 \leq j \leq M: \quad \sum_{i=1}^{n} \alpha_{ik} \phi_j \left(\mathbf{p}_i\right) = 0 \tag{4.65}$$

The constraints can be expressed as a system of linear equations:

$$\begin{pmatrix} \mathbf{C} & \mathbf{D} \\ \mathbf{D}^T & \mathbf{0} \end{pmatrix} \begin{pmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\beta} \end{pmatrix} = \begin{pmatrix} \mathbf{q} \\ \mathbf{0} \end{pmatrix} \tag{4.66}$$

$\boldsymbol{\alpha}$, $\boldsymbol{\beta}$ and $\mathbf{q}$ are column vectors composed of the unknown weights α_{ik}, β_{ik} and the values $(\mathbf{q}_i)_k$, respectively. The elements of the $n \times n$ matrix $\mathbf{C}$ and the $n \times M$ matrix $\mathbf{D}$ are given by:

$$C_{ij} = R\left(\|\mathbf{p}_i - \mathbf{p}_j\|\right) \tag{4.67}$$

$$D_{ij} = \phi_j \left(\mathbf{p}_i\right) \tag{4.68}$$

The weights can be obtained by inverting the matrix:

$$\begin{pmatrix} \boldsymbol{\alpha} \\ \boldsymbol{\beta} \end{pmatrix} = \begin{pmatrix} \mathbf{C} & \mathbf{D} \\ \mathbf{D}^T & \mathbf{0} \end{pmatrix}^{-1} \begin{pmatrix} \mathbf{q} \\ \mathbf{0} \end{pmatrix} \tag{4.69}$$

Although matrix inversion is a costly operation, it only needs to be executed once as the nodes $\mathbf{p}_i$ do not change throughout the registration process. When the parameters in $\mathbf{q}$ are modified, new weights may be obtained quickly by performing the matrix-vector multiplication. An overview of frequently used radial basis functions and the conditions that must be satisfied for each to guarantee that the matrix will be invertible are provided in [FRS99]. The functions are the thin-plate spline, multiquadric, inverse multiquadric and Gaussian:

$$R_{TPS}\left(r\right) = \begin{cases} r^{4-d} \ln r & 4 - d \in 2\mathbb{N} \\ r^{4-d} & \text{otherwise} \end{cases} \tag{4.70}$$

$$R_M\left(r\right) = \left(r^2 + c^2\right)^{\mu} \qquad \mu \in \mathbb{R}_+ \tag{4.71}$$

$$R_{IM}\left(r\right) = \left(r^2 + c^2\right)^{-\mu} \qquad \mu \in \mathbb{R}_+ \tag{4.72}$$

$$R_G\left(r\right) = e^{-\frac{r^2}{2\sigma^2}} \tag{4.73}$$

Some of the weaknesses of these functions are described in [FRS99] and [RM92]. For example, if the value of c is chosen incorrectly for multiquadrics and inverse multiquadrics, the interpolation will not be smooth or cause the registered image to fold over itself.

A very important shortcoming shared by all four functions is their global influence. Thin-plate splines and multiquadrics increase with r, making the modeling of local deformations difficult. The other two functions decrease with r but never reach zero. The polynomial component of equation (4.64) provides an additional global effect. However, it cannot be omitted for thin-plate splines and multiquadrics with $\mu > 1$ if a valid interpolation and the solvability of the system in equation (4.66) are to be guaranteed.

The global nature of these radial basis functions is problematic for two reasons. First, if any of the parameters is changed, the effects of this change on the alignment quality can only be assessed by recalculating the similarity metric for the whole image. Second, for every point $\mathbf{x}_u$ mapped into the registered image, the entire sums in equation (4.64) need to be evaluated causing a computational cost of $O(N + M)$ per point that increases with the number of nodes. Additionally, except for the multiquadrics and inverse multiquadrics with appropriately chosen μ, each addend contains a transcendental function.

4.2.1.2. Radial Basis Functions with Compact Support

Many of these problems can be overcome by using suitable radial basis functions with compact support. A popular choice are the $\psi_{d,l}(r)$ functions introduced by Wendland [Wen95]. They have several desirable properties:

Compact Support Compact support means that there exists a constant t for which $R(r) = 0$ when $r > t$. This gives each radial basis function only local influence within a radius of t. When evaluating the first sum in equation (4.64), functions centered around nodes $\mathbf{p}_i$ farther away from $\mathbf{x}$ than t may therefore be omitted from the calculation. All Wendland functions allways have influence radius $t = 1$.

Positive Definiteness For $\mathbf{x} \in \mathbb{R}^g$, all Wendland functions $\psi_{d,l}(\mathbf{x}) = \psi_{d,l}(\|\mathbf{x}\|)$ with $d \geq g$ are positive definite. This means that the matrix in equation (4.66) is always invertible and the polynomial component of equation (4.64) is not required. An interpolation function f can therefore be constructed that consists solely of a sum of weighted radial basis functions with local influence.

Polynomial Structure Wendland functions are simple polynomials which can quickly be evaluated without the need to calculate transcendental functions.

Continuity For $\mathbf{x} \in \mathbb{R}^d$, each $\psi_{d,l}(\mathbf{x})$ is C^{2l} continuous. The choice of l permits a trade-off between the smoothness of the interpolation and the complexity of the radial basis functions to be made.

Definitions of $\psi_{d,l}$ for all combinations of d and l may be found in [Wen95] and [FRS99]. Here, only the function chosen for this thesis is given. To obtain positive definiteness for three-dimensional image positions, $d = 3$. Because continuity C^0 is mathematically sufficient and leads to the simplest functions, $l = 0$. The resulting Wendland function is:

$$\psi_{3,0}(r) = \begin{cases} (1-r)^2 & 0 \leq r \leq 1 \\ 0 & r > 1 \end{cases} \tag{4.74}$$

As noted in [FRS99], the radius of influence may be changed by applying a scaling factor to r. A radial basis function with all properties of the Wendland function above but influence radius a is therefore given by:

$$R_a(r) = \begin{cases} \left(1 - \frac{r}{a}\right)^2 & 0 \leq r \leq a \\ 0 & r > a \end{cases} \tag{4.75}$$

4.2.2. Transformation Function

The first step in constructing a transformation function based on R_a is to insert the radial basis function into equation (4.64) to obtain an interpolation function. Since no polynomial term is required, the second sum may be omitted:

$$f_k(\mathbf{x}) = \sum_{i=1}^{n} \alpha_{ik} R_a(\|\mathbf{x} - \mathbf{p}_i\|) \tag{4.76}$$

Before this result can be used in either of the transformation functions given by equations (4.59) or (4.60), the separation into individual coordinates introduced in equation (4.61) must be reversed. Because the interpolation functions for the three coordinates differ only in the values of the weights α_{ik}, this is easily done:

$$f(\mathbf{x}) = \sum_{i=1}^{n} \boldsymbol{\alpha}_i R_a(\|\mathbf{x} - \mathbf{p}_i\|) \qquad \text{with } \boldsymbol{\alpha}_i = \begin{pmatrix} \alpha_{i1} \\ \alpha_{i2} \\ \alpha_{i3} \end{pmatrix} \tag{4.77}$$

The calculation of these weights is also identical for each coordinate. Due to the lack of a polynomial term, equation (4.69) simplifies to:

$$\boldsymbol{\alpha} = \mathbf{C}^{-1}\mathbf{q} \tag{4.78}$$

To combine the weight calculations for all coordinates into one expression, $n \times 3$ matrices A and Q consisting of the following elements are introduced:

$$A_{ik} = \alpha_{ik} \tag{4.79}$$
$$Q_{ik} = (\mathbf{q}_i)_k \tag{4.80}$$

All weights are then given as:

$$\mathbf{A} = \mathbf{C}^{-1}\mathbf{Q} \tag{4.81}$$

The $n \times n$ matrix $\mathbf{C}$ must be constructed from the positions of the nodes according to equation (4.67) and inverted only once at the beginning of the registration process. When the parameter values are later changed, new weights can be calculated quickly by executing the matrix multiplication with an updated matrix $\mathbf{Q}$.

The thus obtained scattered data interpolation technique is used in this thesis to interpolate between *displacements* $\mathbf{q}_i$. This has the significant advantage that in an image area far away from all control points, the transformation function in equation (4.60)

reduces to the identity transform. Nodes therefore only need to be placed where adjustments are required. If the $\mathbf{q}_i$ represented node positions in the registered image, the transformation function of equation (4.59) would map such distant areas to the nonsensical position of zero, making a dense arrangement of nodes throughout the entire image necessary. By inserting equation (4.77) into (4.60), the following family of local transformation functions is obtained:

$$T_l(\mathbf{x}_u) = \mathbf{x}_u + \sum_{i=1}^{n} \boldsymbol{\alpha}_i R_a(\|\mathbf{x}_u - \mathbf{p}_i\|) \tag{4.82}$$

As explained in section 1.3.2 and at the beginning of this chapter, registration is performed in two passes. First, the six parameters of a global transformation function are optimized. Then, the mapping is refined by introducing a local transformation. However, T_l has no global component that would allow it to incorporate the mapping calculated in the first pass. The global and local transformation functions are therefore combined to produce a function which can express both the global adjustments found in the first pass and the necessary local modifications determined in the second pass.

The position of a point $\mathbf{x}_u$ in reference image coordinates is first modified by T_l, offsetting it according to the local transformation. The result is then mapped into the registered image by applying the global transformation function from equation (4.12). Because the global transformation requires homogenous coordinates, the result of T_l is extended by a fourth component with the constant value of one:

$$\mathbf{x}_v = T(\mathbf{x}_u) = \mathbf{M}_T \begin{pmatrix} \mathbf{x}_u + \sum_{i=1}^{n} \boldsymbol{\alpha}_i R_a(\|\mathbf{x}_u - \mathbf{p}_i\|) \\ 1 \end{pmatrix} \tag{4.83}$$

It should be noted that the six parameters that determine the matrix $\mathbf{M}_T$ are only adjusted during the first pass of the registration process. When the local transformation is added, the search strategy is concerned only with the parameters of T_l. $\mathbf{M}_T$ remains constant.

4.2.3. Placement of Nodes

Before the family of transformation functions defined by equation (4.83) may be used for image registration, the nodes $\mathbf{p}_i$ must be distributed in the reference image. Their positions should satisfy several goals. Because the local component of the transformation function reduces to identity in areas without nearby nodes, they must be present in each region where local deformations are expected. A single node only allows for very simple adjustments so that to be able to undo complex misalignments, the regions of influence of several nodes should overlap.

During registration, the similarity metric must provide information about the impact of each parmeter on the current alignment. In a feature based metric, alignment quality is only evaluated for distinct features. To obtain information for the adjustment of every parameter vector $\mathbf{q}_i$, the node positions $\mathbf{p}_i$ should therefore coincide with feature

locations in the reference image. Since mutual information is an image based similarity metric, no such restriction exists. Alignment quality is measured not only at the nodes themselves but throughout the image.

Despite the freedom available, no node should be put in a region that contains mainly a uniform background intensity and few significant data as this may lead to severe misregistration. Because most intensity differences in such a region are only due to noise, the optimal displacement found by the similarity metric and search strategy will be based primarily on random noise. This random parametrization then has the potential to corrupt the alignment of significant image data in the vicinity.

Good results may be obtained by manually placing the nodes [FRS99]. However, in this thesis a fully automatic registration technique is desired. A method for automatically distributing the nodes was therefore developed. A grid of $8 \times 8 \times 8$ nodes is constructed and placed over the reference image so that its center coincides with the center of the image. The grid spacing d is identical in all dimensions and calculated such that the nodes fully cover the longest side of the reference image. With $\mathbf{s}'$ the size of the image, d is given by:

$$d = \frac{\max\{s'_1, s'_2, s'_3\}}{8} \tag{4.84}$$

The radius of influence of each node is $r = 1.5d$. This guarantees that at every point in the grid, the influences of several nodes overlap. After the grid has been positioned, nodes that are determined to be unnecessary are removed. First, all nodes that fall outside the image are dropped. Because the grid forms a perfect cube but the tomographic dataset may be a cuboid of arbitrary size, eight nodes may not be required to cover the entire image in each dimension.

Next, the area surrounding each node $\mathbf{p}_i$ is analyzed to determine whether it contains enough significant data. This is illustrated in figure 4.1. The radial basis function $R_{1.5d}$ centered around the node gives it an area of influence that is a sphere of radius $r = 1.5d$. However, because the influence diminishes with the distance from $\mathbf{p}_i$, only a smaller spherical region of radius d in which the radial basis function is deemed to have substantial influence is considered in the analysis.

To ascertain the amount of significant data in this region, the intensity of the reference image is evaluated at a number of points. In order to obtain reproducible results, the points are arranged in a deterministic, regular pattern. They form a smaller three-dimensional grid with a spacing of $\frac{d}{4}$ centered around the node and truncated where the Euclidean distance from $\mathbf{p}_i$ exceeds d. This leads to a total of 257 points around each potential node.

A point is regarded as indicating significant data when its intensity exceeds a threshold t_1. The number of such points is counted and the node is kept only if this count is above another threshold t_2. The values used for these thresholds have been determined empirically and are $t_1 = 7$ and $t_2 = 85$. With the image having integer intensities in the range of $[0, 255]$, this means that a node is only retained if at least one-third of the points evaluated in its region of substantial influence have an intensity that is above a background noise level.

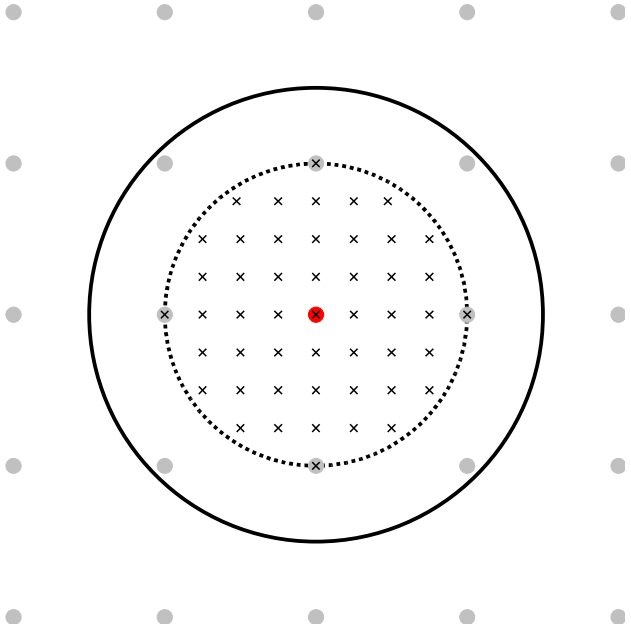


Figure 4.1: Two-dimensional cut through the grid of nodes: node (red); area of influence (solid line); area of substantial influence (dashed line); evaluation points (ticks); neighboring nodes (gray)

Although the technique is very simple, experiments have shown that it is able to automatically produce good node arrangements for different types of tomographic scans. Examples for PET and CT scans serving as reference images are provided in figure 4.2. They show two-dimensional cuts through the images along the grid planes. The number of nodes in each dimension, 8, is chosen arbitrarily and can be increased to provide more local control.

4.2.4. Intensity Derivative

According to section 2.6.2, calculation of the mutual information derivative requires a formula for the evaluation or estimation of $\frac{d}{dT} v\left(T\left(\mathbf{x}_u\right)\right)$. To determine at a suitable expression, the chain rule is applied first:

$$\frac{d}{dT} v\left(T\left(\mathbf{x}_u\right)\right) = \operatorname{grad} v\left(T\left(\mathbf{x}_u\right)\right) \frac{d}{dT} T\left(\mathbf{x}_u\right) \tag{4.85}$$

The gradient of v at $\mathbf{x}_v = T\left(\mathbf{x}_u\right)$ may be estimated in the same way as was done in equation (4.31) for the global transformation. As will be seen, the fourth component is again not required:

$$\operatorname{grad} v\left(\mathbf{x}_v\right) \approx \left(v\left(\mathbf{x}_v + \mathbf{e}_1\right) - v\left(\mathbf{x}_v\right), v\left(\mathbf{x}_v + \mathbf{e}_2\right) - v\left(\mathbf{x}_v\right), v\left(\mathbf{x}_v + \mathbf{e}_3\right) - v\left(\mathbf{x}_v\right), 0\right) \tag{4.86}$$

An expression for the second factor in equation (4.85) is easiest derived for each parameter $\left(\mathbf{q}_i\right)_k$ separately. It is first decomposed into its global and local components

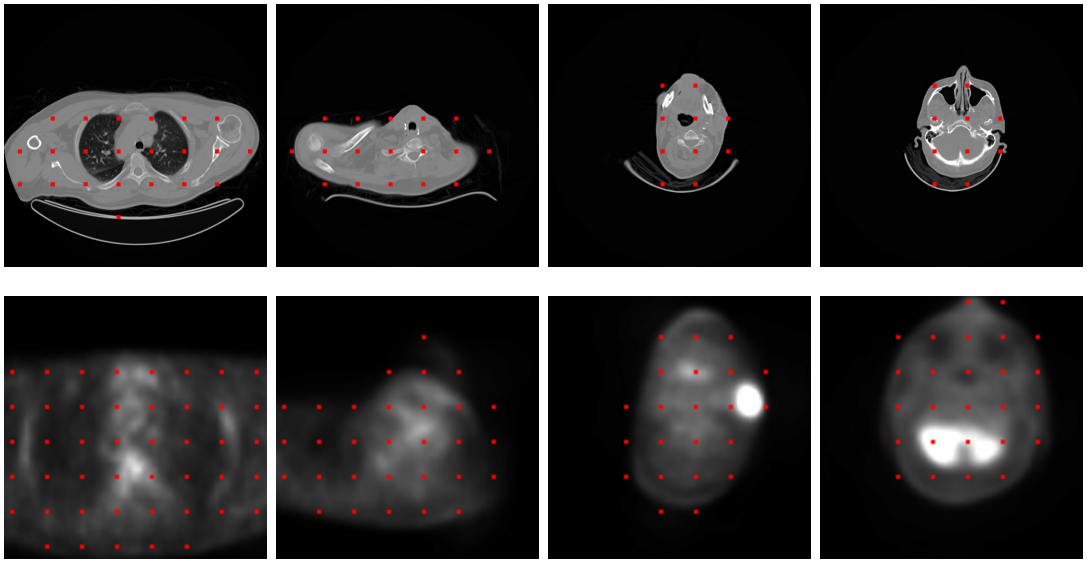


Figure 4.2: Automatically generated node arrangements for a CT scan (top row) and a PET scan (bottom row)

using equations (4.82) and (4.83):

$$\frac{d}{d\left(\mathbf{q}_i\right)_k} T\left(\mathbf{x}_u\right) = \frac{d}{d\left(\mathbf{q}_i\right)_k} \mathbf{M}_T \begin{pmatrix} T_l\left(\mathbf{x}_u\right) \\ 1 \end{pmatrix} = \mathbf{M}_T \begin{pmatrix} \frac{d}{d(\mathbf{q}_i)_k} T_l\left(\mathbf{x}_u\right) \\ 0 \end{pmatrix} \tag{4.87}$$

The resulting column vector is denoted $\mathbf{d}$. Because the interpolation between the $\mathbf{q}_i$ and thus the entire local transformation T_l is calculated independently for each coordinate, the derivative with respect to $(\mathbf{q}_i)_k$ can be non-zero only for the kth coordinate of $T_l\left(\mathbf{x}_u\right)$. All components of $\mathbf{d}$ other than d_k are therefore always zero and do not contribute to the matrix-vector product:

$$\frac{d}{d\left(\mathbf{q}_i\right)_k} T\left(\mathbf{x}_u\right) = \begin{pmatrix} \left(\mathbf{M}_T\right)_{1k} d_k \\ \left(\mathbf{M}_T\right)_{2k} d_k \\ \left(\mathbf{M}_T\right)_{3k} d_k \\ \left(\mathbf{M}_T\right)_{4k} d_k \end{pmatrix} = \begin{pmatrix} \left(\mathbf{M}_T\right)_{1k} \\ \left(\mathbf{M}_T\right)_{2k} \\ \left(\mathbf{M}_T\right)_{3k} \\ \left(\mathbf{M}_T\right)_{4k} \end{pmatrix} d_k \tag{4.88}$$

With $\left(\mathbf{M}_T\right)_k$ denoting the kth column of $\mathbf{M}_T$, this may be written more compactly as:

$$\frac{d}{d\left(\mathbf{q}_i\right)_k} T\left(\mathbf{x}_u\right) = \left(\mathbf{M}_T\right)_k d_k \tag{4.89}$$

Using equation (4.82), d_k expands into:

$$d_k = \frac{d}{d\left(\mathbf{q}_i\right)_k} \left(\left(\mathbf{x}_u\right)_k + \sum_{j=1}^{n} \alpha_{jk} R_a\left(\left\|\mathbf{x}_u - \mathbf{p}_j\right\|\right) \right) \tag{4.90}$$

$$= \sum_{j=1}^{n} \left(\frac{d}{d\left(\mathbf{q}_i\right)_k} \alpha_{jk} \right) R_a \left(\left\| \mathbf{x}_u - \mathbf{p}_j \right\| \right) \tag{4.91}$$

The derivative of α_{jk} can be rewritten by inserting equation (4.81):

$$\frac{d}{d\left(\mathbf{q}_i\right)_k} \alpha_{jk} = \frac{d}{d\left(\mathbf{q}_i\right)_k} \left(\mathbf{C}^{-1} \mathbf{Q} \right)_{jk} \tag{4.92}$$

$$= \frac{d}{d\left(\mathbf{q}_i\right)_k} \sum_{h=1}^{n} \left(\mathbf{C}^{-1} \right)_{jh} Q_{hk} \tag{4.93}$$

$$= \sum_{h=1}^{n} \left(\mathbf{C}^{-1} \right)_{jh} \frac{d}{d\left(\mathbf{q}_i\right)_k} Q_{hk} \tag{4.94}$$

With the definition of Q_{hk} from equation (4.80), this becomes:

$$\frac{d}{d\left(\mathbf{q}_i\right)_k} \alpha_{jk} = \sum_{h=1}^{n} \left(\mathbf{C}^{-1} \right)_{jh} \frac{d}{d\left(\mathbf{q}_i\right)_k} \left(\mathbf{q}_h \right)_k \tag{4.95}$$

The derivative is one for $h = i$ and zero in all other cases so that the sum simplifies to a single term:

$$\frac{d}{d\left(\mathbf{q}_i\right)_k} \alpha_{jk} = \left(\mathbf{C}^{-1} \right)_{ji} \tag{4.96}$$

The result of equation (4.96) inserted into (4.91) produces the following expression for d_k:

$$d_k = \sum_{j=1}^{n} \left(\mathbf{C}^{-1} \right)_{ji} R_a \left(\left\| \mathbf{x}_u - \mathbf{p}_j \right\| \right) \tag{4.97}$$

The desired formula for the derivative of the entire transformation is obtained by inserting d_k into equation (4.89):

$$\frac{d}{d\left(\mathbf{q}_i\right)_k} T\left(\mathbf{x}_u\right) = \left(\mathbf{M}_T\right)_k \sum_{j=1}^{n} \left(\mathbf{C}^{-1} \right)_{ji} R_a \left(\left\| \mathbf{x}_u - \mathbf{p}_j \right\| \right) \tag{4.98}$$

According to equation (4.85), $\frac{d}{d\left(\mathbf{q}_i\right)_k} v\left(T\left(\mathbf{x}_u\right)\right)$ can now be calculated by multiplying the estimate of the gradient from equation (4.86) with equation (4.98). Because according to equations (4.25) to (4.27), the last component of $\left(\mathbf{M}_T\right)_k$ is always zero, its multiplication with the fourth component of the gradient may be omitted. It is therefore not necessary to estimate that part of the gradient, confirming the assumption made in equation (4.86).

It is apparent from equation (4.98) that the calculations required for each of the three parameters in a parameter vector $\mathbf{q}_i$ differ only in the column of $\mathbf{M}_T$ used. By multiplying with the entire matrix, the derivatives with respect to all components of $\mathbf{q}_i$ can be obtained together. The final expression obtained is:

$$\frac{d}{d\mathbf{q}_i} v\left(T\left(\mathbf{x}_u\right)\right) = \operatorname{grad} v\left(T\left(\mathbf{x}_u\right)\right) \mathbf{M}_T \sum_{j=1}^{n} \left(\mathbf{C}^{-1} \right)_{ji} R_a \left(\left\| \mathbf{x}_u - \mathbf{p}_j \right\| \right) \tag{4.99}$$

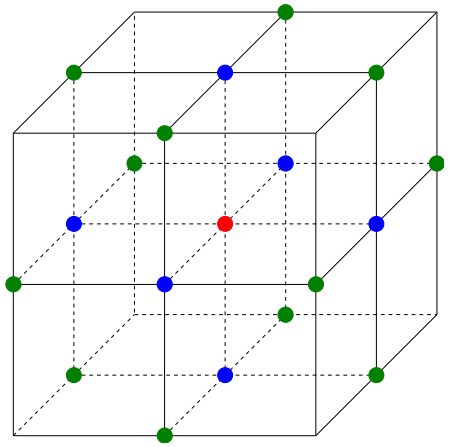


Figure 4.3: Neighboring grid points of node $\mathbf{p}_i$: $\mathbf{p}_i$ (red); grid points offset by one grid spacing in one coordinate (blue); grid points offset by one grid spacing in two coordinates (green)

4.2.5. Optimizations

The nature of the transformation function and the arrangement of nodes allow for several significant optimizations.

4.2.5.1. Matrix $\mathbf{C}^{-1}$ Generation

When constructing the matrix $\mathbf{C}$ according to equation (4.67), a radial basis function is evaluated for all pairs $(\mathbf{p}_i, \mathbf{p}_j)$. This process can be vastly accelerated. Because radial basis functions with compact support and influence radius $r = 1.5d$ are used, only node pairs whose Euclidean distance is smaller than this radius need to be considered. For all other pairs, it follows from equation (4.75) that the matrix elements are zero.

As the node arrangement is a subset of a regular grid, the nodes $\mathbf{p}_j$ who fall into a radius of r around $\mathbf{p}_i$ can be quickly determined. The situation is illustrated in figure 4.3. $\mathbf{p}_i$ has a distance to itself of zero. Six neighboring grid points are offset from $\mathbf{p}_i$ by one grid spacing in one coordinate, leading to a distance of d. Twelve more grid points are offset by one grid spacing in two coordinates, thus having a distance of $\sqrt{2}d$. All other grid points are farther away than $1.5d$. Relevant are thus only the nodes $\mathbf{p}_j$ located at one of the eighteen neighboring grid points. In an implementation, it needs to be taken into account that the grid has finite size and some of the grid positions may not exist.

Because all relevant nodes are located at fixed distances from $\mathbf{p}_i$, the required values of the radial basis functions can be precalculated:

$$c_{ij} = \begin{cases} R_{1.5d}(0) & \mathbf{p}_j = \mathbf{p}_i \\ R_{1.5d}(d) & \mathbf{p}_j \text{ offset from } \mathbf{p}_i \text{ by one grid spacing in one coordinate} \\ R_{1.5d}\left(\sqrt{2}d\right) & \mathbf{p}_j \text{ offset from } \mathbf{p}_i \text{ by one grid spacing in two coordinates} \\ 0 & \text{otherwise} \end{cases} \tag{4.100}$$

The matrix can now efficiently be constructed. Its elements are initially all set to zero. Then, for each $\mathbf{p}_i$ only c_{ii} and up to eighteen c_{ij} are updated. Because $\mathbf{C}$ is sparse, its inversion can also be optimized by using one of the efficient and numerically stable techniques available for such matrices [KSSH02].

4.2.5.2. Local Evaluation of $MI^\star$

Mutual information, as an image based similarity metric, only provides a cumulative assessment of the alignment quality for the entire image. When using a local transformation, it is desirable to define a new metric $MI^\star(\mathbf{q}_i)$ that evaluates the degree of alignment achieved by a single parameter vector $\mathbf{q}_i$. Unfortunately, the radial basis function centered at each node $\mathbf{p}_j$ is weighted with a factor $\boldsymbol{\alpha}_j$ calculated in equation (4.81) as the weighted sum of all parameter vectors. The influence of each vector on the transformation is therefore global and it is impossible to isolate the effects of $\mathbf{q}_i$.

However, the influence of every $\mathbf{q}_i$ can be considered *approximatively* local. Because by virtue of equation (4.60), the displacement at each node $\mathbf{p}_i$ is precisely $\mathbf{q}_i$ and the interpolation function attempts to provide a smooth transition between these displacements throughout the image, the influence of $\mathbf{q}_i$ is strongest at $\mathbf{p}_i$ and decreases with distance while the influences of other $\mathbf{q}_j$ increase. The quality of alignment achieved by $\mathbf{q}_i$ may therefore be estimated by calculating the mutual information for a small image region centered around $\mathbf{p}_i$.

The region chosen in this thesis is the area of substantial influence introduced in figure 4.1. It is defined as a sphere around $\mathbf{p}_i$ with a radius of d. As will be seen in chapter 7, this provides a metric that is largely representative of the alignment quality produced by $\mathbf{q}_i$.

4.2.5.3. Local Evaluation of $\frac{d}{d\mathbf{q}_i} MI^\star$

As it is also image based, the derivative of the mutual information with respect to a parameter vector $\mathbf{q}_i$ expresses how changes to its components would affect the alignment quality of the entire image. Because the influence of the parameter vector is approximatively local, most of the samples used in the calculation are drawn from image areas on which $\mathbf{q}_i$ has almost no effect. To obtain an estimate of the sign and relative magnitude of the mutual information derivative using fewer samples, it is therefore reasonable to use only those located in a small image region centered around $\mathbf{p}_i$ in which $\mathbf{q}_i$ is deemed to have significant influence on the transformation function.

The region used is again the area of substantial influence. This way, the calculated value is $\frac{d}{d\mathbf{q}_i} MI^\star(\mathbf{q}_i)$, the derivative of the local mutual information introduced in the previous section.

4.2.5.4. Calculation of T and $\frac{d}{d\mathbf{q}_i} T$

When using equation (2.41) to estimate mutual information, the transformation function T needs to be evaluated for each sampling point $\mathbf{x}_u$. To estimate the mutual information derivative in equation (2.46), it is additionally necessary to calculate $\frac{d}{d\mathbf{q}_i} T$. As seen in

equations (4.83) and (4.99), the bulk of these calculations is spent evaluating a weighted sum of radial basis functions with the weights denoted here w_j as they have no influence on the following considerations:

$$\sum_{j=1}^{n} w_j R_{1.5d}\left(\|\mathbf{x}_u - \mathbf{p}_j\|\right) \tag{4.101}$$

Because the radial basis functions have compact support, it is unnecessary to iterate over all n of them. Instead, only those centered around nodes $\mathbf{p}_j$ within a radius of $r = 1.5d$ from $\mathbf{x}_u$ need to be considered. With the indices of these nodes stored in $N(\mathbf{x}_u)$, equation (4.101) can be rewritten as:

$$\sum_{j \in N(\mathbf{x}_u)} w_j R_{1.5d}\left(\|\mathbf{x}_u - \mathbf{p}_j\|\right) \tag{4.102}$$

The set of relevant nodes and the corresponding values of the radial basis functions can be precalculated for each sampling point. Because the two samples encompass $N_A + N_B$ points and $MI^\star$ and its derivative are evaluated separately for each node $\mathbf{p}_i$, the total number of sampling points is $(N_A + N_B)\, n$. To reduce the amount of precalculations required, the same pattern of sampling points $\mathbf{x}'_u$ is used for each node. The pattern consists of $N_A + N_B$ points scattered randomly in a sphere of radius d around an imaginary node at the origin. The actual sampling points $\mathbf{x}_u$ for an evaluation at the node $\mathbf{p}_i$ are obtained by translating the pattern so that it is centered around this node:

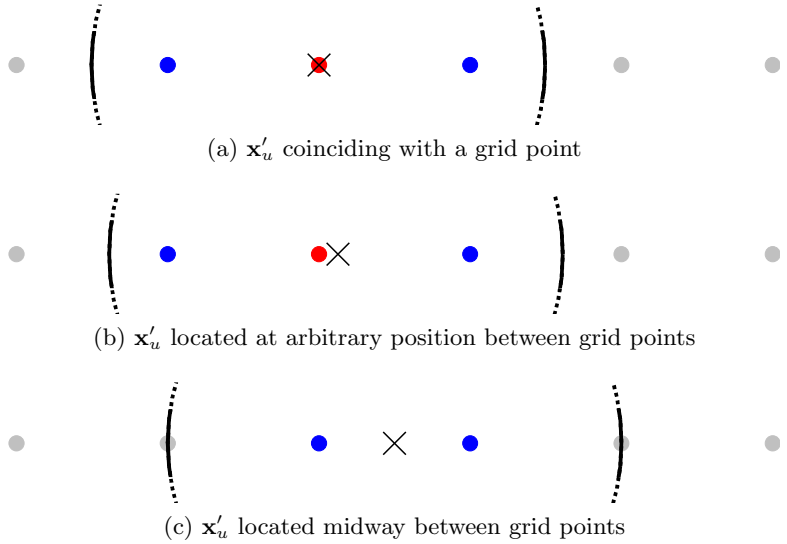
$$\mathbf{x}_u = \mathbf{x}'_u + \mathbf{p}_i \tag{4.103}$$

Precalculations are now only performed for the $N_A + N_B$ points $\mathbf{x}'_u$. It is impossible to directly determine the set of nodes $N(\mathbf{x}_u)$ this way as the nodes are different depending on which $\mathbf{p}_i$ the pattern is centered around. The grid points at which such nodes may be located also depend on $\mathbf{p}_i$. However, their positions are constant if they are expressed relative to the grid point $\mathbf{g}(\mathbf{p}_i)$ of $\mathbf{p}_i$.

Due to this fact, a set of relevant grid point offsets $G(\mathbf{x}'_u)$ can be precalculated for each $\mathbf{x}'_u$. When the pattern is centered around a $\mathbf{p}_i$, the node indices $N(\mathbf{x}_u)$ for each $\mathbf{x}_u$ can then be quickly found by adding each offset in $G(\mathbf{x}'_u)$ to the grid point $\mathbf{g}(\mathbf{p}_i)$ and determining whether a node is present at that location in the grid or not. In an implementation, it needs to be taken into account that because of the finite size of the grid, some of the grid points may not exist. The value of the radial basis function can also be precalculated for each offset in $G(\mathbf{x}'_u)$ as the distances between the grid points and $\mathbf{x}_u$ remain constant when the pattern is centred around different $\mathbf{p}_i$.

To quickly and efficiently determine $G(\mathbf{x}'_u)$, the set is not calculated precisely. Instead, an entire rectangular sub-grid is used that encompasses all points within a radius of $1.5d$ from $\mathbf{x}'_u$ and some number of additional points that are actually farther away. Such spurious points do not corrupt the calculation because their radial basis functions are zero for $\mathbf{x}'_u$ so that they have no influence on the value of equation (4.102).

The size of the sub-grid is calculated independently for each dimension. The largest grid is required when $\mathbf{x}'_u$ falls in line with the grid points in this dimension, as illustrated



(a) $\mathbf{x}'_u$ coinciding with a grid point

(b) $\mathbf{x}'_u$ located at arbitrary position between grid points

(c) $\mathbf{x}'_u$ located midway between grid points

Figure 4.4: One-dimensional view of grid points relevant to $\mathbf{x}'_u$: $\mathbf{x}'_u$ (tick); area of influence (circle segments); closest grid point (red); other relevant grid points (blue); irrelevant grid points (gray)

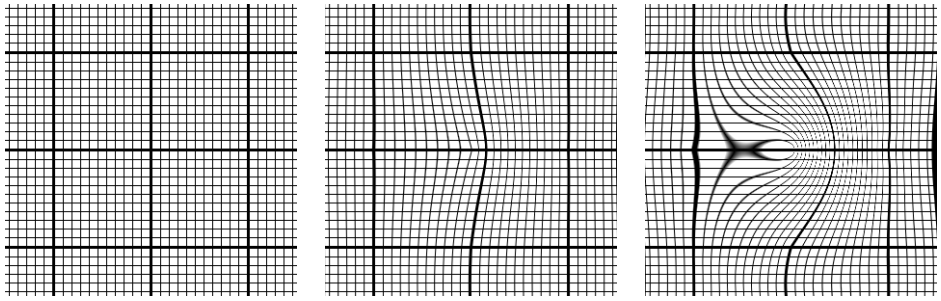
in figure 4.4. As seen in cases (a) and (b), if $\mathbf{x}'_u$ coincides with a grid point or lies at an arbitrary position between two of them, three grid points centered around the one closest to $\mathbf{x}'_u$ fall within a radius of $1.5d$ and are relevant. In the corner case (c), $\mathbf{x}'_u$ lies midway between two grid points and only these two are close enough to be relevant. However, because it is permissible to declare spurious points relevant, this case may be treated in line with the others by choosing one of the two grid points as the closest one and using the three points centered around it.

To be certain all relevant grid points are covered, it is thus sufficient to always use a $3 \times 3 \times 3$ sub-grid centered around the grid point closest to $\mathbf{x}'_u$. Because $G(\mathbf{x}'_u)$ is always a sub-grid of the same shape, it does not actually need to be stored for each sampling point. Instead, only the offset $\mathbf{g}(\mathbf{x}'_u)$ of the grid point closest to $\mathbf{x}'_u$ is recorded and G is reconstructed on the fly by adding or subtracting one from each of the coordinates.

The offset from the grid point the pattern is centered about to the point closest to $\mathbf{x}'_u$ is given directly by the coordinates of $\mathbf{x}'_u$. With Gaussian brackets $[\cdot]$ indicating the rounding operation, the kth coordinate of the offset is:

$$g_k\left(\mathbf{x}'_u\right) = \left[\frac{(\mathbf{x}'_u)_k}{d}\right] \tag{4.104}$$

In summary, what needs to be done while precalculating is to calculate $\mathbf{g}(\mathbf{x}'_u)$ and for each of the 27 points in a sub-grid around it, the value of the radial basis function at $\mathbf{x}'_u$. To evaluate T or its derivative $\frac{d}{d\mathbf{q}_i}T$ at a point $\mathbf{x}_u = \mathbf{x}'_u + \mathbf{p}_i$, $G(\mathbf{x}'_u)$ is constructed on the fly from $\mathbf{g}(\mathbf{x}'_u)$, translated by $\mathbf{g}(\mathbf{p}_i)$ and the nodes found at the resulting grid positions used, along with the precalculated values of the radial basis functions.



(a) (b) (c)

Figure 4.5: Topology of the registered image corrupted by a too large displacement: (a) no displacements; (b) small displacement at center node, preserving topology; (c) large displacement at center node, corrupting topology

4.2.6. Search Space

An important requirement expressed in section 1.2.2 is that the transformation function be injective so that the registered image is not folded over itself. This preservation of topology cannot be guaranteed for the local transformation function developed in this chapter if the displacements $\mathbf{q}_i$ at the nodes are made too large. An example is shown in figure 4.5. Nine nodes are located at the intersections of the bold lines. The center node is displaced to the right so far that the grid's topology is corrupted.

The results of a mathematical analysis of the maximal permissible displacements are presented in [FRS99]. Unfortunately, it is found there that the analysis is very complicated for nodes with intersecting areas of influence. An evaluation is performed for a single isolated node instead and it is suggested to use the results as references when choosing the maximum displacement for a more complex situation.

The largest displacement preserving the topology of the registered image is found to be $0.28r$ if the radial basis function is $\psi_{3,2}$ with influence radius r and $0.34r$ if it is $\psi_{3,1}$ of the same radius. From this data, a maximal allowable displacement of about $0.4r$ can be extrapolated for the $\psi_{3,0}$ function used in this thesis.

To be able to independently adjust the components of the $\mathbf{q}_i$, instead of a maximal total displacement, a limit for the absolute value of each component is chosen. It is set to $0.2r$ so that the total displacement at a node does not exceed $\sqrt{3 \times 0.2^2}r \approx 0.35r$. This provides a margin of $0.05r$ to compensate for the complex interactions of nodes with intersecting areas of influence.

4.2.7. Other Transformation Types

Many alternative approaches to the one used in this thesis have also been proposed for the construction of local transformation functions [ZF03]. Other types of radial basis functions can be considered, for example B-splines and elastic body splines. Instead of interpolating between the $\mathbf{q}_i$, it is possible to use a transformation function that only approximates these values at the nodes $\mathbf{p}_i$. This may result in an overall smoother

transformation that is able to preserve topology when an interpolation would fail to do so.

Based on a completely different idea are elastic transformations. The registered image is treated like a rubber sheet. The parameters adjusted during registration are forces acting on this sheet, deforming it. They can be counteracted by stiffness constraints which limit the deformation. This allows for the interesting possibility of declaring certain image areas, such as those where bones have been identified, as completely rigid and not prone to deformation. The downsides of such transformations are their costly evaluation and the influence on a large image area that each parameter has.

5. Registration System

An overview of the proposed registration technique has been provided in section 1.3.2 and detailed descriptions of its main components in the three chapters that followed. Now, an actual registration system based on these ideas is presented. The implementation focuses on efficiency, attempting to provide the fastest registration possible. The system is modular so that if faster replacements are developed for some of its components, they can be plugged in to speed up the calculations. This will be done in the next chapter when the most time-consuming aspects of the registration are ported to a GPU, or graphics processing unit.

The entire system is split into two applications. The first performs preprocessing steps and stores the images in a format that is easier to use in the context of registration. The second application is concerned with the registration process itself. Such a division is convenient in an academic system as the results of the preprocessing can be reused when changes are made to the registration routines. In a clinical setting, it would be more sensible to combine the two applications providing a one-step solution to the registration of tomographic images.

All code is written in C++ and relies on the utility classes provided by Qt 4[1]. This toolkit provides many useful routines for areas such as GUI development and file access as well as general enhancements to the C++ language. The applications are highly portable and have been developed and tested on both Linux and FreeBSD.

5.1. Preprocessing

The preprocessing application performs several duties. First, the two datasets are loaded from the files generated by tomographic scanners and their intensities brought into a range that is more suitable to efficient calculations. Next, a human operator is given the opportunity of manual intervention by specifying regions of interest. As explained in section 1.3.1, if only small areas within the volumetric images are relevant to the diagnosis, registration can be accelerated by limiting it to these regions.

The reasoning in this case is different. It was feared that the similarity metric and local transformation may be unable to cope with extraneous information such the beds on which the patient was positioned for the two scans being visible in the images. The region of interest specification is therefore optimized to selecting the human shape and clipping away these beds. However, as seen in section 4.2.3, a system for the automatic placement of nodes was developed that reliably positions nodes only where organic tissue

[1] `http://www.trolltech.com/products/qt/`

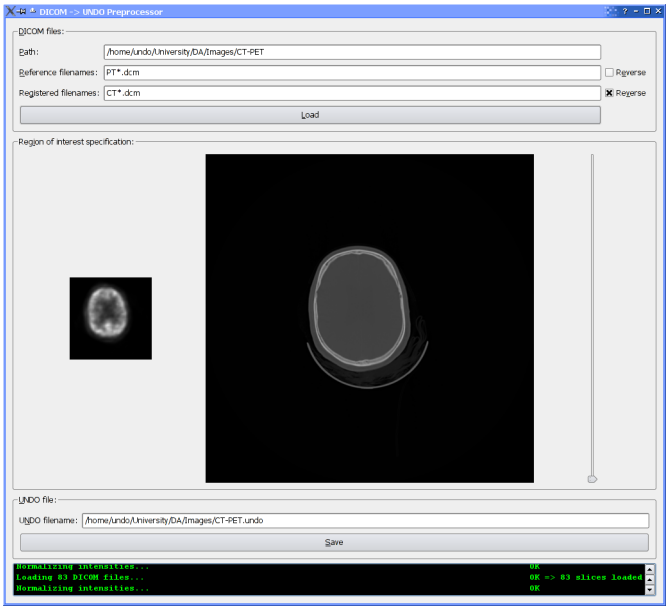


Figure 5.1: Screenshot of the preprocessing application

is present, ignoring the beds. A manual specification of the regions of interest, while still possible, is therefore not required.

As a final step, the two images are saved in a single file that serves as the input of the registration application. A simple file format was developed for this purpose that can efficiently be read and written and stores only the information that is relevant to the registration process. Figure 5.1 shows a screenshot of the preprocessing application.

5.1.1. Loading: The DICOM Standard

The methods and protocols used for exchanging image data between medical devices are specified in the DICOM standard. Part 5 [Nat06] describes the file format to be used when storing medical images. Regardless of modality and manufacturer, tomographic scanners are expected to generate DICOM conformant files. The standard is extensive, allowing for a variety of data types, compression techniques and storage formats. Additionally, most devices are not entirely conformant and generate files that do not strictly adhere to the specifications. The development of DICOM reading routines is therefore a tedious task best done once and abstracted into a library.

The library used in this thesis is GDCM[2]. It is written in C++ and provides a simple class structure for accessing the image intensities and metadata stored in a DICOM file. While the extraction of compressed images is handled transparently, other differences between individual datasets are exposed. Most notably, some scans are stored in a single file while with others, each slice needs to be loaded separately. Also, different bit depths

[2] `http://www.creatis.insa-lyon.fr/Public/Gdcm/`

can be used to express the image intensities. Based on the images available during the development of the registration system, the preprocessing application assumes that slices are stored in separate files and handles several different bit depths. Because it is not apparent from a set of DICOM files whether the first or the last contains the superior slice, a check box is provided to reverse the order in which the slices are arranged.

In a CT scan, intensities are expressed in Hounsfield units where -1000 corresponds to the density of air and 1000 to that of bone. For other modalities, different scales and units are used. In a multi-modal registration system that can handle different types of data, a uniform scale is desirable. The highest and lowest intensities are therefore found in each image and the intensities normalized to an integer scale of $[0, 255]$.

Besides the raw image slices, DICOM files contain metadata. Relevant to registration is the spacing between the voxels in millimeters. Using this information, it is possible to automatically match the scales of the reference and registered image. As noted in section 1.3.2, this is not done by resizing one of the images but by implicitly scaling the coordinates on each access to a voxel intensity. The voxel spacing is therefore only extracted to be forwarded to the registration application.

5.1.2. Regions of Interest

Following the explanation provided in section 5.1, the region of interest specification is optimized for isolating the patient's body and clipping away the beds in the two images. A shape that is very suitable for this purpose is the ellipse. The operator may specify an elliptical region that is to be retained for each slice. Data outside the ellipses is ignored.

For registration to operate only on the specified regions of interest, all voxels not covered by the ellipses must be removed. However, this leads to images with highly irregular shapes and makes registration inefficient as nodes and sampling points must be distributed within such complex bounds. Instead, a bounding box is constructed for each image that covers the areas of all ellipses. Within this box, voxels that fall outside the ellipses are set to a background intensity of zero. All image data within the box is then used for registration. This results in an image that all extraneous information replaced with a neutral background intensity while remaining cuboid like the original dataset. The process is illustrated in figure 5.2.

For each slice, the region of interest can be conveniently specified using a mouse. Dragging moves or resizes the ellipse, depending on whether the left or right button is held down. Clicking the middle mouse button removes the ellipse. If no region of interest is set for a slice, an ellipse is used whose size and position are linearly interpolated between the closest slices for which they have been manually specified. Only when no region of interest is specified for any slice, the clipping process is omitted and all voxels are used in the registration.

5.1.3. Saving: The UNDO File Format

After two tomographic datasets have been loaded, normalized and optionally clipped to smaller regions of interest, they are saved in a file that serves as the input of the

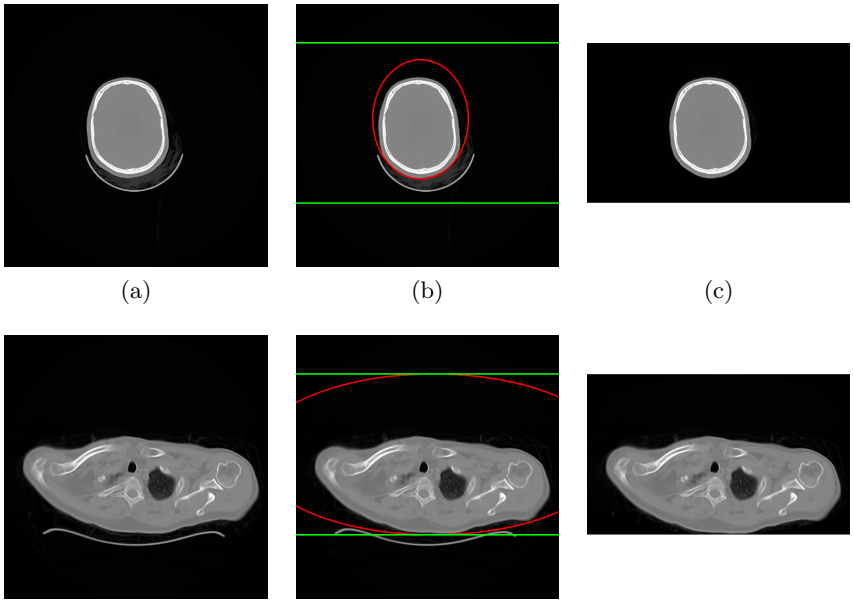


Figure 5.2: Regions of interest for two CT slices: (a), (d): original slices; (b), (e): slices with specified regions of interest (red) and common bounding box (green); (c), (f): slices after filling with background intensity and clipping to the size of the bounding box

registration application. The UNDO file format developed for this purpose is very simple. It consists of a header whose structure is shown in table 5.1, followed first by the voxel intensities of the reference and then those of the registered image.

The intensities are stored uncompressed as one byte per voxel from left to right, top to bottom, front to back. To simplify the code, the 32 bit integer values in the header are saved in the byte order of the CPU, making UNDO files generated on little and big endian machines incompatible.

5.2. Registration

As its name implies, the primary purpose of the registration application is to register the tomographic datasets stored in an UNDO file. Additionally, the program is able to conduct a number of predefined experiments. After a file has been loaded, the two images are roughly aligned by making their centers coincide and presented to the operator. The reference image is shown in blue, the registered in red. This is illustrated in figure 5.3, which is a screenshot of the user interface with an example UNDO file loaded.

The visualization is provided only to verify that the correct datasets have been selected and preprocessing results are satisfactory. No interaction with the images is possible or required. For registration, the only manual input available is the choice between a purely CPU based implementation and one of the GPU accelerated versions which are the topic

Count	Data Type	Fixed Contents	Description
4	`char`	'U' 'N' 'D' 'O'	File signature
1	`uint32`	3	File format version
3	`uint32`		Number of voxels in reference image per dimension
3	`float`		Spacing between voxels in reference image in millimeters per dimension
3	`uint32`		Number of voxels in registered image per dimension
3	`float`		Spacing between voxels in registered image in millimeters per dimension

Table 5.1: UNDO file header

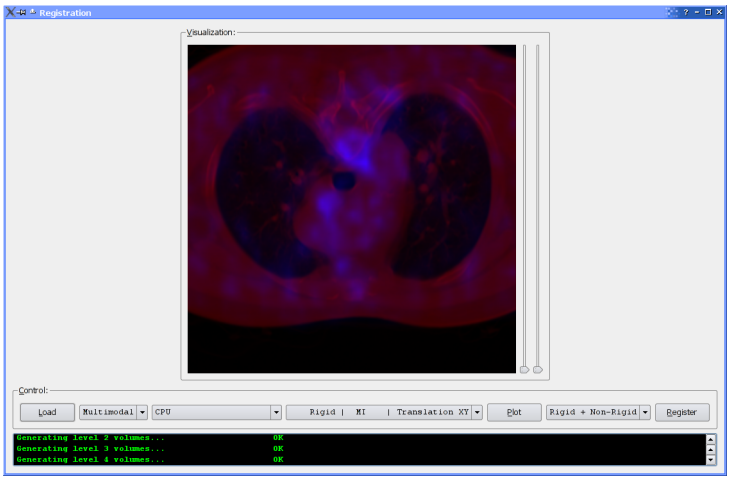


Figure 5.3: Screenshot of the registration application

of the next chapter. The entire registration process is fully automatic.

Although programmatically possible, the results are not presented in the application window. Instead, they are stored in a series of images, each of which shows a reference slice and the corresponding cut through the registered image. This solution was chosen because it allows for a more flexible evaluation of the final alignment. Using the tools of a general purpose image viewer, it is easily possible to zoom in on the images or view multiple slices side by side.

The additional experiments available are not meant to be run by end users. They are only meaningful in an analysis of the registration process such as that conducted in chapter 7. To isolate the effects of each registration pass, it is possible to leave out either the global or the local one. When aligning an image with itself, an artificial misalignment can be introduced to prevent the images from being perfectly registered right away. Finally, it is possible to not align the images at all but plot the mutual information or its derivatives over different values of the transformation parameters.

An important observation underlying the design of the application is that depending on whether registration or one of the other experiments is chosen, either mutual information or its derivatives are required, never both. The calculation of these quantities is therefore strictly split so that when one is needed, the other is not needlessly provided.

5.2.1. Class Structure

The main classes of the registration application and their interactions are illustrated in figure 5.4. The notation used in the code and this diagram differs in one aspect from that found in the previous chapters of this thesis. Instead of *global* and *local*, the two transformation types are referred to as *rigid* and *non-rigid*.

The main loop of the iterative registration process as well as methods controlling the additional experiments are located in the `Registration` class. The number of calculations performed here is minimal with the bulk of the computation effort factored out into other classes.

Raw image intensities to be used in the calculations are administered by the `Data` class. It is responsible for loading UNDO files and generating image pyramids of different resolutions.

Following with the observation made at the end of the previous section, separate classes are used to calculate the mutual information and its derivatives. They are further subdivided into those using a rigid or non-rigid transformation. Each of the four resulting classes, `RigidMI`, `RigidMIDerivative`, `NonRigidMI` and `NonRigidMIDerivative` is only an abstract stub. By exchanging their implementations, different calculation methods can be plugged in without affecting the remainder of the code. These key classes are highlighted by a yellow background in the UML diagram.

In the implementations presented in this chapter, the calculation effort is split between two types of classes. The first, derived from the abstract `Transformation`, are `TransformationRigid` and `TransformationNonRigid`. They provide access to image intensities and registered image gradients by encapsulating the transformation functions. The second type of classes use this input to calculate the mutual information and

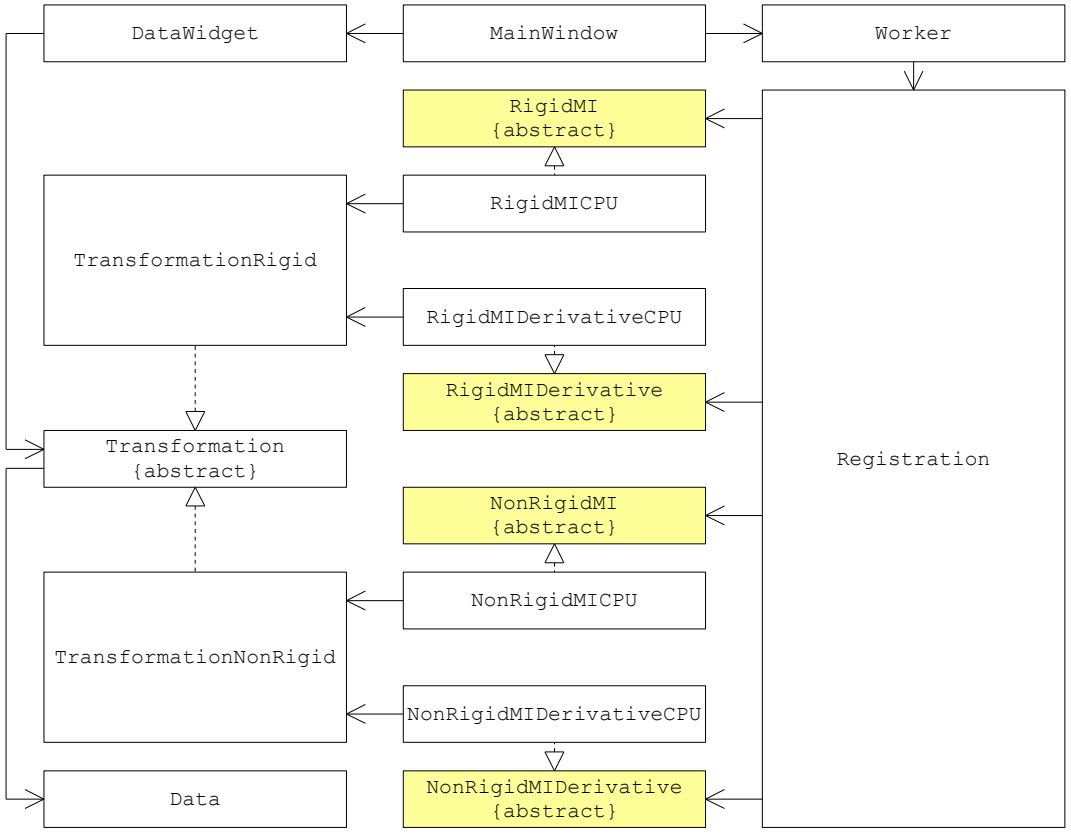


Figure 5.4: Class structure of the registration application

its derivatives. They are `RigidMICPU`, `RigidMIDerivativeCPU`, `NonRigidMICPU` and `NonRigidMIDerivativeCPU`.

The images are displayed on the screen by the `DataWidget`. Because this class accesses the intensities via an implementation of `Transformation`, any alignment of the images can be shown. This provision is used to visualize the initial rough alignment.

All widgets are embedded in an instance of the `MainWindow` class, which implements the application window and handles user interaction. It also starts the registration process or one of the experiments when requested by the user. The `Worker` class is intended to provide a thread that runs in parallel to the GUI so that calculations can be performed without interrupting screen updates. Unfortunately, it has been found that multithreading cannot be used with GPU accelerated code, most likely due to a bug in the X11 server. The `Worker` therefore only forwards requests to the `Registration` class and calculations do block the user interface.

All classes communicate with each other using the light-weight message passing mechanism of slots and signals provided by Qt. For example, when a new UNDO file is loaded, the `Data` class emits a signal and all other classes to whose operation this change is relevant are notified by automatically generated calls to their slot methods.

Although care has been taken to follow proper object-oriented programming techniques and any class may be instantiated multiple times, in practice this is only ever done for the transformations. Of all other classes, only a single object is constructed.

5.2.2. Implementation of Classes

The implementation of the registration system closely follows the methods developed in the previous chapters of this thesis. For the classes that perform large amounts of calculations, much effort has been put into computational efficiency and speed.

5.2.2.1. Registration

The `Registration` class contains the main loop of the iterative registration technique. It can be decomposed into the rigid and non-rigid passes, the first of which is shown in algorithm 5.1.

Algorithm 5.1 Main loop of registration process: Rigid pass

$T_{rigid} \leftarrow$ rough initial alignment
$\lambda_r = r_1$
$\lambda_t = r_2$
for $l = 4$ to 0 **do**
  for $s = 1$ to s_1 **do**
    $\mathbf{d}_r = \frac{d}{d(\alpha,\beta,\gamma)^T} MI^\star$
    $\mathbf{d}_t = \frac{d}{d\mathbf{t}} MI^\star$
    $(\alpha,\beta,\gamma)^T \leftarrow (\alpha,\beta,\gamma)^T + \lambda_r \mathbf{d}_r$
    $\mathbf{t} \leftarrow \mathbf{t} + \lambda_t \mathbf{d}_t$
    $\lambda_r = m_1 \lambda_r$
    $\lambda_t = m_2 \lambda_t$
  end for
end for

A rough initial alignment is obtained by making the centers of the two images coincide. Then, stochastic gradient ascent according to section 3.2 is performed. Following the multiresolution approach described in section 3.3, registration is conducted for pyramid levels of increasing size, beginning with the smallest images at $l = 4$ and finishing with the original datasets at $l = 0$.

Separate step sizes are used for rotation and translation so that a bias of the mutual information toward either of the two transformations can be counterbalanced. The initial rates are given by the constants r_1 and r_2. As explained in section 3.2, to guarantee that registration completes after a finite amount of time, the step sizes must be continually decreased. This is done by multiplying them with factors $0 < m_1, m_2 < 1$ in each iteration.

Instead of looping until the similarity is deemed satisfactory, as proposed by algorithm 1.1, a fixed number of iterations are performed. The reasons for this are twofold.

First, because mutual information is not an absolute measure, it is difficult to judge when it has reached a close to maximal value. Second, as the step sizes are decreased in each iteration, they at some point become so small that registration is effectively halted. The number of iterations s_1 is therefore chosen so that the loop ends when λ_t and λ_r are satisfactorily small.

Good values for the constants r_1, r_2, m_1, m_2 and s_1 need to be determined empirically. A set of values will be proposed in chapter 7.

The second pass is non-rigid registration. It is shown in algorithm 5.2.

Algorithm 5.2 Main loop of registration process: Non-rigid pass

$T_{non-rigid} \leftarrow T_{rigid}$
for all $\mathbf{q}_i$ **do**
  $\mathbf{q}_i \leftarrow \mathbf{0}$
end for
$\lambda \leftarrow r_3$
for $s = 1$ to s_2 **do**
  for $i = 0$ to $n - 1$ **do**
    $\mathbf{d}_i \leftarrow \frac{d}{d\mathbf{q}_i} MI^{\star}(\mathbf{q}_i)$
  end for
  for $i = 0$ to $n - 1$ **do**
    $\mathbf{q}_i \leftarrow \mathbf{q}_i + \lambda \mathbf{d}_i$
    for $k = 0$ to 2 **do**
      if $(\mathbf{q}_i)_k < -0.2r$ **then**
        $(\mathbf{q}_i)_k = -0.2r$
      else if $(\mathbf{q}_i)_k > 0.2r$ **then**
        $(\mathbf{q}_i)_k = 0.2r$
      end if
    end for
  end for
  $\lambda = m_3 \lambda$
end for

First, an initial non-rigid transformation identical to the rigid transformation found in the first pass is constructed. Rotation and translation are retained and the displacements at all nodes set to zero. Then, stochastic gradient descent is performed again to adjust their values. The number of iterations is s_2, the initial step size r_3 and the factor by which it is decreased each time, $0 < m_3 < 1$. Possible values for these constants may be found in chapter 7.

In every iteration, the derivatives of the local mutual information as defined in section 4.2.5.3 are calculated at all nodes and the displacements adjusted accordingly. Each component $(\mathbf{q}_i)_k$ of a displacement vector is then clipped to the interval $[-0.2r, 0.2r]$, respecting the search space set forth in section 4.2.6.

When conducting an experiment that does not align the two images, the transformation parameters are set to a series of predefined values, the mutual information or its

derivatives are calculated for each setting and output into a text file. This file can then be used to generate plots such as those in chapter 7.

5.2.2.2. Data

The `Data` class provides access to raw image intensities. Its two duties are loading UNDO files and generating image pyramids. When loading a file, a number of checks are performed to verify that it adheres to the specifications of section 5.1.3, the header values are sane and all voxel intensities are present. The spacings in millimeters between the voxels of the two images read from the header are stored in the vectors $\mathbf{y}_0$ and $\mathbf{y}'_0$.

As a result of clipping, the two datasets can be cuboids of arbitrary dimensions. To make them easier to handle, they are embedded in three-dimensional arrays whose sizes in all dimensions are powers of two. Because only the cuboids contain relevant data, they are referred to as the *regions of interest* while the entire arrays are known as *volumes*. Each region of interest is centered in the smallest volume possible. Any remaining voxels are set to the background intensity of zero. The offset in voxels of the region of interest within the volume is $\mathbf{o}_0$ for the reference and $\mathbf{o}'_0$ for the registered image.

Image pyramids of five levels each are then constructed for both datasets using the method described in section 3.3. The entire volumes are smoothed and resized. Because they have power of two dimensions, the data at every level again has sizes that are powers of two, avoiding the need to consider the case where an image dimension is odd and cannot precisely be halved. As at subsequent levels, each voxel represents a larger chunk of the original image, the offsets of the regions of interest decrease while the spacings between the voxels increase. For level l of the pyramid, they are recursively given as:

$$\mathbf{o}_l = \frac{1}{2}\mathbf{o}_{l-1} \qquad \mathbf{o}'_l = \frac{1}{2}\mathbf{o}_{l-1} \tag{5.1}$$

$$\mathbf{y}_l = 2\mathbf{y}_{l-1} \qquad \mathbf{y}'_l = 2\mathbf{y}'_{l-1} \tag{5.2}$$

5.2.2.3. TransformationRigid

This class implements the rigid transformation function. Given $\mathbf{x}_u \in \mathbb{R}^3$, it is able to return $u(\mathbf{x}_u)$ and $v(T(\mathbf{x}_u))$, the intensities of the reference and registered images at that point. Additionally, the derivative of the transformation function may be queried.

The transformation is calculated directly as defined by equation (4.12), with $\mathbf{x}_u$ extended to homogenous coordinates by adding a fourth component $(\mathbf{x}_u)_w = 1$:

$$\mathbf{x}_v = \mathbf{M}_T \mathbf{x}_u \tag{5.3}$$

To simplify the implementation, the helper classes `HMatrix` and `HVector` have been developed. The first represents a 4×4 matrix while the second is a four component homogenous coordinate vector. The matrix class features a multiplication operator that performs the required matrix-vector multiplication. According to equation (4.52), the derivative of the transformation function with respect to each parameter p can be expressed as a 4×4 matrix $M_{T,p}$. The derivative matrices for the six parameters are also

stored and returned as `HMatrix` objects. Whenever any of the parameters is changed, the transformation matrix and its six derivatives are updated.

When the intensity of the reference image at a point $\mathbf{x}_u$ is to be looked up, the implicit conversion from millimeters to voxel coordinates introduced in section 1.3.2 must be performed. If level l of the image pyramids is currently being used, the calculations for the kth component of $\mathbf{x}_u$ are:

$$\left(\mathbf{x}_u''\right)_k = \frac{\left(\mathbf{x}_u\right)_k}{\left(\mathbf{y}_l\right)_k} + \left(\mathbf{o}_l\right)_k - 0.5 \tag{5.4}$$

The result is a vector $\mathbf{x}_u''$ of zero based indices into the volume of voxel intensities generated by the `Data` class. The division by the voxel spacing $\left(\mathbf{y}_l\right)_k$ rescales the coordinates so that they are expressed in voxels, not millimeters. A translation by $\left(\mathbf{o}_l\right)_k$ is necessary because the reference image is offset by $\mathbf{o}_l$ in the intensity volume. The final translation of -0.5 is required as each array position corresponds to a voxel center while the origin of the reference coordinate system is the corner of a voxel.

If the three components of $\mathbf{x}_u''$ are integers, they may directly be used as indices into the reference volume. Indices not within the array bounds indicate that $\mathbf{x}_u$ lies outside the image and an intensity of zero should be assumed. When any of the components has a fractional part, $\mathbf{x}_u$ points to a location between voxel centers. In this case, the intensity must be estimated based on those of surrounding voxels. The method used is trilinear interpolation.

To simplify the description, a few notations are introduced. $\mathbf{x}$ is $\mathbf{x}_u''$ with each component rounded down to the nearest integer:

$$x_k = \left\lfloor \left(\mathbf{x}_u''\right)_k \right\rfloor \tag{5.5}$$

Weighting factors τ_k are given by the fractional parts of $\mathbf{x}_u''$:

$$\boldsymbol{\tau} = \mathbf{x}_u'' - \mathbf{x} \tag{5.6}$$

With u'' the volume of reference image intensities, the estimated intensity at the desired position $\mathbf{x}_u''$ then is:

$$\begin{aligned}
&\left(1-\tau_1\right)\left(1-\tau_2\right)\left(1-\tau_3\right)u''\left[\mathbf{x}\right] \\
&+\left(1-\tau_1\right)\left(1-\tau_2\right)\tau_3 u''\left[\mathbf{x}+\mathbf{e}_3\right] \\
&+\left(1-\tau_1\right)\tau_2\left(1-\tau_3\right)u''\left[\mathbf{x}+\mathbf{e}_2\right] \\
&+\left(1-\tau_1\right)\tau_2\tau_3 u''\left[\mathbf{x}+\mathbf{e}_2+\mathbf{e}_3\right] \\
&+\tau_1\left(1-\tau_2\right)\left(1-\tau_3\right)u''\left[\mathbf{x}+\mathbf{e}_1\right] \\
&+\tau_1\left(1-\tau_2\right)\tau_3 u''\left[\mathbf{x}+\mathbf{e}_1+\mathbf{e}_3\right] \\
&+\tau_1\tau_2\left(1-\tau_3\right)u''\left[\mathbf{x}+\mathbf{e}_1+\mathbf{e}_2\right] \\
&+\tau_1\tau_2\tau_3 u''\left[\mathbf{x}+\mathbf{e}_1+\mathbf{e}_2+\mathbf{e}_3\right]
\end{aligned} \tag{5.7}$$

Equation (5.7) calculates the intensity as the weighted sum of those of the eight surrounding voxels. The weights are the distances between the centers of these voxels

and $\mathbf{x}''_u$ in each coordinate. If any of the eight voxels lies outside the array, an intensity of zero should be substituted for it again.

When the intensity of the registered image is to be determined, $\mathbf{x}_u$ is first transformed into $\mathbf{x}_v$ by executing the matrix multiplication in equation (5.3). Then, operations analogous to those used for the reference image are performed with $\mathbf{y}'_l$, $\mathbf{o}'_l$ and the reference volume v'' substituted for $\mathbf{y}_l$, $\mathbf{o}'_l$ and u''.

Additional methods are provided which allow parts of the coordinate transformation to be omitted or altered. If a reference image intensity is being looked up and the indices $\mathbf{x}''_u$ are already known, a conversion from $\mathbf{x}_u$ is not necessary. When estimating the registered image gradient, it is useful to be able to add an arbitrary offset to $\mathbf{x}_v$ after the transformation has been applied but before any other operations have taken place.

5.2.2.4. RigidMIDerivativeCPU

The purpose of this class is to estimate the derivatives of mutual information with respect to the parameters of the rigid transformation function. The transformation itself is represented by an instance of `TransformationRigid`. What remains to be done is to evaluate equation (2.46) for each parameter p:

$$\frac{d}{dp} MI^{\star}(X,Y) = \frac{1}{N_B} \sum_{\mathbf{x}_i \in S_B} \sum_{\mathbf{x}_j \in S_A} \left[W_Y(v_i, v_j) \frac{1}{\sigma_Y^2} - W_{X,Y}(\mathbf{w}_i, \mathbf{w}_j) \frac{1}{\sigma_{Y,Y}^2} \right] (v_i - v_j) \left(\frac{d}{dp} v_i - \frac{d}{dp} v_j \right) \tag{5.8}$$

The definitions used in this expression are provided by equations (2.43) and (2.44):

$$v_i = v(T(\mathbf{x}_i)) \qquad \mathbf{w}_i = \begin{pmatrix} u(\mathbf{x}_i) \\ v(T(\mathbf{x}_i)) \end{pmatrix} \tag{5.9}$$

$$W_Y(v_i, v_j) = \frac{f_{N_{0,\sigma_Y^2}}(v_i - v_j)}{\sum_{\mathbf{x}_k \in S_A} f_{N_{0,\sigma_Y^2}}(v_i - v_k)} \qquad W_{X,Y}(\mathbf{w}_i, \mathbf{w}_j) = \frac{f_{N_{\mathbf{0},\boldsymbol{\Sigma}}}(\mathbf{w}_i - \mathbf{w}_j)}{\sum_{\mathbf{x}_k \in S_A} f_{N_{\mathbf{0},\boldsymbol{\Sigma}}}(\mathbf{w}_i - \mathbf{w}_k)} \tag{5.10}$$

At the beginning of the registration process, sampling points $\mathbf{x}_i$ are generated and stored in S_A and S_B. The reference image intensities at these points can be retrieved without the need for trilinear interpolation if each $\mathbf{x}_i$ coincides with the center of a voxel. This is accomplished by first choosing random array positions $\mathbf{x}''_i$ in the reference volume region of interest and only then converting their coordinates to millimeters, performing the inverse of the transformation in equation (5.4). Each $u_i = u(\mathbf{x}_i)$ may then be looked up directly in the reference volume as $u_i = u''[\mathbf{x}''_i]$.

The key to efficiently evaluating equation (5.8) is precalculation. Whenever a value can be reused, it should be calculated only once and stored for later reference. Because the sampling points do not change over the course of registration, the u_i need only to be determined once at the beginning of the registration process. Precalculation is also possible for the values of $f_{N_{0,\sigma_Y^2}}$ and $f_{N_{\mathbf{0},\boldsymbol{\Sigma}}}$. The values of the first function are stored in

an array G_1 and those of the second, in G_2. Each evaluation of a density function can then be replaced by an array look-up:

$$f_{N_{0,\sigma_Y^2}}(x) = G_1[|x|] \tag{5.11}$$

$$f_{N_{\mathbf{0},\mathbf{\Sigma}}}(\mathbf{x}) = G_2[|x_1|, |x_2|] \tag{5.12}$$

Absolute values may be used because according to their definitions in equations (2.33) and (2.34), Gaussian density functions with a mean of zero are symmetrical. The values for which these functions are evaluated in equation (5.10) are the differences of image intensities. Since only integer intensities in the range $[0, 255]$ are used, $|x|$ is also an integer in $[0, 255]$ while $\mathbf{x}$ is a pair of integers from $[0, 255] \times [0, 255]$. The array G_1 thus has 256 elements while G_2 has 256^2.

After these precalculations have been performed, the mutual information derivatives can repeatedly be evaluated for different alignments. The first step in such an evaluation is to obtain all required values from the `TransformationRigid` class. For every sampling point $\mathbf{x}_i$ in S_A or S_B, the registered image intensity v_i and its derivatives $\frac{d}{dp}v_i$ with respect to all transformation parameters p are needed. While the v_i are directly provided by the transformation class, the derivatives need to be calculated using equation (4.53):

$$\frac{d}{dp}v(T(\mathbf{x}_i)) = \operatorname{grad} v(T(\mathbf{x}_i))\,\mathbf{M}_{T,p}\mathbf{x}_i \tag{5.13}$$

The six matrices $\mathbf{M}_{T,p}$ can be retrieved from the transformation class. To estimate the gradient, equation (4.31) is employed. With $\mathbf{x}_v = T(\mathbf{x}_i)$, it is given as:

$$\operatorname{grad} v(\mathbf{x}_v) \approx (v(\mathbf{x}_v + \mathbf{e}_1) - v(\mathbf{x}_v), v(\mathbf{x}_v + \mathbf{e}_2) - v(\mathbf{x}_v), v(\mathbf{x}_v + \mathbf{e}_3) - v(\mathbf{x}_v), 0) \tag{5.14}$$

Since $v(\mathbf{x}_v) = v(T(\mathbf{x}_i)) = v_i$, only three additional intensities are required for each $\mathbf{x}_i$. They are also obtained from the `TransformationRigid` class using the convenience method that allows an offset $\mathbf{e}_k$ to be added after $\mathbf{x}_i$ has been transformed into $\mathbf{x}_v$.

Next, the weighting factors W_Y and $W_{X,Y}$ are calculated for all pairs of points $(\mathbf{x}_i, \mathbf{x}_j) \in S_B \times S_A$ according to equation (5.10). Because the factors for each $\mathbf{x}_i$ share a common denominator, it only needs to be evaluated once for all of them. Due to the limited precision of a computer, it is possible for the denominator to become zero. This occurs when the values of the Gaussian density function summed are all so small that they get aliased to this value. The weighting factors in this case are set to zero so that the affected pairs of points effectively drop out of the calculation and do not corrupt the result.

Equation (5.8) can now be efficiently evaluated by executing the nested summation and using the precalculated values of v_i, $\mathbf{w}_i = (u_i, v_i)^T$, $\frac{d}{dp}v_i$, W_Y and $W_{X,Y}$. The calculation is performed for all six parameters in parallel, determining the part of the addend that does not depend of p once and multiplying it with $\left(\frac{d}{dp}v_i - \frac{d}{dp}v_j\right)$ for each parameter.

The class as implemented is aware of all events that may invalidate precalculated values and intiates new precalculations whenever required. For example, when the image

pyramid level is changed, new sampling points $\mathbf{x}_i$ must be chosen and the reference image intensities u_i looked up again. If new variances are set for the Gaussian density functions, the arrays G_1 and G_2 need to be recalculated.

5.2.2.5. RigidMICPU

This class estimates the mutual information for the alignment of the two images represented by an instance of `TransformationRigid`. It is not actually needed during registration but only when plotting mutual information over different settings of the transformation parameters. A description of its efficient implementation is provided here only for completeness. According to equation (2.41), mutual information may be estimated as:

$$\begin{aligned} MI^{\star}(X,Y) = & \\ & -\frac{1}{N_B}\sum_{\mathbf{x}_i\in S_B}\log\frac{1}{N_A}\sum_{\mathbf{x}_j\in S_A} f_{N_{0,\sigma_X^2}}\left(u(\mathbf{x}_i)-u(\mathbf{x}_j)\right) \\ & -\frac{1}{N_B}\sum_{\mathbf{x}_i\in S_B}\log\frac{1}{N_A}\sum_{\mathbf{x}_j\in S_A} f_{N_{0,\sigma_Y^2}}\left(v(T(\mathbf{x}_i))-v(T(\mathbf{x}_j))\right) \\ & +\frac{1}{N_B}\sum_{\mathbf{x}_i\in S_B}\log\frac{1}{N_A}\sum_{\mathbf{x}_j\in S_A} f_{N_{\mathbf{0},\mathbf{\Sigma}}}\left(\begin{pmatrix}u(\mathbf{x}_i)\\ v(T(\mathbf{x}_i))\end{pmatrix}-\begin{pmatrix}u(\mathbf{x}_j)\\ v(T(\mathbf{x}_j))\end{pmatrix}\right) \end{aligned} \tag{5.15}$$

Sampling points $\mathbf{x}_i$ and corresponding reference volume indices $\mathbf{x}_i''$ are determined in the same way as in section 5.2.2.4. Also, the same precalculations are used for the reference image intensities $u(\mathbf{x}_i)$ and the values of the Gaussian density functions.

When calculating the mutual information for a new alignment, the only values that need to be obtained from the `TransformationRigid` class are the registered image intensities $v(T(\mathbf{x}_i))$ for all points $\mathbf{x}_i$ in S_A or S_B. The mutual information may then be determined by evaluating the three rows on the right hand side of equation (5.15). They correspond to $H[X]$, $H[Y]$ and $H[X,Y]$, the entropies of the two images and their joint entropy.

Because all intensities are integers in the range $[0,255]$, u and v can take on only 256 distinct values so that when evaluating $H[X]$ or $H[Y]$, only $256^2 = 65536$ different addends are possible. Instead of iterating over all pairs $(\mathbf{x}_i,\mathbf{x}_j)\in S_B\times S_A$, it is therefore sufficient to count how many times each intensity occurs in the samples, iterate over the 65536 intensity combinations and multiply each addend with the number of times this intensity combination occurs.

With N_A and N_B the sample sizes, the complexity of the calculation is reduced from $O(N_AN_B)$ to $O(N_A+N_B)$ as every sample needs to be traversed only once to count the intensities. However, a nested sum with a constant number of 65536 addends still needs to be evaluated so that this approach is only more efficient if $N_AN_B >> 65536$. As will be seen in chapter 7, this actually is the case for tomographic images.

For the joint entropy $H[X,Y]$, no such acceleration is possible. The number of possible addends is 256^4 so that it is far more efficient to directly evaluate the nested sum as given in equation (5.15).

There are two more important points to be considered. First, the values of the inner sums in all three rows may get arbitrarily close to zero. This is an artifact of the small random samples used that occurs when $\mathbf{x}_i \in S_B$ has an intensity whose probability, when estimated using sample S_A, is almost zero. The logarithm in such a case approaches negative infinity so that a single rogue sample could lead to a mutual information of ∞ or $-\infty$. Due to the aliasing described in the previous section, it is even possible that the value of an inner sum becomes precisely zero, for which the logarithm cannot be evaluated. To avoid these problems, if the value of any inner sum is very small (the threshold used in the code is 10^{-9}), the logarithm is not directly evaluated but the definition $\log 0 = 0$ from section 2.6 is used instead.

The second important point is the observation that $H[X]$, the reference image entropy, does not change over the course of the registration. It can therefore be completely precalculated so that for each new alignment, only $H[Y]$ and $H[X, Y]$ must be determined. As was the case with `RigidMIDerivativeCPU`, this class is also aware of all events that could invalidate precalculated values and initiates new precalculations when required.

5.2.2.6. TransformationNonRigid

The `TransformationNonRigid` class implements the non-rigid transformation function of equation (4.83):

$$\mathbf{x}_v = \mathbf{M}_T \begin{pmatrix} \mathbf{x}_u + \sum_{j=1}^{n} \boldsymbol{\alpha}_j R_a \left(\|\mathbf{x}_u - \mathbf{p}_j\| \right) \\ 1 \end{pmatrix} \tag{5.16}$$

As direct evaluation of this function is prohibitively expensive, the optimizations proposed in section 4.2.5 are employed. According to sections 4.2.5.2 and 4.2.5.3, sampling is performed locally for every node $\mathbf{p}_i$. In section 4.2.5.4, it is suggested that the same pattern of sampling points $\mathbf{x}'_u$ be used for each such node. The `TransformationNonRigid` class stores this pattern as well as the node positions and handles the entire sampling process. Methods are provided which with a single call return the reference image intensities, registered image intensities and gradients or transformation function derivatives for all sampling points $\mathbf{x}'_u$ in the pattern centered around all nodes $\mathbf{p}_i$.

Before sampling may commence, the nodes need to be distributed. This is done as described in section 4.2.3. A regular grid of up to $8 \times 8 \times 8$ nodes is placed over the image, for each node the reference image intensity is evaluated at 257 points in the vicinity and the node kept only if enough of these intensities are above a background noise level. The grid is expressed as a three-dimensional array GN where for each grid point, the node number or -1 is stored if no node is present. To simplify further calculations, the array is padded with three additional grid points before and after the actual grid in each dimension, all of which lie outside the image and are assigned the value -1. Another array NG stores the corresponding grid point for each node. The position in the reference image at which the grid begins is recorded in the variable S.

Next, the matrix $\mathbf{C}$ is constructed using the efficient method of section 4.2.5.1. First, the required values of the radial basis functions are precalculated and all matrix elements set to zero. Then, an iteration over the n nodes is performed. For each node $\mathbf{p}_i$, its

position in the grid is looked up in NG and the nodes present at the nineteen relevant surrounding grid points are obtained from GN. Whenever a node $\mathbf{p}_j$ is found at one of these points, the matrix element for the node pair $(\mathbf{p}_i, \mathbf{p}_j)$ is set to the appropriate value of the radial basis function. Because GN is padded with extra grid points at all sides, the surrounding points are guaranteed to exist for every node $\mathbf{p}_i$ and GN may directly be accessed without any further checks.

To invert $\mathbf{C}$, a method from the LinAlg[3] package is used. It is not optimized for sparse matrices and as such, does not achieve optimal computational performance. However, because the inversion needs to be performed only once and at a maximum size of $8^3 \times 8^3$, the matrix is relatively small, this is deemed acceptable. After the inversion, the elements of $\mathbf{C}^{-1}$ are stored in an array W.

How the transformation function may efficiently be evaluated for the generated node arrangement was shown in section 4.2.5.4. For each $\mathbf{x}'_u$, the offset used to locate the relevant $3 \times 3 \times 3$ sub-grid and the values of the radial basis functions for nodes located at these grid points are precalculated. The only deviation from the method proposed earlier is that the offset used points to a corner of the sub-grid, not its center. The offsets are stored in an array F, the radial basis functions values in R.

The matrix $\mathbf{M}_T$ represents the rigid component of the transformation and remains constant throughout the registration process. Its elements are precalculated from the six parameters that affect the rigid transformation using equations (4.13) to (4.28). The weights $\boldsymbol{\alpha}_i$ are not constant and must be recalculated whenever the displacements $\mathbf{q}_i$ are changed. With the elements of $\mathbf{C}^{-1}$ stored in the array W, this may quickly be done by evaluating equation (4.81):

$$\mathbf{A} = \mathbf{C}^{-1}\mathbf{Q} \tag{5.17}$$

The elements of $\mathbf{Q}$ are set as specified in equation (4.80) but additionally negated, so that $Q_{ik} = -(\mathbf{q}_i)_k$. This is done to give the displacement parameters more intuitive directions. For example, a $(\mathbf{q}_i)_1 = 10$ results in a local translation of the registered image at node $\mathbf{p}_i$ by 10 millimeters to the right, not to the left as it otherwise would.

After all precalculations have been completed, efficient sampling is possible. Regardless of which quantity is to be sampled, an iteration over all nodes $\mathbf{p}_i$ and all sampling points $\mathbf{x}'_u$ is performed. The resulting sampling positions are obtained from equation (4.103):

$$\mathbf{x}_u = \mathbf{x}'_u + \mathbf{p}_i \tag{5.18}$$

The positions of the nodes are actually calculated on the fly from the starting point of the grid S and their grid locations stored in the array NG. If reference image intensities $u(\mathbf{x}_u)$ are desired, they are looked up at the points $\mathbf{x}_u$ in the reference volume. The implicit conversion to voxel coordinates and trilinear interpolation are implemented in the same way as described in section 5.2.2.3 for the `TransformationRigid` class.

To calculate registered image intensities $v(T(\mathbf{x}_u)) = v(\mathbf{x}_v)$, each $\mathbf{x}_u$ must first be transformed into the corresponding $\mathbf{x}_v$ by evaluating the transformation function of equation (5.16). Using the precalculated values and the method described in section 4.2.5.4, this is achieved by a series of array look-ups and simple arithmetic operations.

[3] `http://okmij.org/ftp/packages.html`

First, a corner of the relevant sub-grid is located as $NG[\mathbf{p}_i] + F[\mathbf{x}'_u]$. Then, the sub-grid is traversed in the array GN and whenever a node $\mathbf{p}_j$ is found, $\boldsymbol{\alpha}_j$ is multiplied with the precalculated radial basis function value in R and added to $\mathbf{x}_u$. The resulting point is extended to homogenous coordinates and the matrix-vector multiplication executed, yielding $\mathbf{x}_v$. Because GN is padded with extra grid points at all sides, all 27 relevant positions are guaranteed to exist for each $\mathbf{x}_u$, allowing the array to be accessed without any further checks.

The registered image intensity at $\mathbf{x}_v$ is looked up using the same implicit conversion and trilinear interpolation as in section 5.2.2.3. If the registered image gradient is also needed, it is estimated using equation (5.14):

$$\operatorname{grad} v(\mathbf{x}_v) \approx (v(\mathbf{x}_v + \mathbf{e}_1) - v(\mathbf{x}_v), v(\mathbf{x}_v + \mathbf{e}_2) - v(\mathbf{x}_v), v(\mathbf{x}_v + \mathbf{e}_3) - v(\mathbf{x}_v), 0) \quad (5.19)$$

This is most efficiently done together with the calculation of $v(\mathbf{x}_v)$ as the transformation from $\mathbf{x}_u$ to $\mathbf{x}_v$ needs to be performed only once. The `TransformationNonRigid` class provides two separate methods, one of which samples only the registered image intensities and the other both the intensities and the gradients, depending on what is required in subsequent calculations.

The final quantities for which a sampling method is made available are the derivatives of the transformation function with respect to the displacements $\mathbf{q}_i$. The definition from equation (4.99) is followed with additional negations applied due to the directions of the displacements being inverted:

$$\frac{d}{d\mathbf{q}_i} T(\mathbf{x}_u) = \mathbf{M}_T \sum_{j=1}^{n} - \left(\mathbf{C}^{-1}\right)_{ji} R_a(\|\mathbf{x}_u - \mathbf{p}_j\|) \quad (5.20)$$

The same methods as those used to accelerate the calculation of the transformation function are applied to its derivatives. For each combination of a node $\mathbf{p}_i$ and a sampling point $\mathbf{x}'_u$, the derivative is initially set to zero, $\mathbf{x}_u$ is determined and the 27 relevant grid points are traversed in GN. Whenever a node $\mathbf{p}_j$ is found, $\left(\mathbf{C}^{-1}\right)_{ji}$ is multiplied with the precalculated radial basis function value from R and subtracted from the derivative. The multiplication with the matrix $\mathbf{M}_T$ is omitted. This is based on the observation that the matrix is constant for all sampling points. Instead of returning 4×4 derivatives for each point, it is more efficient to calculate only the scalar factor and perform the multiplication on the fly when it is required.

5.2.2.7. NonRigidMIDerivativeCPU

This class estimates the derivatives of mutual information with respect to the parameter vectors $\mathbf{q}_i$ of the non-rigid transformation function. As proposed in section 4.2.5.3, the derivatives are calculated locally using the the same pattern of sampling points $\mathbf{x}'_u$ centered around each node $\mathbf{p}_i$. The transformation is represented by an instance of `TransformationNonRigid`. At the beginning of the second registration pass, a random pattern of points $\mathbf{x}'_u$ is generated for the samples S_A and S_B and forwarded to the transformation class. According to section 4.2.5.3, the points should be scattered in a

sphere of radius d, the spacing of the grid. To simplify the code, a radius of one is used instead and the coordinates are rescaled after they have been forwarded.

Values that do not change over the course of registration are precalculated. The number of nodes n and the rigid transformation matrix $\mathbf{M}_T$ are retrieved from the `TransformationNonRigid` object. Also queried are the reference image intensities and transformation function derivatives at the sampling points for all nodes $\mathbf{p}_i$. Finally, the required values of the Gaussian density functions are calculated and stored into the arrays G_1 and G_2.

After these precalculations, the derivatives may repeatedly be evaluated for different alignments. First, the registered image intensities and gradients are retrieved for all sampling points at all nodes. Then, the derivative with respect to each $\mathbf{q}_i$ is calculated from the samples drawn locally around $\mathbf{p}_i$ using the same steps as those described in section 5.2.2.4 for `RigidMIDerivativeCPU`. The only difference is that the $\frac{d}{dp}v_i$ required by equation (5.8) are obtained by evaluating the negated equation (4.99):

$$\frac{d}{d\mathbf{q}_i} v\left(T\left(\mathbf{x}_u\right)\right) = \operatorname{grad} v\left(T\left(\mathbf{x}_u\right)\right) \mathbf{M}_T \sum_{j=1}^{n} - \left(\mathbf{C}^{-1}\right)_{ji} R_a\left(\left\|\mathbf{x}_u - \mathbf{p}_j\right\|\right) \tag{5.21}$$

The gradient has been sampled and the other values precalculated so that only the multiplications need to be performed. Because the fourth component of the gradient is zero, only its first three components and a 3×3 sub-matrix of $\mathbf{M}_T$ are used. As this equation expresses the derivative with respect to the entire vector $\mathbf{q}_i$, the derivatives for all of its components are obtained together.

5.2.2.8. NonRigidMICPU

The purpose of this class is to calculate the local mutual information at each node $\mathbf{p}_i$ for the alignment of the two images given by an instance of `TransformationNonRigid`. Its implementation closely follows that of the `RigidMICPU` class described in section 5.2.2.5. There are only two significant differences. The first is that the sampling points $\mathbf{x}'_u$ are randomly scattered in a sphere in the same way as in section 5.2.2.7. The other difference is that the entire precalculation and calculation process is repeated for each node $\mathbf{p}_i$ using the local samples obtained from the transformation class.

6. GPU Acceleration

Described in this chapter is a technique by which the computational efficiency of the registration system can be taken beyond what is possible even with the most highly optimized implementation requiring the least number of operations. The calculations are moved from the CPU to a GPU, or graphics processing unit. This is the chip at the heart of every modern graphics card. With the demand for ever more realistic graphics, the performance and versatility of GPUs have steadily increased to the point where for some calculations, they can far outperform much more expensive CPUs. However, as GPUs are designed with graphics in mind, the paradigms and programming models they employ greatly differ from those of traditional CPUs. The algorithms of the registration system therefore need to be adapted before they can benefit from the speed of a GPU.

This chapter covers three topic areas. First, the relevant components of a GPU are presented and the elements that can be reprogrammed introduced. Next, the use of a graphics card for calculations other than the generation of moving images is explained. Finally, these principles are applied to the registration system by describing GPU based implementations of the four classes highlighted in figure 5.4.

6.1. GPU Programming

The two main APIs for controlling graphics processing units are Microsoft's Direct3D [Mic06] and the vendor independent OpenGL [SA04]. Because it is cross-platform, OpenGL 2.0 is used in this thesis. A strength of this API is that it allows independent extensions to be developed. Each vendor can propose a new extension by implementing it in their hardware and drivers. If a proposal is popular and other manufacturers start supporting it, the extension is eventually made official.

An acronym prefixed to its name indicates the status of an extension. Initially, a reference to the original developer is used, such as `ATI`, `NV` or `SGI`. When multiple vendors agree on a proposal, its name begins with `EXT`. If the extension gets approved by the OpenGL Architecture Review Board, the prefix is changed to `ARB`. Even when it has the blessing of the ARB, an extension still is not part of the core API and is not guaranteed to be supported by all GPUs on all platforms. In this thesis, the GLEW[1] library is used to check for the presence of the required extensions. As the development of an alternative code path would have been very time consuming, GPU based registration is aborted and an error message produced should any of the expected extensions be missing. The code has been written and tested using two different GPUs made by NVIDIA and should work with all recent cards based on chips by that manufacturer.

[1] `http://glew.sourceforge.net/`

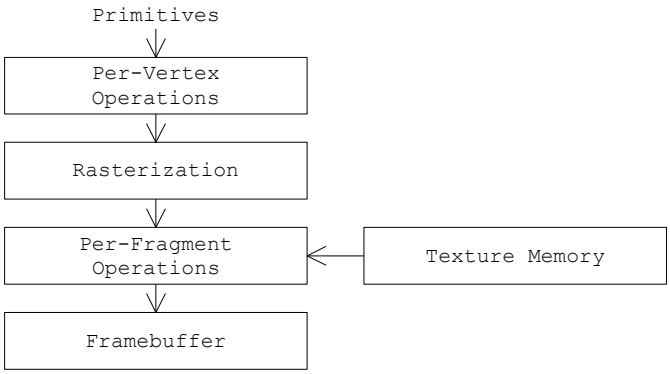


Figure 6.1: Simplified OpenGL rendering pipeline

6.1.1. Rendering Pipeline

All processing on a GPU occurs as part of the rendering pipeline. In OpenGL, the pipeline is modeled as a finite state machine. A simplified view of its structure is presented in figure 6.1. Commands are issued as C function calls and may alter the state of the pipeline or pass primitives through it.

Each primitive is described by a series of vertices and associated properties. At the first stage of the pipeline, the vertices are processed. The operations performed include perspective transformation, lighting and clipping to the visible area. The second stage is rasterization, where for each pixel covered by the primitive, a fragment is generated. At the next stage, the color of each fragment is determined from values interpolated between the vertices. The interpolated value can directly be the color, a set of coordinates used to look it up in a texture map or both with the color obtained as a mixture of different sources. Finally, the fragments are blended with the contents of the framebuffer. This may be a window on the screen or an off-screen buffer that will be displayed later.

While the operations to be performed at each stage may be selected and their parameters adjusted using the appropriate commands, the choices are limited. Only functions explicitly listed in the OpenGL specification or a supported extension are available. More flexibility is offered by a programmable GPU. Here, the fixed functionality at some stages of the pipeline can be disabled and replaced by arbitrary programs, known as *shaders*. The two types of shaders and their positions in the rendering pipeline are shown in figure 6.2.

Despite a large amount of freedom in how vertex and fragment shaders are written, their roles in the pipeline are fixed. The first type of shader is called for each vertex. It is expected to calculate the final position in the frame buffer as well as any lighting effects and the associated texture coordinates. The fragment shader is executed for every fragment. Its duty is to determine a color from the values interpolated between the vertices by the rasterizer. Before the development of shaders is described in more detail, the other elements of the rendering pipeline relevant to their operation are addressed in the next two sections.

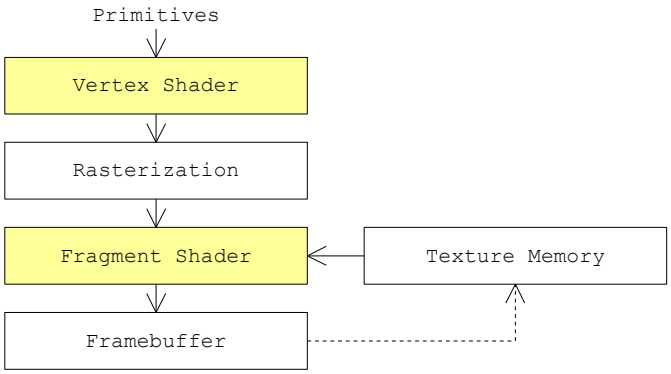


Figure 6.2: Simplified programmable OpenGL rendering pipeline

6.1.2. Texture Maps

The fundamental data structures used by the rendering pipeline are texture maps. OpenGL offers 1D, 2D and 3D textures which correspond to one-, two- or three-dimensional arrays in the memory of the graphics card. The texture elements, known as *texels*, can be encoded in a variety of formats. Each format is defined by the number of components per texel, ranging from one to four, and their data types. A single value is sufficient when the texture expresses only depth, luminosity or transparency. To record colors, three components are used and a fourth is added if an alpha transparency channel is desired. Two values per texel are uncommon but may for example describe luminosity and transparency in a grayscale texture map.

Several formats exist in which the components are allocated unequal numbers of bits. In other formats, the same data type is used for each component. Commonly used options are a byte or a floating point number. Besides IEEE 754 single precision [IEE85], a half precision type is available. It is inspired by the IEEE standard but instead of 32, uses only 16 bits to record a number with 10 mantissa and 5 exponent bits. Floating point textures are not actually part of the OpenGL standard. They are made available by several competing extensions, each of which allows different combinations of texture dimensionality, number of components per texel and data type. To gain access to a wide selection of texture types, the extensions proposed by ATI, NVIDIA and the ARB are all used in the code written for this thesis.

As texture maps are defined in the context of graphics, the addressing of their elements differs greatly from that used for traditional arrays, as illustrated in figure 6.3 (a) and (b). In each dimension, the coordinates at the two edges of a texture are 0 and 1 with the texels in between addressed by floating point numbers in that range. If there are n texels, the center of the first has the coordinate $\frac{1}{2n}$ and each subsequent texel is reached by adding $\frac{1}{n}$. For example, if $n = 4$, the elements of the texture are found at 0.125, 0.375, 0.625 and 0.875.

To overcome the awkwardness of this addressing scheme, the `Rect` texture format is provided. It defines a two-dimensional texture map where the coordinates at one corner

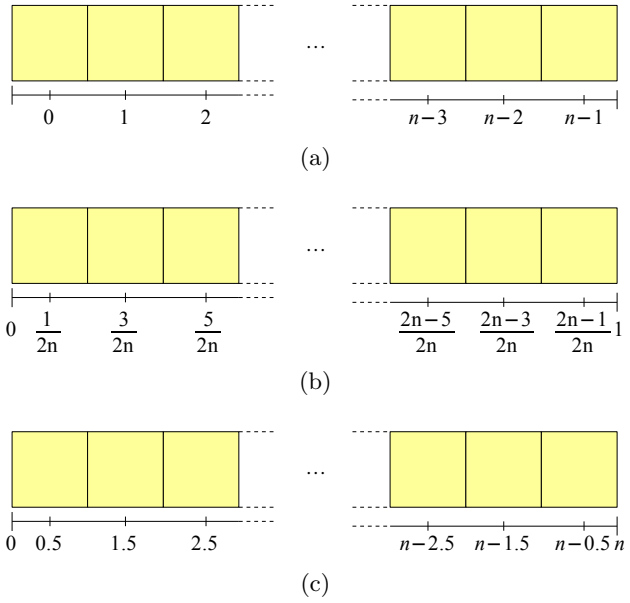


Figure 6.3: Addressing of elements in: (a) array; (b) OpenGL 1D / 2D / 3D texture; (c) OpenGL `Rect` texture

are $(0, 0)$ and those at the opposing, (n, m), with $n \times m$ the size of the texture. As illustrated in figure 6.3 (c), the coordinates of a texel are still not identical to those of the corresponding array element in 6.3 (a). While the increment from one texel to the next is the same as in an array, the address of the first texel in each coordinate is 0.5 and not 0.

All textures in OpenGL 2.0 may have arbitrary integer dimensions. The only limits are the capabilities of the hardware, 4096 texels per coordinate for current GPUs, and the amount of available memory. It is warned, however, that performance may suffer if the extents are not powers of two. Only texture maps with power of two dimensions are therefore used in this thesis.

In contrast to an array, it is not an error to address a position in a texture map that lies outside of its bounds. What happens in such a case depends on the wrap mode. The modes relevant to this thesis are:

Repeat The texture map is treated as if it repeated infinitely in all directions. Coordinates wrap around whenever they reach an edge of the texture.

Clamp to Border Coordinates are clamped to a limited range. When a texel is addressed that lies outside the texture, constant background values are returned instead. These are set beforehand using the appropriate OpenGL commands.

It is also possible to specify coordinates that fall between texel centers. Again, different modes for handling such cases are available. Only two are relevant to the code written for this thesis:

Nearest The values stored at the closest texel center, or background values if no such texel exists, are used.

Linear Values are interpolated between those recorded at the centers of surrounding texels. Depending on the dimensionality of the texture, two, four or eight texels are weighted. The interpolation is linear in each coordinate and for the three-dimensional case, corresponds directly to equation (5.7). This mode is not available for floating point textures where only *nearest* may be used.

One more aspect should be noted. The values of texture components are usually rescaled to a range of $[0, 1]$ when accessed by a shader. The only exception are floating point textures whose values are a returned directly as stored, allowing them to cover the entire range of floating point numbers.

6.1.3. Framebuffer Objects and Multiple Render Targets

This section introduces two concepts which are very valuable when working with shaders. As shown in figure 6.1, the framebuffer is at the end of the rendering pipeline. Values may be written to it but cannot be read back. This limitation can be overcome by using the `ARB_draw_buffers` extension. It allows *framebuffer objects* to be constructed in which a texture map serves as the backing store of the framebuffer. All values written to the framebuffer get recorded in the texture currently bound to it. After the binding has broken, the texture may be accessed in subsequent executions of a fragment shader. This feedback loop is indicated by the dashed arrow in figure 6.2. A texture may never simultaneously be used as both input and output as this leads to undefined results.

Only 2D and `Rect` textures may be bound to framebuffer objects. The framebuffer dimensions, number of components and their data types are given by those of the backing texture. Unfortunately, no authoritative list exists of the texture formats which may be used with framebuffer objects. It depends on the particular GPU and driver combination whether a format is supported or not. As the development of alternative code paths would have been very time consuming, the code written for this thesis assumes that several predefined formats can be used. These were chosen after tests showed them to be available on a wide range of NVIDIA GPUs and Linux driver revisions.

Each framebuffer object offers up to 16 attachment points to which textures may be bound. When rendering into different textures, it is fastest to attach them all to the same framebuffer object and switch only between attachment points. However, the limitation exists that all textures attached to a framebuffer object must have identical type, dimensions and format.

As noted earlier, a texture may have only up to four components. If a shader calculates more than four values per fragment, they cannot all be stored in the same texel. This problem is solved by using *multiple render targets*. Instead of a single framebuffer, potentially backed by a single texture, multiple buffers can be active at the same time. They may be ordinary OpenGL framebuffers or multiple attachment points of a framebuffer object. Because of the limitation noted above, multiple render targets backed by textures always share the same type, dimensions and format.

6.1.4. Shaders

Several extensions have been developed by different vendors that introduce shaders into the OpenGL rendering pipeline. Early proposals were very limited. Shaders had to be written in assembly language, could contain only as few as 16 instructions and were unable to perform branching or looping. Later revisions relaxed some of those restrictions but did not make programming any easier or provide compatibility between manufacturers. The first standard to address these issues was Cg [NVI06]. Developed by NVIDIA in cooperation with Microsoft, this is a high level language with a C-like syntax. The capabilities of the underlying hardware are abstracted into a range of profiles. Every shader is portable between all GPUs that support the profile it is written against. A compiler that produces GPU assembly code is provided as part of the Cg runtime environment.

Instead of adopting Cg, the Architecture Review Board chose a competing proposal by 3Dlabs, GLSL [KBR04]. First provided by means of an extension, it is now part of the OpenGL 2.0 specification. GLSL is also inspired by C, both in its syntax and the steps required to turn source code into an executable. The GLSL compiler is embedded in the OpenGL driver so that each manufacturer can provide a version that generates code optimized for their GPUs. After compilation, shader objects are linked into a program. It may contain both vertex and a fragment shaders or just one of these types. When a program is later activated, the precompiled shaders are loaded into the appropriate pipeline stages.

In the spirit of OpenGL, the GLSL language supports extensions. New commands and data types unique to their hardware can be added by each manufacturer. When no proprietary extensions are used, GLSL is largely a write once, run everywhere language. If a shader is too complex to be executed by the GPU, drivers are encouraged to resort to a software path, running the shader code on the host CPU. This leads to a large performance hit but provides a high degree of shader portability. Unfortunately, no method is provided to unambiguously determine whether a shader will run in hard- or software.

6.1.4.1. Fragment Shaders

Fragment shaders are addressed first because they are the only type used in this thesis. An example introducing several features of the GLSL language is shown in listing 6.1. The strong similarity to C code is apparent. The entry point is the beginning of the `main()` method in line 6. It gets executed for every fragment generated by the rasterizer. As the method has no parameters, the only inputs available are global variables. They fall into two general categories by being either `uniform` or `varying`.

The value of a `uniform` variable, such as those defined in lines 1 and 2, is set to a constant value for the entire primitive using OpenGL commands. A `varying` variable can take on a different value for every fragment. The values are explicitly set only at the vertices and interpolated by the rasterizer for the the individual fragments. This kind of variable is the only way in which a vertex shader may pass information to the fragment shader. If no vertex shader is active, only a set of default `varying` variables is available

Listing 6.1 Example fragment shader

```
1  uniform sampler2DRect data1;
2  uniform sampler2DRect data2;
3  uniform vec2          offset;
4
5  void main() {
6    vec2 coordinates = gl_TexCoord[0].xy + offset;
7    vec2 values      = vec2(texture2DRect(data1, coordinates).r,
8                            texture2DRect(data2, coordinates).r);
9    gl_FragData[0].r = sin(values.r);
10   gl_FragData[1].r = values.g;
11 }
```

whose values are set using OpenGL commands. The most prominent example are the texture coordinates, accessed as `gl_TexCoord` in line 6. The coordinates are expressed as an array because more than one set of coordinates may be required by complex shaders.

When the shader has completed its calculations, the resulting values must be sent to the framebuffer. This is done by writing to the variable `gl_FragColor`. If multiple render targets are being used, the array `gl_FragData` is written to instead, as shown in lines 9 and 10. Each element corresponds to a framebuffer so that different values may be sent to all of them. If `gl_FragColor` is used when multiple render targets are active, the same value is stored into all framebuffers.

Besides data types analogous to those found in other programming languages such as `bool`, `int` and `float` as well as arrays and structures of these, three types unique to GLSL are defined. The first two are `vec`n and `mat`n, defining a column vector of n components and an $n \times n$ matrix, respectively, with $n \in \{2, 3, 4\}$. The base type of the matrix and vector elements is `float`. Vectors of booleans and integers are also available as `bvec`n and `ivec`n. The third data type are samplers. They are used to access texture maps, as exemplified in lines 7 and 8. For each texture type, a corresponding sampler type is available. When a texture is to be accessed by a shader, it is bound to one of the GPU's texturing units using the appropriate OpenGL commands and the sampler variable is set to the number of this unit. How many texturing units are available differs between GPUs.

Vector elements are accessed by appending a period and the names of up to four components, as seen in lines 6 to 10. Three synonymous naming schemes for the components exist. They are $\{\texttt{r}, \texttt{g}, \texttt{b}, \texttt{a}\}$, $\{\texttt{x}, \texttt{y}, \texttt{z}, \texttt{w}\}$ and $\{\texttt{s}, \texttt{t}, \texttt{p}, \texttt{q}\}$. The first names are meant to be used when referring to colors, the second for points and the third for texture coordinates. However, as they are synonymous, the schemes may be exchanged freely. Components may be accessed in a different than their original order and the same component may appear multiple times, such as `coordinates.yxx`. The resulting data type is a vector with the same base type as the original or a scalar of the base type if only one component is specified. As seen in lines 9 and 10, it is also possible to choose a subset of the components on the left hand side of an assignment. Matrix elements may be selected by treating the matrix as an array. If only one array index is provided, a column is chosen.

Other than the selection of a subset of a vector or matrix, no type casting is possible.

Type conversions are performed by constructors, as shown in line 7. Here, a `vec2` is generated from two floats. For every vector and matrix type, a large number of constructors are defined which allow it to be built from various combinations of scalars, vectors and matrices that do not exceed its size. The constructor notation is also used when converting from one scalar data type to another.

Because vectors and matrices are basic data types, arithmetic operations may directly be performed on them, such as `m * v` expressing a matrix-vector multiplication. Most of the other features of the GLSL language directly correspond to those found in C. Local variables may be defined (lines 6 and 7), functions called (line 9) and defined (line 5). C keywords and syntax are used for selection (`if`, `else`), iteration (`for`, `do`, `while`), jumping (`return`, `break`, `continue`), comparison, arithmetic operations and assignment. Also, a subset of the C preprocessor directives is available.

An additional keyword is `discard`. When it is called, execution of the shader is aborted and no changes are made to the framebuffer for the current fragment. Also an addition are preprocessor directives which may be used to declare what extensions are required or desired and to conditionally compile sections of code depending on the available extensions. Within this syntactic framework, a range of built-in variables and functions are defined. The variables, `varying` or `uniform`, provide information about the current state of the rendering pipeline. Built-in functions perform operations such as vector normalization, trigonometric function evaluation and texture access. Notably absent are data types and functions for manipulating strings or pointers.

6.1.4.2. Vertex Shaders

The language features provided for vertex and fragment shaders are nearly identical. The main difference is that in a vertex shader, `uniform` and `attribute` variables are used as inputs while `varying` variables are written to. The `attribute` quantifier indicates a variable whose value is set separately for each vertex using the appropriate OpenGL commands. Other differences include the absence of the `discard` keyword, the lack of some of the built-in functions and no automatic calculation of the level of detail when accessing texture maps. As neither the level of detail nor vertex shaders as a whole are used in this thesis, they are not described here in any more detail. Because no vertex shader is loaded, the fixed functionality of the OpenGL pipeline is retained at the relevant stage. Texture coordinates are passed through as set by OpenGL commands while vertex positions are calculated by performing a perspective transformation whose parameters may be adjusted using the appropriate commands.

6.2. GPGPU Programming

OpenGL and the shader languages that lend it additional flexibility have been developed with the generation of real-time graphics in mind. *GPGPU*, for general purpose GPU programming, is the idea of using these tools to perform other types of calculations. GPUs with support for vertex and fragment shaders have entered the mainstream market only a few years ago. Considerable interest in their use for GPGPU applications has

developed even more recently, when the chips gained the ability to execute complex shaders and began to outperform traditional CPUs for some types of calculations. While the best practices for optimizing performance are still in flow and may change with every new GPU or driver revision, the fundamental paradigms have stabilized.

6.2.1. Paradigms

As described in section 6.1.1, when a primitive is rendered, the rasterizer generates a fragment for every framebuffer pixel covered by it. The fragment shader is the executed for each fragment and the calculated values recorded in the framebuffer. GPGPU equates the rendering of a primitive to a calculation pass, the textures read and written to arrays and the shader to a computation kernel [Fer04]. Values stored in textures or calculated by the shader are interpreted not as colors, luminosities or transparencies but as whatever is required in the context of the particular application.

Rendering of a primitive allows a kernel to be applied to an array of input values and the results to be stored in an output array. The output is captured by means of a framebuffer object, as described in section 6.1.3. OpenGL executes the shader exactly once for every pixel in the framebuffer covered by the primitive. To gain easy control over how many values are calculated and for which array locations, an orthogonal mapping is used at the vertex operations stage of the rendering pipeline.

With the texture backing the framebuffer having $n \times m$ elements, two options are popular. The first is to define a coordinate system in which $(0, 0)$ corresponds to one and (n, m) to the opposing corner of the framebuffer. A rectangular slab of size $n \times m$ is then rendered to execute the shader for every element of the output texture. A smaller slab may be used to calculate values only for some of its elements. The other option is a coordinate system in which the corners of the framebuffer are at $(0, 0)$ and $(1, 1)$. This allows a slab to be specified that covers the desired fraction of the framebuffer without knowing its size in texels. Both techniques are equivalent and may be interchanged.

Input values are read by the shader from two sources. One are the global variables introduced in section 6.1.4.1 and the other, textures. The flexibility of variables in passing data to the shader is very limited. If a variable is `varying`, its value can only be explicitly set at the four vertices. For a `uniform` variable, just a single setting per primitive is allowed. Variables are therefore only used to express constants and texture coordinates. The actual input data is stored in texture maps. A very simple case is the use of a single texture with the same dimensions as the one backing the framebuffer. This leads to a $1 : 1$ relationship where each output texel is calculated based on the values of the corresponding input texel.

Many GPGPU shaders use much more complex arrangements. Not one but multiple input textures are employed with different sets of coordinates pointing to the locations to be queried in each. Texture coordinates may also be calculated within the shader based on the values read from other texture maps. The maximal nesting depth of such dependent texture look-ups depends on the particular GPU. A shader is able not only to read from but also to write to more than one texture map. As explained in section 6.1.3, this is achieved by using multiple render targets and allows a shader to output more

than the four values that can be stored in a single texel.

Each rendering pass in which a shader is executed is preceded by a setup phase. Here, OpenGL commands are called to set up the rendering pipeline. The shader is activated, texture maps bound to texturing units, `uniform` variable values set, an orthogonal mapping specified and framebuffer object attachment points chosen. Then, a slab is rendered with texture coordinates set at each vertex so that after interpolation by the rasterizer, the desired locations are accessed for each fragment. If another rendering pass follows, the state of the pipeline is preserved by OpenGL and only those properties which are to be changed need to be set up anew.

In conjunction with the techniques described in the following three sections, GPGPU can be used to perform arbitrarily complex computations. However, one limitation exists that cannot be easily overcome. GPUs suffer from a lack of precision. While floating point numbers can be represented according to the IEEE 754 single precision specification, not all features of the standard are followed. Denormalized numbers are not supported and rounding rules relaxed, allowing for a range of optimizations and simplifications in the GPU hardware at the cost of precision. The small errors introduced may not be visible in a generated image but can become apparent in scientific applications. Also, the exact deviations from the IEEE standard differ between vendors. The most glaring imprecision is experienced in ATI GPUs of all generations but the most recent which are able to store 32 bit numbers but perform calculations with a precision of only 24 bits.

6.2.1.1. Scatter versus Gather Operations

Every algorithm can be decomposed into a series of steps each of which either scatters or gathers data:

Scatter The input values are used to calculate the location a predefined output will be written to. An example is the calculation of an array index and the subsequent storage of a constant at that position.

Gather The storage location is predetermined while the output value is calculated. An example is the addition of two variables and placement of the result in a third.

Of these two operation types, only gather is naturally supported by a fragment shader. The output location is given by the framebuffer position of the current fragment and cannot be changed. The value to be written is determined by the shader based on data gathered from texture maps and global variables. When an algorithm to be implemented on the GPU requires a scatter operation, several options are available. An alternative formulation of the algorithm may exist that has similar complexity but uses only gather operations. This approach should be chosen whenever possible as it leads to the highest computational efficiency.

If no alternative formulation is found, a vertex shader can be used to perform the scatter. It is able to adjust the positions of vertices, thus scattering the information associated with them throughout the framebuffer. The drawback is that a large number

of vertices usually need to be generated. Because three-dimensional scenes consist of few vertices and many fragments, GPUs are optimized for fragment processing so that vertex shaders and high numbers of vertices negatively affect performance.

The third option is to simulate a scatter operation using two fragment shaders. The first stores value and position pairs into the framebuffer. A second shader then uses the resulting texture as input and for each fragment, checks whether its position is associated with a value anywhere in the texture. Because a series of texels need to be read for each fragment, this approach is also not likely to provide high performance. It is, however, a special case of the technique described in the next section and heavily used in GPGPU applications.

6.2.1.2. Multipass Rendering

Multipass rendering refers to the subsequent execution of shaders, using the output of previous passes as the input of the next. Only with this technique is it possible to decompose an algorithm into a series of shader invocations. The rendering pipeline may be set up differently for each pass by changing the shader, input and output textures as well as the orthogonal mapping and the size of the slab rendered. Because a framebuffer object is always used, the results of every pass are captured into one or more textures and are available until they get explicitly overwritten or discarded.

The same algorithm can be divided into shaders in many different ways. Except for rounding errors and imprecisions, the computation results should be identical in all cases. Efficiency, however, can vary greatly depending on the degree to which the components of the GPU are utilized or idle. Unfortunately, there are no precise rules by which to determine how many or how few operations should be grouped into a shader. Some of the factors that can help with the decision are listed in section 6.2.2.

6.2.1.3. Reduction

Many gather operations include a reduction component where the elements of an input array are combined into a smaller number of output elements. Looping in the fragment shader to directly calculate the final elements, while possible on newer GPUs supporting complex shaders, is not efficient. It is better to combine only small groups of elements, reducing the size of the array and then to repeat this process as often as required. When implemented as multi-pass rendering, reduction requires only a single pair of textures. One serves as input and the other as output with their roles exchanged after each pass. This is often referred to as ping-ponging.

For each pass, the size of the slab and the texture coordinates at its vertices are adjusted to simulate input and output arrays of decreasing size. When performing reduction in one dimension, every pass usually combines two neighboring texels, halving the size of the array. The slab used in the first pass of a horizontal reduction therefore covers only the left half of the output texture while the coordinates at its vertices span the entire input. Both slab width and coordinate range are then halved for each subsequent pass. To reduce a texture map n texels wide to a width of one, $\log_2 n$ passes are required.

In two-dimensional reduction, the texture size is halved in both dimensions by combining four neighboring texels. The number of passes required to reduce an $n \times n$ texture to a single texel is again $\log_2 n$.

Between rendering passes, no changes to the rendering pipeline other than the swapping of the roles of the two textures are necessary. The fastest way to achieve this is to bind both textures to texturing units as well as attachment points of the framebuffer object. After each pass, only the numbers of the texturing unit used for input and the attachment point serving as output need to be adjusted.

6.2.2. Optimizations

A GPU owes its computational efficiency to the stream processing model. Because the calculations performed for each fragment are independent, they can be run in parallel on multiple execution units. NVIDIA's GPUs feature 16 fragment processors in the Geforce 6 and 24 in the Geforce 7 series, each of which can run the shader for a different fragment. Multiple vertex processors are also provided, 6 by the Geforce 6 and 8 by the Geforce 7 series. Every processor can execute several instructions per clock cycle, further increasing performance.

Due to the secretiveness of vendors, little information is available about the design of the processors and their optimal utilization. Much of what is known can be gathered from the GPGPU website and its forums [gpg06], a programming guide by NVIDIA [NVI05] and the GPU Gems series of books [Fer04]. One important insight is that the processors are not truly independent. Not only do they share caches but optimal performance is only achieved when all processors execute the same instructions. It follows that conditional branching should be used sparsely as performance suffers when the execution paths differ between fragments. Another important observation is that to reach peak performance, each rendering pass should consist of at least $10,000$ fragments.

This combined with the fact that texture caches are relatively small, estimated at a few kilobytes, indicates that complex shaders executed for only a few fragments should be broken down into several simpler shaders, each of which calculates a large number of intermediate values. Simpler shaders also reduce the risk of exceeding the limits of the GPU hardware. If a shader needs more resources, for example texturing units, variables or instructions per fragment, than are provided by the GPU, it may run in slow software emulation or fail to compile at all.

Several other aspects should also be kept in mind when designing a GPGPU application. Built-in functions such as `sin`, `cos` and `log` are executed in a single cycle on the GPU. This means that contrary to a CPU based implementation, there is no need to tabulate their values to achieve optimal performance. From sections 6.1.2 and 6.1.3 it follows that textures should have power of two dimensions and the ones that will be rendered into are best attached to framebuffer objects in such a way that only attachment points need to be switched between rendering passes.

Two obstacles in the development of shaders are the lack of proper debugging tools and bugs in the drivers. While a shader can be executed in an emulated fragment processor to analyze its behavior, this does not help in finding problems related to the interaction

with other GPU components. The interaction is also important when profiling shaders to increase their efficiency. The NVShaderPerf tool by NVIDIA[2] can output the assembly code generated for a GLSL source file and count the number of cycles required for its execution. However, because the complex interactions with texturing units and caches are not simulated, all numbers are produced under the fictitious assumption that every texture access needs only one cycle.

Live debugging information is only available in the form of OpenGL error codes, a small number OpenGL state variables that indicate whether previous commands were successful and an info log which contains the messages output by the most recent GLSL compiler run. The only way in which a shader can provide feedback is through the values it outputs into the framebuffer. The generated texture can be read back into main memory and analyzed to verify the results of the shader. Debugging is complicated by bugs in the drivers supplied by GPU vendors. Because they are closed source and only limited support is provided, obscure and undocumented bugs may be present. A prominent example for NVIDIA GPUs is their failure to write into the 4096^{th} column or row of a texture attached to a framebuffer object, limiting the usable texture size to 4095×4095.

6.3. Registration

The registration system of chapter 5 performs all of its calculations on the CPU. However, a series of hooks are included which allow the principles of GPGPU to be applied to it. They are the four abstract classes highlighted in figure 5.4. By providing new implementations, the sampling of the two images and the calculation of mutual information or its derivatives can be moved onto a GPU. These operations encompass the bulk of the computations required in each iteration. Because porting from CPU to GPU causes substantial development and debugging effort, other elements that have less significant impact on execution time are not considered.

As noted in section 5.2.1, sampling and the calculation of mutual information or its derivatives are largely independent. Their GPU based implementations are therefore described separately in the following sections. Each implementation essentially follows its CPU based counterpart from chapter 5. The optimizations applied there are retained and the main difference is that iterations are converted to parallel calculations of the individual elements. If sums are to be evaluated, the values of the addends are determined in parallel and then summed via reduction operations. To simplify the code, the assumption is made that the sizes of the two samples are identical, $N = N_A = N_B$.

All implementations are described in detail, covering every computation step and shader. The reasons for this elaborate treatment are twofold. First, the concepts of GPGPU are still not widely known and thorough explanations may be required to provide clarity. Second, any future work is significantly aided. Only if the interactions of the individual components, shaders and textures in this case, are fully described can the system easily be modified or extended.

[2] `http://developer.nvidia.com/object/nvshaderperf_home.html`

6.3.1. Notations

At the center of the presentation of every implementation is a diagram that illustrates the control and data flows. The first diagram to use all elements available in the notation may be found in figure 6.5. Shown on the left is the control flow. Blue headings indicate the starts of public methods and subroutines. Every implementation contains an `Initialization` method that performs the required precalculations. For sampling, three more public methods are provided. They are `SampleReference`, `SampleRegistered` and `SampleRegisteredDerivatives`. As their names imply, the first two sample the images while the third determines both registered image intensities and their derivatives with respect to the transformation parameters. Each method returns the values for all sampling points. For the calculation of mutual information or its derivatives, the only public method other than `Initialization` is `Calculation`.

Three types of blocks appear in the control flow, all rectangular in shape. The italic label *CPU* expresses a step in which the CPU manipulates textures. Following established conventions, blocks with double vertical borders indicate subroutine or method calls. The name of the routine is written in blue and the size of the texture area in which it stores its return values is given in the lower right corner. It should also be noted that when the sizes of data structures are given, they are provided in the order $width \times height\,(\times depth)$, as is usual for texture maps and arrays and contrary to what is done for matrices.

All other blocks correspond to rendering passes. Each is labeled with the name of the shader active during the pass. The size of the slab rendered, and thus the number of fragments for which the shader is executed, is again given in the lower right corner. When a reduction operation is performed and the same shader used in multiple passes, this is indicated by a circular arrow in the lower left corner of the block. The number of passes is specified next to the arrow and the sizes of the slab are given for the first and the last pass. A loop involving several blocks is shown by a dashed box around them with a circular arrow and the number of iterations in the lower left corner.

The texture maps are represented by columns on the right hand side of the control flow diagram. For each block in the control flow, the input and output textures are indicated by arrows. Green arrows correspond to reads and red, to writes. Because the output of every shader is to be captured for use in later parts of the calculation, each rendering pass manipulates at least one texture map attached to a framebuffer object. Using multiple render targets, it is also possible for a shader to affect more than one texture. In the case of reduction operations and loops, the same shader may be executed several times with different input and output textures. The alternatives are indicated by dashed extensions of the green and red arrows.

In addition to diagrams and textual explanations, the full source code of all shaders is provided in appendix A. Not explicitly shown in the following sections are a series of common initialization steps that must be executed by every class using the GPU:

- Create OpenGL context
- Query available extensions
- Allocate texture maps

- Allocate framebuffer objects
- Attach textures to framebuffer objects
- Load, compile and link shaders

These steps prepare the data structures and set up the rendering pipeline. The context created is a reference to the pipeline for all subsequent OpenGL commands. As noted in sections 6.1 and 6.1.3, if any of the initializations fail due to insufficient hardware resources or driver support, registration is aborted.

While sampling and calculation of mutual information or its derivatives are explained separately, their implementations are placed in joint classes in the actual code. The primary reason for this is better control over the state of the rendering pipeline. When all OpenGL commands that affect the pipeline are executed by the same object, it can keep track of the precise state. As suggested in section 6.2.1, instead of setting it up from scratch for every rendering pass, only the required changes need to be made to the current state. A special case are classes that follow a hybrid approach. Here, sampling is performed on the CPU, using the `TransformationNonRigid` class of section 5.2.2.6. The samples are then stored into texture maps and the GPU based implementation is used to calculate mutual information or its derivatives. Benchmark results comparing the performance of CPU, GPU and hybrid implementations will be provided in chapter 7.

6.3.2. Rigid Sampling

Described in this section is the implementation of the sampling process using the rigid transformation function. Control and data flow are shown in figure 6.4.

During the initialization and precalculation stage, the sampling point positions and the two tomographic images are uploaded to the GPU as texture maps. The images are retrieved from the `Data` class of section 5.2.2.2. Because they are embedded in volumes with power of two dimensions, both images can directly be stored in 3D texture maps. Since the intensities are normalized to an integer range of $[0, 255]$, each voxel is represented by a texel consisting of a single byte.

The modes of the textures, as introduced in section 6.1.2, are set to *clamp to border* and *linear*. The first dictates that a background intensity is returned when a texel outside a volume is addressed. This intensity is set to zero, in line with the handling of such cases by the `TransformationRigid` class. The other mode enables trilinear interpolation in hardware. On every access to one of the textures, the GPU automatically evaluates equation (5.7) and calculates the intensity from those of eight surrounding texels. Together, the two modes ensure that after the implicit conversion to voxel coordinates, only a single look-up is required to retrieve the correct intensity regardless of where a sampling point is located in an image.

The sampling point positions are generated using the same method as employed by the CPU based implementation, described in section 5.2.2.4. Each sampling point coincides with the center of a voxel in the reference image. Per point, two sets of coordinates are stored in `SamplingPositions`. One is expressed in millimeters, the other directly as a position in the `DataReference` texture. It is calculated from reference volume indices $\mathbf{x}''_u$

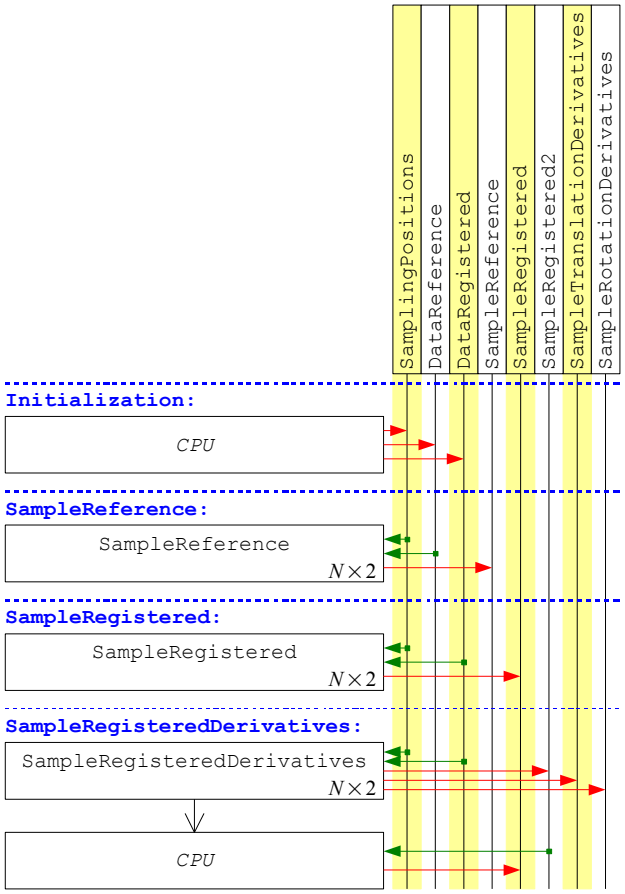


Figure 6.4: Control and data flow: Rigid sampling

by adjusting for the difference in addressing between arrays and 3D textures illustrated in figure 6.3 (a) and (b). To each component $(\mathbf{x}''_u)_k$, 0.5 is added and the result divided by the size of the texture in the kth dimension.

When sampling the reference image, the `SampleReference` shader is executed for the $N \times 2$ sampling points, the first N of which represent the sample S_A and the other, S_B. The shader uses the precalculated texture locations to directly retrieve the desired intensities. The only additional operation is scaling. As explained in section 6.1.2, the values of all textures whose components are not floating point numbers get rescaled to a range of $[0, 1]$ on every access by a shader. To obtain the original image intensities, they are therefore multiplied by 255. The data type of the `SampleReference` texture into which the results are written is half precision float. While this uses 16 bits to store a value for which 8 bits would have been sufficient, it ensures that no further automatic rescaling is applied to the sample by the GPU.

The shader that samples the registered image is similarly simple but implements the rigid transformation function. According to equation (5.3), this can be done as a matrix-vector multiplication, which is a native and fast operation on the GPU:

$$\mathbf{x}_v = \mathbf{M}_T \mathbf{x}_u \tag{6.1}$$

Although only three coordinates are stored for the sampling points $\mathbf{x}_u$, the required homogenous coordinate representations can efficiently be obtained. The fourth component of a texture map is the alpha intensity channel. If it is not part of the texture, the GPU automatically substitutes a value of one. The sampling point positions read from the `SamplingPositions` texture map may therefore directly be treated as `vec4` homogenous coordinate representations. As the implicit conversion from millimeters to voxel coordinates and the adjustment for the addressing scheme of a 3D texture consist of scaling and translation, they can also be expressed as a matrix $\mathbf{M}_C$. Since both $\mathbf{M}_T$ and $\mathbf{M}_C$ are applied to every $\mathbf{x}_u$, they can be premultiplied so that the shader needs to perform only one matrix-vector multiplication. The resulting matrix $\mathbf{M}'_T$ is passed to the shader as a `uniform` parameter of `mat4` type:

$$\mathbf{M}'_T = \mathbf{M}_C \mathbf{M}_T \tag{6.2}$$

The `SampleRegisteredDerivatives` method returns the registered image intensity and its derivatives with respect to all six transformation parameters for each sampling point $\mathbf{x}_u$. The method that the `SampleRegisteredDerivatives` shader uses to estimate the derivatives is that of section 5.2.2.4. First, the gradient is calculated from the intensities at $\mathbf{M}'_T \mathbf{x}_u$ and three neighboring points, as specified in equation (5.14). The offsets that must be added to reach these points are precalculated and passed to the shader as `uniform` parameters. Next, equation (5.13) is evaluated for each parameter p:

$$\frac{d}{dp} v\left(T\left(\mathbf{x}_u\right)\right) = \operatorname{grad} v\left(T\left(\mathbf{x}_u\right)\right) \mathbf{M}_{T,p} \mathbf{x}_u \tag{6.3}$$

As noted in section 4.1.2.2, most of the elements of the $\mathbf{M}_{T,p}$ are zero. For the translation parameters t_x, t_y and t_z, only the last column contains non-zero elements.

Instead of the entire matrix, only that column is therefore given to the shader and used in the calculations. In the case of the rotation parameters α, β and γ, 3×3 matrices are used.

Because the shader calculates an intensity and six derivatives, multiple render targets are required to store its outputs. Following the explanation in section 6.1.3, all texture maps that are written to must have not only the same dimensions but also identical format. Three textures with three single precision floating point components each are used. `SampleTranslationDerivatives` and `SampleRotationDerivatives` store the derivatives with repsect to the translation or rotation parameters, respectively.

`SampleRegistered2` receives the registered image intensity at each point. On every access to this texture, three components are read although only one of them contains useful information. To reduce the total number of bytes that need to be transferred from memory over the course of registration, the intensities are copied to a more suitable texture with just one half precision floating point component. An OpenGL copy command is issued by the CPU which, depending on the GPU and its driver, may be executed in hard- or software.

6.3.3. Rigid Mutual Information Derivative

As suggested in section 6.3, the GPU based estimation of mutual information derivatives largely follows the method described in section 5.2.2.4 with the main difference that instead of being performed iteratively, all summations are executed in parallel by evaluating the addends first and then applying reduction operations. During initialization, the values of the Gaussian density functions are precalculated and stored in the texture maps `Gaussian1D` and `Gaussian2D`. Because it offers the most convenient addressing, the texture type `Rect` is used. Then, the reference image is sampled. This is handled entirely by the `SampleReference` method described in the previous section.

The first step in the actual calculation for a new alignment is the sampling of the registered image intensities and their derivatives with respect to all transformation parameters. This, again, is handled by the appropriate method from the previous section. Next, equation (5.8) must be evaluated:

$$\frac{d}{dp} MI^{\star}(X,Y) = \frac{1}{N_B} \sum_{\mathbf{x}_i \in S_B} \sum_{\mathbf{x}_j \in S_A} \left[W_Y(v_i, v_j) \frac{1}{\sigma_Y^2} - W_{X,Y}(\mathbf{w}_i, \mathbf{w}_j) \frac{1}{\sigma_{Y,Y}^2} \right] (v_i - v_j) \left(\frac{d}{dp} v_i - \frac{d}{dp} v_j \right) \tag{6.4}$$

The definitions used are provided by equations (5.9) and (5.10):

$$v_i = v(T(\mathbf{x}_i)) \qquad \mathbf{w}_i = \begin{pmatrix} u(\mathbf{x}_i) \\ v(T(\mathbf{x}_i)) \end{pmatrix} \tag{6.5}$$

$$W_Y(v_i, v_j) = \frac{f_{N_{0,\sigma_Y^2}}(v_i - v_j)}{\sum_{\mathbf{x}_k \in S_A} f_{N_{0,\sigma_Y^2}}(v_i - v_k)} \qquad W_{X,Y}(\mathbf{w}_i, \mathbf{w}_j) = \frac{f_{N_{\mathbf{0},\mathbf{\Sigma}}}(\mathbf{w}_i - \mathbf{w}_j)}{\sum_{\mathbf{x}_k \in S_A} f_{N_{\mathbf{0},\mathbf{\Sigma}}}(\mathbf{w}_i - \mathbf{w}_k)} \tag{6.6}$$

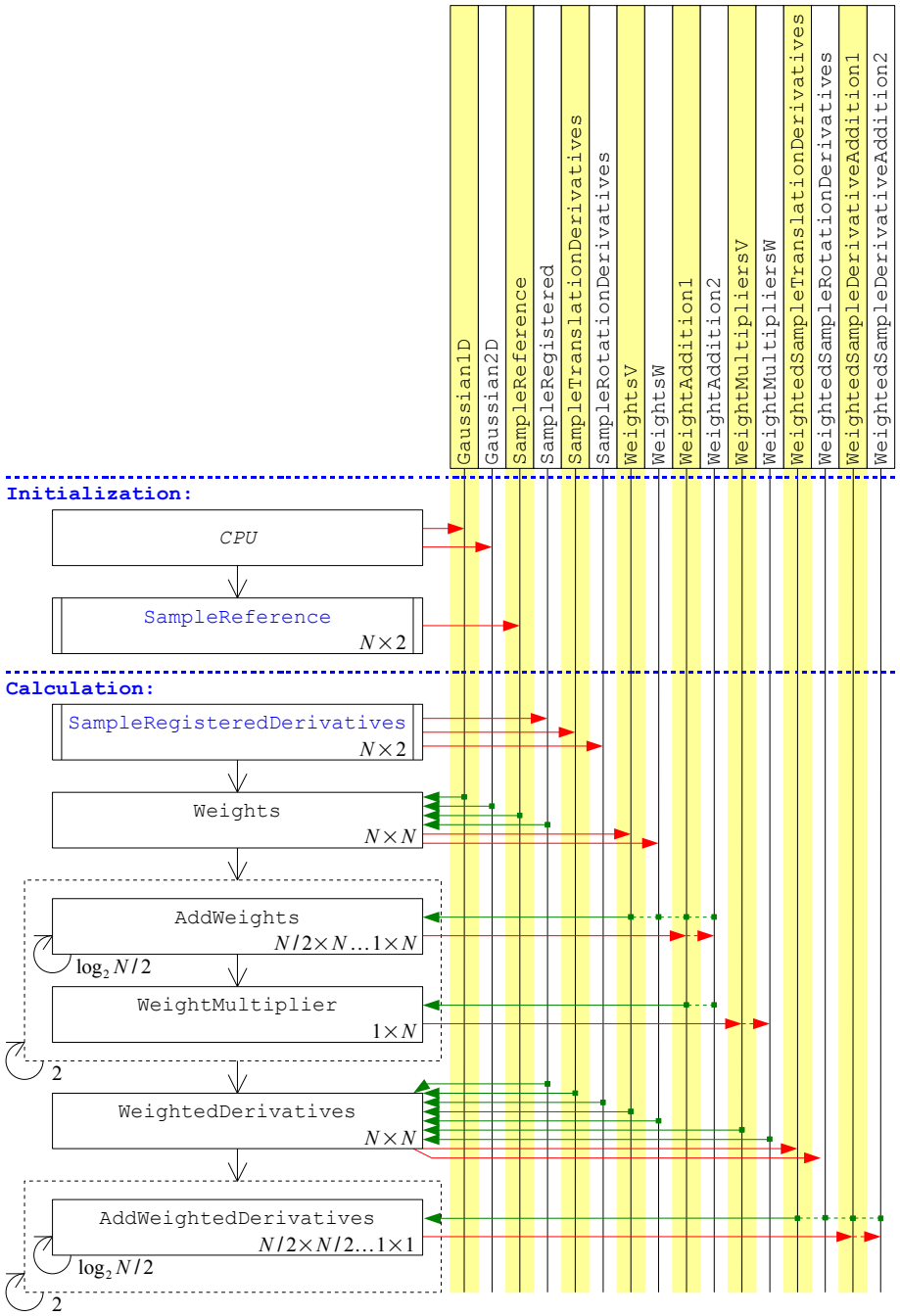


Figure 6.5: Control and data flow: Rigid MI derivative calculation

The weights W_Y and $W_{X,Y}$ are calculated first. This process is distributed among three shaders. `Weights` determines only the numerators of the expressions in equation (6.6) and stores them into the texture maps `WeightsV` and `WeightsW`. The shader is executed for every pair of points $(\mathbf{x}_i, \mathbf{x}_j) \in S_B \times S_A$ and uses the precalculated values of the Gaussian density functions and the sampled image intensities.

Next, the denominators are calculated. Because the computation is identical for the W_Y and the $W_{X,Y}$, the same series of shaders is executed twice with different input and output textures. The W_Y are considered first.

`AddWeights` performs a horizontal reduction operation, summing the numerator values for every $\mathbf{x}_i \in S_B$. As each pass halves the effective width of the texture, a logarithmic number of steps is required. In the first pass, the input texture is `WeightsV`. Then, `WeightAddition1` and `WeightAddition2` are used alternatingly as input and output. After reduction has completed, the result is a single column of N values, each corresponding to the denominator for an $\mathbf{x}_i \in S_B$. Since division on the GPU is slow, the reciprocal of every denominator is calculated only once, converting it into a multiplier. This is handled by the `WeightMultiplier` shader. The reciprocals are additionally premultiplied by σ_Y^{-2}, reducing the number of multiplications required when evaluating equation (6.4). If a denominator is zero, a value of zero is substituted for its reciprocal. The results are stored in `WeightMultipliersV`.

The entire summation and inversion process is then repeated with `WeightsW` as initial input, `WeightMultipliersW` as final output and $\sigma_{Y,Y}^{-2}$ used in the premultiplication.

After the weights have been precalculated, the addends of equation (6.4) can be evaluated. The shader that performs this operation is `WeightedDerivatives`. For each pair of sampling points, the part of the addend that is independent from the parameters p is calculated first using the precalculated numerators and denominators of W_Y and $W_{X,Y}$ and the registered image intensities. Then, the result is multiplied with $\left(\frac{d}{dp}v_i - \frac{d}{dp}v_j\right)$. The derivatives of the registered image intensities with respect to the transformation parameters are obtained from the two textures `SampleTranslationDerivatives` and `SampleRotationDerivatives`. Because each holds the derivatives for three parameters, the subtraction and multiplication are performed for them in parallel. The resulting addends are then output to the textures `WeightedSampleTranslationDerivatives` and `WeightedSampleRotationDerivatives`.

In the final stage of the calculation, the nested sum of equation (6.4) is evaluated by summing the addends. This is done by a two-dimensional reduction operation via the shader `AddWeightedDerivatives`. Beginning with an input texture of size $N \times N$, it adds all elements in a logarithmic number of steps by halving the amount of data in both dimensions per iteration. Reduction is performed twice, first for the derivatives with respect to the translation parameters and then for the rotation parameters. The temporary texture maps used in the ping-ponging are `WeightedSampleDerivativeAddition1` and `WeightedSampleDerivativeAddition2`. At the end of every reduction, the three resulting derivatives are retrieved by the CPU from the first texel of the ping-ponging texture last written to.

Because six derivatives are calculated for each of the $N \times N$ sampling point pairs, the

memory required by intermediary textures grows faster in N than for any of the other GPU based implementations. To increase the usable sample size, an alternative code path is provided that employs smaller textures. The last two steps, derivative calculation and addition, are repeated four times, on every execution considering only a $N/2 \times N/2$ subset of the sampling points. As the texture, framebuffer object and shader switches negatively impact performance, this code path is only used if the sample size is too large for a direct calculation on the given GPU.

6.3.4. Rigid Mutual Information

The estimation of mutual information again closely follows the CPU based implementation, as described in section 5.2.2.5. Initialization begins with the same two steps as used in section 6.3.3. The values of the Gaussian density function are precalculated and the reference image is sampled. Next, its entropy is estimated. The formula used for this purpose is obtained from equation (5.15):

$$H^\star[X] = -\frac{1}{N} \sum_{\mathbf{x}_i \in S_B} \log \frac{1}{N} \sum_{\mathbf{x}_j \in S_A} f_{N_{0,\sigma_X^2}} \left(u(\mathbf{x}_i) - u(\mathbf{x}_j) \right) \tag{6.7}$$

For each pair of sampling points $(\mathbf{x}_i, \mathbf{x}_j) \in S_B \times S_A$, the addend of the inner sum is calculated by the shader `GaussianPDF1D`. It retrieves the reference image intensities from the `SampleReference` texture and uses their difference to look up the precalculated value of the Gaussian density function in `Gaussian1D`. The addends are placed in the `EntropyCalculation1` texture and the `Entropy` subroutine is called to evaluate the remainder of the equation.

The first step in calculating the mutual information for a new alignment is the sampling of the registered image, handled by the appropriate method from section 6.3.2. Then, the registered image entropy is estimated. According to equation (5.15), once the two images have been sampled, their entropies are calculated in the same way. The `GaussianPDF1D` shader and the `Entropy` subroutine are therefore employed again, the only difference being that intensities are read from the `SampleRegistered` texture. For the joint entropy, the summation is identical and only the calculation of the addends differs:

$$H^\star[X,Y] = -\frac{1}{N} \sum_{\mathbf{x}_i \in S_B} \log \frac{1}{N} \sum_{\mathbf{x}_j \in S_A} f_{N_{\mathbf{0},\mathbf{\Sigma}}} \left(\begin{pmatrix} u(\mathbf{x}_i) \\ v(T(\mathbf{x}_i)) \end{pmatrix} - \begin{pmatrix} u(\mathbf{x}_j) \\ v(T(\mathbf{x}_j)) \end{pmatrix} \right) \tag{6.8}$$

The values of the addends are determined by the shader `GaussianPDF2D`. It looks up the values of the two-dimensional Gaussian density function in `Gaussian2D` based on the differences of the intensities read from `SampleReference` and `SampleRegistered`. To sum the addends, the the `Entropy` routine is called once again.

This routine employs three shaders. First, a horizontal reduction is performed by `AddHorizontally`. The initial values are read from `EntropyCalculation1`, the textures `EntropyCalculation1` and `EntropyCalculation2` used for ping-ponging and the result of the reduction placed in one of them, depending on the number of passes required.

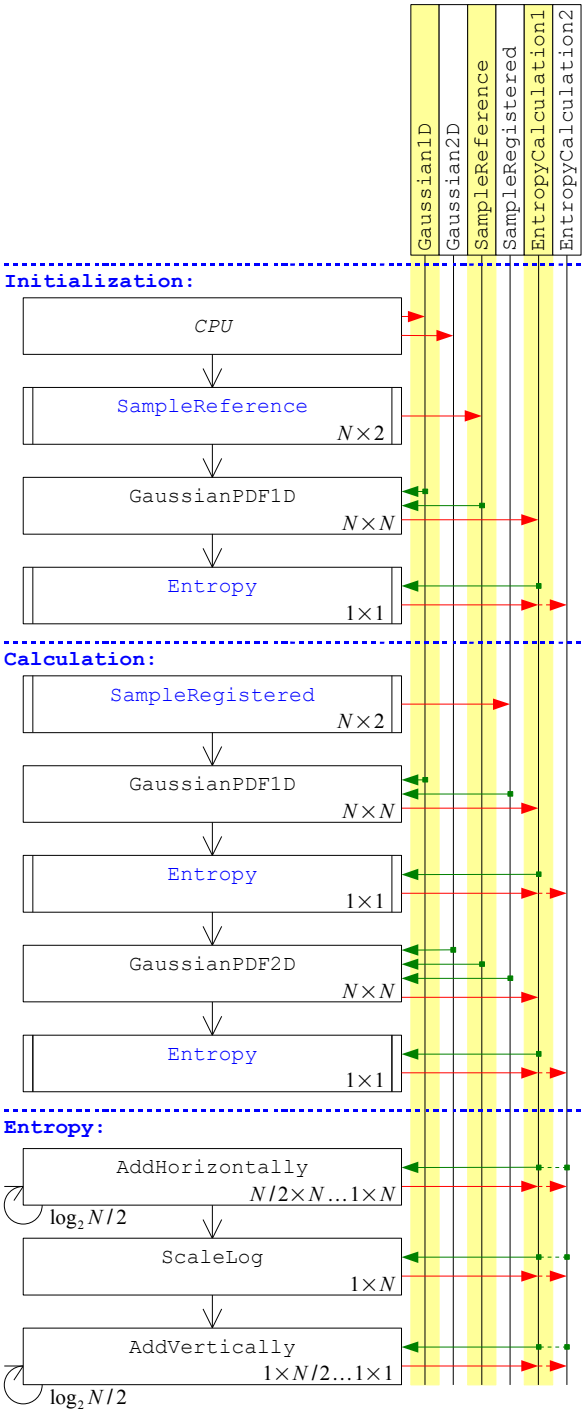


Figure 6.6: Control and data flow: Rigid MI calculation

The shader applied next is `ScaleLog`. It multiplies the resulting sums with N^{-1} and takes their logarithms. As explained in section 5.2.2.5, when the value of a sum is sufficiently close to zero, zero is substituted for its logarithm. The last operation is a vertical reduction, executed by the `AddVertically` shader using the same intermediary textures as before. The result of this reduction is read by the CPU from the first texel of the the ping-ponging texture last written to. In order to obtain the desired entropy, the value is then divided by N and negated. Mutual information is estimated from the entropies using equation (2.40):

$$MI(X,Y) \approx MI^{\star}(X,Y) = H^{\star}[X] + H^{\star}[Y] - H^{\star}[X,Y] \tag{6.9}$$

There is one optimization used in section 5.2.2.5 but omitted on the GPU. When calculating the entropies of the two images, an addend is generated for each pair $(\mathbf{x}_i, \mathbf{x}_j)$ instead of considering every intensity combination only once and weighting its addend with the number of times it occurs. The reason is that counting the occurrences of each intensity in the samples is a scatter operation which cannot directly and efficiently be implemented on a GPU, as explained in section 6.2.1.1.

6.3.5. Non-Rigid Sampling

In the next two sections, different GPU based implementations are proposed for the sampling process using a non-rigid transformation. Both calculate the transformation function and its derivatives using the methods of section 5.2.2.6. The extensive optimizations described there are not repeated here in detail. Only their implementations on the GPU are addressed. The two proposals differ in the granularity with which the calculations have been decomposed into shaders.

6.3.5.1. Monolithic Shader

The first approach is illustrated in figures 6.7 and 6.8.

Initialization

The initialization method serves several purposes. In the first step, the two images are retrieved from the `Data` class and stored into 3D textures. Their modes are set to *clamp to border* and *linear* so that the intensity for any set of voxel coordinates may directly be queried. A grid of $8 \times 8 \times 8$ points is then constructed and their positions recorded in the `GridPositions` texture. Points that fall outside the reference image are pruned so that their total number g_t may be less than 8^3. Placed in the `ClassificationPattern` texture are the offsets from a grid point to the 257 positions at which the reference image should be sampled to determine whether significant data is present in its vicinity. The grid point positions are expressed as locations in the `DataReference` texture and the offsets are scaled so that they may directly be added to these locations.

The shader `Classify` is then executed. It samples the reference image at the 257 offsets from every grid point and classifies the intensities as indicating either significant

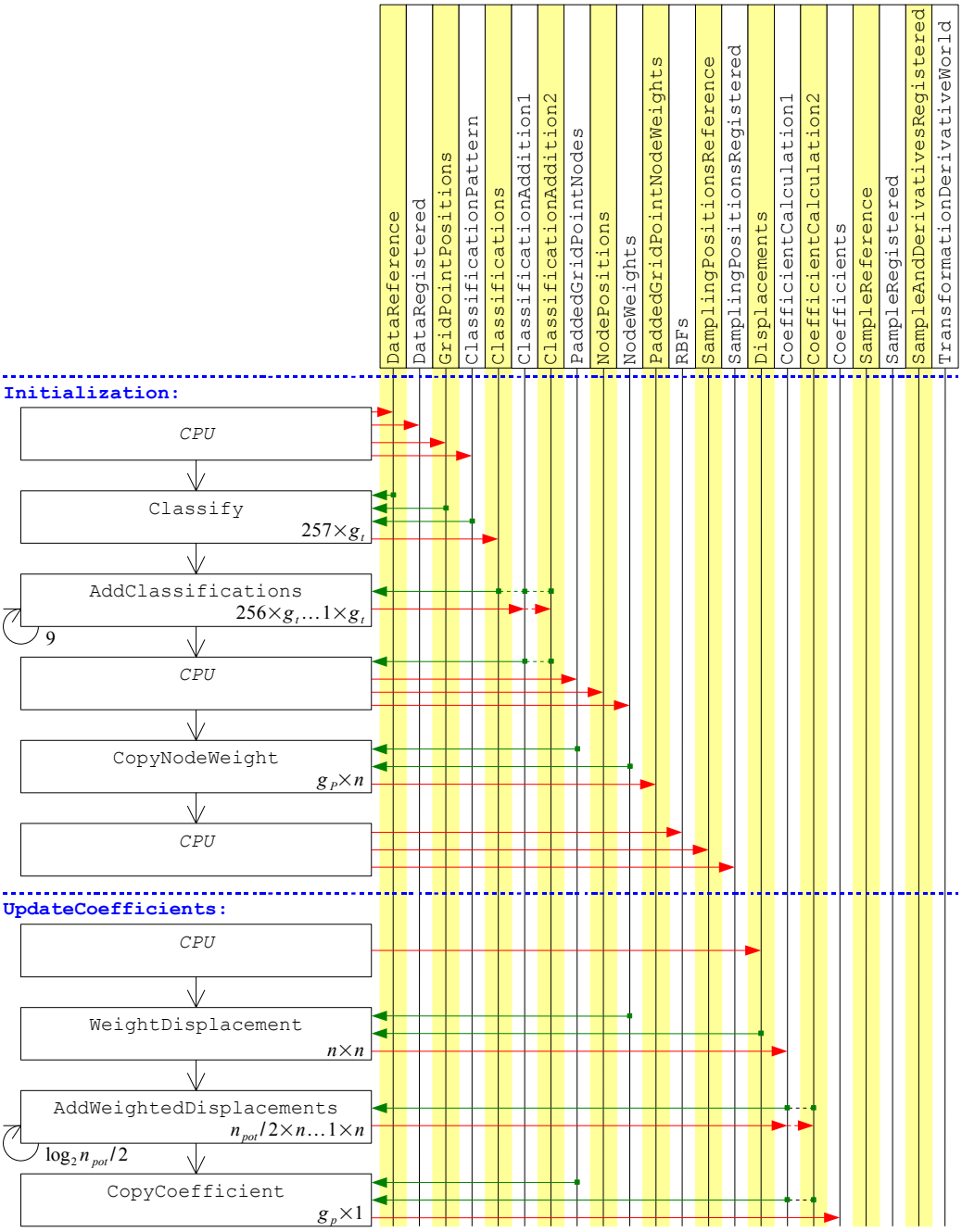


Figure 6.7: Control and data flow: Non-rigid sampling, monolithic shader implementation

information or background noise. According to section 4.2.3, the threshold to be used for this purpose is $t_1 = 7$. Because the intensities are rescaled from a range of $[0, 255]$ to $[0, 1]$ by the GPU, the actual threshold in the shader code is 0.03. The output of the shader is zero when background noise is detected and one if significant information is found. A horizontal reduction operation performed by the `AddClassifications` shader sums the classification results for each grid point. The input is read from the `Classifications` texture in the first pass while `ClassificationAddition1` and `ClassificationAddition2` are used for ping-ponging.

Next, the CPU retrieves the sums and performs further precalculations. Nodes are placed at the grid points for which significant data is present at more than $t_2 = 85$ points in the pattern. The positions of the n resulting nodes are written to the `NodePositions` texture. For convenience, three types of information are recorded in the two columns of the texture. Stored in the first column for each node is its position as a location within the `DataReference` texture. In the second column, the position is expressed in millimeters. Additionally, the number of the grid point at which the node is located is recorded in the otherwise unused fourth component.

The number of the node present at each grid point is put into the texture map `PaddedGridPointNodes`. It is of type `Rect` with a single column and the rows referring to the grid points traversed first in x, then y, then z direction. As was done in section 5.2.2.6, the grid is padded with three additional points before and after it in each dimension. Due to the order in which the grid is traversed, the three padding points at the end of every row are immediately followed by those at the beginning of the next. The same is true for the three padding rows at the beginnings and ends of the slices. In the `PaddedGridPointNodes` texture, consecutive padding grid points are aliased to each other so that rows are separated by exactly three points and slices, by three rows of points. The padding before the first actual grid point and after the last is also omitted. This reduces the maximal number of points for which information must be recorded to $g_p = 8\,(8+3)\,(8+3) - 3\,(8+3) - 3$ while ensuring that there is enough padding before and after the grid in each dimension.

Two components are stored in `PaddedGridPointNodes` for every grid point. The first has a constant value of 0.5 and the second is the zero-based node number plus 0.5 or the constant -128 if no node is present. The components retrieved for a grid point may thus directly be used as indices into a `Rect` texture providing additional information about the node located at its position.

The last texture generated in this step is `NodeWeights`. It contains the elements of the $n \times n$ matrix $\mathbf{C}^{-1}$. They are obtained by constructing and inverting the matrix $\mathbf{C}$ as described in section 5.2.2.6. The wrap mode of this texture is set to *clamp to border* so that if the matrix element for an invalid node pair is requested, the background intensity, which is set to zero, is returned instead.

In the next precalculation step, the `CopyNodeWeight` shader is executed for every grid point and node combination. If a node is present at the grid point, the element of $\mathbf{C}^{-1}$ for the node pair is returned. Otherwise, a value of zero is used. The output of the shader is stored into the `PaddedGridPointNodeWeights` texture. It can later be used to retrieve the element of $\mathbf{C}^{-1}$ for a node pair if for one of the nodes, only the grid point at

which it is located is known. To generate its output, the shader uses the node number offset by 0.5 and the second component of `PaddedGridPointNodes` as indices into the `NodeWeights` texture. If a node is present at the grid point, the correct element of $\mathbf{C}^{-1}$ is retrieved. Otherwise, the value read from `PaddedGridPointNodes` is -128 so that a location outside of `NodeWeights` is queried which, due to its wrap mode, results in zero being returned. This entire step is only required if the derivatives of the reference image will later be queried. When only `SampleReference` and `SampleRegistered` will be called, it may be omitted.

The very last precalculation step is again executed on the CPU. In it, a random pattern of $N \times 2$ sampling points $\mathbf{x}'_u$ to be centered around every node is chosen and the precalculations possible for each point executed. The computations performed are identical to those described in sections 5.2.2.6 and 5.2.2.7. The positions of the sampling points are stored in two textures. In `SamplingPositionsReference`, they are scaled so they can directly be added to locations within the `DataReference` texture. In `SamplingPositionsRegistered`, the positions of the points are expressed in millimeters. The fourth component of the texture expresses the offset used to locate a corner of the relevant $3 \times 3 \times 3$ sub-grid. The 27 precalculated values of the radial basis functions for each of the $N \times 2$ points are recorded in the 3D texture `RBFs`.

UpdateCoefficients

Shown next in the diagram is the `UpdateCoefficients` subroutine. It updates the weights $\boldsymbol{\alpha}_i$ whenever the parameter vectors $\mathbf{q}_i$ at the nodes have changed by evaluating equation (5.17):

$$\mathbf{A} = \mathbf{C}^{-1}\mathbf{Q} \tag{6.10}$$

As defined in section 4.2.2, $\mathbf{A}$ and $\mathbf{Q}$ both have n rows and three columns. They can compactly be represented as texture maps with a single column and three components. The calculation that needs to be performed then is a matrix-vector multiplication, executed in parallel for all three components. The elements of $\mathbf{Q}$ are uploaded into the `Displacements` texture by the CPU. In contrast to section 5.2.2.6, they are not negated at this stage. Because the matrix $\mathbf{C}^{-1}$ and the vectors are too large to be handled natively by a GPU, the matrix-vector multiplication is split into two steps. First, the shader `WeightDisplacement` multiplies the elements of $\mathbf{C}^{-1}$ and $\mathbf{Q}$. Then, they are summed by `AddWeightedDisplacements` in a one-dimensional reduction operation using `CoefficientCalculation1` and `CoefficientCalculation2` for ping-ponging. As every step halves the number of rows, reduction is applied to the whole texture whose height n_{pot} is guaranteed to be a power of two. Rows not containing weighted displacements are set to zero during precalculation to not corrupt the results.

The calculated weights are found in the first column of the ping-pongig texture last written to. They are retrieved by the `CopyCoefficient` shader and stored into the `Coefficients` texture. Using the same mechanism as described for the `CopyNodeWeight` shader, a location outside the piing-ponging texture is accessed and zero weights are recorded whenever no node is present at a point.

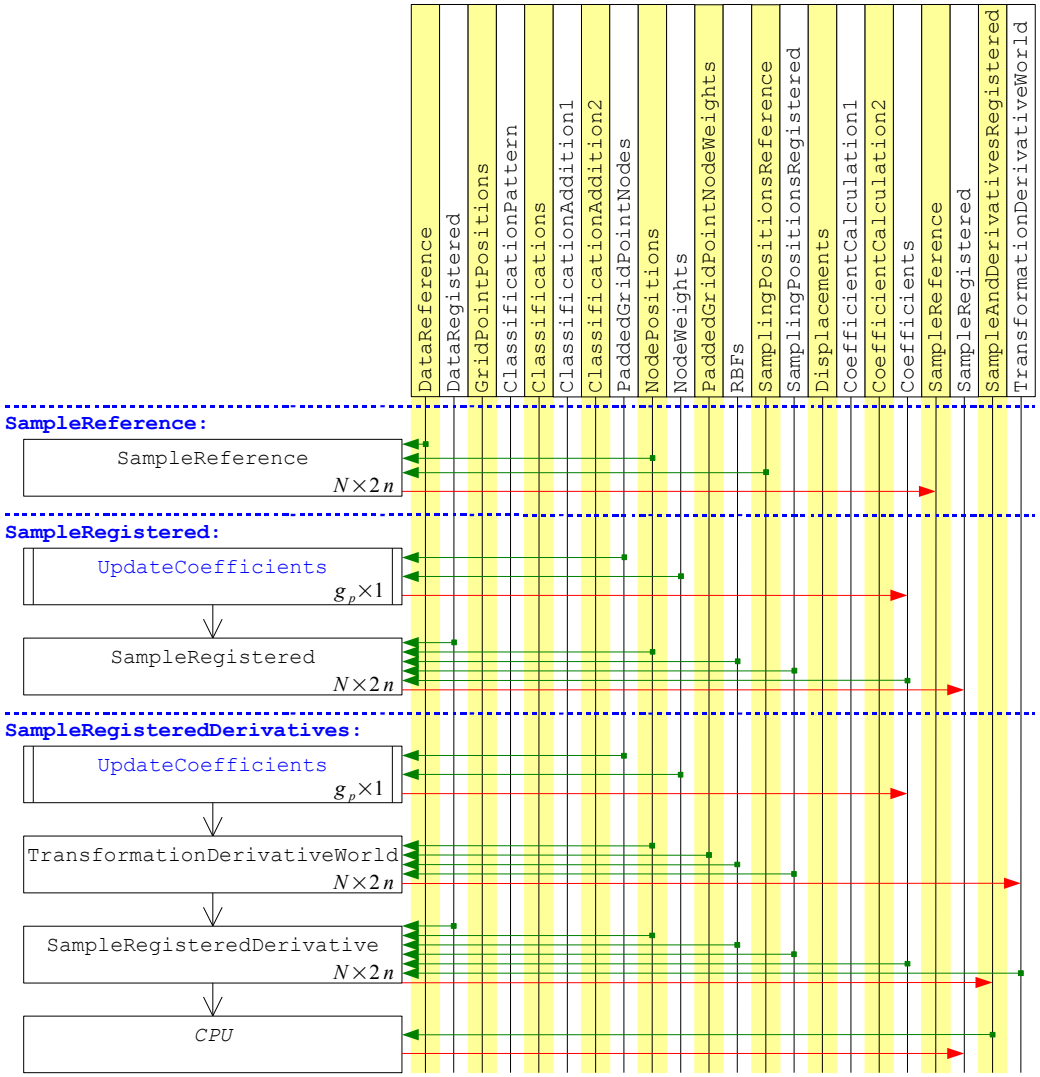


Figure 6.8: Control and data flow: Non-rigid sampling, monolithic shader implementation (continued)

SampleReference

The three sampling methods are described next. `SampleReference` is very simple, consisting of a single rendering pass. For each combination of a node $\mathbf{p}_i$ and a sampling point $\mathbf{x}'_u$, the `SampleReference` shader calculates $\mathbf{x}_u = \mathbf{p}_i + \mathbf{x}'_u$. Because the implicit conversion from millimeters to voxel coordinates has already been applied to the representations of $\mathbf{p}_i$ and $\mathbf{x}'_u$ read from the input textures, the resulting point $\mathbf{x}_u$ can directly be used to look up the reference image intensity in `DataReference`. Before it is returned, the intensity is multiplied by 255 to undo the rescaling applied by the GPU.

SampleRegistered

When sampling the registered image, the `Coefficients` texture is first updated by calling the `UpdateCoefficients` subroutine. Then, the `SampleRegistered` shader is executed for every node $\mathbf{p}_i$ and sampling point combination $\mathbf{x}'_u$. Implemented in this shader is the entire evaluation of the transformation function of equation (5.16) with additional negations to invert the directions of the displacements and match the CPU based implementation of section 5.2.2.6:

$$\mathbf{x}_v = \mathbf{M}_T \begin{pmatrix} \mathbf{x}_u + \sum_{j=1}^{n} -\boldsymbol{\alpha}_j R_a \left(\|\mathbf{x}_u - \mathbf{p}_j\| \right) \\ 1 \end{pmatrix} \tag{6.11}$$

All optimizations described in section 5.2.2.6 are used. The positions of the node and the sampling point expressed in millimeters are added, yielding $\mathbf{x}_u$ in the first three components and the number of the grid point at a corner of the relevant $3 \times 3 \times 3$ sub-grid in the fourth.

Then, the 27 relevant grid points are traversed in an unrolled loop that spans several dozen lines of shader source code. For every grid point, the product of the weights $\boldsymbol{\alpha}_j$ read from `Coefficients` texture and the precalculated value of the radial basis function found in `RBFs` is subtracted from $\mathbf{x}_u$. Because the grid is padded and weights of zero have been recorded for the grid points at which no nodes are located, the multiplication and addition may be performed without checking whether the grid point actually exists and a node is present at its location.

After all relevant grid points have been considered, the calculated vector is extended to a homogenous coordinate representation and the rigid transformation applied. Instead of $\mathbf{M}_T$, the matrix $\mathbf{M}'_T$ is used, which is constructed as described in section 6.3.2 and integrates the implicit conversion to voxel coordinates. The registered image intensity is then looked up at the resulting point in the `DataRegistered` texture.

SampleRegisteredDerivatives

The sampling of the registered image and its derivatives is split into four steps. After `UpdateCoefficients` has been called to update the `Coefficients` texture, the shader `TransformationDerivativeWorld` is executed for every node $\mathbf{p}_i$ and sampling point $\mathbf{x}'_u$

pair. It calculates the derivatives of the transformation function with respect to the three components of the parameter vector $\mathbf{q}_i$, as defined in equation (5.20):

$$\frac{d}{d\mathbf{q}_i} T(\mathbf{x}_u) = \mathbf{M}_T \sum_{j=1}^{n} - \left(\mathbf{C}^{-1}\right)_{ji} R_a(\|\mathbf{x}_u - \mathbf{p}_j\|) \tag{6.12}$$

Only the sum is actually evaluated and the multiplication with $\mathbf{M}_T$ deferred until later. This way, its output is not a matrix that would require multiple render targets and three textures to record but a scalar value that can be stored in the single component of `TransformationDerivativeWorld`. The summation is again unrolled in the shader source code. $\mathbf{x}_u$ and a corner of the relevant sub-grid are determined in the same way as done by `SampleRegistered`.

Then, the 27 relevant grid points are traversed. For each grid point at which a node $\mathbf{p}_j$ is located, $\left(\mathbf{C}^{-1}\right)_{ji}$ is retrieved and multiplied with the precalculated value of the radial basis function from `RBFs`. Because the grid is padded and the elements of $\mathbf{C}^{-1}$ have been copied into `PaddedGridPointNodeWeights` such that a value of zero is obtained for points at which no node is found, the texture access and multiplication may be performed without further checks. The output of the shader is the negated sum of the 27 products.

The next shader, `SampleRegisteredDerivative`, uses these results to calculate the desired derivatives of the registered image intensities with respect to the transformation parameters, as defined in equation (5.21):

$$\frac{d}{d\mathbf{q}_i} v(T(\mathbf{x}_u)) = \operatorname{grad} v(T(\mathbf{x}_u))\, \mathbf{M}_T \sum_{j=1}^{n} - \left(\mathbf{C}^{-1}\right)_{ji} R_a(\|\mathbf{x}_u - \mathbf{p}_j\|) \tag{6.13}$$

First, calculations identical to those found in the `SampleRegistered` shader and described above are performed to determine $\mathbf{x}_u$ and apply the transformation function and the implicit conversion to it. Then, the registered image intensities are looked up at the resulting position and three neighboring points to estimate the gradient according to equation (5.14). The gradient is then multiplied by the matrix $\mathbf{M}_T$ and the sum precalculated for the current node and sampling point combination.

The shader writes four values into the `SampleAndDerivativesRegistered` texture, the first of which is the registered image intensity at the transformed point and the other three, its derivatives with respect to the three components of the displacement vector $\mathbf{q}_i$. As noted in section 6.3.2, it is a waste of memory bandwidth to read back four components also for those subsequent calculations that only require the registered image intensity. Thus, in the last step, the intensities are copied into the `SampleRegistered` texture.

6.3.5.2. Multiple Shaders

In an attempt to accelerate the evaluation of the non-rigid transformation function, an alternative implementation has been developed. As explained in section 6.2.2, a

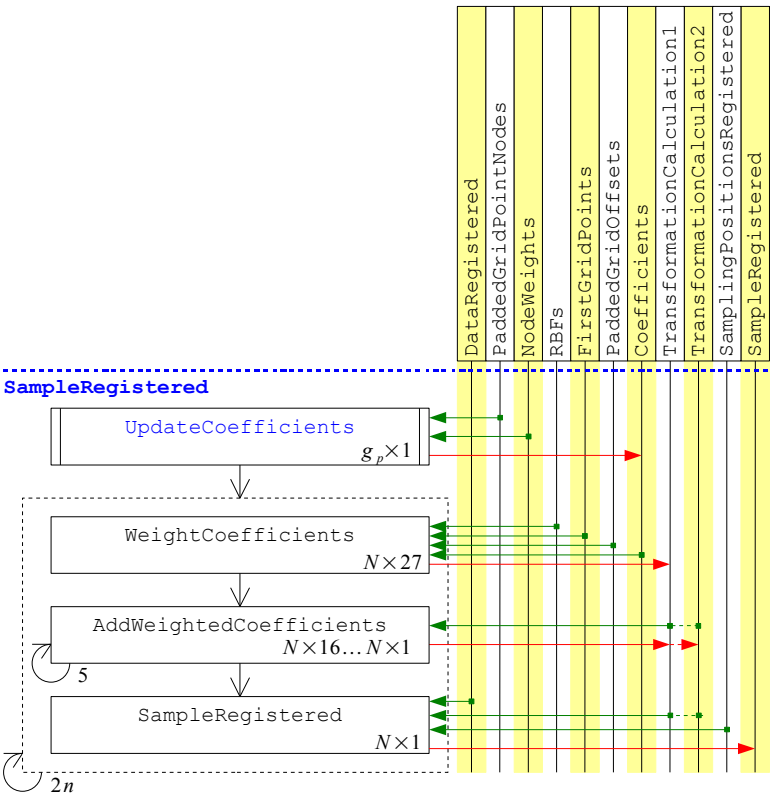


Figure 6.9: Control and data flow: Non-rigid sampling, multiple shader implementation

GPU achieves high performance when calculations are executed in parallel for a large number of fragments. The single monolithic shader that calculates the transformation is therefore replaced by a series of smaller shaders which produce a significantly larger number of intermediary values that are then summed in reduction operations. Shown in this section is only the method for sampling the registered image. Precalculation and coefficient update are largely unchanged and where changes are necessary, they are noted in the following explanation.

After the `Coefficients` texture has been updated via the `UpdateCoefficients` method, the transformation function is evaluated in three steps. Because large textures are required to hold intermediary results, not all node and sampling point combinations can be considered at once. Instead, a single node $\mathbf{p}_i$ and the N sampling points $\mathbf{x}'_u$ for one sample are used and the entire process is repeated $2n$ times, covering S_A and S_B for all n nodes.

The first shader, `WeightCoefficients`, calculates the addends in equation (6.11). As only $3 \times 3 \times 3$ grid points are relevant to the transformation of a sampling point, the shader is invoked 27 times for each $\mathbf{x}'_u$. A corner of the relevant sub-grid is located by adding two components. The first is the number of the grid point at which $\mathbf{p}_i$ is located. Since this value is constant throughout the rendering pass, it is passed to the shader as a `uniform` variable. Second is the offset from $\mathbf{p}_i$ to the corner, stored in `FirstGridPoints` for all $\mathbf{x}'_u$ during precalculation.

To traverse all 27 relevant grid points, a different offset from the first point is used for generating each row of the $N \times 27$ texel output texture. The offsets from the first grid point to itself and the 26 others are stored in the 1D texture `PaddedGridOffsets`, also generated during precalculation. After the addends have been determined, they must be summed for each $\mathbf{x}'_u$. This is done by a vertical reduction using the shader `AddWeightedCoefficients`.

The `SampleRegistered` shader uses the precalculated sums to evaluate the remainder of equation (6.11) and obtain the desired intensity from the `DataRegistered` texture. $\mathbf{x}_u$ is calculated as $\mathbf{x}_u = \mathbf{p}_i + \mathbf{x}'_u$, the precalculated sum subtracted, the result extended to homogenous coordinates and a matrix-vector multiplication with $\mathbf{M}'_T$ performed. Finally, the registered image intensity at the calculated point is looked up.

Save for imprecisions due to the different order in which the calculations are executed, the image positions calculated by this approach are identical to those obtained from the monolithic shader.

6.3.6. Non-Rigid Mutual Information Derivative

This section addresses the estimation of the derivatives of local mutual information. The calculation steps are analogous to those described in section 6.3.3 with few small changes. Except for `WeightedDerivatives`, all shaders are identical. During initialization and precalculation, the only difference is that the `SampleReference` method which uses the non-rigid transformation function is called.

In the `Calculation` method, the registered image intensities and their derivatives are also sampled using the non-rigid transformation function. Because local mutual

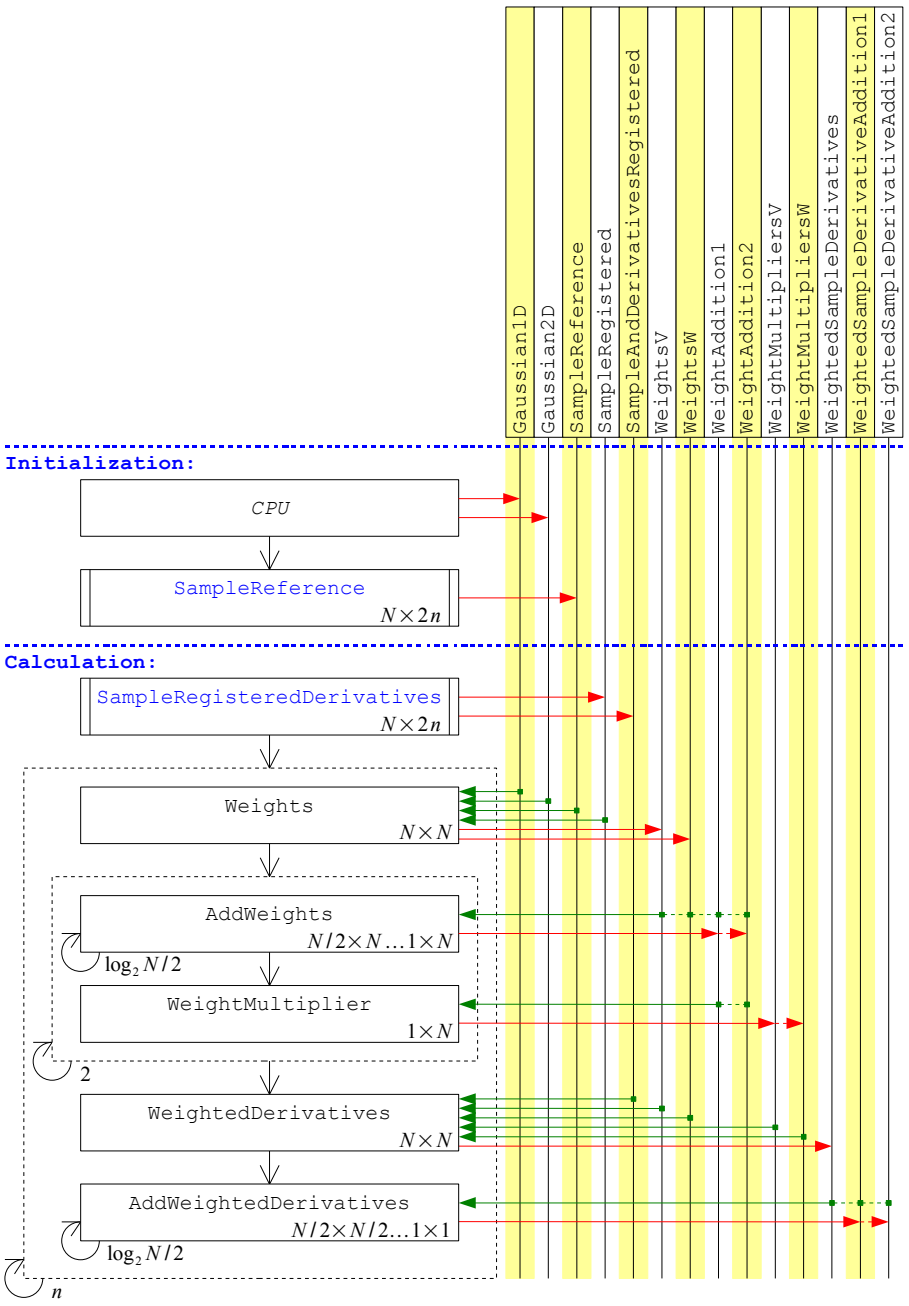


Figure 6.10: Control and data flow: Non-rigid MI derivative calculation

information at each node $\mathbf{p}_i$ depends only on three parameters, the intensities and their derivatives with respect to the parameters are returned as the four components of a single texture. The calculation that follows is repeated n times, once for each node $\mathbf{p}_i$ using the samples obtained by centering the $\mathbf{x}'_u$ around its location.

The steps performed to determine the numerators and denominators of the W_X and $W_{X,Y}$ are completely identical to those of section 6.3.3. The `WeightedDerivatives` shader that is then executed to calculate the addends of equation (6.6) is very similar to its counterpart in that section but simplified because once the part of the addend that does not depend on p has been obtained, it is multiplied only with the difference of intensity derivatives retrieved from a one texture, not from two. The result is also output into a single texture, `WeightedSampleDerivatives`

The shader and steps performed to sum the addends via a two-dimensional reduction are again identical to section 6.3.3. However, because all derivatives may be found in one texture, the reduction needs to be executed only once. The three resulting derivatives are retrieved by the CPU from the ping-ponging texture last written to and the entire calculation is repeated for the next node.

6.3.7. Non-Rigid Mutual Information

The calculation of local mutual information for a node $\mathbf{p}_i$ is completely identical to the determination of mutual information in the rigid case. The calls to rigid sampling methods are simply replaced with their the non-rigid versions and all calculations repeated n times. This results in the precalculation of a reference image entropy for each node and, during the calculation phase, the determination of all n local mutual information measures.

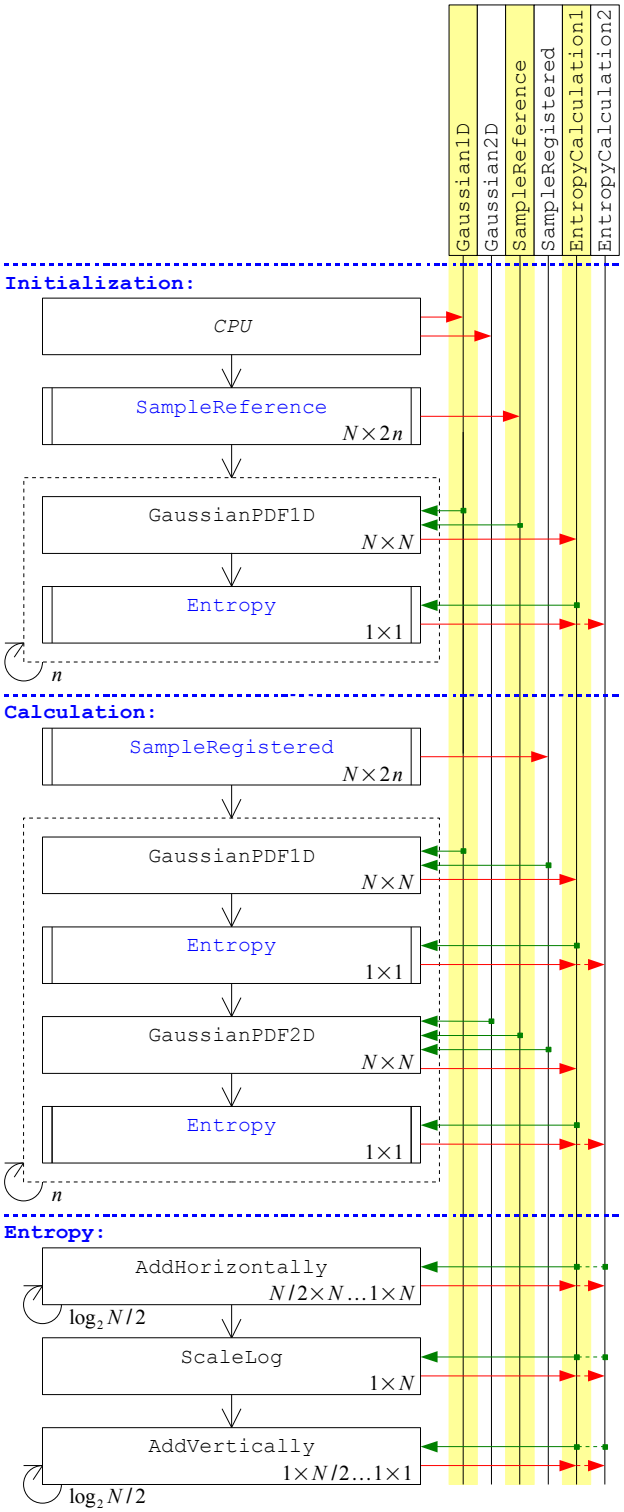


Figure 6.11: Control and data flow: Non-rigid MI calculation

7. Results

The actual performance of the registration system is analyzed in this chapter based on a range of experiments. First, mutual information is considered on its own and the question answered whether it is a suitable similarity metric for multi-modal tomographic datasets. Then, the entire registration system is evaluated. Assessed are both registration quality and speed, including the accelerations possible by using GPGPU techniques.

To obtain realistic results, DICOM images of actual tomographic scans are used in all experiments. Unfortunately, access to such images has proven to be very difficult and only a single pair could be located. Obtained from the website of the OsiriX DICOM viewer[1], the images are used here with permission. Each consists of 83 slices showing transversal cuts through the head and upper thorax regions of a cancer patient. The anatomical dataset is a CT scan with slice resolutions of 512×512 pixels. Functional information is provided by a PET scan whose slices have resolutions of 128×128. Cuts through the two datasets illustrating the node arrangements generated for them are shown in figure 4.2. Because the PET scan covers a smaller region of the patient's body and the nodes are more concentrated, allowing for finer adjustments, it is used as the reference image.

A second pair of datasets has been constructed by using the CT scan as both reference and registered image. This allows a number of additional measurements to be conducted. First, by considering both the PET-CT and CT-CT cases, the statistical basis of all experiments can be increased. Second, the node arrangements generated for the CT dataset can be tested. Third, because the optimal alignment of an image with itself is exactly known, the registration quality achieved in the CT-CT case can precisely be measured.

7.1. Mutual Information

The first aspect of the registration system analyzed is its similarity metric. Only if mutual information provides a reliable assessment of alignment quality and its derivatives have the correct signs is registration possible. The estimates of mutual information and its derivatives calculated by equations (2.41) and (2.46) depend not only on the image data and transformation function, but also on the sample sizes and Parzen window variances. To reduce the number of parameters whose values may have to be adjusted, only one sample size $N = N_A = N_B$ and one variance $\sigma^2 = \sigma_X^2 = \sigma_Y^2 = \sigma_{X,X}^2 = \sigma_{Y,Y}^2$ are used. As noted in section 2.6.3, the Parzen window technique is relatively insensitive to the precise values of the variances, allowing for this simplification.

[1] `http://homepage.mac.com/rossetantoine/osirix/`

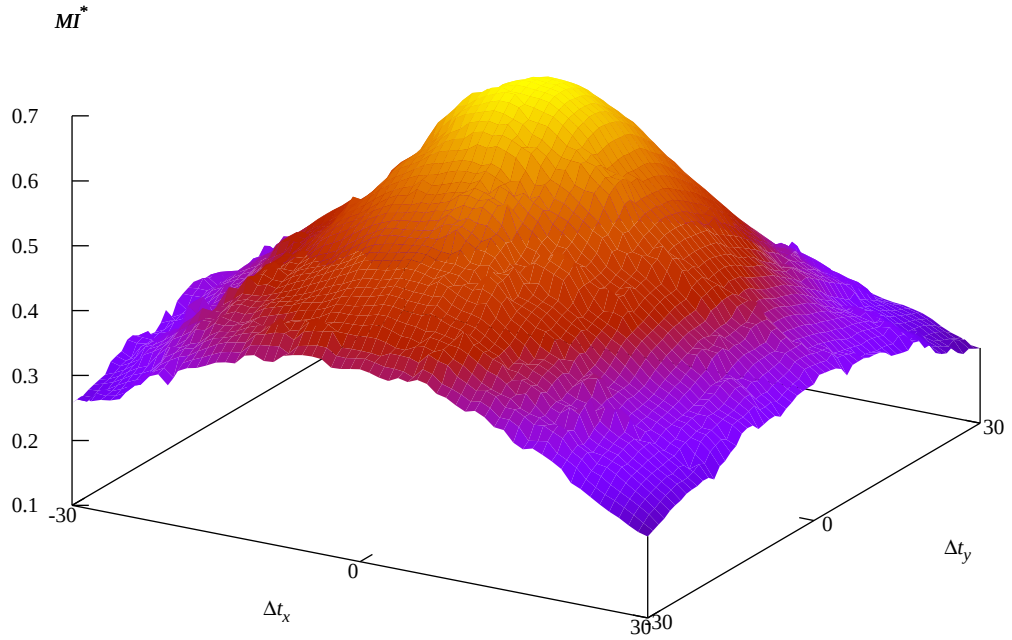


Figure 7.1: $MI^\star$ for different PET-CT alignments

Shown in figure 7.1 is a typical plot of $MI^\star$. It is obtained for the PET-CT case at pyramid level $l = 3$ using $N = 2048$ and $\sigma^2 = 30$. The center of the plot corresponds to a manually determined optimal alignment. Along the x and y axes, the reference image is translated by up to 30 millimeters. Plotted in figure 7.2 for the same data and parameters is $\frac{d}{dt_x} MI^\star$, the derivative of $MI^\star$ with respect to the translation in x direction. The most important observation to be made from these plots is that $MI^\star$ is a viable similarity metric for multi-modal tomographic datasets. As can be seen in figure 7.1, the value of mutual information increases with alignment quality and reaches its maximum at the optimal alignment. However, also apparent is the random noise that corrupts the smoothness of the surface.

More relevant in the context of registration is figure 7.2. The estimated value of $\frac{d}{dt_x} MI^\star$ is plotted here in green where it is positive and in red otherwise. It is apparent that regardless of the value of t_y, the derivative with respect to t_x has the correct sign. When the registered image is translated too far to the left, $\frac{d}{dt_x} MI^\star > 0$ and for a translation to the right, $\frac{d}{dt_x} MI^\star < 0$. Stochastic gradient ascent, which modifies t_x by adding a value proportional to the derivative, thus adjusts the translation toward optimal alignment. Similar plots are obtained for other pyramid levels and combinations of transformation parameters, indicating the suitability of the metric for their adjustment. It should be noted that the noise exhibited by the derivatives is larger than that of $MI^\star$ itself.

How the estimates are influenced by N and σ^2 is addressed in figure 7.3. The plots shown here have been generated for the same image data and range of misalignments

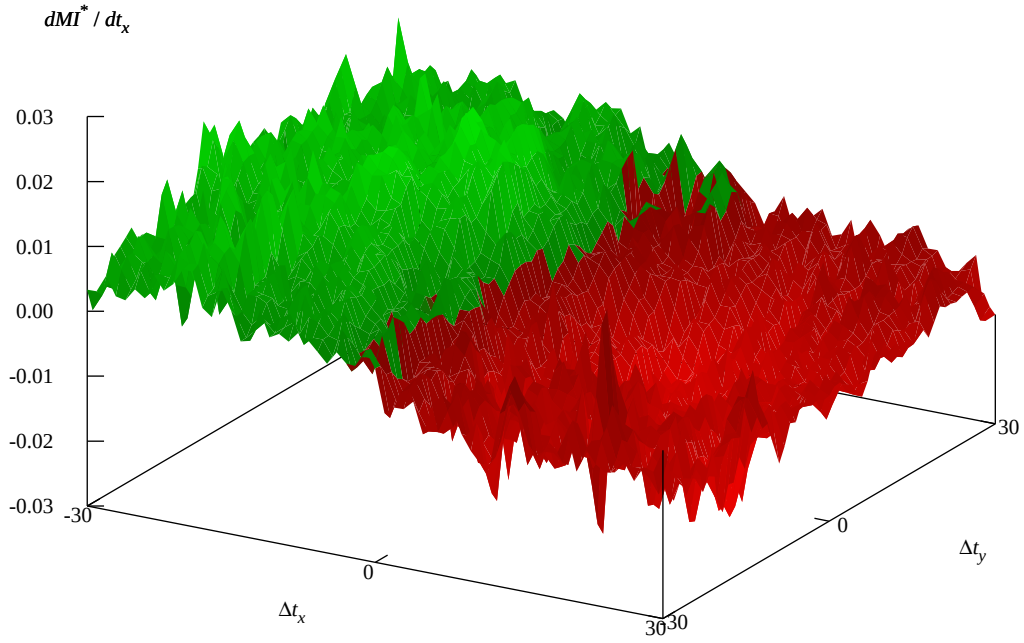
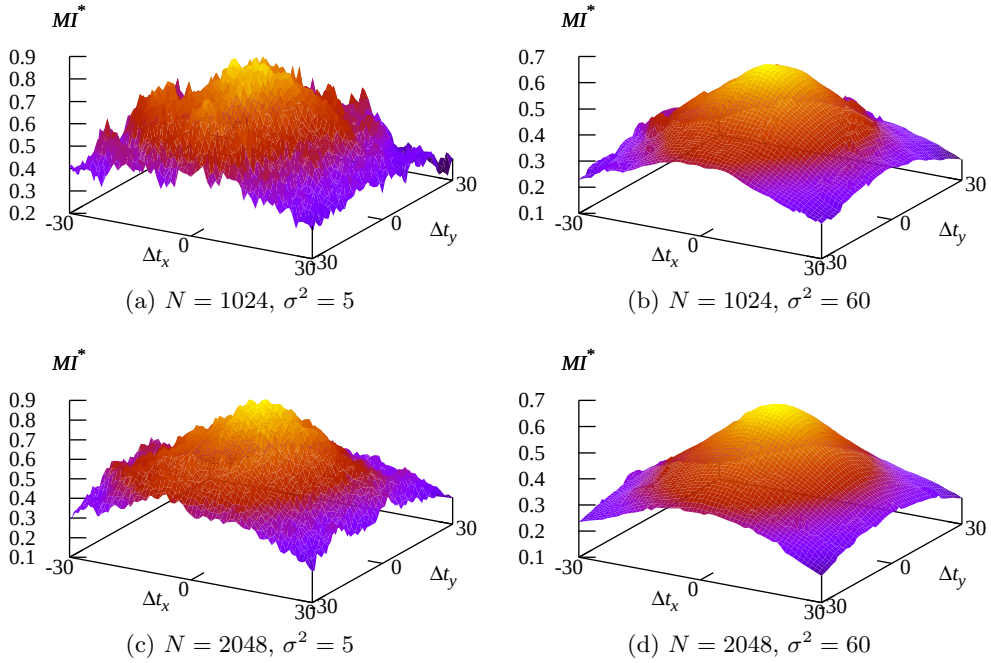


Figure 7.2: $\frac{d}{dt_x} MI^\star$ for different PET-CT alignments

but different values of N and σ^2. From the first row to the second, the sample size is doubled. As is to be expected, the larger amount of information gathered for the images leads to a higher quality and less noisy metric. Noise can also be reduced by increasing σ^2, as illustrated by the difference between the two columns. This leads to a smoothing that removes extraneous spikes but also flattens and blurs the desired maximum. While it is important to tune these parameters so that a smooth and sufficiently steep surface is obtained, it can be concluded that in general, $MI^\star$ truly is a non-parametric measure of alignment quality. Adjustments to N and σ^2 can make the surface easier to work with, but they do not change its principal shape or the location of the maximum.

Based on the tomographic datasets available, the parameter values chosen for the rigid registration pass are $N = 2048$ and $\sigma^2 = 30$. Because the multiresolution approach of section 3.3 is used, the effect of these settings is different at each pyramid level. For $l = 4$, the first level considered, the sample size is relatively large. This leads to a very smooth, high quality metric that can correctly assess substantial misalignments. As registration progresses and higher resolution images are used, the relative sample size decreases and the metric becomes more noisy. While this limits the range of misalignments for which a reliable assessment is possible, the alignment is continuously improved and expected to be close to optimal when the later pyramid levels are reached.

Only at $l = 0$ is the noise so substantial that a sample size beyond 2048 would have been desirable. Unfortunately, due to the NVIDIA driver bug mentioned in section 6.2.2, the maximal texture size that can effectively be used in GPU based calculations is 2048×2048. All rendering passes working on the entire samples would thus have to



(a) $N = 1024$, $\sigma^2 = 5$ (b) $N = 1024$, $\sigma^2 = 60$

(c) $N = 2048$, $\sigma^2 = 5$ (d) $N = 2048$, $\sigma^2 = 60$

Figure 7.3: $MI^\star$ for different PET-CT alignments and N, σ^2 parameter values

be split into multiple smaller steps processing subsets of the data if N was larger than 2048. Since this would have caused considerable additional development effort for little benefit, $N = 2048$ is used instead and the noise accepted.

For the non-rigid pass, the sample size can be decreased as mutual information is calculated only locally for small image regions. A value found to work well is $N = 512$. However, the noise in the estimated derivatives with respect to the displacements $(\mathbf{q}_i)_k$ is much larger than for the parameters of the rigid transformation function. σ^2 is therefore increased to 480, providing additional smoothing.

It is interesting to note that $N = 50$, as proposed in [Vio95], is not sufficient for any pyramid level, not even $l = 4$ containing the lowest resolution images. When the amount of information gathered is this small, the measure is dominated by noise and no amount of smoothing will allow the actual maximum to be located. The smallest value for which a reasonable estimate of mutual information can be obtained at $l = 4$ is $N = 256$. Due to the larger amount of noise, usable estimates for the derivatives are not possible below $N = 512$.

7.2. Registration System

The registration system uses a number of constants for which values must be chosen before its performance can be assessed. As defined in section 5.2.2.1, these are r_1, r_2, r_3, m_1, m_2, m_3, s_1 and s_2. The constants control the step sizes and the number of steps executed. It is difficult to determine universally valid values form just two datasets. By

optimizing the step sizes for the PET-CT and CT-CT scenarios, it is possible to obtain perfect registration in a single step per pyramid level. However, this would lead to a completely unrealistic situation and skew the results. Values less precisely tuned to the two specific datasets are used instead.

The number of steps is set to $s_1 = 10$ per pyramid level for the rigid and to $s_2 = 50$ for the non-rigid pass. These values are arbitrary but are believed to represent a realistically long registration process. The step sizes are set so that large adjustments are possible at $l = 4$ and only small corrections at $l = 0$. If only one step size was used, rigid registration would strongly favor rotation over translation. This is because an increase by one unit in t_x, t_y or t_z moves the registered image by just a single millimeter while the same adjustment applied to any of the angles rotates it by a whole radian, or about 57°. The derivatives of $MI^\star$ with respect to the rotation parameters are thus much greater in magnitude than those for the translations. To compensate for this effect, the rotation step size λ_r is set to a thousandth of λ_t, the step size for translation. Their initial values are $r_1 = 2000$ and $r_2 = 2$, the factors by which they are multiplied after each step, $m_1 = m_2 = 0.84$. In the non-rigid pass, an initial step size of $r_3 = 400$ and the multiplier $m_3 = 0.97$ are used.

7.2.1. Quality

The quality of alignment achieved by the registration system is evaluated separately for the rigid and non-rigid passes.

7.2.1.1. Rigid

Four scenarios are used to assess the quality obtained after 50 steps of rigid registration. The transformation parameters before and after registration are summarized in tables 7.1 and 7.2, exemplary cuts through the alignments shown in figures 7.4 to 7.6. The first scenario analyzed is most realistic and closest to an actual clinical application. Using the registration system exactly as it was designed, PET and CT images are loaded and registration is started. No manual adjustments are applied so that the initial alignment is determined solely by the registration system, which, according to section 5.2.2.1, chooses the translation that makes the centers of the two images coincide. In the other three scenarios, the initial alignment is set manually to determine the robustness of the system at different magnitudes of misalignment. Because no radiologist was available to evaluate the results, the medical correctness of the PET-CT alignments could not be verified. To nevertheless obtain an objective measure of alignment quality, two CT-CT registration scenarios are used for which the optimum is known to be the identity transform.

Figure 7.4 illustrates the situations before and after registration for PET-CT with automatic initial alignment. The differences are small but important. The blue reference image remains unchanged while the placement of the red registered image is adjusted. As can be seen in (a), there is a bright spot of activity outside the body. In (b), the spot coincides with a lump in the patient's body, a plausible location. In (c), the brain is not properly aligned with the skull. Brain tissue can clearly be seen to intersect with

Automatic Initial Alignment						
	t_x	t_y	t_z	α	β	γ
Before	−118.201	−118.201	−1.037	0	0	0
After	−113.013	−122.154	−8.734	−0.004	−0.002	0.001
Artificial Misalignment						
	t_x	t_y	t_z	α	β	γ
Before	−80	−80	−20	0.2	0.2	0.2
After	−113.255	−121.211	−10.191	−0.004	0.007	−0.027

Table 7.1: Transformation parameters before and after PET-CT registration for different initial alignments

Experiment 1						
	t_x	t_y	t_z	α	β	γ
Before	−30	−30	−30	0.2	0.2	0.2
After	−0.257	−0.069	−1.539	0	0.003	−0.001
Experiment 2						
	t_x	t_y	t_z	α	β	γ
Before	30	30	30	−0.2	−0.2	−0.2
After	−0.390	−0.267	1.143	0.002	−0.006	0.001

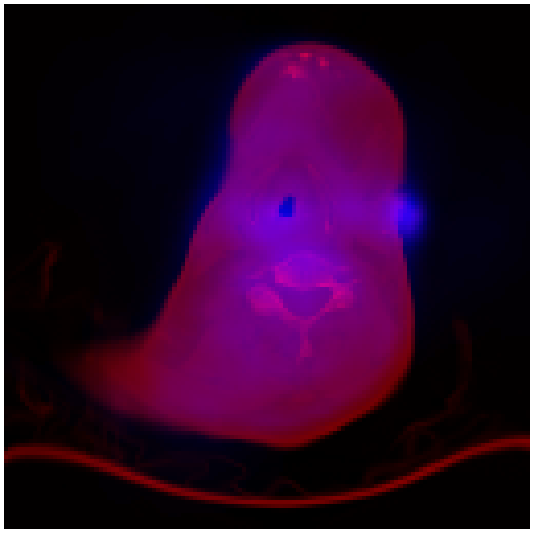
Table 7.2: Transformation parameters before and after CT-CT registration for different initial misalignments

parietal and petrosal bones. In (d), the position of the skull is adjusted so that the brain does not intersect with it. The values of the six transformation parameters before and after registration are given in the first two rows of table 7.1.

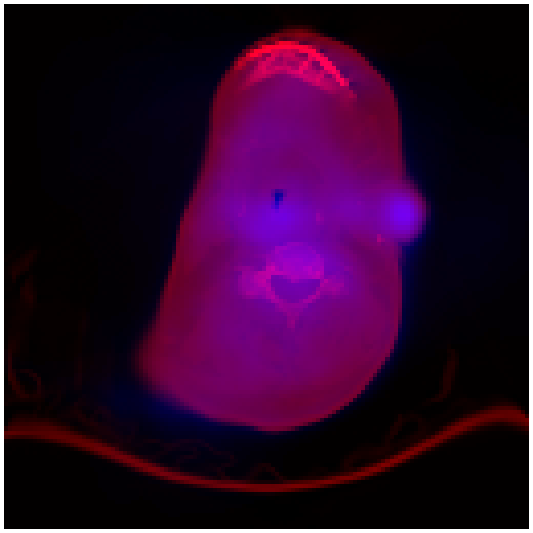
In the second scenario, the CT image is translated by up to 40 millimeters per coordinate from the previously found alignment and rotated by 0.2 radians, or about 11°, around every axis to evaluate the ability of the registration system to cope with larger initial misalignments. The situations before and after registration are illustrated in figure 7.5 and expressed in numbers in the last two rows of table 7.1. As can be seen both in the images and the table, the resulting alignment is almost identical to that found for the first registration scenario. The maximal deviations in parameter values are approximately 1.5 millimeters and 1.5°.

To obtain a broader statistical basis and precisely measure the quality of alignment, CT-CT registration is performed in the next two experiments. The parameter values before and after registration are given in table 7.2 and the corresponding alignments illustrated in figure 7.6. It can be seen that despite the large initial misalignments, the images are reliably registered to within at most approximately 1.5 millimeters and 0.006 radians, or 0.3°, of the optimal alignment. In figure 7.6, the blue and red images blend into almost uniformly purple colored shapes.

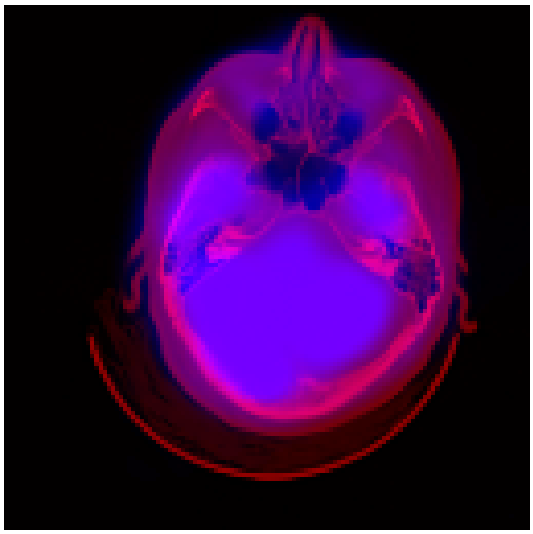
It can be concluded from these experiments that using the same set of constants



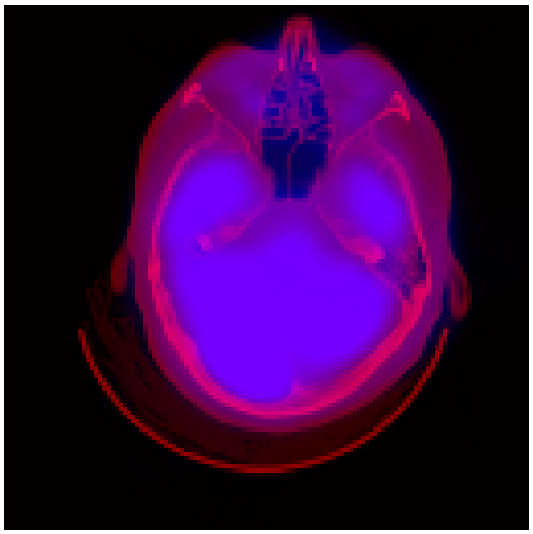
(a) Reference slice 32 before registration



(b) Reference slice 32 after registration

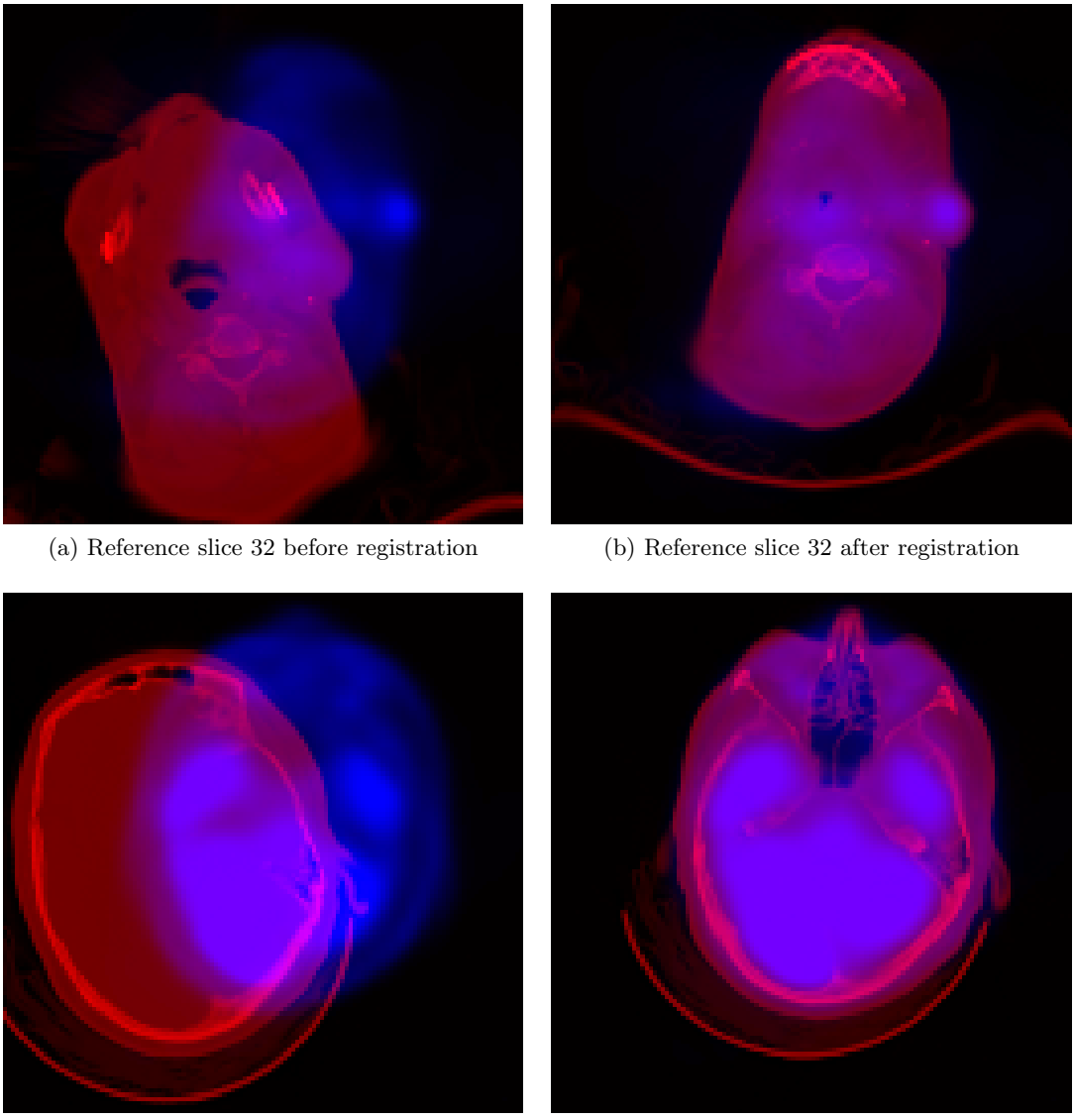


(c) Reference slice 64 before registration



(d) Reference slice 64 after registration

Figure 7.4: Cuts through PET-CT alignment before and after rigid registration with automatic initial alignment: reference image (blue); registered image (red)



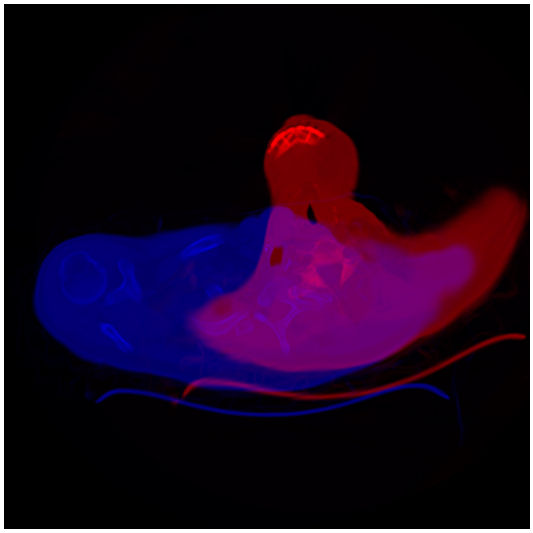
(a) Reference slice 32 before registration

(b) Reference slice 32 after registration

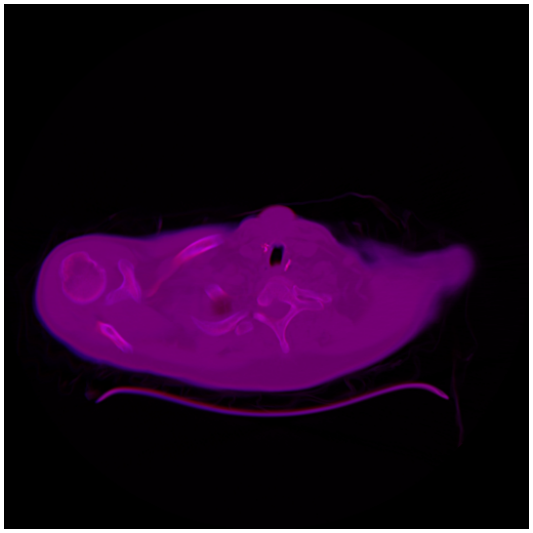
(c) Reference slice 64 before registration

(d) Reference slice 64 after registration

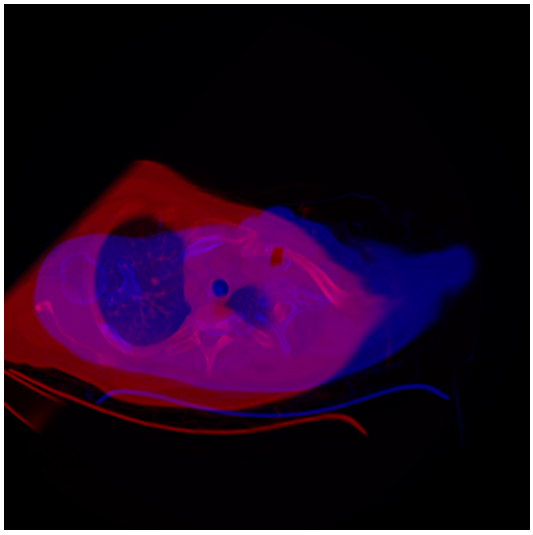
Figure 7.5: Cuts through PET-CT alignment before and after rigid registration with artificial initial misalignment: reference image (blue); registered image (red)



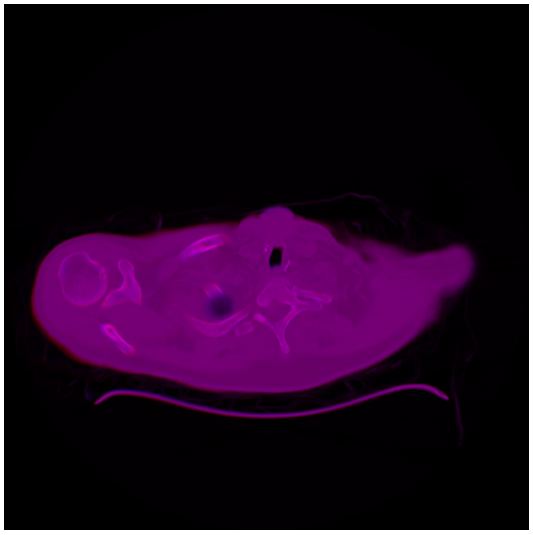
(a) Experiment 1 before registration



(b) Experiment 2 after registration



(c) Experiment 2 before registration



(d) Experiment 2 after registration

Figure 7.6: Cuts through CT-CT alignments before and after rigid registration for two scenarios with different artificial initial misalignments: reference image (blue); registered image (red)

and without manual intervention, rigid registration is able to reliably undo a range of misalignments for different pairs of images. While the statistical basis is not large and the medical correctness of the PET-CT alignments has not been verified, the plausible results obtained in all cases make it appear likely that high quality alignments can be produced for a wide range of tomographic datasets.

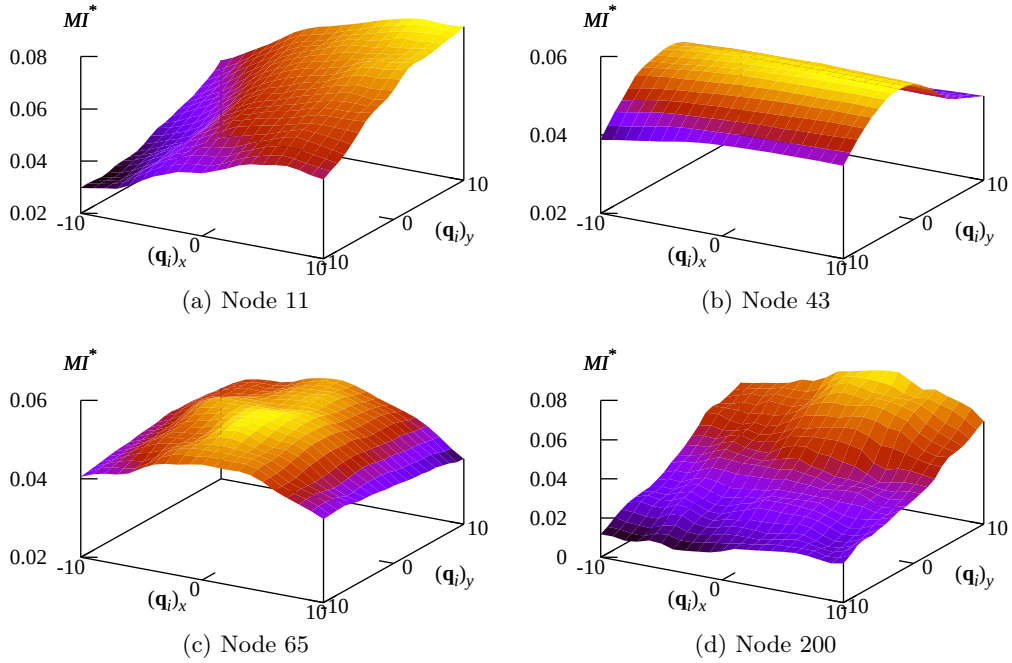
7.2.1.2. Non-Rigid

The most realistic scenario used again is PET-CT. After the images have been rigidly aligned, 50 steps of non-rigid registration are performed. As specified in section 4.2.6, the maximal displacement that can be applied per node and coordinate is $0.2r$, with r the influence radius of the radial basis functions. When the PET scan is used as the reference image, this corresponds to just 10.5 millimeters. Because the differences between the alignments before and after registration are in this small range and their correctness again cannot be assessed without the help of a radiologist, no cuts through the images are shown. Exemplary plots of $MI^\star$ for several nodes are presented instead, illustrating some of the problems encountered during the non-rigid pass.

Three CT-CT scenarios are employed next to objectively measure the ability of non-rigid registration to undo complex misalignments. In each case, the rigid transformation parameters are set to zero and initial displacements introduced at the nodes. Then, 50 steps of non-rigid registration are executed and the degree to which the displacements are corrected determined.

Before the CT-CT results are considered in more detail, the PET-CT scenario is addressed. Shown in figure 7.7 are plots of the estimated local mutual information over the displacements in x and y directions at four different nodes. Because a constant displacement in z direction of zero is used, the maximums seen in these plots do not have to correspond to the absolutely maximal values of $MI^\star$ in the vicinity. Nevertheless, the images are able to illustrate tendencies and problems. A desirable case is shown in (c). There is a clear maximum, offset several millimeters from the null displacement. This is a situation in which stochastic gradient ascent can locate the optimal alignment. The maximum in (b) is more difficult to locate precisely because the area around it is flat and protruded. A plot of this shape is obtained when the region around a node is dominated by a straight, elongated structure such as a large bone or the border between patient body and background intensity. Translations of the registered image along the structure lead to little visible change and have only small influence on the value of mutual information, resulting in a relatively flat mutual information surface.

Examples of another frequent problem are seen in (a) and (d). Mutual information continuously increases in some direction of the parameter plane and it is obvious that a higher value of $MI^\star$ could be achieved if the displacement per coordinate was not limited to 10.5 millimeters. Unfortunately, the limitation cannot be waived because it is required to guarantee that the topology of the registered image is preserved, as explained in section 4.2.6. There are two ways in which the maximum of mutual information could be reached. One is to position the nodes further apart, which increases the maximal permissible displacements. However, this would lead to fewer nodes being



(a) Node 11 (b) Node 43

(c) Node 65 (d) Node 200

Figure 7.7: Local $MI^\star$ for different PET-CT nodes and alignments

generated, leading to less precise control over the non-rigid transformation. The other option is to use a more sophisticated non-rigid transformation function that allows larger displacements without reducing the number of parameters.

Another important reason to reconsider the transformation is the lack of a notion of bones. Because the entire registered image is treated as non-rigid, the deformation applied to it is based solely on the estimated derivatives of $MI^\star$. This can result in a medically incorrect transformation where rigid bones are bent, stretched and arbitrarily deformed instead of altering only the shapes of internal organs. To provide a distinction between soft tissue and bones, a model based approach is required that segments and classifies image regions first and then refrains from deforming those considered rigid.

While the results of PET-CT registration cannot be precisely evaluated and their medical plausibility is in question, an exact assessment is possible for CT-CT. The three scenarios employed differ in the initial misalignments introduced at the nodes. First, the displacements are set to $(7,7,7)^T$ millimeters for every node. In the second scenario, displacements are alternatingly set to $(7,7,7)^T$ and $(-7,-7,-7)^T$. Finally, for the last scenario, $(-7,-7,-7)^T$ is used throughout. Because the two identical images are perfectly aligned when all displacements are zero, the mean square error of the $(\mathbf{q}_i)_k$ at the beginning of the non-rigid registration pass is $7^2 = (-7)^2 = 49$. How it changes over the course of the 50 registration steps can be seen in figure 7.8

It is apparent that for all three scenarios, the error decreases quickly at first and then asymptotically approaches a constant value. The number of steps $s_2 = 50$ has been chosen so that calculations are halted and registration is finished when the rate at which

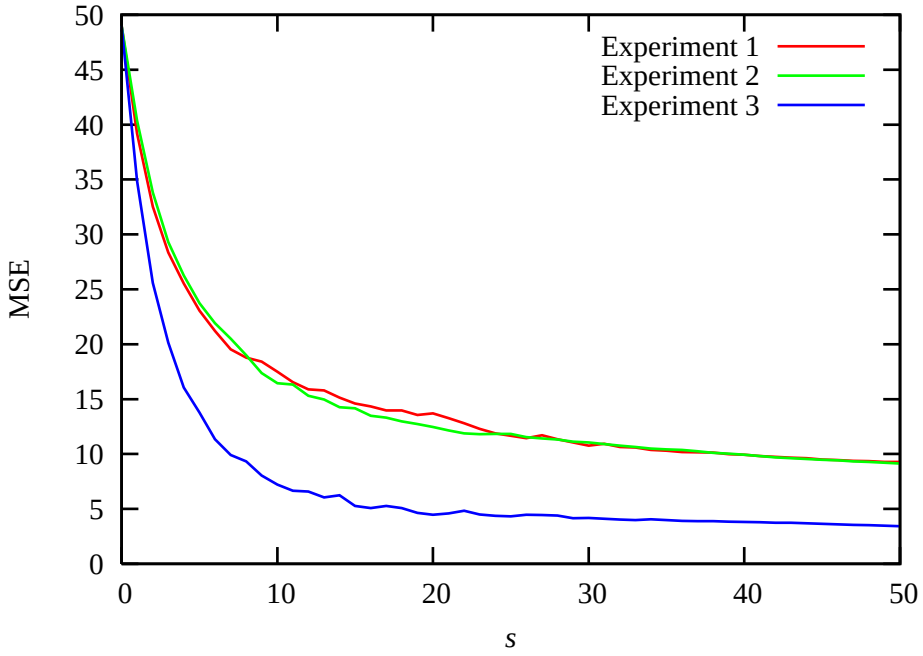


Figure 7.8: Mean square error of displacements $(\mathbf{q}_i)_k$ for CT-CT registration with different initial misalignments

errors are being corrected becomes sufficiently small. Errors remaining can be attributed to two types of nodes. The first are those whose local area is dominated by an elongated structure. In analogy to the PET-CT scenario, local mutual information takes on a shape such as that of figure 7.7 (b). The broad flat region around the maximum is quickly reached but many more steps would be required to locate it precisely. A different type of problematic nodes are those for which mutual information is highly multimodal and stochastic gradient ascent finds an incorrect local extremum. Fortunately, the number of such nodes is very small. They are located primarily in the area showing the patient's teeth which, due to metallic dental fillings, reflect the X-rays and generate smudges.

Experiments show that by tuning the constants of the registration system, the mean square error after 50 registration steps can reliably be reduced to 0.02 for all three CT-CT registration scenarios. However, the results presented above indicate that even without such adjustments, the registration system is able to reliably produce a substantial reduction in alignment error for a range of different initial misalignments. The second CT-CT scenario especially shows that assessment and correction of misalignment truly occur locally at every node so that even if the initial displacements point in opposing direction for adjacent nodes, each is adjusted toward optimal alignment.

7.2.2. Speed

The final aspects addressed are registration speed and the impact that the use of GPGPU techniques can have on it. Three computer systems, two of them equipped with modern

System	1	2	3
CPU	Pentium M 1.6GHz	Athlon XP 1.53GHz	Pentium D 3.2GHz
RAM	768 MB	1024 MB	4096 MB
GPU	—	GeForce 6600GT	GeForce 7800GT
GPU RAM	—	128 MB	256 MB
Bus	—	AGP 4×	PCI-Express 16×

Table 7.3: Computer systems used in speed experiments

NVIDIA GPUs, are used. Their relevant specifications are listed in table 7.3

Save for the GPUs, which have been fitted later, the systems roughly represent technological progression in two year steps. System 2 is oldest, assembled in 2002. System 1 is a laptop bought in 2004. The last system dates from 2006. The two GPUs are one generation apart with the GeForce 6600GT not only older but also positioned slightly lower within the spectrum of cards offered by NVIDIA. Besides a higher number of fragment processors, faster clock speed, more performant bus interface and generally improved architecture, the GeForce 7800GT has double the RAM of the 6600GT. This is significant when calculating mutual information derivatives for the rigid transformation function with a sample size of $N = 2048$. Because 128 MB are not sufficient to hold the large intermediate textures required, the alternative code path described in section 6.3.3 is used that splits some of the computation steps into several rendering passes using smaller textures. For all other calculations, no more than 128 MB are required. The CPU based implementations do not need significantly more RAM than their GPU based counterparts so that the main memory available on all three systems suffices.

Measured first is the time required for a full PET-CT registration using the different implementations. The results are presented as averages over multiple replications in table 7.5. Because the benefits of the non-rigid pass are dubious, it may be desirable to execute only rigid registration, timings for which are given in table 7.6. In both cases, the processes whose duration is assessed are precalculation and actual registration. For GPU based implementation, this includes the time required to set up the rendering pipeline and upload all necessary texture maps.

Only two implementations are available for the rigid pass, performing sampling and calculation either entirely on the CPU or on the GPU. In the non-rigid case, a hybrid approach additionally exists. The entirely GPU based solution uses the monolithic shader of section 6.3.5.1 to sample the registered image, while the hybrid method samples on the CPU and uploads the results to the GPU for further processing as textures. When both rigid and non-rigid registration are executed, the implementations are combined as shown in table 7.4.

Several observations can be made from tables 7.5 and 7.6. Looking at only the rigid pass, it can be seen that the GPU based implementation provides a considerable speed-up. Compared to the fastest CPU, which needs 27 seconds, the GeForce 7800GT is 6.75 times faster, requiring only 4 seconds for texture upload, preprocessing and 50 registration steps. The 6600GT is slower but still almost twice as fast as the best

Implementation Combination	Rigid Pass	Non-Rigid Pass
CPU	CPU	CPU
GPU	GPU	GPU
Hybrid	GPU	Hybrid

Table 7.4: Combinations of implementations used for rigid and non-rigid passes

Implementation Combination	Pentium M 1.6GHz —	Athlon XP 1.53GHz GeForce 6600GT	Pentium D 3.2GHz GeForce 7800GT
CPU	274	301	239
GPU		1025	709
Hybrid		116	82

Table 7.5: Duration of rigid and non-rigid PET-CT registration in seconds

Implementation	Pentium M 1.6GHz —	Athlon XP 1.53GHz GeForce 6600GT	Pentium D 3.2GHz GeForce 7800GT
CPU	36	35	27
GPU		15	4

Table 7.6: Duration of a rigid PET-CT registration in seconds

CPU. When both passes are considered, it is apparent that non-rigid registration takes significantly longer and dominates the results. It is also clearly seen that the GPU based implementation does not achieve the desired speed-up. Instead of being faster, it is up to 2.4 times slower than the CPU in the same system. The hybrid approach, however, is able to shorten registration time by a factor of 2.6 for system 2 and 2.9 for system 3.

This demonstrates that GPU based sampling is slow while the derivative calculation that follows can significantly be accelerated. The lack of profiling hooks on a GPU makes the precise cause of the slow sampling difficult to determine. However, the large number of random accesses to texture maps required when evaluating the non-rigid transformation function combined with the small texture caches make it likely that pipeline stalls while waiting for data are responsible for the poor performance.

As noted in section 7.2, the number of steps conducted during registration is chosen arbitrarily. To be able to extrapolate the results for other values of s_1 and s_2, it is interesting to time a single registration step. This is done for the rigid pass in table 7.7 and for the non-rigid, in 7.8. The numbers agree with the findings of tables 7.5 and 7.6. For rigid registration, the GPU based implementation is fastest while for non-rigid, the hybrid approach takes the lead.

Although not part of registration, the calculation speeds for mutual information itself are provided for completeness. In table 7.9, the average time for a single evaluation of $MI^\star$ in the rigid case is shown. As was the case for the derivatives, the GPU based

Implementation	Pentium M 1.6GHz —	Athlon XP 1.53GHz GeForce 6600GT	Pentium D 3.2GHz GeForce 7800GT
CPU	752	605	577
GPU		214	63

Table 7.7: Duration of a $\frac{d}{dT}MI^{\star}$ calculation for rigid PET-CT alignment in milliseconds

Implementation	Pentium M 1.6GHz —	Athlon XP 1.53GHz GeForce 6600GT	Pentium D 3.2GHz GeForce 7800GT
CPU	5108	5417	4139
GPU		20509	14441
Hybrid		2005	1651

Table 7.8: Duration of a $\frac{d}{dT}MI^{\star}$ calculation for all nodes in non-rigid PET-CT alignment in milliseconds

implementation is faster. Interestingly, the oldest CPU, made by AMD, outperforms the other two chips by intel.

The results for non-rigid $MI^{\star}$ are shown in table 7.10. A total of four implementations are evaluated. As before, GPU corresponds to non-rigid sampling using a monolithic shader and GPU based calculation. GPU 2 employs the multiple shader approach of section 6.3.5.2 instead. This allows the question to be answered whether better performance may be achieved by breaking up the large shader into a series of smaller steps. As can clearly be seen, the opposite is the case. A series of smaller shaders not only does not provide a benefit but leads to an additional increase in computation time. The fastest technique again is the hybrid approach of CPU based sampling and calculation on the GPU.

Implementation	Pentium M 1.6GHz —	Athlon XP 1.53GHz GeForce 6600GT	Pentium D 3.2GHz GeForce 7800GT
CPU	116	64	86
GPU		30	12

Table 7.9: Duration of a $MI^{\star}$ calculation for rigid PET-CT alignment in milliseconds

Implementation	Pentium M 1.6GHz —	Athlon XP 1.53GHz GeForce 6600GT	Pentium D 3.2GHz GeForce 7800GT
CPU	1675	1626	1349
GPU		17088	12220
GPU 2		41808	32301
Hybrid		861	636

Table 7.10: Duration of a $MI^{\star}$ calculation for all nodes in non-rigid PET-CT alignment in milliseconds

8. Discussion

The goal of this thesis has been to develop a registration system that is suited as best as possible for the solution of the problem stated in section 1.3.1. The main aims were to achieve fully automatic, precise registration of multi-modal tomographic datasets and to perform the required calculations quickly and efficiently, despite the large amounts of data involved. Experiments have shown that the system meets some of these goals while in other areas, future improvements are possible.

At the core of the registration system is the similarity metric $MI^\star$. Based on the work published in [Vio95], it is an estimate of mutual information calculated from two random samples of size N using a Parzen window technique. The concept of mutual information is popular in multi-modal registration because it does not assume a linear relationship between the intensities of corresponding points in the two images. As the plots presented in the previous chapter indicate, the measure has shown to be appropriate for multi-modal tomographic datasets. Save for the noise introduced by randomized estimation and a few problematic nodes in the non-rigid pass, a unimodal metric is obtained which smoothly increases toward a single maximum that coincides with the optimal alignment of reference and registered image.

More important to the registration process is the fact that the estimated derivatives of $MI^\star$ also show the desired behavior. While they are more noisy than mutual information itself, they generally have correct signs which allows them to be used in an iterative search for the optimal alignment. Whenever a stochastic approximation is used, noise is to be expected and must be tolerated to some degree. However, experiments have shown that the noise amplitude can be reduced to an acceptable level by increasing N and σ^2.

Some of the results of [Vio95] could not be reproduced. To obtain a reliable measure whose derivatives are of sufficient quality for robust registration, the sample size N had to be increased far beyond the proposed value of 50, even at the pyramid level $l = 4$ with the smallest images. Also, the automatic calculation of Parzen window variances has been found not to work due to the limited precision of a computer. However, since the variances have shown to have mostly a smoothing effect and the metric is largely insensitive to their precise values, good results have been achieved with empirically determined constant values.

While $MI^\star$ provides a reliable measure of alignment quality and can be used to drive the registration process, the question should be posed whether a faster metric exists. The Parzen window technique proposed in [Vio95] has the advantages of being simple and nonparametric. Its major weakness is the complexity of the resulting estimated density function. A Gaussian bell shape is centered at every sampling point in sample S_A so that to evaluate the function, an iteration over N points is necessary. Because to estimate either mutual information or its derivatives, the density function must be

evaluated for each point in S_B, the resulting complexity is $O\left(N^2\right)$.

Instead of centering a Gaussian bell shape at each point in S_A, a substantially smaller number of Gaussians could be used, leading to the Gaussian mixture model approach described in section 2.5.2. Its downside is that the mean and covariance matrix of each Gaussian must be calculated from S_A before the density function can be used. However, the benefit then obtained is that the resulting estimated density function consists of only a small constant number of Gaussians and can be evaluated in $O\left(1\right)$.

One possibility for the estimation of the parameters of a Gaussian mixture model is described in [Bil98]. Using a fixed number of iterative steps of the EM algorithm, an estimate can be obtained in $O\left(N\right)$. The evaluation of this estimate for every point in S_B then again requires $O\left(N\right)$, reducing the overall complexity from $O\left(N^2\right)$ to $O\left(N\right)$. This reduces not only the number of computation steps but also the amount of memory needed. In the GPU based implementations developed in this thesis, the large intermediary textures of size $N \times N$ required effectively limit the usable sample size. With both time and space complexity $O\left(N\right)$, much larger sample sizes could be used. As noted in the previous chapter, this is desirable at pyramid level $l = 0$.

Another factor that greatly affects the performance of the similarity metric is the programming model used for its implementation. The registration system of chapter 5 uses traditional, sequential implementations on a single CPU. This type of code is simple to develop and understand but does not necessarily lead to optimal performance. On the GPU, parallel execution is possible and the implementations in chapter 6 are adapted to this paradigm. Whenever calculations are independent and can be executed concurrently, they are represented as different fragments so that the current shader may be run for them in parallel. This approach has led to considerable speed-ups for some calculations while for others, computation time has increased.

It is impossible to precisely analyze the execution of a GPU based implementation due the lack of profiling hooks and exact documentation. However, generic models and rules for parallel computation are available which can help to understand why some algorithms experience a speed-up and others do not. One important observation is that not all computations benefit from parallel execution. With T_s the amount of time spent in operations that must be executed sequentially, T_p the time for those which can be parallelized and n_p the number of processing units, such as shaders, Amdahl's law [KGGK94] states that an upper bound on the speed-up possible is:

$$S = \frac{T_s + T_p}{T_s + \frac{T_s}{n_p}} \tag{8.1}$$

This value could only be reached if the parallelizable calculations can be split into n_p equally sized portions and no overhead is incurred. The equation also shows that with growing n_p, a maximal speed-up is approached asymptotically. A more realistic view of parallel computation is incorporated into the *LogP* model [CKP+93], which also considers the costs of communication between the processing units. Each unit is additionally limited by the speed of its memory interface so that massively parallel execution of small steps may be slower than larger, less parallelized steps due to the constant reading and writing of intermediate results.

Because of its specialized hardware, not all of these points apply to a GPU. While the facts that some calculations can only be performed sequentially and that the memory interface can become a bottleneck are true, direct communication between the fragment shaders is not possible. An exchange of data is only possible by writing to the framebuffer and then reading back these values in another rendering pass. This, in turn, benefits from a shared graphics memory used by all processors. No messages need to explicitly be passed as the same textures are available to all fragment processors. Nevertheless, it follows that if a calculation either cannot be parallelized to a high degree or performs too many memory accesses, its performance on the GPU may suffer. This could be observed in the previous chapter in the difference between the monolithic and multiple shader implementations of the non-rigid transformation function. The multiple shader version is more parallelized but the much higher number of memory accesses required completely eliminates this advantage.

Although an upper bound on the speed-up could be calculated, only experimentation can show how much is actually gained. For the GPU based implementations developed in this thesis, the results have been presented in the previous chapter. It has been found that rigid registration can be performed in just 4 seconds with a speed-up of 6.75 over the fastest CPU. In the non-rigid pass, the evaluation of the transformation function has shown to perform poorly on the GPU. If, however, sampling remains on the CPU but the actual calculation of mutual information derivatives is moved onto a GPU, a speed-up of 2.9 is obtained. How further improvements could be made will be addressed in the next section.

Considered now are the transformation functions and search spaces. For the rigid pass, the experiments conducted have clearly shown the chosen transformation type to be suitable. Since the scales of the two datasets can be matched automatically using DICOM metadata, the only rigid differences between the images to be expected are translations and small rotations. As seen in the previous chapter, the transformation function could be used to undo these alignments with a very high precision for CT-CT. Equally convincing results were obtained for PET-CT, as far as this can be judged without the help of a radiologist. Because it can be expressed as a simple 4×4 matrix, the transformation function and its derivatives have also proven to be quick to calculate via matrix-vector operations.

The situation is different for the non-rigid transformation function. Based on the results of the previous chapter, it must be questioned whether the non-rigid pass is actually useful and if so, what needs to be done to ensure that plausible adjustments are obtained. Some of the problems encountered were multimodal or partially flat local $MI^\star$ surfaces, making the localization of the precise maximum very difficult. However, even when the maximum can be located, the question remains in how far the resulting alignment is medically valid. While the search space is constrained so as to preserve topology, no rules are present to ensure anatomical correctness.

A model of the anatomical structure in the registered image is required. The aim of the non-rigid pass is to undo small local misalignments due to changes in the shapes of the patient's internal organs. Only these should be deformed and rigid bones left untouched. This can be ensured only if constraints are introduced that express which areas may be

deformed and by how much. The model may influence the transformation parameters by clipping them to valid ranges or removing nodes for which no local displacements are desired. Another option is to leave the parameters unaltered but integrate the model into the evaluation of the transformation function. In this case, the $\mathbf{q}_i$ as well as the output of the model are interpreted as forces acting on the registered image and deforming its shape.

In both cases, before the model may be used, it must be constructed. It can be built only from an anatomical scan as a physiological dataset contains no information about the structures to be modeled. Unfortunately, reliable automatic segmentation is still an unsolved problem and it is unlikely that a satisfactory distinction between soft tissue and bones could automatically be computed. This step therefore necessitates human intervention, incurring a cost for the time spent by a skilled operator directing the segmentation process.

The node arrangements used in this thesis are relatively widely spaced. If the changes to be undone are minimal, the radius of influence of each node should be decreased to gain more local control. However, this also means an increase in the total number of nodes and as a consequence, in registration time. If the number of nodes is increased, an option to be considered would be the use of a completely different transformation function. While the Wendland polynomials used are simple and in principle fast to evaluate, no GPU acceleration of their computation has been possible. Experiments may show that a different type of non-rigid transformation function can be more efficiently evaluated on such hardware.

While a model and a tight node spacing may be able to limit the displacements applied to anatomically plausible ranges, the adjustments required can be expected to be very small. Deformations of internal organs occur between pre- and postoperative images where for example tissue has been removed or a fractured bone adjusted. When the patient is placed in almost identical poses and the scans are taken in close succession, the probability of changes to the shapes of internal organs that are large enough to affect diagnosis should be minimal. The computationally expensive and time-consuming non-rigid pass, especially when coupled with the manual or semi-automatic construction of a model, thus appears to provide little benefit at great cost.

The last component of the registration system is its search strategy. With stochastic gradient ascent, a very simple approach has been chosen. However, experiments have shown that it is able to locate the maximum of $MI^\star$ despite random noise and large initial misalignments. For the non-rigid pass, the CT-CT experiments have also shown that parameters are adjusted toward the optimum. Only nodes for which the local image structure leads to a multimodal or very flat local $MI^\star$ surface have proven problematic. A different strategy may be able to overcome these problems. One approach for multimodal surfaces is to perform the registration multiple times with different initial alignments and return the alignment produced that leads to the highest value of $MI^\star$. A completely different family of search strategies is found in the genetic algorithms, which treat each potential alignment as an individual and generate new alignments by mutation, recombination and survival of the fittest.

Also part of the search strategy is the multiresolution technique. As was the case with

stochastic gradient descent, it has shown to be successful. The small images used at the beginning of the registration process lead to a very high quality metric that allows the region in which the optimal alignment may be found to be quickly located. The original images at pyramid level $l = 0$ allow the precise maximum of $MI^\star$ to be found. What could be reconsidered is whether all intermediate pyramid levels are required. The difference in the location of the maximum from one level to the next is so small that gradient ascent should be able to locate the optimum also when one or more pyramid levels are skipped, allowing registration time to be shortened.

It would have been desirable to compare the entire registration system to other similar systems, both in terms of quality and speed. Unfortunately, this was not possible. Publications do not contain source code and use different example scenarios so that results cannot directly be compared. Also, many systems require manual intervention making a fair comparison with a fully automatic system impossible. However, the computation times obtained for large volumetric images are reasonable, as is the registration quality, at least for the rigid pass where it can be judged.

8.1. Future Work

Work based on the registration system developed in this thesis could be taken in different directions. However, before any adjustments or modifications are made, it appears sensible to properly test the system. While speed has been measured thoroughly, the registration quality achieved for two tomographic datasets is hardly representative. A larger body of physiological and anatomical dataset pairs should be obtained, the constants of the registration system adjusted using some of these pairs and the system then evaluated using the remaining. With the help of a radiologist, this would allow the quality of registration to be properly assessed. It should also highlight the shortcomings of the system and areas where improvements are necessary.

Although much work has been put into an efficient implementation and considerable speed-ups have been achieved on the GPU, more registration speed may be desired. It appears unlikely that the shaders can be substantially improved to produce a significantly faster computation. The approach to be followed instead should be to use different, more powerful hardware. An obvious step would be to use SLI, the liking of two GPUs. This would require no changes to shaders or the code that drives the registration, as the entire coordination between the two GPUs is handled by their driver. However, only minimal control is provided over how data and work is split betwen the two graphics cards so that no additional optimizations are possible and a large overhead may have to be accepted for the exchange of information between the GPUs.

To gain better control, a parallel implementation on a cluster of CPUs could be investigated. The implementations could largely follow those developed for the GPU and described in chapter 6 as they are already parallelized. However, because the distribution of work and data can be controlled, additional optimizations are possible. The goal should be to equally divide the work among the CPUs and minimize the need for communication between them.

How the GPU based implementations may be adapted is shown here using the calculation of $\frac{d}{dT}MI^{\star}$ for the rigid transformation as an example. To be evaluated are equations (6.4) to (6.6). The steps of the GPU implementation are outlined in figure 6.5. The most important observation to be made about the formulas is that except for the nested sum in equation (6.4), which is evaluated last, the calculations performed for all points $x_i \in S_B$ are independent of each other. Data and work should thus be distributed so that each CPU handles a subset $S'_B \subseteq S_B$.

The sampling of image intensities and derivatives can directly be parallelized as the calculation of the transformation function and the retrieval of image intensities are done independently for each point. Every CPU should handle equally sized subsets S'_B and $S'_A \subseteq S_A$. After sampling has completed, the data obtained for sample S_A must be exchanged to provide a complete sample at each CPU. Then, all subsequent steps can be performed concurrently by the CPUs as each only considers pairs $(x_i, x_j) \in S'_B \times S_A$. After weight numerators have been calculated and added, denominators computed, the derivatives weighted and summed, the sums calculated by each CPU have to be retrieved by one and added to calculate the final derivatives.

Similar implementations are possible for the other parts of the registration system. In all cases, the communication overhead is minimal. However, communication is still required several times over the course of each calculation. To further reduce this need, parallelization may be performed at a different level. Each CPU may calculate the entire derivatives $\frac{d}{dT}MI^{\star}$ for an alignment, beginning with the sampling up to the computation of the final values. Parallelism is achieved by giving each CPU a different alignment to work on, thus implementing multiple initial alignments or a genetic algorithm. For the non-rigid pass, every CPU could also evaluate the local derivatives at a different node as the calculations for all nodes are completely independent. Because the communication overhead is small and most computations can be parallelized, CPU cluster based versions should be evaluated for all implementations in chapter 6.

It could be attempted to further accelerate the non-rigid transformation function as well, but it appears more important to first add a model, as described in the previous section, so that medically plausible results are produced. If the non-rigid pass is to be retained, which is a question that cannot be fully answered until thorough experiments on a larger statistical basis have been performed, a model is clearly necessary. While it is doubtful that automatic segmentation will be of sufficient quality, it should be investigated to ensure that manual intervention is not needlessly introduced into the system. Different models could be used, possibly combined with another family of transformation functions. Again, the plausibility of their results must be assessed by a radiologist and a model chosen that protects bones from deformation while allowing it for internal organs.

Finally, the results of this thesis could be applied to an entirely different area. As outlined in section 1.2.1, multi-modal registration is found in many other application areas and a fast and efficient system could be beneficial to them. Because mutual information is a very robust metric applicable to far more than medical images, it is likely that the registration system can easily be adapted to many different areas. Yet again, the non-rigid transformation function may be the component that proves not viable and has to be removed.

A. Shaders

A.1. Rigid Sampling

A.1.1. SampleReference

```
uniform sampler3D data;
uniform sampler2D samplingPositions;

void main() {
  vec3 samplingPoint = texture2D(samplingPositions, gl_TexCoord[0].xy).xyz;
  gl_FragColor.r     = 255. * texture3D(data, samplingPoint).r;
}
```

A.1.2. SampleRegistered

```
uniform sampler3D data;
uniform sampler2D samplingPositions;
uniform mat4      transformation;

void main() {
  vec3 samplingPoint = (transformation
                        * texture2D(samplingPositions, gl_TexCoord[0].xy)).xyz;
  gl_FragColor.r     = 255. * texture3D(data, samplingPoint).r;
}
```

A.1.3. SampleRegisteredDerivatives

```
uniform sampler3D data;
uniform sampler2D samplingPositions;
uniform mat4      transformation;
uniform vec3      offset[3];
uniform mat3      transformationTranslationDerivative;
uniform mat4      transformationRotationDerivative[3];

void main() {
  vec4  worldPosition   = texture2D(samplingPositions, gl_TexCoord[0].xy);
  vec3  samplingPoint   = (transformation * worldPosition).xyz;

  float intensity       = 255. * texture3D(data, samplingPoint).r;

  vec3  gradient        = vec3(
    255. * texture3D(data, samplingPoint + offset[0]).r - intensity,
    255. * texture3D(data, samplingPoint + offset[1]).r - intensity,
    255. * texture3D(data, samplingPoint + offset[2]).r - intensity);

  mat3  innerDerivative = mat3(
    (transformationRotationDerivative[0] * worldPosition).xyz,
    (transformationRotationDerivative[1] * worldPosition).xyz,
    (transformationRotationDerivative[2] * worldPosition).xyz);
```

```
  gl_FragData[0].r      = intensity;
  gl_FragData[1].rgb    = gradient * transformationTranslationDerivative;
  gl_FragData[2].rgb    = gradient * innerDerivative;
}
```

A.2. Rigid Mutual Information Derivative

A.2.1. Weights

```
uniform sampler2DRect gaussian1D;
uniform sampler2DRect gaussian2D;
uniform sampler2DRect sampleX;
uniform sampler2DRect sampleY;

void main() {
  vec2 sampleACoordinates = gl_TexCoord[0].xy;
  vec2 sampleBCoordinates = gl_TexCoord[0].zw;
  vec2 sampleA            = vec2(texture2DRect(sampleX, sampleACoordinates).r,
                                 texture2DRect(sampleY, sampleACoordinates).r);
  vec2 sampleB            = vec2(texture2DRect(sampleX, sampleBCoordinates).r,
                                 texture2DRect(sampleY, sampleBCoordinates).r);
  vec2 sampleDifference   = abs(sampleA - sampleB) + vec2(.5, .5);

  gl_FragData[0].r        = texture2DRect(gaussian1D,
                                          vec2(sampleDifference.g, .5)).r;
  gl_FragData[1].r        = texture2DRect(gaussian2D, sampleDifference).r;
}
```

A.2.2. AddWeights

```
uniform sampler2DRect inputData;

void main() {
  float left     = texture2DRect(inputData, gl_TexCoord[0].xz).r;
  float right    = texture2DRect(inputData, gl_TexCoord[0].yz).r;
  gl_FragColor.r = left + right;
}
```

A.2.3. WeightMultiplier

```
uniform sampler2DRect inputData;
uniform float         varianceReciprocal;

void main() {
  float denominator = texture2DRect(inputData, gl_TexCoord[0].xy).r;

  if (denominator > 0.)
    gl_FragColor.r = varianceReciprocal / denominator;
  else
    gl_FragColor.r = 0.;
}
```

A.2.4. WeightedDerivatives

```
uniform sampler2DRect sample;
```

```
uniform sampler2DRect weightMultipliersV;
uniform sampler2DRect weightMultipliersW;
uniform sampler2DRect weightsV;
uniform sampler2DRect weightsW;
uniform sampler2DRect sampleTranslationDerivatives;
uniform sampler2DRect sampleRotationDerivatives;

uniform float         varianceReciprocal;

void main() {
  vec2  coordinatesA                = gl_TexCoord[0].xy;
  vec2  coordinatesB                = gl_TexCoord[0].zw;
  vec2  coordinatesWeightMultiplier = gl_TexCoord[0].yz;
  vec2  coordinatesWeight           = gl_TexCoord[0].xz;

  float multiplier                  =
      (texture2DRect(sample, coordinatesB).r
    -  texture2DRect(sample, coordinatesA).r)
    * (texture2DRect(weightMultipliersV, coordinatesWeightMultiplier).r
    *  texture2DRect(weightsV, coordinatesWeight).r
    -  texture2DRect(weightMultipliersW, coordinatesWeightMultiplier).r
    *  texture2DRect(weightsW, coordinatesWeight).r);

  gl_FragData[0].rgb                =
       multiplier
    * (texture2DRect(sampleTranslationDerivatives, coordinatesB).rgb
    -  texture2DRect(sampleTranslationDerivatives, coordinatesA).rgb);
  gl_FragData[1].rgb                =
       multiplier
    * (texture2DRect(sampleRotationDerivatives, coordinatesB).rgb
    -  texture2DRect(sampleRotationDerivatives, coordinatesA).rgb);
}
```

A.2.5. AddWeightedDerivatives

```
uniform sampler2DRect inputData;

void main() {
  vec3 topLeft     = texture2DRect(inputData, gl_TexCoord[0].xz).rgb;
  vec3 topRight    = texture2DRect(inputData, gl_TexCoord[0].yz).rgb;
  vec3 bottomLeft  = texture2DRect(inputData, gl_TexCoord[0].xw).rgb;
  vec3 bottomRight = texture2DRect(inputData, gl_TexCoord[0].yw).rgb;

  gl_FragColor.rgb = topLeft + topRight + bottomLeft + bottomRight;
}
```

A.3. Rigid Mutual Information

A.3.1. GaussianPDF1D

```
uniform sampler2DRect gaussian;
uniform sampler2DRect sample;

void main() {
  float sampleA  = texture2DRect(sample, gl_TexCoord[0].xy).r;
  float sampleB  = texture2DRect(sample, gl_TexCoord[0].zw).r;
  gl_FragColor.r = texture2DRect(gaussian,
```

```
                                vec2(abs(sampleA - sampleB) + .5, .5));
}
```

A.3.2. GaussianPDF2D

```
uniform sampler2DRect gaussian;
uniform sampler2DRect sampleX;
uniform sampler2DRect sampleY;

void main() {
  vec2 sampleACoordinates = gl_TexCoord[0].xy;
  vec2 sampleBCoordinates = gl_TexCoord[0].zw;
  vec2 sampleA            = vec2(texture2DRect(sampleX, sampleACoordinates).r,
                                 texture2DRect(sampleY, sampleACoordinates).r);
  vec2 sampleB            = vec2(texture2DRect(sampleX, sampleBCoordinates).r,
                                 texture2DRect(sampleY, sampleBCoordinates).r);
  gl_FragColor.r          =
    texture2DRect(gaussian, abs(sampleA - sampleB) + vec2(.5, .5)).r;
}
```

A.3.3. AddHorizontally

```
uniform sampler2DRect inputData;

void main() {
  float left     = texture2DRect(inputData, gl_TexCoord[0].xz).r;
  float right    = texture2DRect(inputData, gl_TexCoord[0].yz).r;
  gl_FragColor.r = left + right;
}
```

A.3.4. ScaleLog

```
void main() {
  float probability = texture2DRect(inputData, gl_TexCoord[0].xy).r
                    * sampleSizeReciprocal;
  if (probability > .000000001)
    gl_FragColor.r = log(probability);
  else
    gl_FragColor.r = 0.;
}
```

A.3.5. AddVertically

```
uniform sampler2DRect inputData;

void main() {
  float top      = texture2DRect(inputData, gl_TexCoord[0].xy).r;
  float bottom   = texture2DRect(inputData, gl_TexCoord[0].xz).r;
  gl_FragColor.r = top + bottom;
}
```

A.4. Non-Rigid Sampling, Monolithic Shader

A.4.1. Classify

```
uniform sampler3D     data;
uniform sampler2DRect gridPointPositions;
uniform sampler2DRect classificationPattern;

void main() {
  vec3 gridPoint    = texture2DRect(gridPointPositions, gl_TexCoord[0].yz).xyz;
  vec3 patternPoint = texture2DRect(classificationPattern,
                                    gl_TexCoord[0].xz).xyz;
  gl_FragColor.r    = step(.03, texture3D(data, gridPoint + patternPoint).r);
}
```

A.4.2. AddClassifications

```
uniform sampler2DRect inputData;

void main() {
  float left     = texture2DRect(inputData, gl_TexCoord[0].xz).r;
  float right    = texture2DRect(inputData, gl_TexCoord[0].yz).r;
  gl_FragColor.r = left + right;
}
```

A.4.3. CopyNodeWeight

```
uniform sampler2DRect gridPointNodes;
uniform sampler2DRect nodeWeights;

void main() {
  vec2 nodes     = vec2(gl_TexCoord[0].z,
                        texture2DRect(gridPointNodes, gl_TexCoord[0].xy).w);

  gl_FragColor.r = texture2DRect(nodeWeights, nodes).x;
}
```

A.4.4. WeightDisplacement

```
uniform sampler2DRect nodeWeights;
uniform sampler2DRect displacements;

void main() {
  float weight       = texture2DRect(nodeWeights, gl_TexCoord[0].xy).r;
  vec3  displacement = texture2DRect(displacements, gl_TexCoord[0].xz).xyz;
  gl_FragColor.rgb   = weight * displacement;
}
```

A.4.5. AddWeightedDisplacements

```
uniform sampler2DRect inputData;

void main() {
  vec3 left        = texture2DRect(inputData, gl_TexCoord[0].xz).rgb;
  vec3 right       = texture2DRect(inputData, gl_TexCoord[0].yz).rgb;
  gl_FragColor.rgb = left + right;
}
```

A.4.6. CopyCoefficient

```
uniform sampler2DRect gridPointNodes;
uniform sampler2DRect coefficients;

void main() {
  vec2  node       = texture2DRect(gridPointNodes, gl_TexCoord[0].xy).xw;
  gl_FragColor.rgb = texture2DRect(coefficients, node).xyz;
}
```

A.4.7. SampleReference

```
uniform sampler3D     data;
uniform sampler2DRect nodePositions;
uniform sampler2D     samplingPositions;

void main() {
  vec3 samplingPoint = texture2DRect(nodePositions, gl_TexCoord[0].xz).xyz
                     + texture2D(samplingPositions, gl_TexCoord[0].yz).xyz;
  gl_FragColor.r     = 255. * texture3D(data, samplingPoint).r;
}
```

A.4.8. SampleRegistered

```
#define CoordinateNodes 8

#define offsetX 1
#define offsetY (CoordinateNodes + 3)
#define offsetZ (CoordinateNodes + 3) * (CoordinateNodes + 3)

uniform sampler3D     data;
uniform sampler2DRect nodePositions;
uniform sampler2D     samplingPositions;
uniform sampler3D     rbfs;
uniform sampler2DRect coefficients;
uniform mat4          rigidTransformation;

void main() {
  vec4  samplingPointInformation =
      texture2DRect(nodePositions, gl_TexCoord[0].xz)
    + texture2D(samplingPositions, gl_TexCoord[0].yz);
  vec3  worldSamplingPoint       = samplingPointInformation.xyz;
  vec2  firstGridPoint           = vec2(samplingPointInformation.w, .5);

  worldSamplingPoint            -=
      texture3D(rbfs, vec3( 1./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
                                                    ).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3( 3./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX,                             0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3( 5./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX,                             0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3( 7./64., gl_TexCoord[0].yz)).r
```

```
  * texture2DRect(coefficients, firstGridPoint
  + vec2(                  offsetY,               0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3( 9./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(    offsetX +     offsetY,               0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(11./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(2 * offsetX +     offsetY,               0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(13./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(              2 * offsetY,               0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(15./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(    offsetX + 2 * offsetY,               0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(17./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(2 * offsetX + 2 * offsetY,               0.)).xyz;

worldSamplingPoint            -=
    texture3D(rbfs, vec3(19./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(                                  offsetZ, 0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(21./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(    offsetX +                     offsetZ, 0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(23./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(2 * offsetX +                     offsetZ, 0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(25./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(                  offsetY +     offsetZ, 0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(27./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(    offsetX +     offsetY +     offsetZ, 0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(29./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(2 * offsetX +     offsetY +     offsetZ, 0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(31./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(              2 * offsetY +     offsetZ, 0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(33./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(    offsetX + 2 * offsetY +     offsetZ, 0.)).xyz;
worldSamplingPoint            -=
    texture3D(rbfs, vec3(35./64., gl_TexCoord[0].yz)).r
  * texture2DRect(coefficients, firstGridPoint
  + vec2(2 * offsetX + 2 * offsetY +     offsetZ, 0.)).xyz;
```

```
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(37./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(                             2 * offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(39./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX +                2 * offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(41./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX +                2 * offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(43./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(                   offsetY + 2 * offsetZ, 0.)).xyz;
  worldSamplingPoint              -=
      texture3D(rbfs, vec3(45./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX +      offsetY + 2 * offsetZ, 0.)).xyz;
  worldSamplingPoint              -=
      texture3D(rbfs, vec3(47./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX +      offsetY + 2 * offsetZ, 0.)).xyz;
  worldSamplingPoint              -=
      texture3D(rbfs, vec3(49./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(              2 * offsetY + 2 * offsetZ, 0.)).xyz;
  worldSamplingPoint              -=
      texture3D(rbfs, vec3(51./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX + 2 * offsetY + 2 * offsetZ, 0.)).xyz;
  worldSamplingPoint              -=
      texture3D(rbfs, vec3(53./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX + 2 * offsetY + 2 * offsetZ, 0.)).xyz;

  vec3 samplingPoint               =
    (rigidTransformation * vec4(worldSamplingPoint, 1.)).xyz;
  gl_FragColor.r                   = 255. * texture3D(data, samplingPoint).r;
}
```

A.4.9. TransformationDerivativeWorld

```
#define CoordinateNodes 8

#define offsetX 1
#define offsetY (CoordinateNodes + 3)
#define offsetZ (CoordinateNodes + 3) * (CoordinateNodes + 3)

uniform sampler2DRect nodePositions;
uniform sampler2D     samplingPositions;
uniform sampler3D     rbfs;
uniform sampler2DRect weights;

void main() {
  vec2 firstGridPoint = vec2(
```

```
    texture2DRect(nodePositions, gl_TexCoord[0].xz).w
  + texture2D(samplingPositions, gl_TexCoord[0].yz).w,  gl_TexCoord[0].z);
gl_FragColor.r      = 0.;

gl_FragColor.r     -=
    texture3D(rbfs, vec3( 1./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
                                                      ).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3( 3./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(     offsetX,                              0.)).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3( 5./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(2. * offsetX,                              0.)).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3( 7./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(                     offsetY,              0.)).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3( 9./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(     offsetX +       offsetY,              0.)).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3(11./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(2. * offsetX +       offsetY,              0.)).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3(13./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(               2. * offsetY,               0.)).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3(15./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(     offsetX + 2. * offsetY,               0.)).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3(17./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(2. * offsetX + 2. * offsetY,               0.)).r;

gl_FragColor.r     -=
    texture3D(rbfs, vec3(19./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(                                   offsetZ, 0.)).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3(21./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(     offsetX               +       offsetZ, 0.)).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3(23./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(2. * offsetX               +       offsetZ, 0.)).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3(25./64., gl_TexCoord[0].yz)).r
  * texture2DRect(weights, firstGridPoint
  + vec2(                     offsetY +       offsetZ, 0.)).r;
gl_FragColor.r     -=
    texture3D(rbfs, vec3(27./64., gl_TexCoord[0].yz)).r
```

```
    * texture2DRect(weights, firstGridPoint
    + vec2(     offsetX +      offsetY +      offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(29./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(2. * offsetX +      offsetY +      offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(31./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(               2. * offsetY +      offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(33./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(     offsetX + 2. * offsetY +      offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(35./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(2. * offsetX + 2. * offsetY +      offsetZ, 0.)).r;

  gl_FragColor.r     -=
      texture3D(rbfs, vec3(37./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(                              2. * offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(39./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(     offsetX               + 2. * offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(41./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(2. * offsetX               + 2. * offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(43./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(                     offsetY + 2. * offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(45./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(     offsetX +      offsetY + 2. * offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(47./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(2. * offsetX +      offsetY + 2. * offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(49./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(               2. * offsetY + 2. * offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(51./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(     offsetX + 2. * offsetY + 2. * offsetZ, 0.)).r;
  gl_FragColor.r     -=
      texture3D(rbfs, vec3(53./64., gl_TexCoord[0].yz)).r
    * texture2DRect(weights, firstGridPoint
    + vec2(2. * offsetX + 2. * offsetY + 2. * offsetZ, 0.)).r;
}
```

A.4.10. SampleRegisteredDerivative

```
#define CoordinateNodes 8

#define offsetX 1
#define offsetY (CoordinateNodes + 3)
#define offsetZ (CoordinateNodes + 3) * (CoordinateNodes + 3)

uniform sampler3D     data;
uniform sampler2DRect nodePositions;
uniform sampler2D     samplingPositions;
uniform sampler3D     rbfs;
uniform sampler2DRect coefficients;
uniform sampler2DRect transformationDerivativesWorld;
uniform vec3          offset[3];
uniform mat4          rigidTransformationToVoxelTexture;
uniform mat3          rigidTransformationToVolume;

void main() {
  vec4  samplingPointInformation =
      texture2DRect(nodePositions, gl_TexCoord[0].xz)
    + texture2D(samplingPositions, gl_TexCoord[0].yz);
  vec3  worldSamplingPoint       = samplingPointInformation.xyz;
  vec2  firstGridPoint           = vec2(samplingPointInformation.w, .5);

  worldSamplingPoint            -=
      texture3D(rbfs, vec3( 1./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
                                                    ).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3( 3./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX,                             0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3( 5./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX,                             0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3( 7./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(                  offsetY,               0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3( 9./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX +     offsetY,               0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(11./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX +     offsetY,               0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(13./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(              2 * offsetY,               0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(15./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX + 2 * offsetY,               0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(17./64., gl_TexCoord[0].yz)).r
```

```
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX + 2 * offsetY,               0.)).xyz;

  worldSamplingPoint            -=
      texture3D(rbfs, vec3(19./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(                                 offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(21./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX +                    offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(23./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX +                    offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(25./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(                  offsetY +      offsetZ, 0.)).xyz;
  worldSamplingPoint              -=
      texture3D(rbfs, vec3(27./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX +     offsetY +      offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(29./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX +     offsetY +      offsetZ, 0.)).xyz;
  worldSamplingPoint              -=
      texture3D(rbfs, vec3(31./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(              2 * offsetY +      offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(33./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX + 2 * offsetY +      offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(35./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX + 2 * offsetY +      offsetZ, 0.)).xyz;

  worldSamplingPoint            -=
      texture3D(rbfs, vec3(37./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(                            2 * offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(39./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX +               2 * offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(41./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX +               2 * offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(43./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(                  offsetY + 2 * offsetZ, 0.)).xyz;
  worldSamplingPoint              -=
      texture3D(rbfs, vec3(45./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
```

```
    + vec2(    offsetX +     offsetY + 2 * offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(47./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX +     offsetY + 2 * offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(49./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(              2 * offsetY + 2 * offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(51./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(    offsetX + 2 * offsetY + 2 * offsetZ, 0.)).xyz;
  worldSamplingPoint            -=
      texture3D(rbfs, vec3(53./64., gl_TexCoord[0].yz)).r
    * texture2DRect(coefficients, firstGridPoint
    + vec2(2 * offsetX + 2 * offsetY + 2 * offsetZ, 0.)).xyz;

  vec3 samplingPoint            =
     (rigidTransformationToVoxelTexture * vec4(worldSamplingPoint, 1.)).xyz;

  float intensity               = 255. * texture3D(data, samplingPoint).r;

  vec3  gradient                = vec3(
    255. * texture3D(data, samplingPoint + offset[0]).r - intensity,
    255. * texture3D(data, samplingPoint + offset[1]).r - intensity,
    255. * texture3D(data, samplingPoint + offset[2]).r - intensity);

  gl_FragColor                  = vec4(intensity,
      texture2DRect(transformationDerivativesWorld, gl_TexCoord[1].xy).r
    * rigidTransformationToVolume * gradient);
```

A.5. Non-Rigid Sampling, Multiple Shaders

A.5.1. WeightCoefficients

```
uniform sampler3D     rbfs;
uniform sampler2D     firstGridPoints;
uniform sampler1D     gridOffsets;
uniform sampler2DRect coefficients;
uniform vec2          nodeGridPoint;

void main() {
  float rbf         = texture3D(rbfs, gl_TexCoord[0].xyz).r;
  vec2  gridPoint   = nodeGridPoint
    + texture2D(firstGridPoints, gl_TexCoord[0].yz).xw
    + texture1D(gridOffsets, gl_TexCoord[0].x).xw;
  vec3  coefficient = texture2DRect(coefficients, gridPoint).xyz;

  gl_FragColor.rgb  = rbf * coefficient;
}
```

A.5.2. AddWeightedCoefficients

```
uniform sampler2DRect inputData;

void main() {
```

```
  vec3 top         = texture2DRect(inputData, gl_TexCoord[0].xy).xyz;
  vec3 bottom      = texture2DRect(inputData, gl_TexCoord[0].xz).xyz;
  gl_FragColor.rgb = top + bottom;
}
```

A.5.3. SampleRegistered

```
uniform sampler3D     data;
uniform sampler2DRect samplingPositions;
uniform sampler2DRect nonRigidTransformations;
uniform vec3          nodePosition;
uniform mat4          rigidTransformation;

void main() {
  vec3 worldSamplingPoint =
      nodePosition
    + texture2DRect(samplingPositions, gl_TexCoord[0].xy).xyz
    - texture2DRect(nonRigidTransformations, gl_TexCoord[0].xz).xyz;
  vec3 samplingPoint      =
    (rigidTransformation * vec4(worldSamplingPoint, 1.)).xyz;
  gl_FragColor.r          = 255. * texture3D(data, samplingPoint).r;
}
```

A.6. Non-Rigid Mutual Information Derivative

A.6.1. Weights

Identical to A.2.1.

A.6.2. AddWeights

Identical to A.2.2.

A.6.3. WeightMultiplier

Identical to A.2.3.

A.6.4. WeightedDerivatives

```
uniform sampler2DRect sampleAndDerivatives;
uniform sampler2DRect weightMultipliersV;
uniform sampler2DRect weightMultipliersW;
uniform sampler2DRect weightsV;
uniform sampler2DRect weightsW;
uniform float         varianceReciprocal;

void main() {
  vec2  coordinatesA                  = gl_TexCoord[1].xy;
  vec2  coordinatesB                  = gl_TexCoord[1].zw;
  vec2  coordinatesWeightMultiplier   = gl_TexCoord[0].yz;
  vec2  coordinatesWeight             = gl_TexCoord[0].xz;

  vec4  sampleAndDerivativeDifference =
      texture2DRect(sampleAndDerivatives, coordinatesB)
```

```
    - texture2DRect(sampleAndDerivatives, coordinatesA);

  float multiplier                    =
       sampleAndDerivativeDifference.r
    * (texture2DRect(weightMultipliersV, coordinatesWeightMultiplier).r
    * texture2DRect(weightsV, coordinatesWeight).r
    - texture2DRect(weightMultipliersW, coordinatesWeightMultiplier).r
    * texture2DRect(weightsW, coordinatesWeight).r);

  gl_FragColor.rgb                    =
    multiplier * sampleAndDerivativeDifference.gba;
}
```

A.6.5. AddWeightedDerivatives

Identical to A.2.5.

A.7. Non-Rigid Mutual Information

A.7.1. GaussianPDF1D

Identical to A.3.1.

A.7.2. GaussianPDF2D

Identical to A.3.2.

A.7.3. AddHorizontally

Identical to A.3.3.

A.7.4. ScaleLog

Identical to A.3.4.

A.7.5. AddVertically

Identical to A.3.5.

Bibliography

[AS65] M. Abramowitz and I.A. Stegun, editors. *Handbook of Mathematical Functions*. Dover Publications, Inc., 1965.

[BBR89] H.F. Brink, M.D. Buschmann, and B.R. Rosen. NMR chemical shift imaging. *Computerized Medical Imaging and Graphics*, 13(1):93–104, January–February 1989.

[Bil95] P. Billingsley. *Probability and Measure*. John Wiley & Sons, Inc., third edition, 1995.

[Bil98] J.A. Bilmes. A gentle tutorial of the EM algorithm and its application to parameter estimation for gaussian mixture and hidden markov models. April 1998.

[Bro92] L.G. Brown. A survey of image registration techniques. *ACM Computing Surveys*, 24(4):325–376, December 1992.

[BS72] D.I. Barnea and H.F. Silverman. A class of algorithms for fast digital registration. *IEEE Transactions on Computers*, C-21(2):179–186, February 1972.

[Bur81] P.J. Burt. Fast filter transform for image processing. *Computer Graphics and Image Processing*, 16(1):20–51, May 1981.

[Can86] J.F. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986.

[CB94] M.S. Cohen and S.Y. Bookheimer. Localization of brain function using magnetic resonance imaging. *Trends in Neurosciences*, 17(7):268–277, July 1994.

[CKP+93] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, San Diego, CA, USA*, pages 1–12, May 1993.

[Fer04] R. Fernando, editor. *GPU Gems*. Addison-Wesley Professional, March 2004.

[FRS99] M. Fornefett, K. Rohr, and H.S. Stiehl. Elastic registration of medical images using radial basis functions with compact support. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition CVPR '99, Fort Collins, CO, USA*, pages 402–407, June 1999.

[FvDFH95] J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice in C.* Addison-Wesley Professional, second edition, 1995.

[gpg06] GPGPU. Website, July 2006. `http://www.gpgpu.org/`.

[GW06] R.C. Gonzalez and R.E. Woods. *Digital Image Processing.* Prentice Hall, Inc., third edition, 2006.

[IEE85] IEEE P754. *IEEE 754-1985: IEEE Standard for Binary Floating-Point Arithmetic.* IEEE and ANSI, August 1985.

[IVA+96] W.M. Wells III, P. Viola, H. Atsumi, S. Nakajima, and R. Kikinis. Multimodal volume registration by maximization of mutual information. *Medical Image Analysis*, 1(1):35–51, March 1996.

[KBR04] J. Kessenich, D. Baldwin, and R. Rost. *The OpenGL® Shading Language.* 3Dlabs, Inc. Ltd., April 2004. Language Version 1.10, Document Revision 59.

[KGGK94] V. Kumar, A. Grama, A. Gupta, and G. Karypis, editors. *Introduction to Parallel Computing: Design and Analysis of Parallel Algorithms.* Benjamin-Cummings Publishing Company, January 1994.

[KSSH02] N. Kojekine, V. Savchenko, M. Senin, and I. Hagiwara. Real-time 3D deformations by means of compactly supported radial basis functions. In *Eurographics 2002 EG 2002, Saarbrücken, Germany*, pages 35–43, September 2002. Short Presentation.

[KWT88] M. Kass, A. Witkin, and D. Terzopoulos. Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321—-331, January 1988.

[Lew95] J.P. Lewis. Fast normalized cross-correlation. Website, 1995. `http://www.idiom.com/~zilla/Work/nvisionInterface/nip.pdf`.

[LWS97] S. Lee, G. Wolberg, and S.Y. Shin. Scattered data interpolation with multilevel B-splines. *IEEE Transactions on Visualization and Computer Graphics*, 3(3):228–244, July 1997.

[MB88] G.J. McLachlan and K.E. Basford. *Mixture Models: Inference and Applications to Clustering.* Marcel Dekker, Inc., 1988.

[MH94] J.E. Marsden and T.J.R. Hughes. *Mathematical Foundations of Elasticity.* Dover Publications, Inc., reprint edition, 1994.

[Mic06] Microsoft Corporation. DirectX software development kit, June 2006. `http://msdn.microsoft.com/directx/sdk/`.

[Nat06] National Electrical Manufacturers Association. *Part 5: Data Structures And Encoding*, volume 5 of *Digital Imaging and Communications in Medicine (DICOM)*. National Electrical Manufacturers Association, 2006.

[NGL+98] Z. Nahas, M.S. George, J.P. Lorberbaum, S.C. Risch, and K.M. Spicer. SPECT and PET in neuropsychiatry. *Primary Psychiatry*, 5(3):52–59, March 1998.

[NVI05] NVIDIA Corporation. *NVIDIA GPU Programming Guide*, July 2005. Version 2.4.0.

[NVI06] NVIDIA Corporation. *Cg Toolkit User's Manual*, March 2006. Release 1.4.1.

[Par62] E. Parzen. On estimation of a probability density function and mode. *Annals of Mathematical Statistics*, 33(3):1065–1076, September 1962.

[Pau84] R.P. Paul. *Robot Manipulators: Mathematics, Programming, and Control*. MIT Press Series in Artificial Intelligence. MIT Press, 1984.

[PP93] N.R. Pal and S.K. Pal. A review on image segmentation techniques. *Pattern Recognition*, 26(9):1277–1294, September 1993.

[RM92] D. Ruprecht and H. Müller. Image warping with scattered data interpolation methods. Technical Report 443, Universität Dortmund, November 1992.

[Rou96] M. Roux. Automatic registration of SPOT images and digitized maps. In *IEEE International Conference on Image Processing ICIP '96, Lausanne, Switzerland*, pages 625–628, September 1996.

[SA04] M. Segal and K. Akeley. *The OpenGL® Graphics System: A Specification (Version 2.0 — October 22, 2004)*. October 2004.

[Sha48] C.E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[She94] J.R. Shewchuk. An introduction to the conjugate gradient method without the agonizing pain. August 1994.

[SHH99] C. Studholme, D.L.G. Hill, and D.J. Hawkes. An overlap invariant entropy measure of 3D medical image alignment. *Pattern Recognition*, 32(1):71–86, January 1999.

[Vio95] P.A. Viola. *Alignment by Maximization of Mutual Information*. PhD thesis, Massachusetts Institute of Technology, June 1995.

[Wen95] H. Wendland. Piecewise polynomial, positive definite and compactly supported radial functions of minimal degree. *Advances in Computational Mathematics*, 4(1):389–396, December 1995.

[Zai97] H. Zaidi. Medical imaging: Current status and future perspectives. In *Conference Record of The International Conference on the Arab World and the Society of Information, Tunis, Tunisia*, pages 1–21, May 1997. Invited Talk.

[ZF03] B. Zitová and J. Flusser. Image registration methods: A survey. *Image and Vision Computing*, 21(11):977–1000, October 2003.