University of Applied Sciences, Dresden

Department of Computer Science

Development of a Parallel Computing Optimized Head Movement Correction Method in Positron-Emission-Tomography

Submitted in partial fulfillment of the requirements for the degree
„Master of Computer Science“

Author:	Jens Langner
Student number:	10895
Supervisor:	Prof. Dr. rer. nat. habil. Heino Iwe
	Prof. Dr. biol. hum. habil. Dr. rer. nat. Jörg van den Hoff
Submission date:	December 03, 2003

Abstract

As a modern tomographic technique, Positron-Emission-Tomography (PET) enables non invasive imaging of metabolic processes in living organisms. It allows the visualization of malfunctions which are characteristic for neurological, cardiological, and oncological diseases. Chemical tracers labeled with radioactive positron emitting isotopes are injected into the patient and the decay of the isotopes is then observed with the detectors of the tomograph. This information is used to compute the spatial distribution of the labeled tracers.

Since the spatial resolution of PET devices increases steadily, the whole sensitive process of tomograph imaging requires minimizing not only the disturbing effects, which are specific for the PET measurement method, such as random or scattered coincidences, but also external effects like body movement of the patient.
Methods to correct the influences of such patient movement have been developed in previous studies at the PET center, Rossendorf. These methods are based on the spatial correction of each registered coincidence. However, the large amount of data and the complexity of the correction algorithms limited the application to selected studies.

The aim of this thesis is to optimize the correction algorithms in a way that allows movement correction in routinely performed PET examinations. The object-oriented development in C^{++} with support of the platform independent Qt framework enables the employment of multiprocessor systems. In addition, a graphical user interface allows the use of the application by the medical assistant technicians of the PET center. Furthermore, the application provides methods to acquire and administrate movement information directly from the motion tracking system via network communication.

Due to the parallelization the performance of the new implementation demonstrates a significant improvement. The parallel optimizations and the implementation of an intuitive usable graphical interface finally enables the PET center Rossendorf to use movement correction in routine patient investigations, thus providing patients an improved tomograph imaging.

Zusammenfassung

Die Positronen-Emissions-Tomographie (PET) ist ein modernes medizinisches Diagnoseverfahren, das nichtinvasive Einblicke in den Stoffwechsel lebender Organismen ermöglicht. Es erfasst Funktionsstörungen, die für neurologische, kardiologische und onkologische Erkrankungen charakteristisch sind. Hierzu werden dem Patienten radioaktive, positronen emittierende Tracer injiziert. Der radioaktive Zerfall der Isotope wird dabei von den umgebenden Detektoren gemessen und die Aktivitätsverteilung durch Rekonstruktionsverfahren bildlich darstellbar gemacht.

Da sich die Auflösung solcher Tomographen stetig verbessert und somit sich der Einfluss von qualitätsmindernden Faktoren wie z.B. das Auftreten von zufälligen oder gestreuten Koinzidenzen erhöht, gewinnt die Korrektur dieser Einflüsse immer mehr an Bedeutung. Hierzu zählt unter anderem auch die Korrektur der Einflüsse eventueller Patientenbewegungen während der tomographischen Untersuchung. In vorangegangenen Studien wurde daher am PET Zentrum Rossendorf ein Verfahren entwickelt, um die nachträgliche listmode-basierte Korrektur dieser Bewegungen durch computergestützte Verfahren zu ermöglichen. Bisher schränkte der hohe Rechenaufwand den Einsatz dieser Methoden jedoch ein.

Diese Arbeit befasst sich daher mit der Aufgabe, durch geeignete Parallelisierung der Korrekturalgorithmen eine Optimierung dieses Verfahrens in dem Maße zu ermöglichen, der einen routinemässigen Einsatz während PET Untersuchungen erlaubt. Hierbei lässt die durchgeführte objektorientierte Softwareentwicklung in C^{++} , unter Zuhilfenahme des plattformübergreifenden Qt Frameworks, eine Nutzung von Mehrprozessorsystemen zu. Zusätzlich ermöglicht eine graphische Oberfläche die Bedienung einer solchen Bewegungskorrektur durch die medizinisch technischen Assistenten des PET Zentrums. Um darüber hinaus die Administration und Datenakquisition der Bewegungsdaten zu ermöglichen, stellt die entwickelte Anwendung Funktionen bereit, die die direkte Kommunikation mit dem Bewegungstrackingsystem erlauben.

Es zeigte sich, dass durch die Parallelisierung die Geschwindigkeit wesentlich gesteigert wurde. Die parallelen Optimierungen und die Implementation einer intuitiv nutzbaren graphischen Oberfläche erlaubt es dem PET Zentrum nunmehr Bewegungskorrekturen innerhalb von Rou-tineuntersuchungen durchzuführen, um somit den Patienten ein verbessertes Bildgebungsverfahren bereitzustellen.

Acknowledgments

My sincere thanks goes to all who supported me throughout the work on this thesis, and to all those who helped me whenever and wherever required.

Special thanks go to the head of the PET center Rossendorf, Prof. Jörg van den Hoff and to Prof. Heino Iwe of the Univerisity of Applied Sciences Dresden who gave me the opportunity to write my thesis in the interesting field of nuclear medicine. The gathered experience during the work on this thesis and the way I was allowed to engage myself in medical research have been unique.

Close interdisciplinary teamwork at the PET center contributed significantly to the success of this thesis. Especially, Dr. Paul Bühler has to be named here, as he guided me throughout the whole time, always willing to answer my questions and concerns in fields of physics and movement correction foundations. I would also like to thank Dr. Edmund Will, Christian Pötzsch and Uwe Just for interesting discussions.

Last, but by no means least, I would like to thank my family for all their support during my study, as well as Jens Tröger (aka *savage*) for being the most supportive and stunning friend ever.
And Sylvia for holding on.

Contents

Introduction

Modern tomography is a medical imaging technique which allows non invasive visualization of internal structures in organisms. There exist different variants of tomography like X-ray Computed Tomography (CT) or Magnetic Resonance Imaging (MRI), which all are used as diagnostic tools in medicine and as scientific analysis tools in the life sciences in general.

After the development of the first CT by Hounsfield in 1972 [Hou72] this method had a steadily increasing impact in the field of radiologic diagnostics and found a large distribution. In CT, an external X-ray source produces radiation which penetrates the examination object and is attenuated during the process. The remaining intensity is measured by X-ray sensitive detectors around the object. Such methods are also called *transmission based tomographic methods* and allow to calculate the regional tissue density.

In parallel, emission tomography has developed which allows the examination of metabolic processes to gain a better understanding of organic functions, or to diagnose metabolism related diseases like cerebral diseases[1]. For discovering alterations like tumors or metastases emission tomography can be helpful, because tumor diseases frequently manifest themselves in changes within the metabolism before tissue modifications are discovered via transmission tomography. In fields of treatment planning and control emission tomography becomes more and more important because it allows physicians to track changes of the metabolism during the patient treatment.

Positron-Emission-Tomography (PET) is the most sensitive variant where the physician injects a positron emitting tracer into the bloodstream of the patient. Electron-positron-annihilation leads to γ-radiation, which is measured by the surrounding detectors, so that the distribution of the labeled substance within the body can be calculated and evaluated. In the beginning, the expensive production of suited radio nuclides was the reason why PET had been a scientific analysis tool only, but with the area-wide appropriation of such tracers PET has become a routinely used method in the diagnosis of metabolic diseases.

In contrast to older PET systems, modern systems allow measurements in a three dimensional mode. Such 3D-PET systems measure more data yielding higher sensitivity, but are also more susceptible to several sources of external errors like scattered coincidences or the partly inevitable patient movement during the acquisition. Some error sources like scattered coincid-

[1]e.g. Depression, Schizophrenia, *Parkinson's*, or *Alzheimer's* disease.

ences can be minimized by better shielding[2] of the Field-of-View (FOV) or through advanced compensation methods. On the other hand, the patient movement can have a high impact on the overall quality of the resulting data because PET examinations can take up to two hours and it is improbable that the patient remains completely still during this acquisition time. Therefore, means to compensate uncontrolled body movements have become more important, and some PET research centers have started to develop methods to include movement correction within PET examinations. The research center at Rossendorf, Germany started working on such methods in mid 2002, when a stereoscopic infrared camera system was installed to allow motion tracking of the patient during data acquisition.

After the physical and mathematical foundations for head motion corrections have been developed [Büh03], the aim of this thesis is to extend the development into the fields of computer science. During the physical study, many computer related problems arise which have a high impact on the performance of such movement corrections. The huge number of coincidence channels and high count rate tolerance of PET scanners leads to a data output which can reach several gigabyte[3] of data. This data needs to be processed and synchronized with the motion tracking data before image reconstruction can be performed. It is obvious that this process puts a high pressure on the underlying computer systems. The aim is to relieve that pressure by using computer science related techniques, especially in fields of parallel computing so that the computation times can be minimized sufficiently to allow routine use of movement corrections in PET.

The main tasks can be summarized as follows:

1. *Parallel Computing Optimization*
   Multi-processor machines are common. Since all modern operating systems do support the Symmetric Multi Processing (SMP) architecture, developers should always consider designing software in a way that gives the underlying operating system the chance to distribute independent parts on different processors. This requires to analyze algorithms and find areas which can be computed in parallel using multithreading. Different multiprocessor machines[4] have to be supported natively, and the parallel computing implementation is required to be based on a POSIX threads (*pthread*) compatible model to keep it portable to other operating systems. Independent computational areas have to be identified and data access needs to be synchronized via semaphores and mutual-exclusive mechanisms to avoid race conditions. The parallel optimizations have to increase the performance of the main algorithms sufficiently to allow processing of the movement corrections for all routinely performed patient examinations.

[2]e.g. by using a so called *neuro shield* during head acquisition.
[3]an ordinary 3D-PET examination of 1 hour produces ≈ 5 gigabyte of raw data.
[4]one 4 processor Sun Ultra v480 with 16GB RAM and one 4 processor Sun v450 with 2GB RAM are available at the PET center.

2. *Platform independent and object-oriented design*
   Keeping software development seminal is an important factor today. Therefore, the implementation of this thesis have to be platform independent. It have to be done with a modern programming language that not only allows to maintain the source code on different platforms, but also to reuse many of the individual parts for future developments. Thus, it has to be implemented by using an object-oriented language and to be designed with the Unified-Modeling-Language (UML), including a class based developer documentation.

3. *Graphical user interface and optional command-line execution*
   To guarantee an intuitive usage, the application have to have a user interface that provides graphical elements for all necessary parts of the movement correction. It have to provide an expert and a novice mode to hide elements that are not necessary for routine operation. Presenting the different functionality in separate parts of the user interface should make the application more easier to use. In addition to the graphical user interface, a batched command line execution have to be possible where the complete functionality is available to the user.

4. *Overall extensibility*
   Especially with medical application development, extensibility plays an important role. The implemented methods and algorithms have to be extensible in their design so that they could be easily adapted to other PET systems or distributed computing techniques. Where applicable, all external interfaces have to use modern and interchangeable data description standards like the eXtensible-Markup-Language (XML).

The layout of this thesis is as follows: Chapter one and two talk about the physical and mathematical foundations. Chapter three discusses the performed analysis in fields of existing solutions and user requirements. Chapter four and five discuss the undertaken parallel analysis as well as the UML specific implementation details. In chapter six a validation of the implementation is presented. Chapter seven and eight discuss possibilities for future developments as well as summarizing the work on this thesis. Appendix A summarizes the functionality and options of the developed application in a user documentation. And finally, in appendix B all developed classes together with their respective filename are presented.

Chapter 1

Positron-Emission-Tomography

As a non invasive, nuclear-medical imaging method, Positron-Emission-Tomography (PET) allows to examine functional processes within a living organism. An injected chemical tracer substance transports positron emitting radio-nuclides through the metabolism of the organism leading to a characteristic distribution, thus making metabolic processes visible. PET is used as an examination method to analyze the cerebral and myocardial metabolism as well as for tumor diagnostics and support of tumor treatment planning and control.

1.1 Physical Fundamentals

According to the atomic model of *Ernest Rutherford*, atoms have a nucleus, which consists of neutrons (n) and protons (p), and is surrounded by electrons (e^-). The number of protons and neutrons within the nucleus controls if an atom is *stable* or if it is *radioactive* and changes its structure by transforming a proton into a neutron or vise versa.
An example of an instable atom is ${}^{13}_{7}N$ which has a half-life of 597,9 seconds and transforms into the stable ${}^{13}_{6}C$. This kind of transformation is also called a β^+-decay where a positron (e^+) and neutrino (ν) are emitted [VBTM03]:

$$\boxed{p \rightarrow n + e^+ + \nu} \tag{1.1}$$

There exist many other positron emitting isotopes. Only those with short half-lifes are of interest in PET because radiation protection is an important aspect of an examination.

The energy difference between the instable element and its stable product is carried away by the emitted particles. While the almost massless uncharged neutrino can fly away unhindered, the electrical positive charged positron interacts with the ambient matter. This continues until it has lost a large portion of its initial kinetic energy and finally ends up in a matter-antimatter reaction with an electron where both masses are transformed into energy. This is also called an *annihilation* process that produces two γ-quanta (photons) with energies of $511keV$ which are emitted in diametrically opposite directions[1], as shown in figure 1.1.

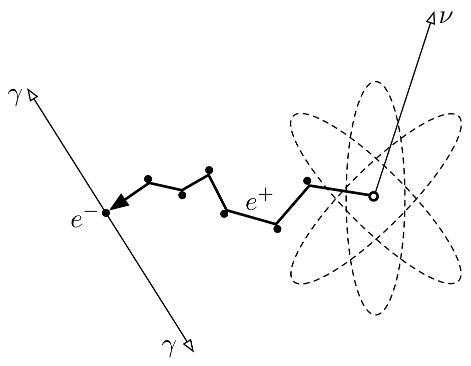


Figure 1.1: β^+-decay and subsequent positron-electron annihilation into two $511keV$ γ-quanta.

1.2 Coincidence Tomography

The annihilation process is the basis of coincidence[2] tomography. If an annihilation within the body of a patient takes place, the γ-quanta fly through the surrounding matter until they leave the body and reach gamma sensitive detectors. These detectors consist of scintillator crystals in which, for their physical characteristics, light flashes are produced when a quantum is absorbed. The flashes are then converted by a photomultiplier into electrical signals which are processed by a coincidence electronic to filter out those events that are received within a limited time window (e.g. $10 - 20ns$). Two γ-quanta detected within this time window describe a so called Line Of Response (LOR) on which the annihilation process must have taken place.

While the γ-quanta are flying through the examined object they interact with the surrounding matter and get attenuated. This attenuation depends on the type of matter, differs from object to object and has to be recorded during the PET examination with a so called *transmission* scan. By taking the data of a transmission and emission scan into account, the image reconstruction can compute the spatial distribution of the tracer. This allows the physician to analyze the distribution of the accumulated tracer at arbitrary positions within the object. This way it is possible to draw conclusions about the metabolism or to visualize tumors and metastases which normally have an elevated metabolism and accumulate the radioactive substance more strongly.

The used radio nuclides have a relatively short half-life which is the reason why a medical facility providing PET examinations needs to produce those nuclides on demand and within a short time frame. Particle accelerators (Cyclotrons) are being used to produce such radio nuclides where stable elements like $^{11}_{5}B$ are bombarded with protons or deuterons which results

[1]with a typical FWHM of angular spread of 0.5°

[2]the term *coincidence* refers to the nearly simultaneous detection of the two annihilation quanta.

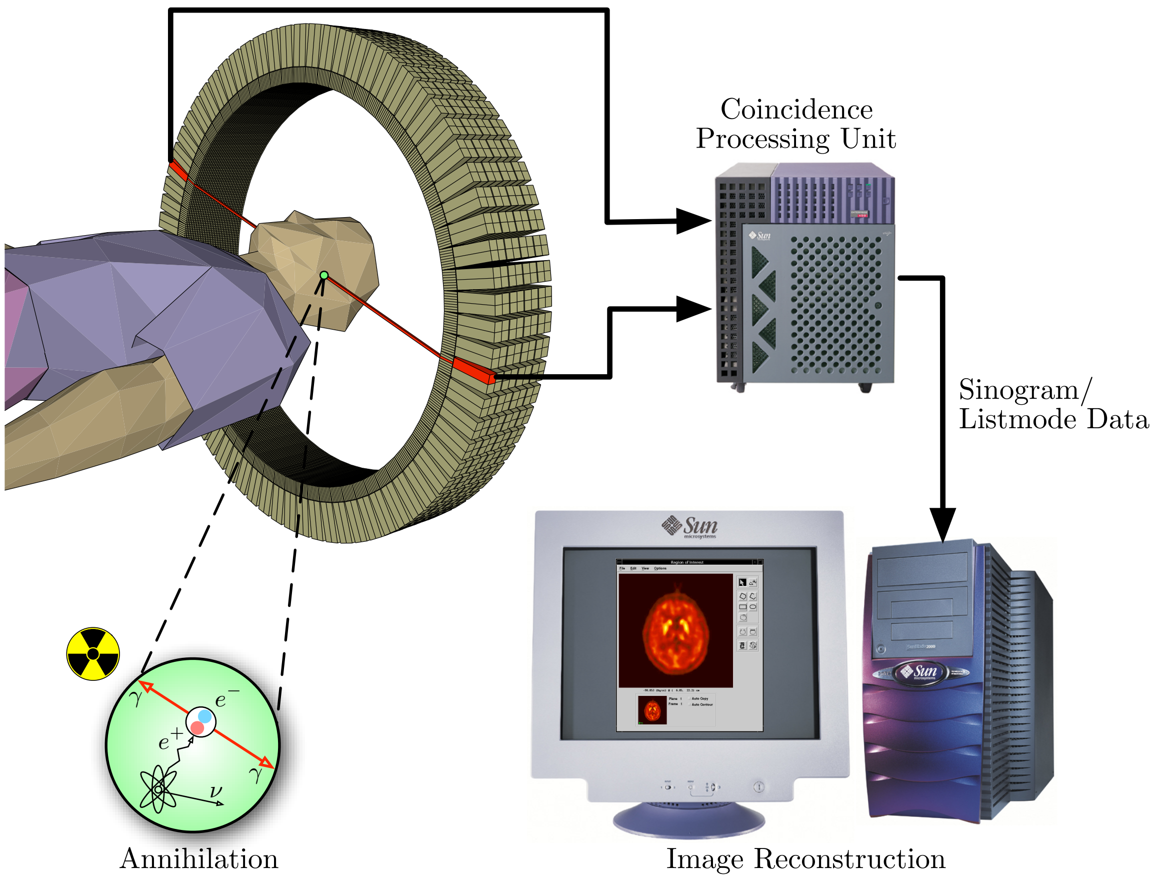


Figure 1.2: Schema showing the different processing steps of the Positron-Emission-Tomography: Starting with the annihilation process through registering the photons at the scanner ring until the final image reconstruction.

in a nuclear reaction that transforms them to e.g. $^{11}_{6}C$. Such a cyclotron is shown in figure 1.3.

1.3 Quality Limitations

The spatial resolution of PET is limited by the physical characteristics of the radioactive decay and the annihilation, but also by technical aspects of the coincidence registration and by external sources of errors, e.g. object movement during the examination. While the range of the positron and its angular deviation limits the resolution to 0.5-3 mm [LH99], the resolution achieved in modern scanners is about 5 mm. The following sections give a short description of the different sources of errors and how they can be reduced.

1.3.1 Physical Influences

1.3.1.1 Positron Lifetime and Angular Deviation

The location where the positron was emitted by the radioactive nucleus is the point of interest. After emission, the positron interacts with electrons of the surrounding matter and moves ran-

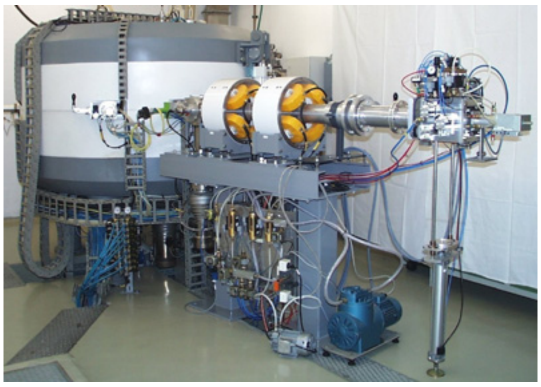


Figure 1.3: Cyclotron for production of ^{11}C, ^{13}N, ^{15}O, ^{18}F

Probes	Usage
$H_2^{15}O$, ^{15}O-buntanol, ^{11}CO, $^{13}NH_3$ ...	hemodynamic parameters
^{18}F-FDG, $^{15}O_2$, ^{11}C-palmitic acid ...	substrate metabolism
^{11}C-leucine, ^{11}C-methionine, ^{11}C-tyrosine	protein synthesis
^{11}C-deprenyl, ^{18}F-deoxyuracil ...	enzyme activity
^{11}C-cocaine, ^{13}N-cisplatin, ^{18}F-fluorouracil ...	drugs
^{11}C-raclopride, ^{11}C-carfentanil, ^{11}C-scopalamine	receptor affinity
^{18}F-fluorodopa, ^{11}C-ephedrine ...	neurotransmitter biochemistry
^{18}F-penciclovir, ^{18}F-antisense oligonucleotides ...	gene expression

Table 1.1: Some tracers and their application in Positron-Emission-Tomography

domly away from the original decay location. The positron range depends on the initial energy of the positron and on the kind of ambient matter it has to pass through (e.g. 1.1 mm in H_2O for $^{11}_{6}C$).

Further influence on the spatial resolution has the angular deviation of the opposed photons. On annihilation the positrons still have a residual energy of approx. $10keV$ and the *conservation of momentum* causes the γ-quanta to be emitted diametrically ($180°$) with an angular deviation of $\pm 0.5°$.

In contrast to the influence of the positron range, the effects of the angular deviation can be limited by reducing detector distances in the PET scanner, as it is done in small animal scanners.[3]

[3]e.g. For a detector radius of $100cm$ the deviation of the coincidence line is $\approx 2.6mm$ [Keh01].

1.3.1.2 Photon Attenuation

The two annihilation photons are attenuated while traversing through the examination object. The attenuation can amount up to 95% in a human body examination[4] [Keh01]. However, it can be measured by a *transmission scan* where a γ-radiating source like ^{68}Ge is used to irradiate the object from the outside. In addition to this transmission measurement a *blank scan* without any object in the FOV is performed.
Taking the data of the transmission scan and the blank scan, allows to compute the overall photon attenuation and therefore compensate its effects:

$$PhotonAttenuation = \frac{TransmissionScan}{BlankScan}$$

1.3.1.3 Isotope Lifetime

The radioactive decay of the injected nuclides causes the counted coincidences to decrease exponentially with time. This is normally compensated to obtain the intensity (A_0) at the beginning of the acquisition. The decay rate depends on the lifetime of the isotope and a correction factor (f) can be calculated according to

$$f = \frac{N_c}{N_m} = \frac{(t_e - t_s)A_0}{\int\limits_{t_s}^{t_e} A_t dt} \tag{1.2}$$

where N_c is the number of corrected counts, N_m the number of measured counts, A_0 the activity of the radio nuclide at start time (t_s), t_e the end time and $A_t = A_0 e^{-\lambda t}$ the radioactivity at a specific time [VBTM03].

1.3.2 Scanner Influences

1.3.2.1 Random Coincidences

The time window that the coincidence electronic applies does not only contain true coincidences. It happens that *random coincidences* are counted because two quanta that do not originate from the same annihilation event, arrive within the same time window. Such a situation can also happen if a quantum from outside the F̲ield O̲f V̲iew (FOV) arrives at the same time like another one[5]. Random coincidences are calculated by

$$N_{rand} = 2\tau N_i N_j \tag{1.3}$$

where N_i and N_j are the γ-rates (Singles) of both detectors and τ the length of the time window.

Random coincidences can also be measured directly by applying a second time window where the signal of one detector is delayed, so that simultaneous detections in the two time windows is a direct measure of the random events [HHPK81].

[4]head acquisitions have an attenuation of approx. 75-80%.
[5]often the bladder of a patient accumulates lots of radioactivity and produces random coincidences.

1.3.2.2 Scattered Coincidences

Within an electron-free environment the emitted γ-quanta do hit the detectors straight from the annihilation source position. In reality the photons interact with electrons (*Compton Scattering*) so that a photon hits the wrong detector and is assigned to a false LOR. Even if the probability of Compton Scattering is very high[6], the resulting angle and therefore the probability that a wrong detector is being hit is relativly low [Dav55].
During this scattering the photons loose some of their energy which allows to filter out those coincidences by setting energy limits for events to be accepted. Unfortunately, the average energy loss is low so that this filtering is only useful for strongly scattered photons. Photons loosing only a small amount of energy during the scattering, can however be calculated by applying mathematical methods [WWH88].
Finally, the real coincidences (Trues) are calculated by using the formula

$$N_{true} = N_{tot} - N_{rand} - N_{sc} \tag{1.4}$$

where N_{tot} are the counted coincidences (Prompts), N_{true} the Trues, N_{rand} the Randoms and N_{sc} the scattered coincidences.

1.3.2.3 Variable Detector Sensitivity

Because of differences between the photo multipliers and scintillator crystals, each detector has a different sensitivity. If kept uncorrected, those differences do result in an inhomogeneous distribution of the counted coincidences. By performing a scan with a low-activity phantom[7] radiating γ-quanta, it is possible to build a map of this inhomogeneous distribution which then is merged with the data from the final scan to compensate those differences. This is also referred to as *Normalization*.

1.3.2.4 Electronic Dead Time

The *electronic dead time* describes the limitation that electronic components process events only at a limited rate. The same applies for components of the PET scanner where it happens that the coincidence processor is busy during the arrival of other coincidences and is not able to measure them. The event rate dependence of the dead time can be measured by performing a phantom scan with an initially high activity. With help of this measurement the dead time influence is compensated within the final image reconstruction.

1.3.2.5 Crystal Characteristics

The scintillator crystals transform parts of the energy deposited by the γ-quanta into visible light. This conversion depends on the material characteristics of the crystals like density, light

[6] $\approx 50\%$ during a brain examination [Keh01].
[7] a phantom is a radiation source for calibration and study purposes.

efficiency, decay time and atomic number. Some common used crystal materials are listed in table 1.2.

Material $[g/cm^3]$	Density [%]	rel. light efficiency $[ns]$	Decay time	Hygroscopic	Atomic number
NaI	3.67	100	230	yes	50
BGO	7.13	15	300	no	75
YAP	5.37	40	25	no	36
LSO	7.4	75	40	no	66
GSO	6.71	23	60	no	59
CsF	4.64	6	5	yes	52

Table 1.2: Physical characteristics of common scintillator crystals [Pie99]

Geometry and arrangement of those scintillator crystals is also important for the coincidence recognition. As the crystals are highly packed, it happens that when a photon hits a crystal with an angle different from 90° it traverses the first crystal and is absorbed by a neighboring one. This is the reason why typically the best resolution of a PET scanner can be achieved within the center of the FOV, where all LORs hit the crystals under 90°. However, such effects can be lowered by reducing the crystal lengths.

1.3.3 External Influences

1.3.3.1 Organ Movement

Since PET is being used to retrieve metabolic information on a living organism, the normal object of interest is, in contrast to static objects like phantoms, an object that has a dynamic behavior. Sometimes this is a volitional dynamic behavior[8], but more often it degrades the final results of the examination so that those dynamics need to be compensated.

Periodic movements like those from heart or respiration can be compensated by recording the periodicity with tools like an electrocardiogram (ECG) or respiration belt, and by mapping the different phases of the motion to different time frames (gates) to summarize only the coincidences at a specific position of the organ. Whereas heart examinations with help of a ECG are supported natively by most of the modern PET scanners, respiratory movements or other organ movements are not supported and cannot be compensated easily.

1.3.3.2 Patient Movement

It cannot be expected that patients keep still during the data acquisition of a PET examination of up to two hours, especially if they have diseases like *Parkinson* or *Epilepsy*, where uncon-

[8]in case of a blood flow examination.

trolled and unpredictable movements are unavoidable. Even if patients are supported with special devices like vacuum cushions, such movements cannot be totally avoided and have to be compensated by other means. The impact of such movements on the resulting image quality is often underestimated, and since the resolution of the scanners is improving, the importance of compensating such movements is steadily increasing.

1.4 Quantification

To draw quantitative conclusions about the metabolic function of an examined organism it is necessary to analyze the measured radioactive level in a specific Region Of Interest (ROI). A scale factor calculation allows to map the counted coincidences back to a radioactive level. This factor is determined by using a scan of a phantom of which the specific activity is known and the difference to the actually measured coincidences is calculated. With this method it is possible to provide quantitative statements in $Becquerel/cm^3$ for a specific ROI.

1.5 The PET Scanner - ECAT EXACT HR$^+$

During the studies for this thesis a PET scanner was used to perform tests and analysis of the implementation of the correction algorithms. This scanner, as shown in figure 1.4, was developed by CTI and Siemens in 1996 and supports 2D and 3D body examinations.

The γ-sensitive scintillator crystals of this tomograph are combined into groups of 8×8 crystals, that form a *detector block* that is connected to 2×2 photo multipliers. 72 of these detector blocks form a *detector ring* and four of these rings are arranged in axial direction, resulting in the total number of 18432 *BGO* crystals, as shown in figure 1.5.

Figure 1.4: EXACT HR$^+$ PET scanner

The scanner supports the extraction of 0.8 *mm* thick *septa* rings for a 2D measurement. If running in 2D mode, those 66.5 *mm* long barriers are physically restricting the maximum possible ring difference within the detector system, thus limiting the influence of scattered coincidences. In contrast, if running in 3D mode (cf. figure 1.6) the overall sensitivity of the PET scanner is increased by a factor of 3-5 [Keh01], so that the injected dose can be decreased, the examination time reduced or the statistical accuracy improved.

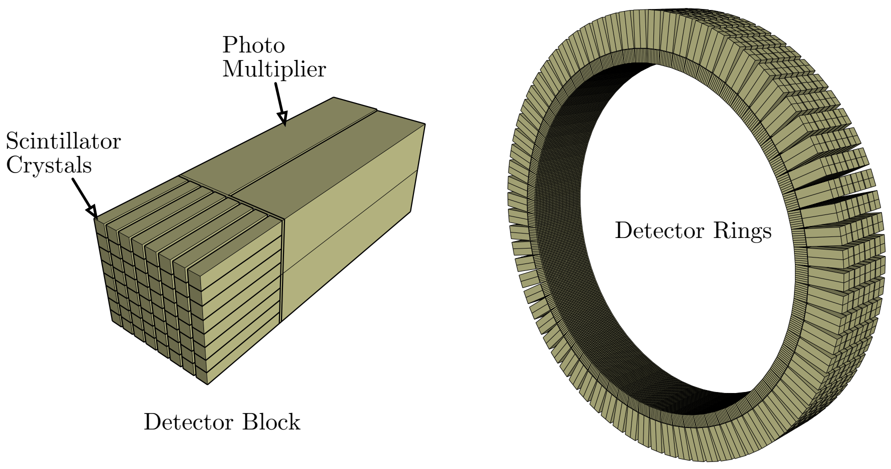


Figure 1.5: Detector system layout showing a detector block and 4 detector rings.

By assuming that coincidences of a point source which is located exactly in between two neighbouring detectors are not recognized, *interleaving* describes a technique to increase the possible angular combinations during sorting LORs into their respective bins[9]. This is illustrated in figure 1.7. It increases the angular combinations such that the scanner accepts 576 different

[9]Here a *bin* refers to a single element in a two dimensional histogram.

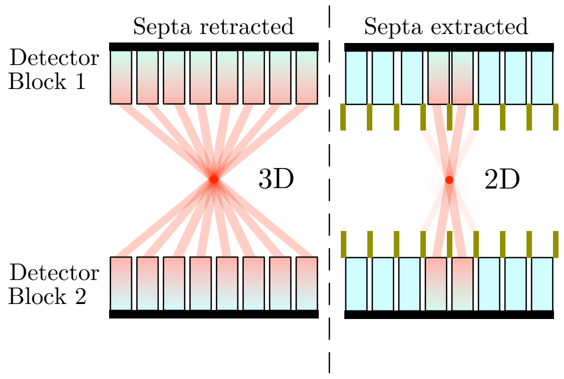


Figure 1.6: PET scanner 2D/3D Mode principles

angles on one ring unit with 576 crystals instead of just the normal 288. However, those 288 additional interleave angles are combined with the other ones so that the final dataset still consist of 288 angles [Sie96]. Due to this technique the resolution especially near the center of the FOV is increased.

In addition, the scanner provides a method to reduce the amount of data an acquisition produces. The so called *angular compression*[10] allows the scanner to summarize successive angles to one logical unit and is per default set to mash 2 so that in combination with the interleaving the resulting dataset only consists of 144 angles per layer. Unfortunately, this does not only decrease the amount of data but also the resolution within the outer areas of each plane.

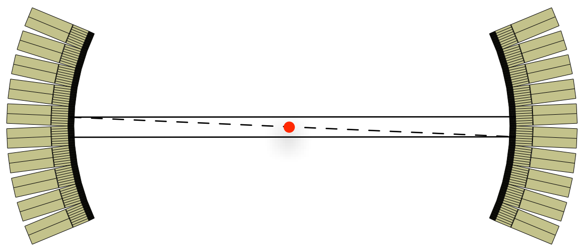


Figure 1.7: Interleaving Technique

There exist two different types of *planes*, direct and indirect ones (cf. figure 1.8). The combination of accepted angles within the direct and indirect layers is called a *Span* level where a *Span7* (3 + 4) refers to three direct plus four indirect layers. A layer itself consist of LOR combinations of one or more detector rings. By increasing the axial accepted angle of LOR combinations, the sensitivity of the scanner can be increased. However, it also leads to a decrease

[10]also known as *mashing*

of the resolution within the outer ranges of the layers. Thus, this technique is understood not only to take a LOR as a single coincidence line but to consider a LOR a logical volume of coincidences.

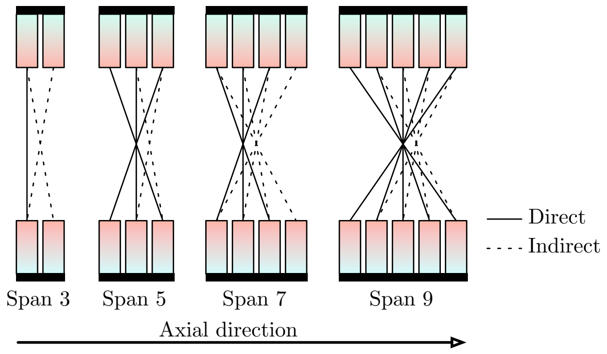


Figure 1.8: Axial Acceptance Angle

As infinitely increasing the Span causes too much of a resolution loss, the scanner divides the dataset into different segments, where each segment is scanned with the same Span, but includes differently inclined LORs as shown in figure 1.9. The number of segments is defined by the maximum available ring difference (RD_{max}) and Span and can be calculated by

$$N_{seg} = \frac{2 \times RD_{max} + 1}{Span} \tag{1.5}$$

where the number of segments has always to be odd, and is per default 5 for the ECAT EXACT HR$^+$ scanner with Span 9.

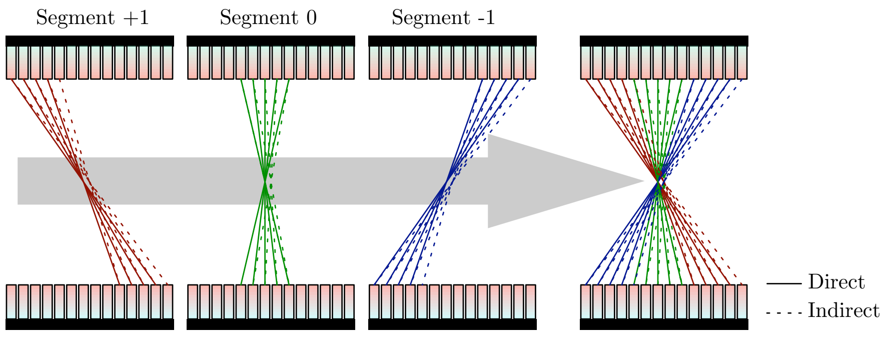


Figure 1.9: Sinogram Segments for Span 9 and $RD_{max} = 13$

To illustrate the connection between the Span, the maximum ring difference and the separation into several segments *Michelograms* are used, developed by Christian Michel (Université Catholique de Louvain, Belgium) and CTI [Sie96], where the axes of the diagram enumerate the different detector rings so that each point within the diagram stands for a possible ring combination. The connected points are referring to combined LORs where the diagonal of each segments shows the different layers within the segment. Such a Michelogram is given in figure 1.10 and the default parameters of a 2D and 3D scan are listed in table 1.3.

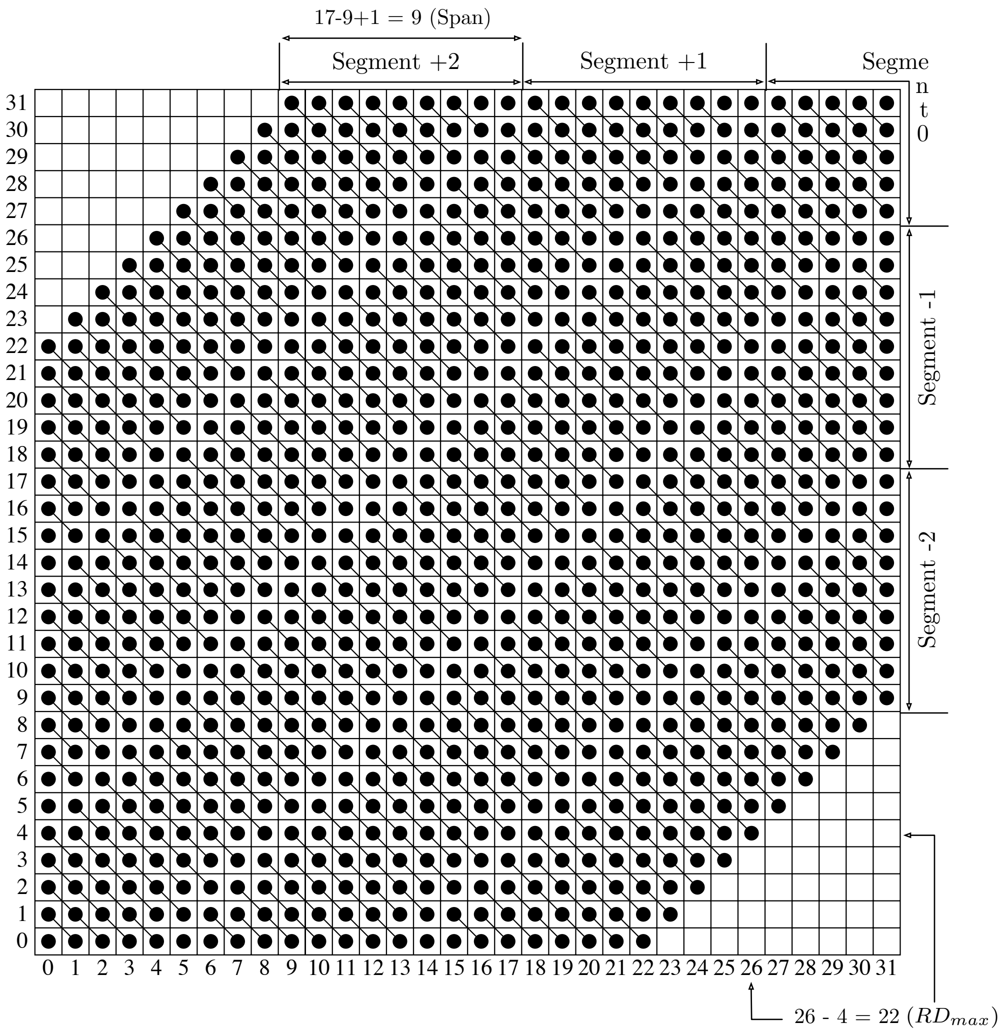


Figure 1.10: Michelogram for Span 9 and $RD_{max} = 22$ [Keh01]

Mode	N_{seg}	$Span$	Segment	RD_{min}	RD_{max}	RD	Planes
2D-Measurement	1	15	0	-7	7	0	63
3D-Measurement	5	9	0	-4	4	0	63
			±1	5	13	9	53+53
			±2	14	22	18	35+35

Table 1.3: Default parameters of a 2D/3D measurement

Chapter 2

Coincidence Position Correction

The unavoidable patient movement during the PET acquisition has impacts on the quality of the examination. As scanner resolutions are being steadily improved, any movement which is comparable to the size of the intrinsic spatial resolution causes a blurring and therefore a loss of information. Therefore, corrections of such movements have to be performed and recent studies show that the herein presented *coincidence position correction* is a appropiate technique to compensate such movements. The following sections discuss this techniques fundamentals which are used to compensate the patient's head movement [Büh03].

2.1 Different Coordinate Systems

The PET scanner and the motion tracking system have distinct coordinate systems. As illustrated in figure 2.1, a point $\vec{P}$ that moved during the acquisition to point $\vec{Q}$ has different coordinates in the respective systems. To use the motion tracking data for correcting patient movement, it is necessary to be able to convert movements measured in the coordinate system CS_{trk} of the tracking camera to the scanner coordinate system CS_{pet} and vice versa.

As the coincidence data of the PET scanner have to be corrected, the measured transformation within CS_{trk} has to be converted into the corresponding transformation in CS_{pet}, such that the position of coincidences can be corrected within the scanner's coordinate system.

2.1.1 Cross-Calibration

Being able to map the coordinates of a given point $\vec{P}$ to the corresponding coordinates $\vec{P}'$ in another system is called *cross-calibration.* Two systems are cross-calibrated to each other if both the rotation matrix $\hat{T}_{cc}$ and the translation vector $\vec{t}_{cc}$ which transform a vector $\vec{P}$ in coordinate system CS_{trk} to a vector $\vec{P}'$ in the system CS_{pet}, are known:

$$\vec{P}' = \hat{T}_{cc} \cdot \vec{P} + \vec{t}_{cc} \tag{2.1}$$

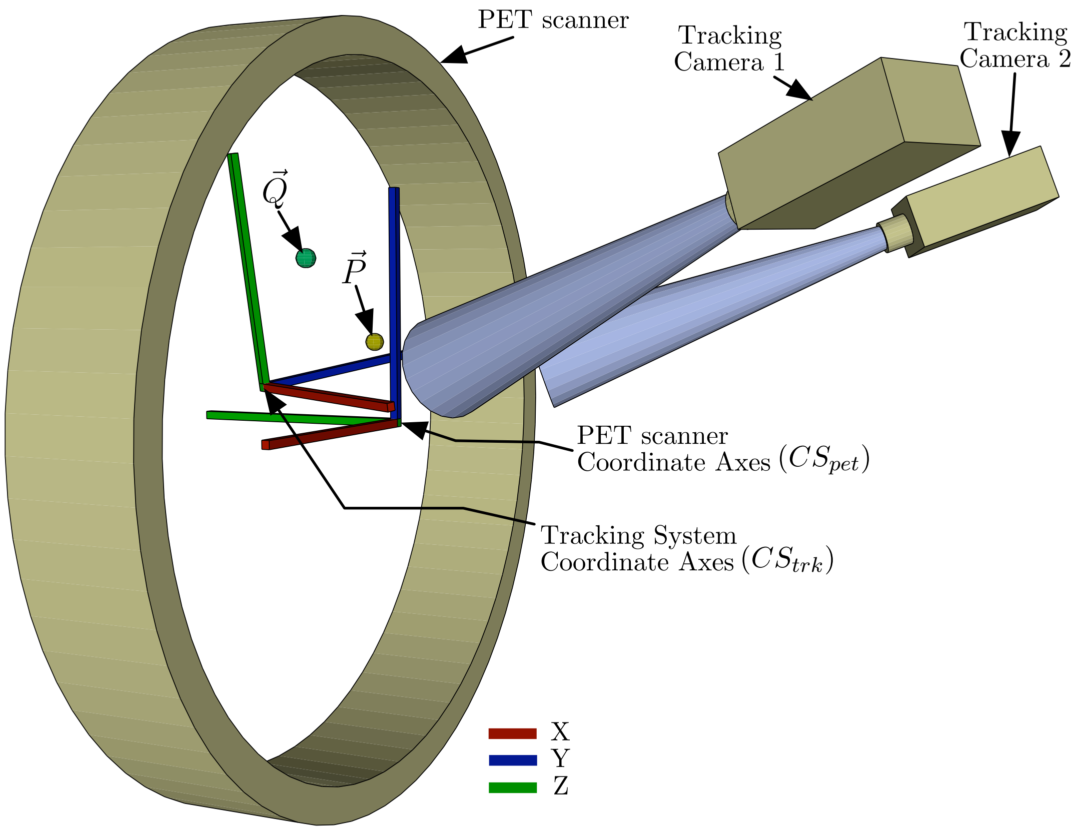

PET scanner
Tracking Camera 1
Tracking Camera 2
$\vec{Q}$
$\vec{P}$
PET scanner Coordinate Axes (CS_{pet})
Tracking System Coordinate Axes (CS_{trk})
X
Y
Z

Figure 2.1: The different coordinate systems involved in movement correction. A point $\vec{P}$ that moved to point $\vec{Q}$ has different coordinates in both systems, thus a calibration of both coordinate systems is necessary to map the movement between the systems.

In order to apply the measured spatial transformations in CS_{trk} for correction of the coincidences in CS_{pet}, the cross-calibration ($\hat{T}_{cc}$, $\vec{t}_{cc}$) between those two systems have to be determined prior to the movement correction. Therefore, a simultaneous measurement of a set of spheric bodies within the FOV of the tracking system and PET scanner is performed, which allows to measure the orientation of the systems to each other, i.e. to compute $\hat{T}_{cc}$ and $\vec{t}_{cc}$.

For our motion tracking system and PET scanner this measurement is accomplished through a transmission scan where an object, that can be tracked by the motion tracking system, is fixed within the FOV of the scanner during the scan. The object does not only have to be visible to the tracking system, but also need to cause a significant attenuation during the transmission scan so that it is visible in the final PET image and can be mapped onto the coordinates provided by the tracking system.

This object has to provide a minimum of four motion-trackable bodies so that the motion tracking system outputs four different vector coordinates $\vec{P}_1, ..., \vec{P}_4$. This requirement is based on the mathematical procedure where by solving the system of equations (2.2), $\hat{T}_{cc}$ and $\vec{t}_{cc}$ can be computed with the required accuracy.

$$\begin{aligned}
P'_{1_x} &= T_{cc1,1} \cdot P_{1_x} + T_{cc1,2} \cdot P_{1_y} + T_{cc1,3} \cdot P_{1_z} + t_{cc_x} \\
P'_{1_y} &= T_{cc2,1} \cdot P_{1_x} + T_{cc2,2} \cdot P_{1_y} + T_{cc2,3} \cdot P_{1_z} + t_{cc_y} \\
P'_{1_z} &= T_{cc3,1} \cdot P_{1_x} + T_{cc3,2} \cdot P_{1_y} + T_{cc3,3} \cdot P_{1_z} + t_{cc_z} \\
&\vdots \\
P'_{4_x} &= T_{cc1,1} \cdot P_{4_x} + T_{cc1,2} \cdot P_{4_y} + T_{cc1,3} \cdot P_{4_z} + t_{cc_x} \\
P'_{4_y} &= T_{cc2,1} \cdot P_{4_x} + T_{cc2,2} \cdot P_{4_y} + T_{cc2,3} \cdot P_{4_z} + t_{cc_y} \\
P'_{4_z} &= T_{cc3,1} \cdot P_{4_x} + T_{cc3,2} \cdot P_{4_y} + T_{cc3,3} \cdot P_{4_z} + t_{cc_z}
\end{aligned} \tag{2.2}$$

$\vec{P}'_i$ is a position vector of object i within the PET attenuation image and $\vec{P}_i$ the position of the same object, but obtained from the data of the motion tracking system.

By using algorithms like *Singular Value Decomposition* [PTVF92] the system of equations (2.2) can be solved so that with the homogeneous coordinate based matrix-vector multiplication

$$\begin{pmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{pmatrix} = \begin{pmatrix} T_{cc1,1} & T_{cc1,2} & T_{cc1,3} & t_{cc_x} \\ T_{cc2,1} & T_{cc2,2} & T_{cc2,3} & t_{cc_y} \\ T_{cc3,1} & T_{cc3,2} & T_{cc3,3} & t_{cc_z} \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} P_x \\ P_y \\ P_z \\ 1 \end{pmatrix} \tag{2.3}$$

vector $\vec{P}'$ within CS_{pet} can be computed from $\vec{P}$ in CS_{trk} or vice versa.

As the two systems are normally located at fixed positions, such a cross-calibration has to be performed only in routinely based intervals to avoid drifting effects, or in case that one system changes its coordinate system, e.g. by recalibration of the motion tracking system.

2.1.2 Spatial Movement

Patient movement during acquisition is a spatial transformation and can be expressed by

$$\vec{Q} = \hat{T}_{mc} \cdot \vec{P} + \vec{t}_{mc} \tag{2.4}$$

where $\vec{P}$ is the initial position, $\hat{T}_{mc}$ the rotation matrix, $\vec{t}_{mc}$ the translation vector and $\vec{Q}$ the final position.

The used motion tracking system directly provides the rotation matrix and translation vector of the movement so that, after applying the cross-calibration transformation of section 2.1.1 the position of $\vec{P}'$ (cf. equation 2.3) can immediately be shifted to $\vec{Q}'$. The combination of cross calibration and spatial movement can be expressed in one equation:

$$\begin{pmatrix} Q'_x \\ Q'_y \\ Q'_z \\ 1 \end{pmatrix} = \begin{pmatrix} \hat{T}_{cc}\hat{T}_{mc}\hat{T}_{cc}^{-1} & -(\hat{T}_{cc}\hat{T}_{mc}\hat{T}_{cc}^{-1}) \cdot \vec{t}_{cc} + \hat{T}_{cc} \cdot \vec{t}_{mc} + \vec{t}_{cc} \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} P'_x \\ P'_y \\ P'_z \\ 1 \end{pmatrix} \tag{2.5}$$

2.2 Position Correction Procedure

Single coincidence position correction requires that every coincidence is recorded separately. Modern PET scanners like the EXACT HR$^+$ achieve this in *listmode*, where all coincidence events are stored as 32bit encoded words (cf. section 3.3.2.1). Those words are separated into *time words* and *event words* where an event word is the representation of a single LOR and a time word an absolute time information since scan start.

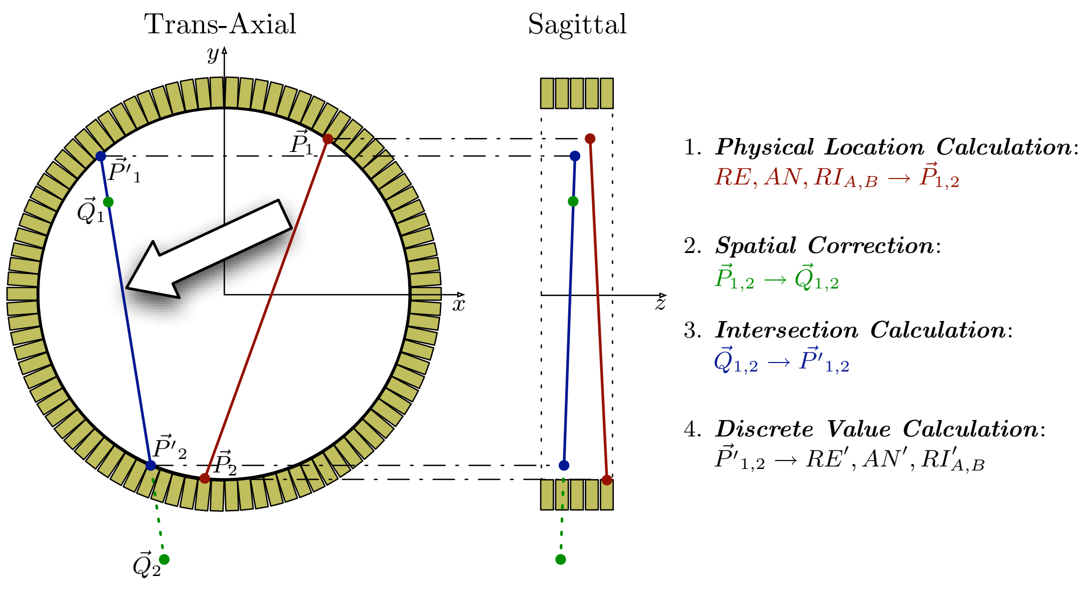

Trans-Axial
Sagittal
y
x
z
$\vec{P}_1$
$\vec{P'}_1$
$\vec{Q}_1$
$\vec{P'}_2$
$\vec{P}_2$
$\vec{Q}_2$
1. ***Physical Location Calculation***: $RE, AN, RI_{A,B} \rightarrow \vec{P}_{1,2}$
2. ***Spatial Correction***: $\vec{P}_{1,2} \rightarrow \vec{Q}_{1,2}$
3. ***Intersection Calculation***: $\vec{Q}_{1,2} \rightarrow \vec{P'}_{1,2}$
4. ***Discrete Value Calculation***: $\vec{P'}_{1,2} \rightarrow RE', AN', RI'_{A,B}$

Figure 2.2: Image illustrating the different steps of the position correction procedure. After having transformed the discrete values of the raw acquisition data into absolute physical location (1), the spatial transformation is applied (2). Afterwards the new positions are mapped onto the scanner circumference (3) and converted to discrete scanner values again (4). Those are then used and finally sorted into a sinogram file.

As shown in figure 2.2, coincidence position correction can be divided into four different steps which we discuss in the following paragraph. Scanner parameters relevant to the procedure are summarized in table 2.1. The last three items are discrete parameters ($RE, AN, RI_{A,B}$) which are encoded in the event words of the listmode stream.

1. ***Physical Location Calculation*** ($RE, AN, RI_{A,B} \rightarrow \vec{P}_{1,2}$)
   Before the LOR dependent data of a listmode file can be used to correct the position of a coincidence, discrete values of the listmode event words have to be converted to absolute physical values within the PET scanner coordinate system.
   In case of the EXACT HR$^+$, a listmode event word consists of four discrete values RE, AN and $RI_{A,B}$ which unambiguously define a LOR. They can be used to compute the two

Parameter	**Notation**	**EXACT HR$^+$**
detector ring radius	R	41 cm
minimum Z-axis value of axial FOV	z_0	7.76 cm
axial width of planes	d_z	0.485 cm
trans-axial FOV angle	β	45°
number of radial bins	N_{RE}	288
number of angular bins	N_{AN}	288
radial bin number	RE	0...287
angular bin number	AN	0...287
detector ring numbers	$RI_{A,B}$	0...31

Table 2.1: Coincidence position correction relevant scanner parameters. The last three items are encoded within the provided *event words* during listmode of the EXACT HR$^+$ scanner.

endpoints $\vec{P}_{1,2}$ of the LOR on the crystal rings:

$$\vec{P}_{1,2} = \begin{pmatrix} -\rho \cdot \sin\alpha \pm r \cdot \cos\alpha \\ \rho \cdot \cos\alpha \pm r \cdot \sin\alpha \\ z_0 + (RI_{A,B} + 0.5) \cdot d_z \end{pmatrix} \tag{2.6}$$

where

$$\alpha = \frac{AN \cdot \pi}{N_{AN}}$$
$$\rho = R \cdot \sin\left(\frac{(RE + 0.5) \cdot 2\beta}{N_{RE}} - \beta\right)$$
$$r = \sqrt{R^2 - \rho^2}$$

Here α, ρ and r are physical quantities which are temporary used to calculate the endpoints. Furthermore, it is assumed that a LOR is the connecting line between the centers of the two involved crystal detectors.

2. ***Spatial Correction*** ($\vec{P}_{1,2} \rightarrow \vec{Q}_{1,2}$)
After the physical endpoints of a LOR have been calculated, the spatial LOR correction is applied by transforming both points $\vec{P}_{1,2}$ with equation (2.5) using the corresponding transformation information. This results in the two intermediate points $\vec{Q}_1$ and $\vec{Q}_2$ which represent the transformed endpoints.

3. ***Intersection Calculation*** ($\vec{Q}_{1,2} \rightarrow \vec{P'}_{1,2}$)
In the next step, the intersection points $\vec{P}'_{1,2}$ with the detector ring of the line through

$\vec{Q}_{1,2}$ have to be determined. The coordinates of $\vec{P}'_{1,2}$ can be calculated from

$$\begin{aligned} \vec{P} &= \vec{Q}_2 + q \cdot (\vec{Q}_1 - \vec{Q}_2) \\ P_x^2 + P_y^2 &= R^2 \end{aligned} \tag{2.7}$$

which results in a quadratic equation for q

$$\begin{aligned} 0 &= q^2 \cdot (dx^2 + dy^2) + 2q \cdot (Q_{2_x} \cdot dx + Q_{2_y} \cdot dy) + (Q_{2_x}^2 + Q_{2_y}^2 - R^2) \\ dx &= Q_{1_x} - Q_{2_x} \\ dy &= Q_{1_y} - Q_{2_y} \end{aligned} \tag{2.8}$$

with two possible solutions $q_{1,2}$ so that $\vec{P}'_{1,2}$ can be computed by

$$\begin{aligned} \vec{P}'_1 &= \vec{Q}_2 + q_1 \cdot (\vec{Q}_1 - \vec{Q}_2) \\ \vec{P}'_2 &= \vec{Q}_2 + q_2 \cdot (\vec{Q}_1 - \vec{Q}_2) \end{aligned} \quad \text{with} \quad q_1 > q_2 \tag{2.9}$$

4. ***Discrete Value Calculation*** $(\vec{P'}_{1,2} \rightarrow RE', AN', RI'_{A,B})$
Finally, $\vec{P'}_{1,2}$ are converted into listmode compatible discrete values. This is done by the inverse of equation (2.6):

$$\begin{aligned} RE' &= int\left(\frac{(\arcsin(\rho'/R) + \beta)}{2\beta} \cdot N_{RE}\right) \\ AN' &= int\left(\frac{\alpha' \cdot N_{AN}}{\pi}\right) \\ RI'_{A,B} &= int\left(\frac{z'_{1,2} - z_0}{d_z}\right) \end{aligned} \tag{2.10}$$

where

$$\begin{aligned} \rho' &= \frac{P'_{2_x} \cdot dy - P'_{2_y} \cdot dx}{\sqrt{dx^2 + dy^2}} \\ \alpha' &= \arctan\left(\frac{dy}{dx}\right) \\ z'_i &= P'_{i_z} \end{aligned} \quad \text{with} \quad \begin{aligned} dy &= P'_{1_y} - P'_{2_y} \\ dx &= P'_{1_x} - P'_{2_x} \end{aligned} \tag{2.11}$$

and $int()$ is the integer fraction of a floating point value.

Before such transformed LORs can be sorted into a sinogram or appended to a separate listmode file, a number of corrections have to be performed as described below.

2.2.1 Problems and Solutions

When spatially transforming LORs, neglecting scanner characteristics like variable detector sensitivities, the trans-axial bin widths within one plane and the limited FOV will produce artifacts in the final data. Therefore, a thorough correction of these effects is mandatory.

2.2.1.1 Normalization Correction

As discussed in section 1.3.2.3, the detectors of a scanner have different sensitivities which are compensated by *Normalization* during the reconstruction. Therefore, the sensitivity of the detectors which originally detected the event is not the same as the one of the detectors computed from equation (2.10) for the corrected LOR. This difference results in incorrect normalization during reconstruction and produces typical *ring-artifacts* in the final image, cf. figure 2.3.

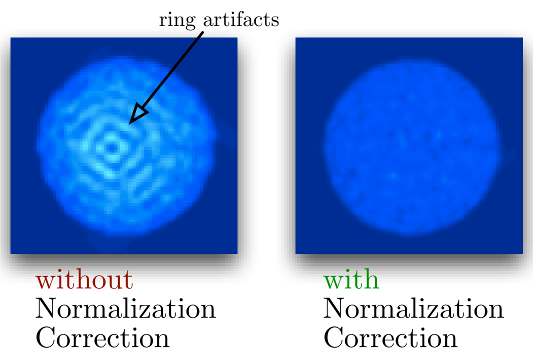

ring artifacts
without Normalization Correction
with Normalization Correction

Figure 2.3: Images showing the impact of an uncorrected normalization during a movement correction. If kept uncorrected typical ring artifacts within the final image show up, which is caused by the different detector sensitivities of a PET scanner.

These artifacts can either be eliminated by modifying the normalization used during image reconstruction, or by calculating correction factors to each registered LOR so that the standard normalization can be used despite movement correction.

By assuming a *mash* value of zero and *span* of one (cf. section 1.5) each logical LOR consists of one possible detector combination. Therefore, the normalization factor $\eta_{(i,j)}$ of a LOR consists of two detector efficiencies, and is calculated by

$$\eta_{(i,j)} = \eta_i \cdot \eta_j \cdot \phi \tag{2.12}$$

where $\eta_{i,j}$ are the single detector efficiencies and ϕ a geometric factor which describes the influence of the angle of incidence of a γ-quantum on the detection efficiency [CGN95], and will not further be discussed here.
If we now transform a LOR from detector pair (k, l) via the methods described in section 2.2, into a LOR at pair (i, j) the normalization correction factor is:

$$f_{(k,l),(i,j)} = \frac{\eta_{(k,l)}}{\eta_{(i,j)}} \tag{2.13}$$

Weighting each corrected event with this factor allows to use the default normalization during reconstruction, as desired.

2.2.1.2 LOR Discretization Correction

The assumption that a single LOR is the connecting line between the centers of two detectors leads to problems during the movement correction.

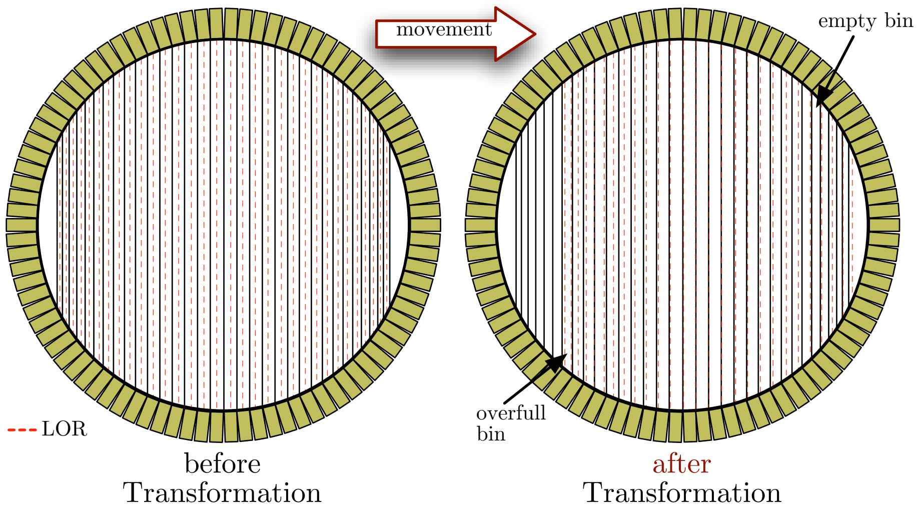


Figure 2.4: Trans-axial plane of a scanner showing the different bin widths due to the radial placement of the detectors. After considering a spatial transformation, detector-center-aligned LORs are moved so that either *empty bins* or *overfull bins* are produced. This causes artifacts in the final reconstructed image, if kept uncorrected.

The sketch 2.4 of a trans-axial plane of a PET scanner illustrates the problem. The left panel shows the initial situation where the LORs connect the centers of two opposite detectors. The width of the *bins* varies depending on the distance of the LOR from the center of the FOV. The right panel shows the same trans-axial plane but with spatially transformed LORs. By concentrating on the dotted LORs, the drawing illustrates that due to the spatial transformation the left side of the FOV carries some LORs that fell into the same bin (*overfull bin*), whereas the right side contains some *empty bins* which leads to artifacts within the reconstructed image.

Referring to figure 2.5, that problem can be avoided by considering LORs as a volume confined by the planes connecting the edges of two detector crystals (rod with rectangular cross-section), rather than a simple connecting line between the detectors. In this case, a transformation can not lead to uncovered bins (except for the fringes, cf. section 2.2.1.3). However, such a transformed LOR is generally not exactly aligned to a single bin and does intersect with several of them. In order to assign such an intersecting LOR to a bin, a weighting factor has to be calculated which is proportional to the amount of overlap between a LOR and a particular bin.

Computing this overlap is time consuming. Therefore, a simplified scheme is used, assuming a constant bin width in axial direction.

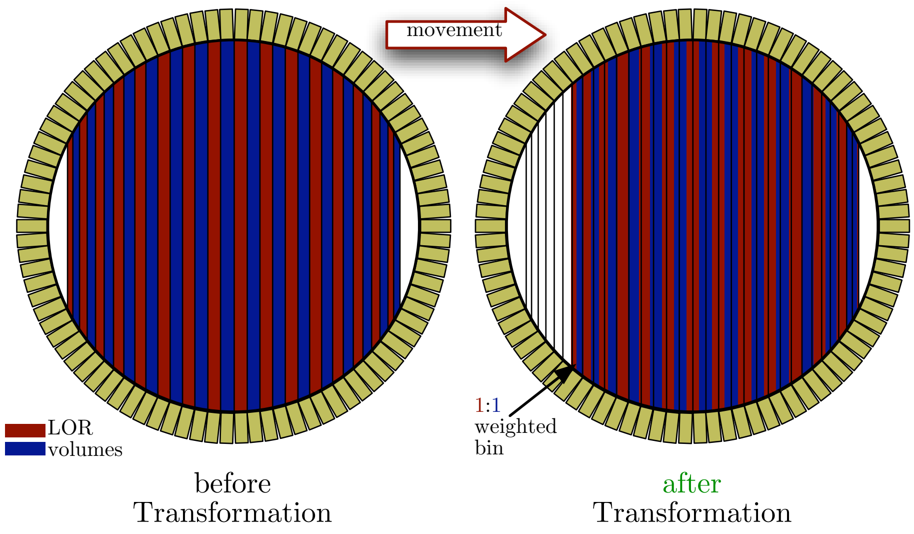


Figure 2.5: By considering transformed LORs to be volumes instead of lines connecting the center of two detectors, the LOR discretization correction is able to calculate weights on each transformed LOR. This way *overfull bins*, cf. figure 2.4, are prevented whereas *empty bins* at the outer ranges of the FOV have to be compensated by an additional Out-of-FOV correction (see section 2.2.1.3).

2.2.1.3 Out-of-FOV Correction

Movement of a LOR leads to uncovered bins at the fringes, even if a LOR discretization correction was performed. This happens because the acceptance range for LORs is restricted by the maximum accepted ring difference[1] and the maximum accepted radial distance of a PET scanner. Therefore it happens that LORs fall out of this Field-of-View (FOV) during spatial transformation and are *lost counts*. On the other hand regions, which in the measured position are outside of the FOV, are moved into the FOV of the scanner after correction and cause the sinogram to carry empty bins (*missing counts*). Whereas the lost counts do not affect the reconstructed image, the missing counts in the second case lead to an underestimation of the activity in the affected regions, as shown in the sum projections in figure 2.6.

In order to recognize if a transformation causes a LOR to result in a missing count, this LOR has to be transformed with the inverse of the transformation and tested whether it is still within the FOV of the scanner or if it is a missing count. As each new transformation causes different LORs to create missing counts, the underestimation within a sinogram bin (b) can be compensated by accumulating the durations $dt(n,l)$ of all affected transformations (n) to a factor

$$t_{outFOV_b} = \sum_{n=t_{fs}}^{t_{fe}} dt(n,l) \tag{2.14}$$

for each LOR (l), where ($t_{fs}...t_{fe}$) are the boundaries of a specific time interval (i.e. *frame*) in a

[1]which is e.g. 22 for a ECAT EXACT HR$^+$ scanner.

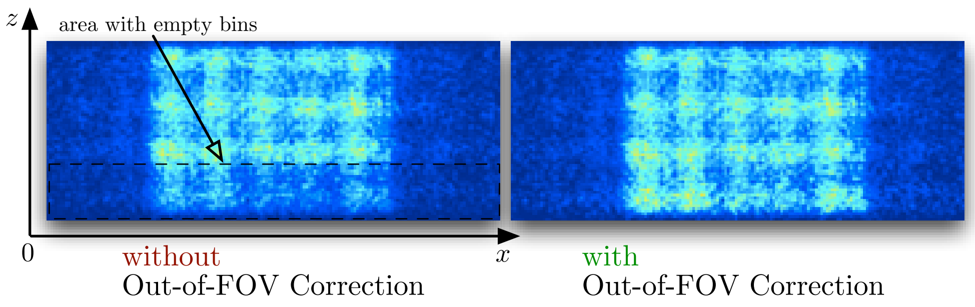


Figure 2.6: Sum projections showing the impact of the Out-of-FOV correction. The left projection shows the underestimation caused by empty bins due to the spatial transformation of the LORs. In contrast, the right projection shows the same acquisition data, but after having processed the Out-of-FOV correction during the movement correction.

dynamic PET study. A frame-based Out-of-FOV correction factor f_{outFOV_b} is calculated under the assumption that the count rate is constant during this frame[2]:

$$f_{outFOV_b} = \left(\frac{t_{frame}}{t_{frame} - t_{outFOV_b}} \right) \quad \text{with} \quad t_{frame} = t_{fe} - t_{fs} \tag{2.15}$$

This factor is then used to scale the count rate of the sinogram bin:

$$N_{corr_b} = N_{norm_b} \cdot f_{outFOV_b} \tag{2.16}$$

where N_{corr_b} is the corrected count rate within bin (b) and N_{norm_b} the count rate after the normalization correction (see 2.2.1.1).

2.3 Transformation Discretization

The ARTtrack^{TM}motion tracking system used in this thesis (cf. section 2.4) provides motion information with sampling rates of up to $60Hz$. This requires to distinguish between significant movements that have to be spatially corrected and movements that can be neglected. The acceptable threshold depends on the field of application of such a tracking system and in case of the usage in PET it is moreover dependent on the examined area and required resolution of the performed study. One must therefore be able to dynamically define, depending on the PET examination, which patient movement is to be significant enough to have to be correction. Figure 2.7 illustrates such a selection of relevant movements. It shows a short sequence of head tracking data with vertical lines confining areas of a persistent transformation is shown.

This selection is determined by considering the surface of a sphere, which is centered at the origin of the PET system. For each motion information this sphere is then spatially transformed like discussed in section 2.1.2 and a factor d_{max}, which is the maximum displacement of a point on the surface of the sphere, is computed with iterative mathematical methods like the *Downhill*

[2]which is not true, but is in many practical cases a valid approximation.[Büh03]

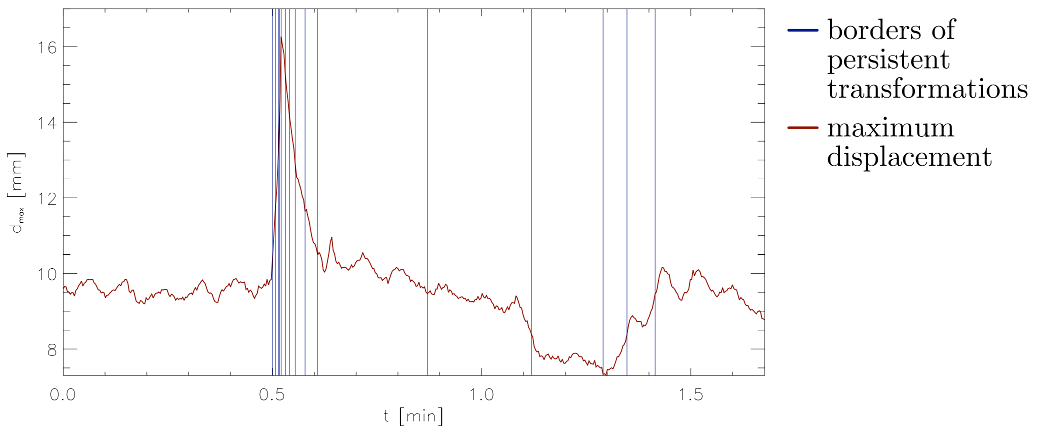


Figure 2.7: Plot showing the maximum displacements d_{max} of a sphere that is computed for each transformation information received from the tracking system. It shows that, as soon as $\triangle d_{max} > 1.0\ mm$ a new persistent transformation is considered, thus allowing to only use those transformation which cause a significant movement.

Simplex Method [PTVF92]. A maximum threshold for the changes of d_{max} is used to decide whether a transformation is relevant enough or if it can be neglected.

2.4 ARTtrack^{TM}Motion Tracking System

During the studies of this thesis, a commercial motion tracking system has been used to measure the movement of a patient. The system consists of two cameras equipped with CCD imaging sensors.They are mounted behind the PET scanner, and are pointing into the scanner tube, cf. figure 2.8.

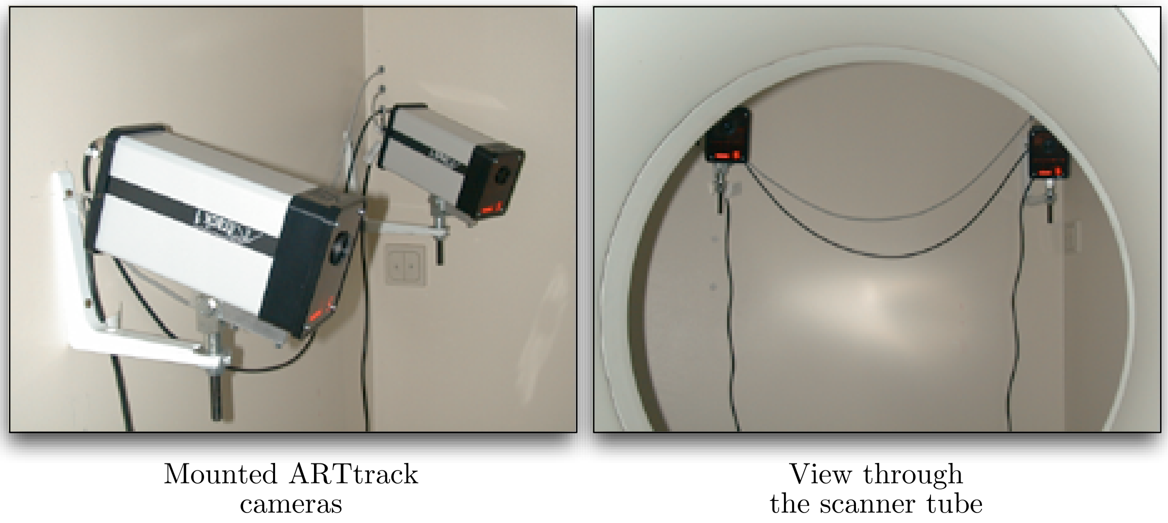
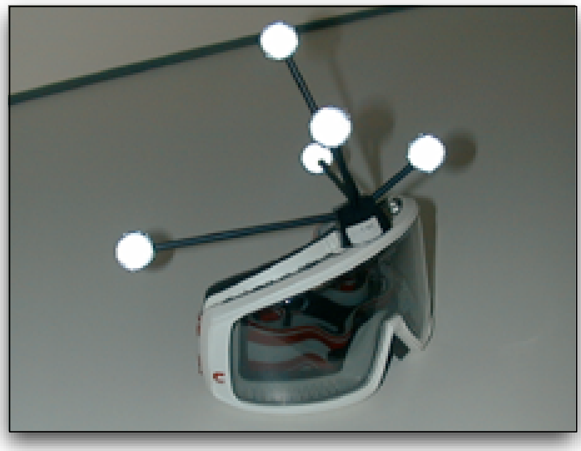


Figure 2.8: The ARTtrack^{TM}motion tracking system installed at the PET center Rossendorf consists of two infrared cameras. They are located behind the scanner tube and are mounted behind the PET scanner, pointing into the scanner tube.

Infrared light flashes integrated in each camera illuminate the tracking area periodically. To be able to recognize a movement within the cameras field of view, infrared light reflecting objects (i.e. *markers*) have to be used. Those objects are passively reflecting the infrared light back to the cameras where an embedded Linux system analyzes the produced images and deduces the 2D-centers of the markers in the camera specific image plane. To prevent interference of simultaneous emitted flashes, both systems are controlled by an external synchronization source. The tracking information is transmitted via Ethernet to a central computing unit where, due to the stereoscopic setup of the cameras, the full 3D positions of the markers is computed. However, to be able to provide the complete 3D movement information, the tracking system has to be used with a *body* as the tracking object.

Such a tracking body, as shown in figure 2.9, consists of a fixed setup of several spherical markers where all distances between the markers are different so that the tracking system can identify every single marker. The figure shows a special body that has been developed for PET brain investigations, where five markers are mounted to a glasses frame and are fixed to the patients head. By tracking all markers of the body, the motion tracking system computes the location of the center of gravity (CG) of the body and the rotation and translation components relative to a calibration reference position. This information is then provided in binary or text-based format (cf. section 3.3.2.3) within an application running on the controlling computer system, and can also be sent to other systems via UDP network datagrams.


6D-body with 5 markers

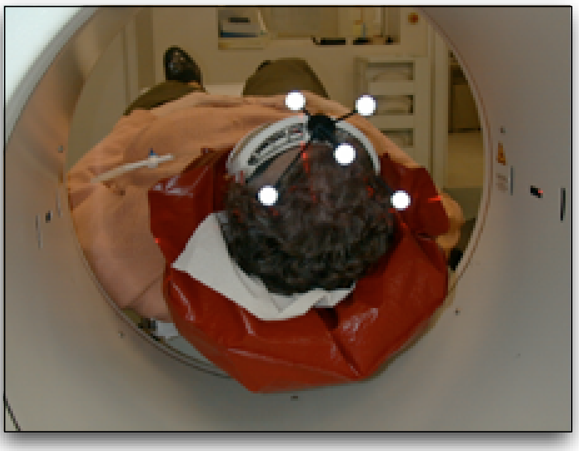
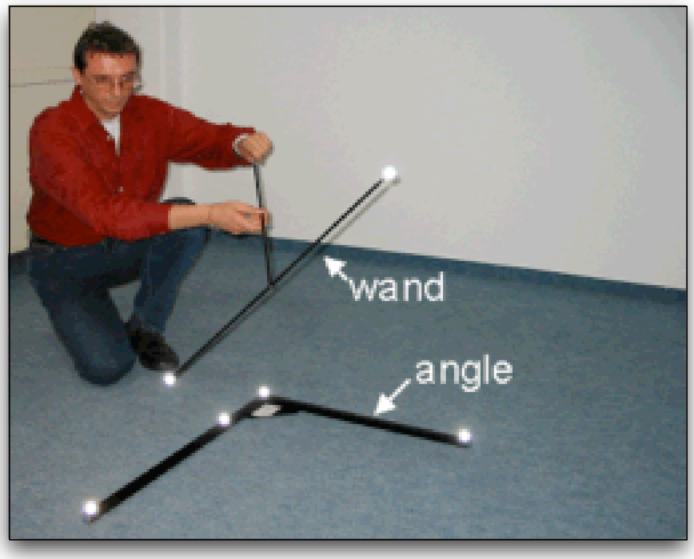

Body attached to patient's head

Figure 2.9: During a PET head examination, a special 6D-body is attached to the head of a patient. With the 5 spherical markers available at the body object, the motion tracking system is able to provide spatial movement information during an examination.

The accuracy of the tracking system depends not only on the setup conditions like the position of cameras or the tracked volume, but also on the necessary calibration of the system: The coordinate calibration (*Room Calibration*) of the system and the calibration of the bodies (*Body Calibration*). First, the coordinate system of the cameras themselves has to be calibrated. This is done through the room calibration where a fixed and unique body (*calibration angle*) has to be positioned within the field of view of both cameras and another unique body (called *wand*) has to be moved during an initial calibration (cf. figure 2.10).

Due to this setup, the motion tracking system can calibrate its coordinate system depending on the measured differences during this process. In addition, each tracking body has to be calibrated where the body itself has to contain markers with different distances to each other so that the tracking system can compute the distances of each marker to another one and use this combination of markers as a single unique body. With a standard setup and calibration of the ARTrack system, the achievable spatial resolution is below 1 mm and the maximum sampling rate can be tuned up to $60Hz$.



Special bodies for
calibration of ARTtrack.

Figure 2.10: Special type of bodies are using during the *room calibration* of the ARTtrack^{TM}system. While one unique body (*angle*) has to be positioned within the FOV of the cameras, another body (*wand*) will be moved during the calibration. Due to the measured differences in this setup, the motion tracking system is able to calibrate its coordinate system.

Chapter 3

Implementation Aspects

After the discussion on the fundamentals of PET and coincidence position correction, the following chapter will concentrate on several aspects relevant for an implementation of those fundamentals. By discussing the existing scientific implementations, outlining user and developer requirements and by analyzing external interfaces that are common in PET, this chapter clarifies the different steps that were taken for a development of such a movement correction application.

3.1 Existing Solutions

The previously discussed coincidence position correction algorithm was initially implemented with several small programs and script based tools. The main correction algorithms were implemented by an experimental application that was written in unoptimized C source code (called `trans_lm`).

Study and analysis of this implementation showed that the correction algorithms produce the expected results [Büh03], but revealed that there are computer science related issues that have to be solved before this new technique is used for routine application:

- the given implementation of the correction algorithm yields computation times that exceed the overall duration of a PET examination by a factor of ≈10, which is not tolerable,
- amount and complexity of the different software tools and applications involved limit the user group to scientists only,
- the current implementation is limited to static PET studies, but is required to support dynamic (multi-frame) studies, which are common with routine PET examinations,
- the whole process of movement correction involves several partial steps by using different software tools, which makes its application error prone,
- the cross-calibration of the tracking system and PET scanner has to be performed within a separate application by manual invocation of several script-based tools, and

- data acquisition with the motion tracking system and preprocessing of this data are done manually by using different software utilities and have to be automated.

3.2 Requirement Analysis

One important part of a software development process is the specification of requirements for the potential groups of users and how their needs can be satisfied. As there are different groups of expected users of a software application, a well-defined implementation has to suit the different needs of each group.

In our case there are three groups of potential users. The main group are the medical technicians who carry out the PET examinations. The second group are scientists (generally physicists) who are interested in improving and administrating the movement correction process. Finally, the third group are generally computer scientists who are interested in reusing or enhancing the implementation.

3.2.1 User Requirements

The primary user group, medical technicians, are familiar with the different steps of performing a PET acquisition. They are trained in using different software solutions to perform the PET examination from data acquisition to image reconstruction. Figure 3.1 summarizes these steps within a schematic drawing.

By introducing a new step into this process, it is important not only to carefully review the requirements of the personnel that is routinely performing those examinations, but also to account for the requirements of the involved scientists. In contrast to the medical technicians, the scientists that are using the application have to be provided with a more advanced and flexible setup that allows the usage of such an application for individual studies.

The following user requirements for a routine-based implementation have to be reviewed:

- loading PET coincidence data from 32bit *listmode* files[1],
- intuitive *Graphical User Interface* (GUI) with the possibility to switch between a novice and expert mode,
- batch-able *Command-Line Interface* (CLI) with all common features that the GUI provides,
- movement correction of dynamic (multi-frame) 3D PET studies,
- direct data acquisition of motion information from the tracking system,
- enhanced import and export functions to load and save all patient and study relevant data,

[1]The native PET acquisition data format, cf. section 3.3.2.1.

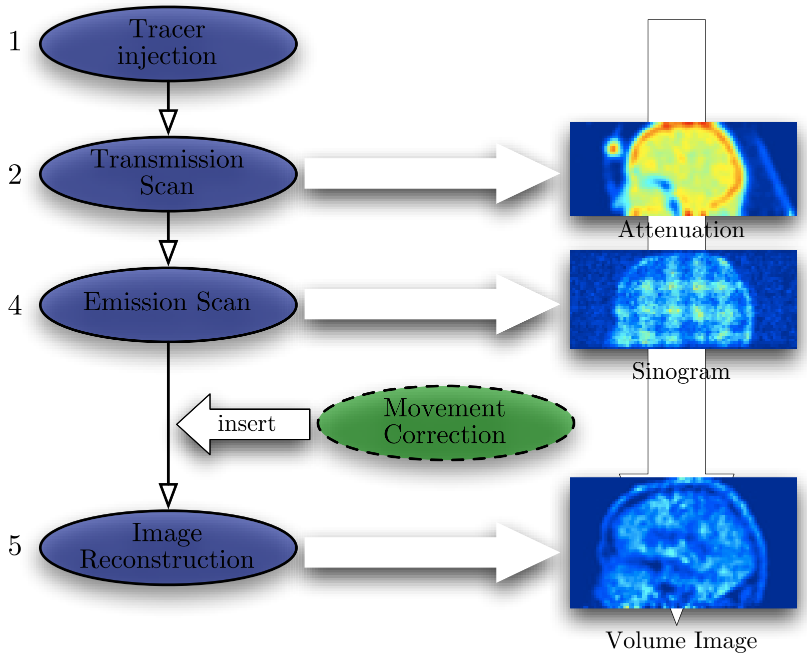


Figure 3.1: Illustration of the four main steps of a PET examination performed by medical assistant technicians. By introducing a new step (*movement correction*) into this process, several user dependent requirements have to be reviewed first.

- allow to save all information of a movement correction study, so that a reevaluation of the data is possible at any time,
- optimized implementation such that a PET examination including a movement correction is possible in routine use, and
- providing final motion corrected data in ECAT7 sinogram files[2].

These issues require to find an adequate programming language, graphical framework and implementation structure that allows to develop an application that is usable by an ordinary user, but also flexible enough to enable a future development by other developers.

[2]The native data format of a Exact HR$^+$ tomograph which is used prior to the image reconstruction, cf. section 3.3.2.2.

3.2.2 Developer Requirements

In contrast to former software development, developers today have different expectations on modern software implementations. Whereas several years ago a software implementation was generally only focused on a particular case, today's software development more and more tends to be highly flexible in design. Properties like *reusability* and *portability* are important for modern software development as is support for well-defined standards like *XML*. This assures that in future whole or parts of a software implementation can be used for, or easily be adapted to new technologies.

The developer requirements for an implementation of the movement correction can be summarized as followed:

- a standardized high-level programming language has to be used,
- *Object Oriented Programming* (OOP) paradigms like *data encapsulation* and an *abstract data types* based implementation to ensure reusability have to be applied,
- a *platform independent* implementation to ensure future migration to other operating systems and potential user groups has to be achieved,
- a *multithreading*, *POSIX-threads* (pthreads) compatible implementation to ensure optimization on the utilized multiprocessor machines[3] has to be assured,
- an *Application Programming Interface* (API) based developer documentation to ensure further development has to be created,
- where applicable, standardized third-party libraries have to be used during development, and
- *Distributed Computing* paradigms have to be considered so that an easy adaption is possible in future.

3.3 Specification

The specification of an implementation and the different involved interfaces, is an important step during the software development process. It is necessary to perform a work-flow analysis of the existing solution, taking the requirements into account and to define the boundaries of the new implementation within this main work-flow. The transitions at these boundaries form the external interfaces between different involved systems that have to be supported by an implementation. Such a specification also has to cover internal interfaces like the chosen programming language or the GUI framework so that within the implementation the right tools are used to achieve the desired outcome.

[3]We used a 4x900MHz Sun Solaris$^{\text{TM}}$v480 system during development.

Therefore, a work-flow analysis was performed during the specification process of this thesis. By referring to the existing implementation discussed in section 3.1, the command-line tools and applications involved in that movement correction implementation are summarized in figure 3.2.

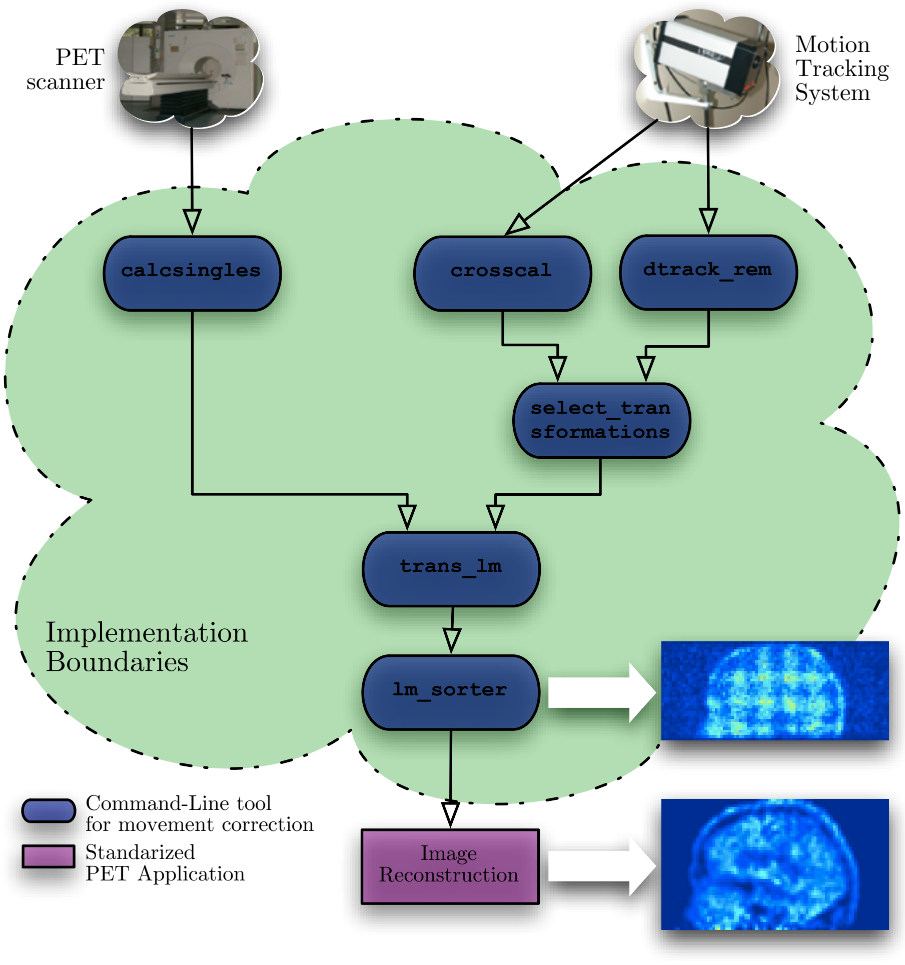


Figure 3.2: The work-flow of the existing movement correction implementation is shown. It consists of the successive execution of several different applications and command-line tools. In addition, the surrounding implementation boundary illustrates the coverage of the new implementation `lmmc`.

In addition to the different tools that are used in the existing solution, the figure shows the coverage of the new implementation with the involved transitions to other external systems, thus specifying the required *external interfaces*. According to that, the transitions visible within the implementation boundaries specify the *internal interfaces* that have to be supported.

3.3.1 Internal Interfaces

By referring to the different user and developer requirements of section 3.2, there exist different demands on the internal interfaces for our movement correction implementation. The following subsections will discuss the fundamental parts of this implementation and outlines the accounted internal interfaces.

3.3.1.1 Programming Language

The correct choice of a programming language for a specific implementation depends on the field of application, the operating systems involved, and on the particular requirements of a software developer.The best programming language to choose for the movement correction was found by evaluating the following criteria:

- *portability*,
- *reusability*,
- *performance*,
- *object-orientation*, and
- *multithreading* extension.

Table 3.1 Based on personal experience and tests on the target platform, table 3.1 shows potential languages that are candidates for an implementation.

Name	**Portability**	**Reusability**	**Per-formance**	**Object-Orientation**	**Multi-threading**
JAVA	very good	JAVA only	moderate	yes	proprietary
C	good	good	good	limited	through *pthread* library
C^{++}	very good	very good	good	yes	through *pthread* library
$C^{\#}$	poor	very good	moderate	yes	proprietary
Assembler	n/a	same platform	very good	no	no

Table 3.1: Comparison of common Programming Languages

Only the fully object-oriented JAVA, C^{++} and $C^{\#}$ were serious candidates. Since the programming languages $C^{\#}$ and JAVA only provide own proprietary multithreading frameworks and

C$^{\#}$ is not yet directly supported by the Solaris^{TM}operating system, these two languages were also excluded. Therefore, we choose C^{++} as the programming language for the final implementation.

3.3.1.2 GUI Framework

In contrast to the developer driven choice of the programming language, the choice of the graphical user interface framework depends on the requirements of the particular users but also on the available operating systems. Therefore, the criteria for a suitable GUI framework are:

- *portability*,
- *graphical component variety*,
- *multithreading* capabilities,
- *performance*,
- *object-oriented* integration.

Table 3.2 lists GUI frameworks that can be used with the C^{++} programming language. It is based on performed tests and personal experiences.

Name	**Portability**	**Component Variety**	**Multi-threading**	**Per-formance**	**Object-Oriented integration**
GTK+	good	good	good	very good	limited
Qt	very good	very good	very good	good	very good
KDE	good	very good	very good	good	very good
Motif++	poor	poor	poor	good	poor

Table 3.2: Comparison of GUI frameworks

Another relevant aspect for the choice of the GUI framework is the existence of other software development projects at the PET center. Beside the fact that Qt fulfills most of the required criteria, it is also the preferred GUI framework for other software projects carried out at the PET center in Rossendorf. We therefore chose Qt as the graphical user interface framework for an optimized implementation of the movement correction.

3.3.1.3 Multithreading Framework

An important part of of this thesis is the application of parallel computing techniques in the development process. This requires to analyze the multithreading capabilities of the involved operating systems as well as to search for an appropriate multithreading framework.

Since most computer systems at the PET center Rossendorf are running the Solaris operating system, we choose this platform as the primary platform for the implementation. As this operating system natively supports multithreading through the *Symmetric Multiprocessing* (SMP) architecture, a multithreading framework was used to separate computational independent parts and distribute them on different processors. Available multithreading frameworks that can be used with Solaris^{TM}are either the operating system's own framework (Solaris^{TM}Threads) or the available POSIX Threads (pthread) library.

Solaris threads and pthreads are very similar in both API action and syntax. In contrast to Solaris threads, the pthread framework is based on the POSIX standard and therefore allows to port an application to another platform where an implementation of pthreads exists. Additionally, the Qt framework provides low-level classes such as multithreading classes that are directly based upon the POSIX thread framework, acting as a wrapper and providing the same pthreads functionality, but within an object-oriented environment.

The features of the pthreads framework and the cooperation between pthreads and Qt therefore supports the choice of Qt framework. Moreover, it allows to develop multithreading enabled graphical applications which are then also portable to other operating systems. In addition, there exist Qt implementations for all major Unix systems such as for Solaris^{TM}and Linux etc., as well as for MacOSX^{TM}and Microsoft Windows^{TM}.

3.3.2 External Interfaces

As the application will be used to preprocess the acquisition data of a PET tomograph prior to image reconstruction, several interfaces have to be defined. The application has to support the native file formats like the *listmode* and the *sinogram* file format of the PET scanner, and must be able to read the movement information from the motion tracking system. Therefore, the main external interfaces are discussed in the following subsections, which concentrate on the raw formats and features that have to be supported.

3.3.2.1 PET Listmode Format

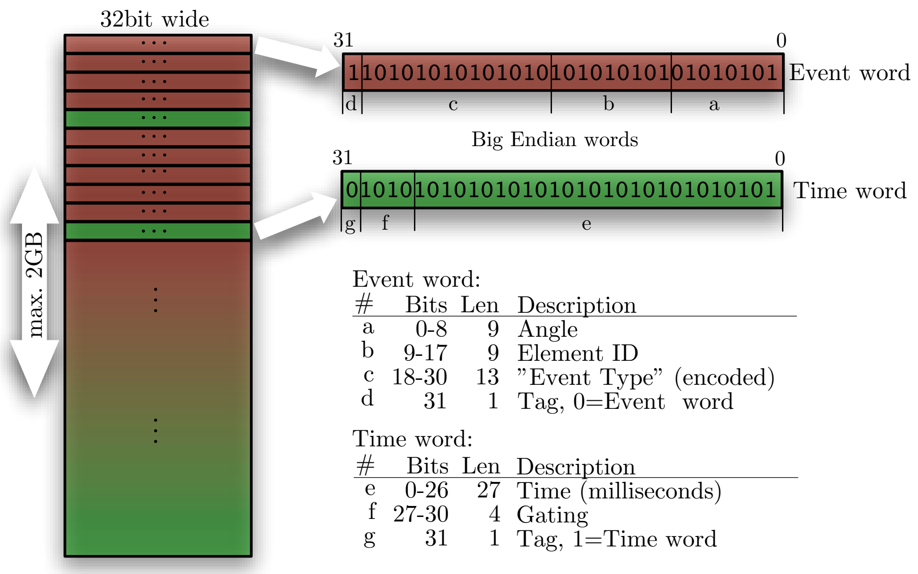
A *listmode* file consist of 32bit big-endian words. No header exists, and the maximum file size is only limited by the file system's capabilities[4]. Two different types of listmode words exist, *time words* and *event words*. They are distinguished by a tag, the most significant bit 31.

Time words are inserted into the listmode stream every millisecond. A time word contains the time in milliseconds since the start of the listmode acquisition within its first 27 bits[5]. In addition, bits 27-30 are reserved for *gating* to signal a special event[6]. The time at which a

[4]the system controlling a EXACT HR^{+} scanner limits the file size to a maximum of 2GB and creates successive files to cover the whole data set.

[5]which limits a listmode acquisition duration to $\approx$ 37 hours.

[6]e.g. during a heart examination with a connected ECG the start of a new heart beat cycle.



Event word:

#	Bits	Len	Description
a	0-8	9	Angle
b	9-17	9	Element ID
c	18-30	13	"Event Type" (encoded)
d	31	1	Tag, 0=Event word

Time word:

#	Bits	Len	Description
e	0-26	27	Time (milliseconds)
f	27-30	4	Gating
g	31	1	Tag, 1=Time word

Figure 3.3: A PET scanner can be configured to save the acquisition data in *listmode* format. This format consists of 32bit long big endian words where two different types of words (*time word* and *event word*) exist. This figure shows the bit definitions for each word type [Nic98].

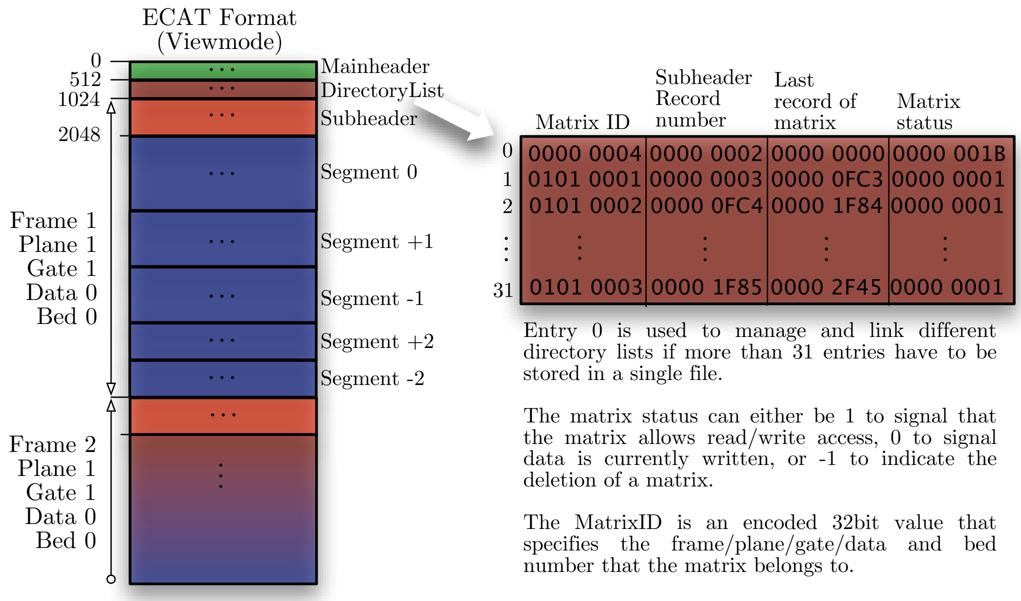
specific coincidence is registered is determined by the most recent time word in the file. Event words at the beginning of the file have time zero until occurrence of the first time word.

As shown in figure 3.3, event words describe the LOR of a single coincidence - the discrete values of the angle (AN) and radial element (RE) of the LOR and the numbers of the involved detector rings ($RI_{A,B}$). Please note that the information of the ring numbers are encoded separately in a *EventType* bit field, which will not further be discussed here [Nic98].

3.3.2.2 ECAT File Format

By default, the EXACT HR$^+$ sorts the data of an acquisition directly into a *sinogram* and represents the input for image reconstruction process. As the movement correction is based on *listmode* acquisition, support for the native sinogram file format is necessary. The EXACT HR$^+$ outputs the sinogram data in the proprietary *ECAT file format* that is based on a general matrix file format specification for image processing [Col85].

ECAT files are logically divided into 512 byte long blocks. The first block is always the *main-header* that contains general information like patient data, used isotope and other acquisition relevant meta data. In addition, it contains a type identifier such that an ECAT file is able to contain different types of matrix data. The second block is a *directory list* that contains start and end position of a maximum of 31 data blocks that contain the matrix data of the sinogram, along with an additional flag to signal if a particular data block is valid or is still pending to be written. The directory list allows the distinction of each of these blocks by a unique 32bit



	Matrix ID	Subheader Record number	Last record of matrix	Matrix status
0	0000 0004	0000 0002	0000 0000	0000 001B
1	0101 0001	0000 0003	0000 0FC3	0000 0001
2	0101 0002	0000 0FC4	0000 1F84	0000 0001
⋮	⋮	⋮	⋮	⋮
31	0101 0003	0000 1F85	0000 2F45	0000 0001

Entry 0 is used to manage and link different directory lists if more than 31 entries have to be stored in a single file.

The matrix status can either be 1 to signal that the matrix allows read/write access, 0 to signal data is currently written, or -1 to indicate the deletion of a matrix.

The MatrixID is an encoded 32bit value that specifies the frame/plane/gate/data and bed number that the matrix belongs to.

Figure 3.4: Illustration of the ECAT file format with a brief description of the binary format of a single directory list [ECA99].

identifier which consists of the *frame*, *gate*, *bed*, and *plane* the data block belongs to. In addition, it contains another identifier that allows the reference of succeeding and preceding directory lists, and thus provides an endless linked-list functionality.

As shown in figure 3.4, each referred data block in the directory list starts with a *sub-header* that contains type-specific meta information about the succeeding matrix data. More information on the definition of the ECAT file format can be found in [ECA99].

To implement the ECAT sinogram file format output routines, the following possibilities have been evaluated:

- Usage of the ECAT file I/O libraries of Uwe Just [Jus00],
- usage of the ECAT file I/O libraries of Merence Sibomana, or
- development and use of a custom multithreading enabled ECAT file I/O library implementation.

Even if the usage of an existing library should be preferred in most cases, the evaluation of the available ECAT file format implementations showed that they are all written in a non object-oriented programming language, and do not support the integration within a multithreading environment. As this thesis discusses the parallel computing optimization of the movement correction, it was necessary to implement multithreading capable file I/O routines for writing ECAT sinogram files. These routines were integrated in an own ECAT library implementation and are available for use in other ECAT supporting applications at the PET center.

3.3.2.3 Motion Tracking System

The patient movement information is directly provided by the computer system that controls the motion tracking system. It sends out UDP datagrams through a network interface periodically which are either binary or ASCII encoded. These datagrams carry the movement relevant position information of a tracked body relative to a reference position and can be send out to a maximum number of 4 different computer systems simultaneously, cf. [ART02]. As the binary encoded format has become obsolete with the latest version of the motion tracking software, this section concentrates on discussing the ASCII based format specification only.

Identifier	Description	Example
`fr <int>`	continuous frame number	`fr 47`
`ts <float>`	optional continuous time stamp since 0:00am with an accuracy $\triangle t_{err} \approx \pm 10ms$	`ts 39596.024`
`3d <int> ...`	all 3D-marker position information	`3d 1 [0 1.000][210.730 -90.669 -108.554]`
`6d <int> ...`	all 6D-body position/rotation information	`6d 1 [0 1.000][326.848 -187.216 109.503 -160.4704 -3.6963 -7.0913][-0.9452 -0.3392 -0.0190 0.3335 -0.9325 0.1377 -0.0644 0.1231 0.990286]`

Table 3.3: Prior to the movement correction computation, the motion tracking information is provided by ASCII encoded string in send out UDP datagrams. Although the motion tracking system supports more data strings, the table lists only those strings that are required for the movement correction processing.

A single ASCII datagram consists of multiple lines which are separated through a CRLF (hex: `0D 0A`) line break. Depending on the *sampling rate* and a *data divisor* the tracking system sends a datagram for each measurement (i.e. *frame*). Each line starts with an identifier that specifies the type of the remaining data in this line. Although the motion tracking system can provide more data and identifiers, we only concentrate on the ones listed in table 3.3.

Each datagram starts with a `"fr <int>"` line where `<int>` refers to a continuous identification number of the measured frame. An optional `"ts <float>"` line provides the timing information, when the motion information has been tracked by the system. The rest of the datagram contains any number of lines with tracking information of *markers* or *bodies* (cf. section 2.4) which have the following format:

- **3D-Markers:** `"3d` n `[1` q_1`][`P_{1_x} P_{1_y} P_{1_z}`]...[`n q_n`][`P_{n_x} P_{n_y} P_{n_z}`]"`
  A line starting with `"3d"` contains n marker locations (counting from one), where each marker is identified by a `"[`n q_n`][`P_{n_x} P_{n_y} P_{n_z}`]"` section. A provided floating point quality factor q_n ranges from zero to one and describes the tracking stability at the time

the position of $\vec{P}_n$ has been tracked [7].

- **6D-Bodies:** `"6d` n `[0` q_0`][`P_{0_x} P_{0_y} P_{0_z} η_0 θ_0 ϕ_0`][`$T_{0_{1,1}}$ $T_{0_{2,1}}$ $T_{0_{3,1}}$ $T_{0_{1,2}}$ $T_{0_{2,2}}$ $T_{0_{3,2}}$ $T_{0_{1,3}}$ $T_{0_{2,3}}$ $T_{0_{3,3}}$`].."`
  A line starting with `"6d"` contains positions ($\vec{P}_n$) and orientation (η_n, θ_n, ϕ_n) of n different bodies (counting from zero) within the FOV of the tracking system. The center of gravity of each body is specified by a vector $\vec{P}_n$. The orientation is specified by three angles η, θ and ϕ (in degrees) that can be used to calculate the rotation matrix $\hat{R}$ with three successive rotations $R_i(\alpha)$ ($x \rightarrow y \rightarrow z$) around an axis i with angle α:

$$\hat{R} = R_x(\eta) \cdot R_y(\theta) \cdot R_z(\phi) \tag{3.1}$$

  The final rotation matrix $\hat{R}$ can be expressed as:

$$\hat{R} = \begin{pmatrix} \cos\phi\cos\theta & -\sin\phi\cos\theta & \sin\theta \\ \sin\phi\cos\eta + \cos\phi\sin\theta\sin\eta & \cos\phi\cos\eta - \sin\phi\sin\theta\sin\phi & -\cos\theta\sin\eta \\ \sin\phi\sin\eta - \cos\phi\sin\theta\cos\eta & \cos\phi\sin\eta + \sin\phi\sin\theta\cos\phi & \cos\theta\cos\phi \end{pmatrix} \tag{3.2}$$

  For convenience the rotation matrix (3.2) is directly provided in the separate `"[`$T_{0_{1,1}} \ldots T_{0_{3,3}}$`]"` section of the ASCII line, and the indexes follow the same layout as shown in equation (2.3).

Obviously, the routines that receive the UDP datagrams from the motion tracking system have to provide string parsing routines to extract the required information.

Command string	**Action**
`dtrack 10 0`	Stop a currently running measurement.
`dtrack 10 1`	Start the ARTtrack cameras (no marker/body measurement).
`dtrack 10 3`	Start the ARTtrack cameras and marker/body measurement.
`dtrack 31`	Start of an continuous performance data transfer.
`dtrack 32`	Pause a performance data transfer.
`dtrack 33` n	Request n performance datagrams.

Table 3.4: DTrack remote-control command strings

In addition to the UDP data transmission, the motion tracking system provides a *remote control* facility, where an UDP based listener at the tracking system waits for `NUL`-terminated ASCII command strings. Table 3.4 lists the possible commands which have also to be supported by the movement correction application.

In summary, the implementation has to provide methods to receive and send UDP datagrams from and to a specific IP address and port. As the Qt framework (cf. section 3.3.1.2) provides a way to keep the source code fully portable over multiple platforms, it also ships with several socket classes which have been chosen for the purpose of communication with the motion tracking system.

[7] $\vec{P}_n$ is also the center of gravity (CG) of each marker and body.

3.4 Implementation Prospect

After having discussed the requirements and the internal and external interfaces, that are going to be supported by the introduced application, the following summarized prospects can be listed:

- Due to computational optimizations, such as parallel computing optimizations through a multithreaded implementation, the movement correction performance is expected to be significantly improved such that a routinely usage will be possible.
- An intuitive graphical user interface (GUI) will allow the use of the movement correction by the technicians using the tomograph for routine examinations.
- The implementation of network I/O functionality will allow the direct communication and control of the tracking system through the application.
- An expert mode within the GUI and the additional command-line interface will allow scientists to use the application as an administration and scientific analysis tool.
- An object-oriented design and implementation as well as the portable development using Qt/C^{++} will assure the future use and compatibility to other operating systems.
- The development of custom multithreading optimized file I/O routines of ECAT compatible files will allow the use of standard image reconstruction software.

Chapter 4

Parallel Computing Analysis

Parallelization is a common optimization technique in computer science. However, in order to benefit from this optimization, analysis on the involved algorithms have to be performed to uncover independent areas that can be computed in parallel.

4.1 Fundamentals

The goal of parallel computing is to speed up execution of a program by dividing computations into independent fractions (which are implemented by a *thread*[1]) and to distribute them on multiple processor units. The ideal assumption, that a multiple processor machine with N processors speeds up a parallelized program N times, does hold in practice, since the overhead of the underlying operating system limits this factor. In addition, the maximum achievable speedup depends on the size of the remaining fraction that, due to data dependencies, can only be executed on one processor.

Assuming that the size of a sequential fraction f is known, the expectable speedup $S(N)$ is defined by *Amdahl's Law* [Amd67],

$$S(N) = \frac{1}{f + \frac{1-f}{N}} \quad \text{with} \quad f > 0 \tag{4.1}$$

which has a limiting value of $1/f$ for an infinite number of processors. This law illustrates that the amount of expectable speedup depends on the sequential fraction f, where in practice a value of $f = 0$ is unachievable. This leads to the conclusion, that even on a multiprocessor machine with unlimited number of processors, a sequential fraction of 10% limits the total achievable speedup to $S(\infty) \approx 10$.

Another more general estimation for the expectable speedup is, that the speedup does not scale with the number of processors, but with $N/\log_2 N$. However, analysis and tests have shown that an application can be practically speed up to a maximum of $S(4) \approx 1.9$ on a four processor system [Lan02].

[1] A thread is defined as an independent fraction of code that runs and shares memory with others in parallel.

Therefore, it is important to reduce the amount of sequential computations. In addition, if some threads have to share the same data, special techniques have to be used to synchronize the access to this data. The used Qt/POSIX multithreading functionality generally provides the following different methods to synchronize such an access:

A. **Mutual-Exclusives (*Mutex*)**

A *mutex* is generally used to protect an object, data structure or section of code so that only one thread can access it at a time[2]. For example, say there are two methods which modify the value of the same `int` variable `number`. If these two methods are called simultaneously from two threads then the result of this variable is undefined, because both threads could have changed the contents of the variable at the same time. Listing 4.1 illustrates the use of a mutual-exlusive mechanism, with the provided $\boxed{\texttt{QMutex}}$ class in Qt to synchronize the access.

Listing 4.1: Use a QMutex to protect shared data

```
QMutex mutex;
int number = 6;

void Thread1::calc_method()
{
  mutex.lock();    // lock access to "number"
  number *= 5;
  number /= 4;
  mutex.unlock(); // unlock access to "number"
}

void Thread2::calc_method()
{
  mutex.lock();    // lock access to "number"
  number *= 3;
  number /= 2;
  mutex.unlock(); // unlock access to "number"
}
```

Here the calls to methods `mutex.lock()` and `mutex.unlock()` are atomic, i.e. are indivisible operations, that try to lock and unlock the access to the mutex. If a mutex could successfully be locked, the thread which called the `mutex.lock()` method continues with its execution, or otherwise halts until another thread is unlocking it with `mutex.unlock()`. Listing 4.1 illustrates that without the use of such mutex mechanisms, both threads are able to change the value of the shared variable while another thread is currently also altering the value in parallel, thus resulting in an undefined state of `number`.

In addition to protect a shared data area, mutexes are also often used to prevent so called *race conditions*. These are situations where the result of multiple events depends on the execution order and this order cannot be guaranteed. For example, if one thread is

[2]which is also similar to the `synchronized` keyword in Java.

currently in the process of checking if a particular file exists and tries to open it afterwards, another thread could possibly remove that file before it could be opened by the initial thread.

B. **Semaphores**

A semaphore can be used to serialize thread executions, similar to a mutex. However, it differs from a mutex in that a semaphore can be accessed by more than one thread at a time. Although a semaphore can be used instead of a mutex, it is often used as a *resource control facility*.

For example, suppose we have an application that generates a large number of threads operating on some data (a *thread pool*), but wants to limit the amount of currently running threads to the available number of processors. As shown in Listing 4.2 the `QSemaphore` class of the Qt framework is used to instantiate a `semaphore` object for that purpose.

Listing 4.2: Use a QSemaphore to manage thread resources

```
int num_processors = 4;
QSemaphore sempahore(num_processors);

void MyThread::started()
{
  ++semaphore;  // get access if semaphore.available() > 0 or wait.
}

void MyThread::finished()
{
  --semaphore;  // release access to the semaphore and
                // increase semaphore.available() by one.
}
```

This object is used in a separate `started()` and `finished()` method within each thread class, such that a maximum of `num_processors` threads is guaranteed to run at the same time.
During the execution of these methods the access is synchronized by operator-overloaded calls to the semaphore. Each call to the `++semaphore` method tries to obtain access to the semaphore, and thus tries to decrease the number of available slots for a simultaneous execution. When the access slots are exhausted (`semaphore.available()` == 0), another call to the `++semaphore` method suspends the requesting thread until another one is signaling with a call of the `--semaphore` method, that it has finished its processing.

In the present context, several semaphores are used to synchronize the execution of threads as shown in listing 4.2. In section 4.3 we discuss use of these semaphores within *thread pools*.

C. **Wait Conditions**

As an inter-thread communication method, *wait conditions* allow a thread to notify other threads that some condition is met and that those waiting for this condition should ”wake up”.

For example, if we have two tasks that run every time the user presses a key, each of these tasks is assigned to a thread that waits until a third thread signals it to start its computation. With using the Qt framework, such a wait condition is implemented by using a global object of the `QWaitCondition` class, as illustrated in Listing 4.3.

Listing 4.3: Use a QWaitCondition to let threads communicate

```
QWaitCondition key_pressed;

void ComputeThread::run()
{
  while(true)
  {
    key_pressed.wait(); // wait until another thread signals to wake up.

    // Key was pressed, start computations
    ...
  }
}

void KeyboardThread::readKeyboard()
{
  while(true)
  {
    getchar();              // wait until a key on the keyboard has been pressed

    key_pressed.wakeAll();  // Causes any thread in key_pressed.wait() to return
                            // from that method and continue processing
  }
}
```

The listing shows that as soon as the keyboard thread calls the `key_pressed.wakeAll()` method, all threads waiting for this condition wake up immediately and resume their execution.

In addition, a thread signaling other threads to wake up can issue a method that only wakes up a single thread out of the waiting queue. In Qt this is implemented within the `wakeOne()` method of the wait condition class. However, due to the implementation of this wait condition methods, it is nondeterministic which thread will wake up first.

Of course, the use of synchronization methods also has an impact on the overall possible increase in speed, due to the use of multithreading capabilities. This is the reason why a separate dependency analysis of the different involved computations has to be performed, such that the shared data areas are identified and encapsulated properly.

4.2 Dependency Analysis

In terms of the optimization of the movement correction process the performed analysis concentrated on trying to separate the partial position correction steps (cf. section 2.2), such that a large number of steps can be performed in parallel.

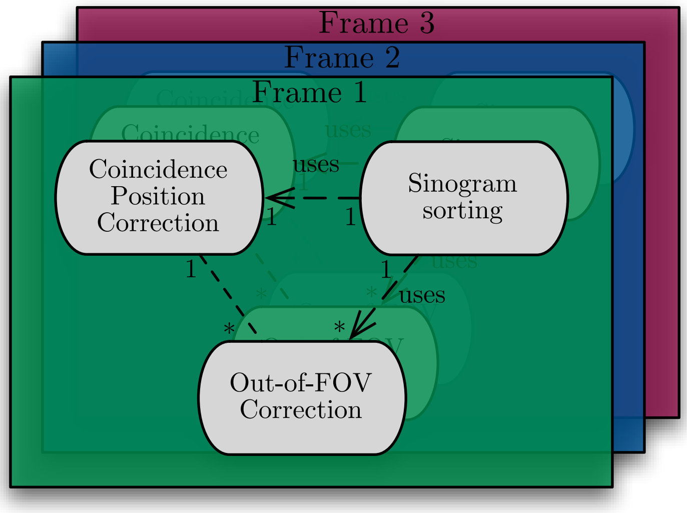


Figure 4.1: Dependency graph of Movement Correction entities

It turned out that only the three frame-based steps, shown in figure 4.1, have a significant potential for a parallel computing optimization. The independence of the intrinsic coincidence position correction algorithms and the Out-of-FOV computations, as well as the fact that each frame can be handled independently, are reasons why the further analysis was split into three different optimizing stages.

4.2.1 Stage 1 - Frame optimized parallelism

In a PET examination, the acquisition data is sorted into different sinograms, depending on either time based (*frame*), position based (*bed*) or trigger based (*gate*) criteria. However, for brain examinations only the frame based separation has been reviewed. The acquisition data of each of the frames are independent from other frames. Therefore, this natural separation was adopted and each frame has been encapsulated within a separate thread.

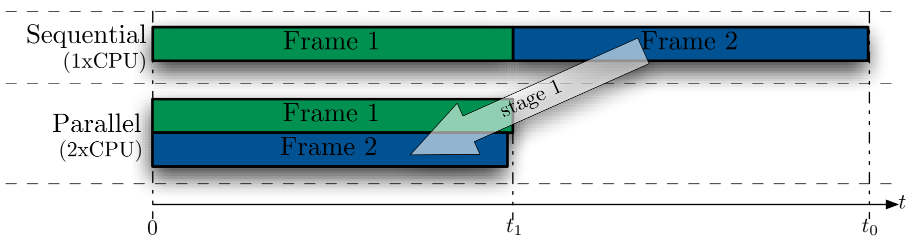


Figure 4.2: Stage 1 - Frame based dependency splitting

Figure 4.2 illustrates that in contrast to sequential execution, the synchronous computation of the frames decreases the maximum required execution time ($t_0 > t_1$). However, the minimum possible execution such a *frame-based* parallelism can provide, is limited by the number of processors available and the duration of the longest frame computation.

4.2.2 Stage 2 - Out-of-FOV optimized parallelism

The movement correction computations that are performed for a single frame imply computations of several Out-of-FOV corrections (OFC). While the number of required OFC corrections within a frame is proportional to the number of available movement informations (transformations) for this frame, the complexity and therefore the theoretical execution duration of each OFC is equal (c.f. section 2.2.1.3). However, an OFC is independent of the intrinsic coincidence position correction, because of the *data independence* of both types of correction.

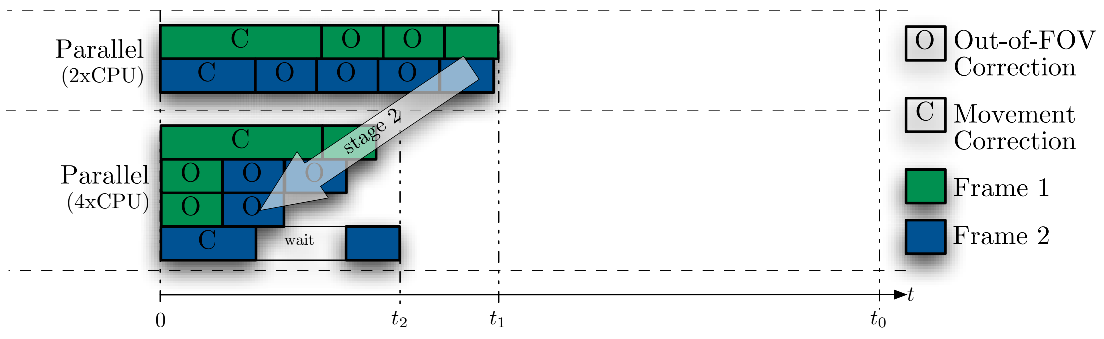


Figure 4.3: Stage 2 - Out-of-FOV based dependency splitting

Therefore each OFC was encapsulated within a thread, sharing and synchronizing data only between other OFCs operating on the same frame. Figure 4.3 illustrates how the overall execution time is decreased by executing each OFC in parallel. The example shows that due to the additional splitting of the movement correction process into several independent OFC corrections the total execution time has been further reduced to t_2 ($t_0 > t_1 > t_2$).

However, due to the processing dependencies (cf. figure 4.1) of the movement correction process, this figure illustrates also that for a frame in which more OFC computations than movement correction computations have to be performed, a *waiting phase* is included during which the end of the OFC computations has to be waited for. This ”waiting phase” limits the maximum possible reduction of the execution time, because no other computations can be performed during that time until the corrected data is finalized and sorted into the sinogram.

4.2.3 Stage 3 - Sinogram sorting optimized parallelism

The movement correction process of a frame is finalized by sorting the corrected coincidence data into a sinogram file. Due to the data dependency between the OFC correction and the sinogram sorting, the stage 2 optimization (cf. section 4.2.2) showed that if this sorting is done within the same computation thread as the coincidence correction, it happens that this thread has to wait until all OFC are finished and therefore puts one processor unit to sleep.

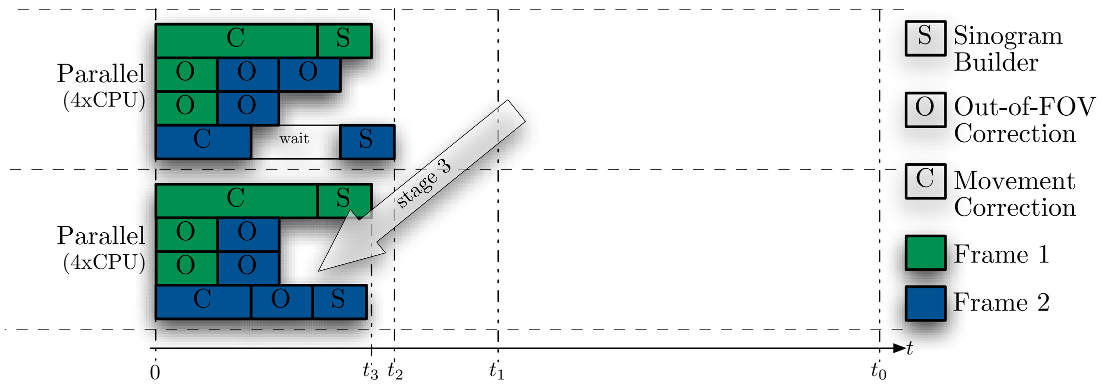


Figure 4.4: Stage 3 - Sinogram sorting based dependency splitting

This situation is avoided by running the final sinogram sorting in a separate thread, thus allowing the processor to be used by other threads during the waiting phase. The example shown in figure 4.4 illustrates that this technique can lead to a further reduction of the overall execution time ($t_0 > t_1 > t_2 > t_3$).

In addition, if more frames have to be corrected, this technique allows to start the computations of the successive frame earlier, thus having the results of each frame computation available earlier.

4.3 Hierarchical Thread Modeling

When a process is split into several threads, the proper processing and relational computation of these fractions have to be assured. This requires to have a central management unit, a *thread dispatcher*, that "knows" all threads within the running application. In addition, the modeling of the threads within a hierarchical system is often preferred, such that a thread manages the execution of potential sub-threads on its own. This assures an additional security and data encapsulation within the object-oriented context.

Figure 4.5 illustrates how the different threads discussed in section 4.2 are managed within a hierarchical system. It includes a thread dispatcher which acts as the central communication unit between the sub-threads of each frame and the user interface.

As the number of required threads is generally greater than the number of available processors, the thread model has to provide a *processing queue*. This avoids putting too much load on the operating system by processing more simultaneously running threads than there are available processor units. Therefore, each thread containing sub-threads within the hierarchical thread model provides a *thread pool* that manages the execution and distribution of its sub-threads. The number of simultaneous processing movement corrections is managed by a main

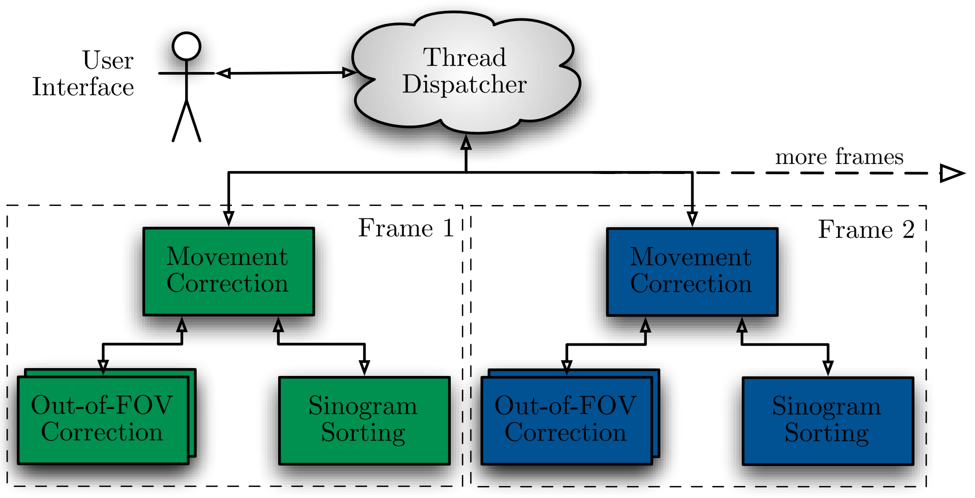


Figure 4.5: Hierarchical thread modeling including a Thread Dispatcher

thread pool within the thread dispatcher, while the number of parallel running OFC corrections is handled by a second thread pool within each movement correction thread.

This hierarchical management, in which only the parent entity ”knows” about its sub-threads, allows not only to manage these entities within separate pools, it also assures that due to the invisibility of a sub-thread to another one, the security of the execution of such a thread is increased.

4.4 User Interface Communication

Having multiple threads within an application requires a proper communication between the threads and the provided user interface. In case of a simple command-line application, the communication is generally limited to signaling progress or error events to the terminal which is perfectly possible from each thread via standard I/O methods like `cout` or `cin` in C^{++}

In contrast, in an application with a graphical user interface or advanced event handling, the communication between threads and the user interface dependents on the implementation of the underlying event handling system of the employed GUI framework.

In the Qt framework, graphical objects are generally communicating with a *signal-slot* mechanism, where an object can connect a predefined signal to a slot of another object such that as soon as the signal is *emitted*, the corresponding slot method is called.

Listing 4.4: Use the `connect()` method to use the signal-slot mechanism of Qt

```
QPushButton startButton;

void MyApplication::init()
{
  connect(&startButton, SIGNAL(clicked()),  // connect signal clicked() of startButton
          this, SLOT(processClicked());       // to the processClicked() method of
                                              // our application
}

void MyApplication::processClicked()
{
  // will be called as soon as startButton is pressed
  ...
}
```

For example, Qt allows to connect a `clicked()` signal of a `QPushButton` object to an own method via a special `connect()` method, as shown in listing 4.4. This method is then called as soon as the connected button is pressed.

As the entire GUI communication in Qt is running in the same thread as the application, and the signal-slot mechanism requires the connected objects to run within the same thread, a direct interaction between a sub-thread and the GUI is not possible through the signal-slot connection. However, like other modern GUI frameworks, Qt provides an advanced event handler that allows to send custom event objects via an application wide method `postEvent()` to a custom event method `customEvent()` of any object inherited from the base class `QObject`.

Listing 4.5: Use the Qt event system via an own `customEvent()` method

```
void MyThread::signalProgressToGUI()
{
  // To send a progress information to the GUI, we forward
  // the event to the thread dispatcher, which can then use the signal-slot mechanism
  QApplication::postEvent(dispatcher, new QCustomEvent(65432));
}

// To receive an event of this custom event type:
void ThreadDispatcher::customEvent(QCustomEvent* e)
{
  if(e->type() == 65432)
  {
    // it must be the progress event from MyThread
    // so emit the signal "progress" to the GUI elements
    // connected to this signal
    emit progress(65432);
  }
}
```

In our application, this event system is used to implement the communication between the computation threads and the GUI. The *thread dispatcher* of section 4.3 acts as the communication unit between the GUI and the threads, forwarding each information coming from the threads via signals to each GUI elements. Listing 4.5 illustrates this, by showing an example of how a thread `MyThread` posts an event that will be received by the `customEvent()` method of the thread dispatcher, that will then emit a specific signal that another GUI element can receive.

This mechanism is used to signal special events like progress, warning, and error events and allows the GUI elements to react upon the reception of these events, e.g. forwarding the error messages to GUI requesters that allow to abort a movement correction execution.

Chapter 5

Implementation

This chapter discusses the object-oriented implementation of the head movement correction application. Where applicable, UML diagrams illustrate the implemented functionality and the hierarchies between the different classes [Gro03, Oes98].

5.1 Mathematical Interfaces

During the movement correction process, methods perform basic mathematical tasks such as spatial transformations or other computations on huge data arrays. For example, each Out-of-FOV correction has to iterate and perform spatial transformations over a multi-dimensional matrix that contains $\approx 8.49 \cdot 10^7$ floating point values, which results in ≈300MB[1] of data. In addition, the intrinsic coincidence correction uses a temporary matrix with the same amount of data to store each corrected LOR within this matrix, and thus results in ≈600MB of minimum data required per frame.

In contrast to the management of matrices with the C programming language, in C^{++} such multi-dimensional data is generally handled within *storage classes* that provide different *accessor methods* for transparent access to data elements by their dimensional position. Unfortunately, the flexibility provided by such full object-oriented storage class implementations puts additional load on the performance of the algorithms that use these methods to access the data objects.

We therefore implemented two different approaches to the data management of our movement correction, and discuss them in the following subsections.

5.1.1 Direct-Access Matrix Management

The high frequency of data access within time critical computations requires the ability to access each data element quickly. Within movement correction, computational data is managed in huge three or four dimensional matrices, and the intrinsic coincidence correction algorithms access different elements of such a matrix at a time.

[1]assuming a floating point value to have 32bit.

Therefore, in contrast to a full object-oriented implementation of multidimensional matrix classes, a simple and quick-access implementation of a matrix storage class hierarchy was implemented.

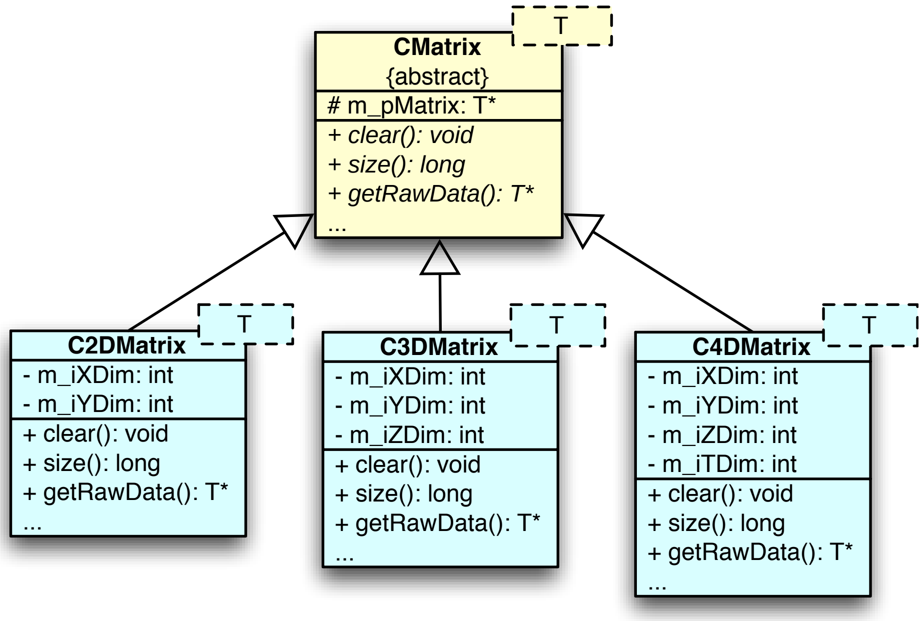


Figure 5.1: Class hierarchy of *Direct-Access* matrix classes

As the type of data stored within these classes varies on the different types of computations, the storage classes were implemented by *template classes*, that are instantiated with an appropriate data type at declaration time, thus providing data type independence. Figure 5.1 shows the different implemented classes which provide direct access via a `getRawData()` method to the matrix data stored within each instance. The increase of performance to access each element of the matrix data results from the fact that the data for these matrices is allocated and freed with standardized ANSI-C functions like `malloc()`, `free()` and not with their more complex C^{++} counterparts `new` and `delete`. In addition, the access to each matrix element is implemented with standard C^{++} multi-dimensional array access operators, e.g. `matrix[x][y][z] = 3`, rather than by calls to custom *accessor methods* within each matrix object, e.g. `setElement(x, y, z, 3)`.

This type of matrix storage class implementation allows to profit from the object-oriented way to manage hierarchical types, but also provides a method to have direct and efficient access to each matrix data, thus allows to quickly iterate through the data.

5.1.2 Indirect-Access Matrix Management

The implementation of *indirect-access* storage classes refers to the technique to access a specific data element with centralized accessor methods. In contrast to the quick-access matrix classes of the above section, these classes provide the possibility to dynamically change the size and dimensions of the stored data, and thus provides more flexibility in their application.

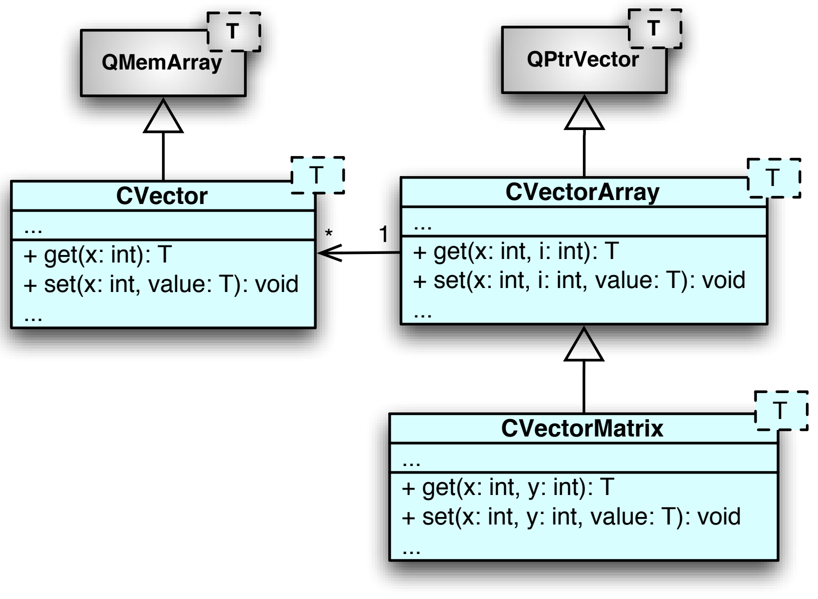


Figure 5.2: Class hierarchy of *Indirect-Access* matrix classes

In the movement correction application the classes shown in figure 5.2 were implemented and are mainly used for the spatial calibration procedures of the motion tracking system support, where no time critical computations are performed. As indicated in figure 5.2, these classes provide `get()` and `set()` *accessor methods* to access each data element separately. In addition, the classes provide basic vector/matrix computation methods to e.g. perform a vector-matrix multiplication, or to compute the inverse of a matrix.

5.1.3 GNU Scientific Library (GSL)

In addition to the provided matrix data storage functionality, the use of available third-party numerical C/C^{++} libraries has been evaluated during the implementation phase. In fact, the freely available GNU Scientific Library (GSL) is used within the application. In contrast to other available libraries, this library provides a wide range of advanced, well-proofed and up to date numerical algorithm implementations, such as algorithms for linear algebra or solving multidimensional minimization [GDT03].

For example, as discussed in section 2.3, the selection criteria for the transformations obtained from the motion tracking system will be calculated with help of such iterative multidimensional minimization. Within the indirect-access storage class `CVectorMatrix` these computations are implemented by a separate `analyzeMovement()` method that uses the GSL library to filter out transformations representing movements which can be neglected.

Furthermore, the `CVectorArray` class provides a `calcTransMatrix()` method that uses linear algebra functions of the GSL library to compute the homogeneous transformation matrix required for a proper cross-calibration, as discussed in section 2.1.1.

Due to the use of the GSL library, many numerical computations within the movement correction application are simplified by implementing the functionality within a flexible class-based hierarchy. Especially in fields of vector/matrix computations, the wide range of available numerical functions in the GSL library allowed to simplify the implementation wherever no time critical computations were necessary.

5.2 External Interfaces

During the development of the movement correction application, several required external interfaces, such as file formats, were implemented. After having specified the required data formats in section 3.3.2, the following paragraph concentrates on discussing the implementation to support these formats in own I/O routines or libraries.

5.2.1 Listmode File Management

Running in listmode, a PET scanner generally outputs the acquisition data stream to several sequential files. The sequential layout and the several gigabyte large set of data required the implementation of optimized file I/O routines, instead of using the standarized sequential access methods provided by Qt.

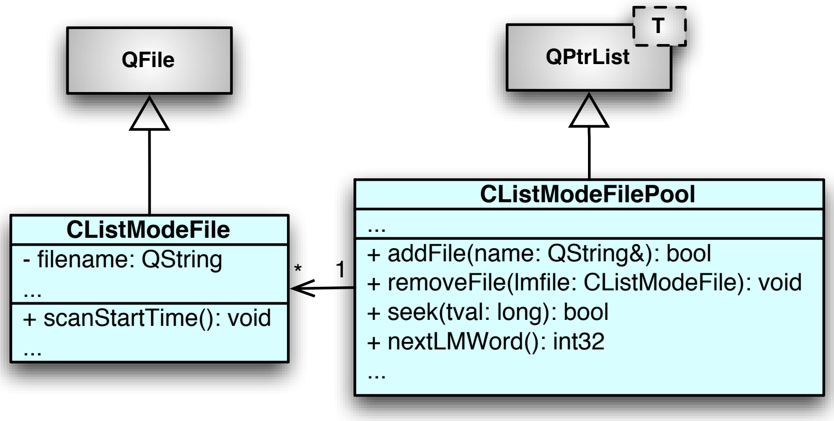


Figure 5.3: Class diagram of Listmode File Management classes

As each frame-based coincidence correction in the application accesses different data areas depending on the start and end time of a the frame, separate buffers were used to cache a preset amount of data in advance. This prevents the routines requesting the data to have to read directly from the raw files each time, thus increases the performance.

This mechanism, together with a quick-sort optimized file seeking algorithm are implemented in a `CListModeFilePool` class. It allows to add an unlimited number of listmode files by encapsulating each file in a `CListModeFile` object, and thus provides a *listmode pool* (cf. figure

5.3). In addition, the movement correction application uses such a pool as a transparent interface to obtain successive listmode words of a data acquisition. During the computations, each frame-based correction thread calls the `nextLMWord()` method of its own listmode file pool to retrieve the next listmode word without having to care about possibly existing file boundaries or position seeking mechanisms. Also, each frame thread using the pool does not need to care in which listmode file the first frame relevant data word starts and where exactly the last one is located.

5.2.2 ECAT Sinogram File I/O

Similar to the listmode input file format, output of the corrected coincidence data in a sinogram file according to the ECAT standard is an elementary requirement for the discussed application.

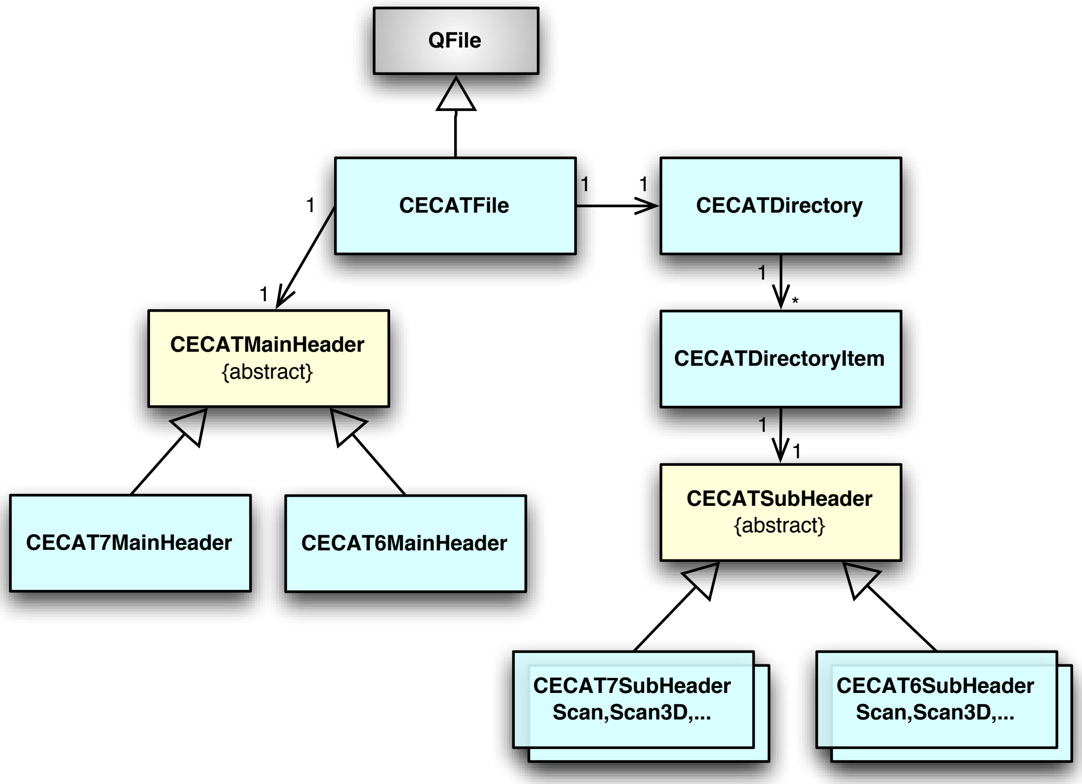

QFile
CECATFile
CECATDirectory
CECATMainHeader {abstract}
CECATDirectoryItem
CECAT7MainHeader
CECAT6MainHeader
CECATSubHeader {abstract}
CECAT7SubHeader Scan,Scan3D,...
CECAT6SubHeader Scan,Scan3D,...
1
1
1
1
1
*
1
1

Figure 5.4: Class diagram of ECAT C^{++} file I/O Library classes

As outlined in section 3.3.2.2, an own C^{++} library implementation of this standarized file format has been considered and during the development a separate branch to implement the whole functionality of this file format has been started. However, as only the 3D sinogram file output routines are required, other ECAT supported data types were not implemented, but scheduled for a future development.

The ECAT file format exists in two versions, the old ECAT6 and the current ECAT7 version. The main differences are the main header and sub header meta formats, as illustrated in figure 5.4. As the ECAT EXACT HR^{+} PET scanner outputs the sinogram data in ECAT7 format, the library implementation concentrates on supporting this version but owing to the object-oriented design of the library an implementation of the ECAT6 functionality is easy to achieve.

In contrast to other rarely existing ECAT6/7 file format libraries, the developed solution has been implemented with multithreading support. Besides the reentrant implementation, it supports the modification of two different directory items (matrices) in parallel. This allows the movement correction application to finalize a particular frame-based correction as soon as it is finished, and thus allows the user to evaluate the results of that frame while the correction of following frames is still running.

5.2.3 Motion Tracking Data

The movement information provided by the motion tracking system can either be obtained by directly receiving datagram packets via a network interface, or by using a command-line tool to pipe the received data into a predefined ASCII-based data file. Both ways are supported by the movement correction application. And as discussed in section 3.3.2.3, the data received from the motion tracking system generally contains information about each 3D-*marker* and 6D-*body* tracked within the FOV of the tracking system.

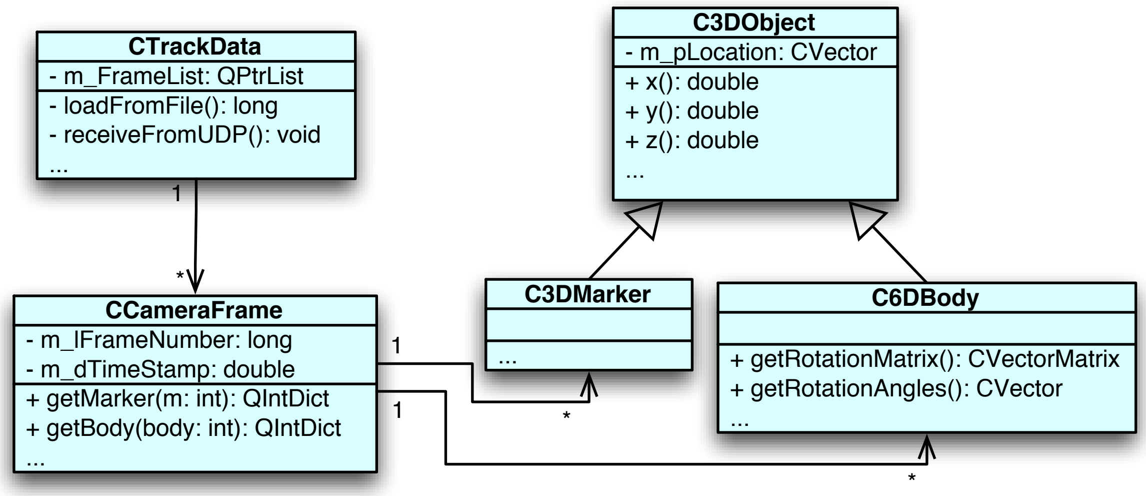


Figure 5.5: Class diagram of Tracking System Data classes

Figure 5.5 illustrates the class `CTrackData` which implements several methods to obtain the data either directly from a network interface or via loading it from a file. As the tracking system provides the data within *frames*[2] "time intervals" the `CCameraFrame` class acts as a container for the different markers and bodies within a single frame. As shown, this container class carries variables of the `C3DMarker` and `C6DBody` class which inherits the functionality from a base class `C3DObject`.

This way of implementation allows the movement correction to use the same classes for

[2]not to be confused with the PET scanner specific time frames.

both, loading motion tracking data from an ASCII file or by directly receiving data via network routines that are implemented within the `CTrackData` class.

5.2.4 XML based Import/Export

The eXtensible Markup Language (XML) is a meta description language to layout and manage data hierarchically [Con03]. In the movement correction application, XML is used in many different places throughout the implementation, but especially to import and export user or configuration dependent data. As discussed in the requirements, the applications have to provide a possibility to save the entire relevant data to a *listmode study* file, thus the format in which the data was designed is XML.

Listing 5.1: Using XML to map your data

```
<LMMClms version="1.0">
   <correction>
      <time start="" />
      <lorDiscretization mode="" />
      <scannerData>
         <lmFilePool>
            <lmFile filename="file1.lm" />
            <lmFile filename="file2.lm" />
            ...
         </lmFilePool>
         <framePool>
            <frame start="0" end="60000" />
            <frame start="60000" start="90000" />
            ...
         </framePool>
         <singlesRates filename="" />
         <normalization idnorm="" filename="" />
      </scannerData>
      <trackingData mode="" >
         <crossCalibration date="" />
         <tmatrix filename="" />
         <trackdata refBody="" filename="" />
      </trackingData>
      <outFOVCorrection relemmash="" ringmash="" anglemash="" enabled="" />
      <outputFormat normalizedOutput="" filetype="" filename="" />
      <headerData isotope_name="" ... patient_name="" />
   </correction>
</LMMClms>
```

As the Qt framework provides several portable classes to manage data within large XML trees, these classes were used to develop import and export methods to handle study relevant data. Aside from the XML tree structure, listing 5.1 shows that together with the correction relevant data like frame start and end time, the format contains an additional `<headerData />` tag that handles all patient and study relevant data which is also stored into the final sinogram main header.

By using XML-based study files, which are identifiable by their `.lms` file extension, a user can save all relevant data so that a recalculation of the movement correction is possible at later time.

5.3 Application Use Cases

An application does generally address several different user groups, and therefore also needs to support different use cases. In fact, the movement correction application has two main user groups.

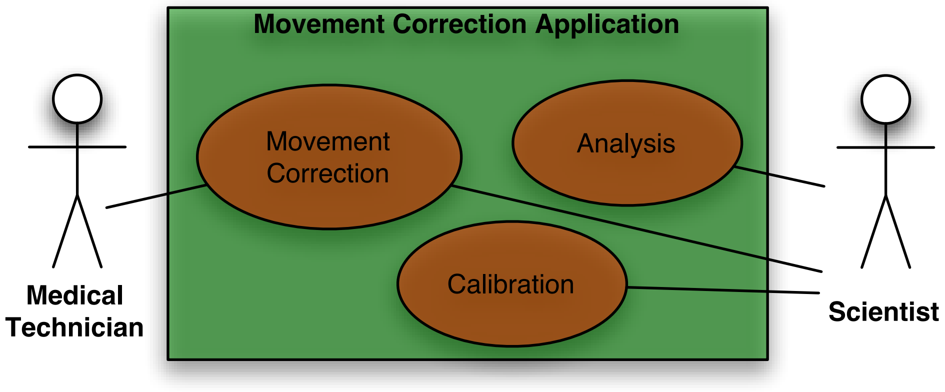


Figure 5.6: Use-Case diagram of Movement Correction Application

The *use-case diagram* in figure 5.6 summarizes these groups and shows the different use cases of the movement correction application. It illustrates that while a medical technician is interested in using the application for the intrinsic movement correction, the more experienced scientist uses the application for different purposes. He is interested in administrating the calibration of the tracking system or performing study based analysis. While the analysis use case is more related to providing graphical elements to output statistics within the application, the following subsections will discuss the two main use cases, *calibration* and *movement correction*.

5.3.1 Calibration Management

The tracking system includes a room and body based calibration (cf. section 2.4). As those calibrations have to be repeated regularly and as performed movement corrections are always based on a specific calibration, each performed calibration is permanently stored in a database like tree in the application. This tree is then exported to an application wide configuration file `lmmcsys.cfg` which uses XML as the data description language. This file includes, in addition to other information all calibration data available to the application, and acts as a repository of all performed calibrations since the first usage of the application.

When a new calibration of the tracking system is required, the application is used to either directly communicate with the motion tracking system or load the calibration relevant tracking

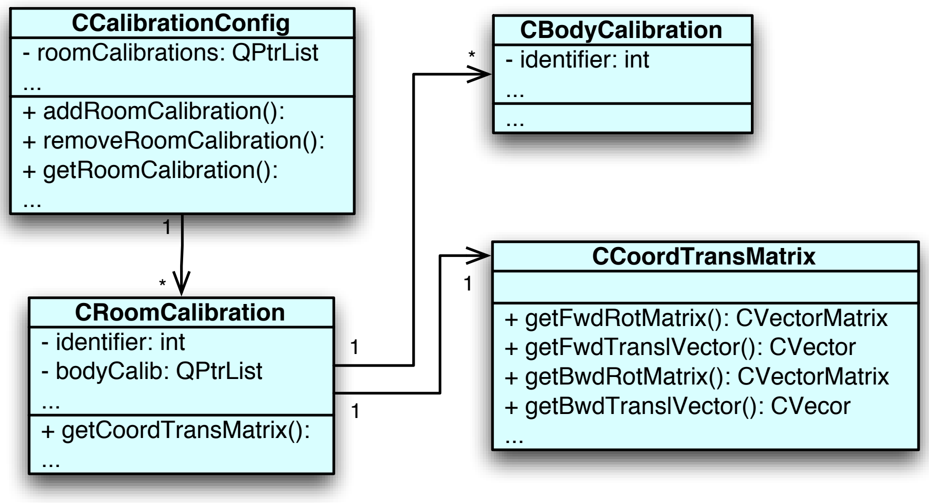


Figure 5.7: Class diagram of Calibration Management classes

data and calculate the cross-calibration (cf. section 2.1.1). Figure 5.7 shows, that with each new calibration the application instantiates a new object of the `CRoomCalibration` class and appends it to an object of the general configuration class `CCalibrationConfig`, which acts as a pool for all calibration relevant data. When the calibration data is obtained from the motion tracking system and a transmission scan of the PET scanner, the data is used to compute the forward and backward transformation matrix through the `calcTransMatrix()` method of the `CVectorArray` class. The results of this computation are then stored into an instance of the `CCoordTransMatrix` class, as illustrated in figure 5.7, and are used by each movement correction and Out-of-FOV thread.

As new room calibrations require the recalibration of all used 6D-*bodies* of the motion tracking system, a room calibration object stores a `CBodyCalibration` object for every existing body in a linked list. Within these body calibration objects, the specific body calibration information is stored and due to the hierarchical relationship between the room calibration object and its body calibration children, a logical dependency is always ensured. Thus, storing all room calibrations from the first use of the application in the calibration config object is required, to ensure that a user is able to recalculate a specific movement correction study at a later time.

5.3.2 Movement Correction

As discussed in chapter 4, the intrinsic movement correction happens within separate *frame thread*. Each of these threads contains several *Out-of-FOV threads* in which the OFC corrections are computed from the available motion tracking information. Finally, the results of the computations of these threads, together with additional corrections, are processed by a *Sinogram-Builder thread* and sorted into a sinogram.

Figure 5.8 shows the movement correction relevant classes that have been implemented.

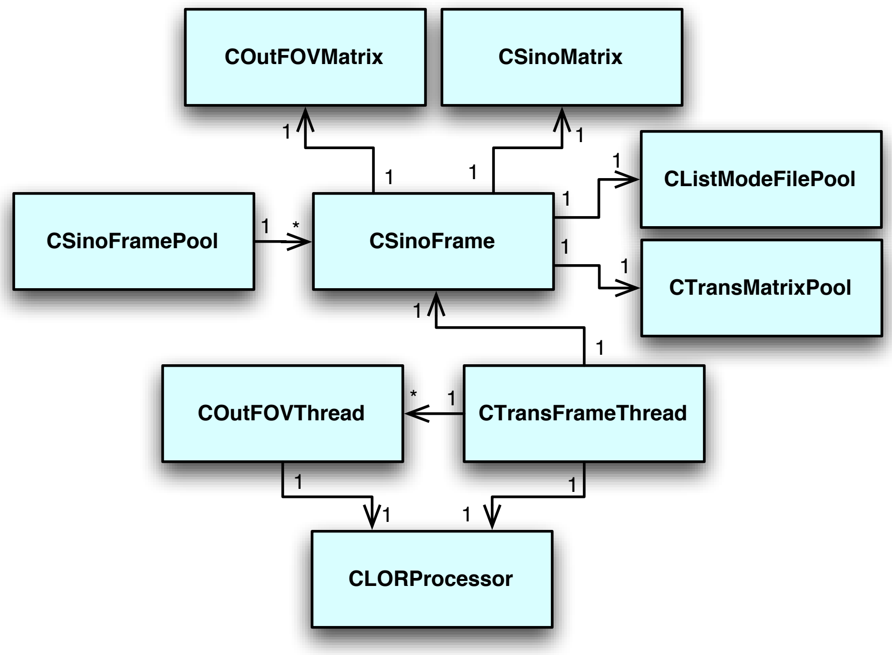


Figure 5.8: Collaboration diagram of Movement Correction classes

Because both, the intrinsic movement correction and the Out-of-FOV correction spatially transform LORs from one position to another, the figure illustrates the two main thread classes `COutFOVThread` and `CTransFrameThread` that use objects of the same `CLORProcessor` class. Within this class the functionality of the coincidence correction, as discussed in section 2 is implemented. In fact in a `process()` method the LOR processor distinguishes between a movement correction or an OFC correction. This allows to maintain the affected algorithm implementations more easily, thus having only a *single point of failure*.

Additionally, the central functioning of the `CSinoFrame` class is shown in the same figure. Looking at the movement correction process, this class was designed to implement a single sinogram frame. Through this class, both the frame and OFC correction thread access their data storage objects of the `COutFOVMatrix` and `CSinoMatrix` classes. When both threads have finished their computations, a third `CSinoBuilderThread` processes the data stored within these classes and sorts the finally corrected coincidences into a separate frame.

Similar to that, both threads obtain their input data (listmode and transformation matrix data) through the same sinogram frame object by using *sub-instances*[3] of the `CListModeFilePool` and `CTransMatrixPool` classes as discussed in section 5.2.1.

5.3.2.1 Thread Management

The usage of parallel computations in an application requires an additional management of the execution of these threads. As discussed in chapter 4, aside from the data synchronization

[3] *sub-instances* are copies of an existing object whereas only partially data is inherited depending on a condition.

mechanisms an application preferably uses a *thread dispatcher* to manage and optimize the execution of all threads involved.

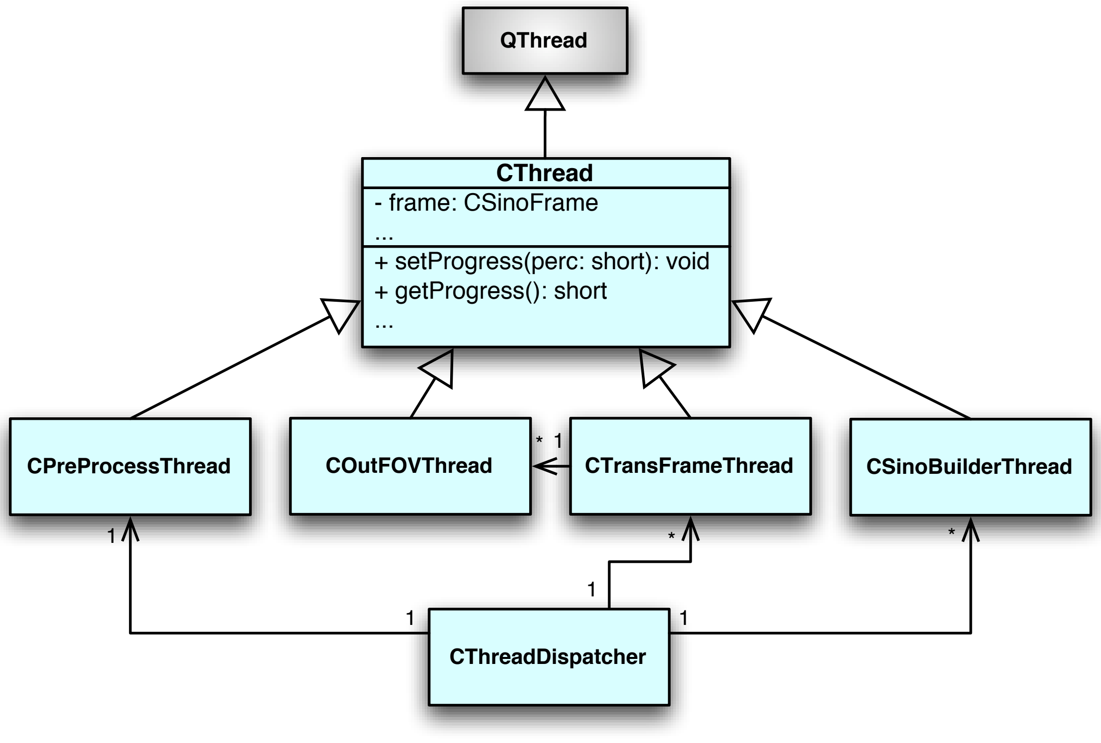


Figure 5.9: Class diagram of Multithreading classes

Therefore, in case of the movement correction application, the thread-based computations are managed by an object of the `CThreadDispatcher` class which interacts as the central communication unit between the threads and the user interface. All correction relevant data is passed to the dispatcher in advance so that as soon as the movement correction is started, the dispatcher acts as an independent interface that administrates the execution of all different threads until the computations are finished and the sinogram is saved.

Figure 5.9 illustrates the thread dispatcher together with the four implemented thread classes. All of these classes are derived from a `CThread` base class which itself inherits the functionality of the Qt multithreading class `QThread`. This way, the derived thread classes can access all data and communicate directly with the thread dispatcher to signal progress and error events.

The figure also shows, in addition to the frame and Out-of-FOV thread classes, the pre and post processing classes, `CPreProcessThread` and `CSinoBuilderThread`. As the name implies, the `CPreProcessThread` is used to preprocess data before the intrinsic movement correction takes place. It is used to calculate conversion tables and precalculates the correction matrices, so that the movement correction can use the precalculated data instead of having to calculate it on demand. On the other hand, the `CSinoBuilderThread` is used to sort and store the final data in the sinogram file. As soon as the movement correction thread and all Out-of-FOV threads of a frame are finished, this sinogram building thread will be started by the dispatcher, and sorts all corrected coincidences into the sinogram.

The fact that multiple `COutFOVThread` objects are accessing a shared `COutFOVMatrix` object at the same time requires synchronization (cf. chapter 4). Within the movement correction application, the synchronization of the data access with the Out-of-FOV matrix is implemented by logically dividing the data area of the matrix into sections that are protected by `QMutex` objects. As soon as an Out-of-FOV thread wants to access a data area it locks it with a mutex. Other OFC threads trying to access the same area are then not able to work on the same area at the same time. However, the access to this data areas is registered in a separate queue in each OFC thread so that if one OFC thread cannot access a particular data area, it proceeds with the next one and queues the computation of that locked area for a later time. This way, all OFC threads can operate on the same data matrix without waiting until another one has finished its computations.

5.4 User Interface

The implementation of the user interface concentrated on the required user interface types defined in chapter 3. The previous experimental implementation of the movement correction supported the use of a command-line interface only, which was a severe limitation for its usability in routine operation and a graphical user interface is therefore an important key feature.

Nevertheless, an advanced command-line interface was implemented, to support the already discussed needs of the scientists at the PET center.

5.4.1 Graphical Interface

The Qt framework was chosen as the graphical user interface development kit. Due to the different discussed use cases the graphical interface has been implemented by separating the GUI elements into the two main use cases, *Movement Correction* and *Calibration*. Aside from the main window and menu bar, implemented by the `CMainWindow` and `CMainMenuBar` classes, the functionality of these two use cases was graphically separated by dividing the GUI with *tabbed panes*.

This tabbed browsing functionality has been implemented in the `CMainWidget` class of the application. The use of Qt's `QTabWidget` class within the main widget allows to dynamically show or hide a specific tab, e.g. allowing to show and hide the `CCalibrateWidget` as soon as the user requests the novice or expert mode of the application.

Figure 5.10 illustrates the different class relationships between the main GUI classes of the application. The fact that all relationships shown in this figure represent a `[1:1]` relation was used to design the classes as so called *singleton classes*. By using this type of design pattern for a class, only a single instance of a class is guaranteed to exist during runtime [GHJV03]. This ensures an additional safety during the development of the graphical user interface and makes

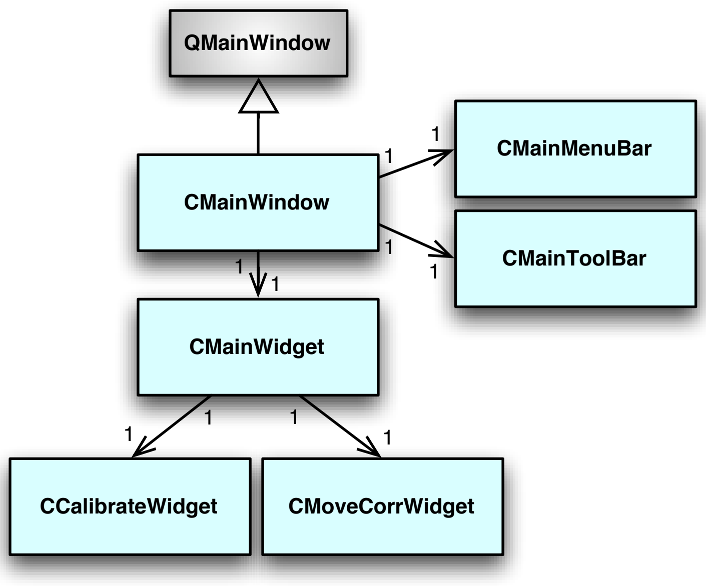


Figure 5.10: Class and Collaboration diagram of Main GUI classes

it easier to maintain the different GUI elements within an application.

5.4.2 Command-Line Interface

Another possibility to use the movement correction is by using the command-line interface (CLI). To distinguish between a GUI dependent application start and a command line based start, the application checks if the user has supplied some additional commands at the command-line (in `argc/argv`). If so, this command string is forwarded by a `CCmdLineStart` object to a `CGetOpt` object that handles all available command line options.

The object-oriented implementation of the command line parsing classes, as shown in figure 5.11, allows to manage an unlimited number of options. Options are separated in several `CGetOptCategory` category objects depending on their purpose. And each of these category objects contain several command line options. During the parsing of the command line, through the `CCmdLineStart` object, all option objects within the `CGetOpt` object are checked for validity. If true, a special method within the command line start class is executed which processes the requested operation.

5.5 Advanced Memory Management

Having numerous threads computing in parallel generally causes an increase of the total memory usage of an application. In our case, the fact that for every thread several sub-threads are computing on the movement correction, causes a steady increase of the memory usage. For each additional frame computation running in parallel, the overall memory usage increases by 600MB. This is due to the fact that every frame computation carries a 300MB large sinogram

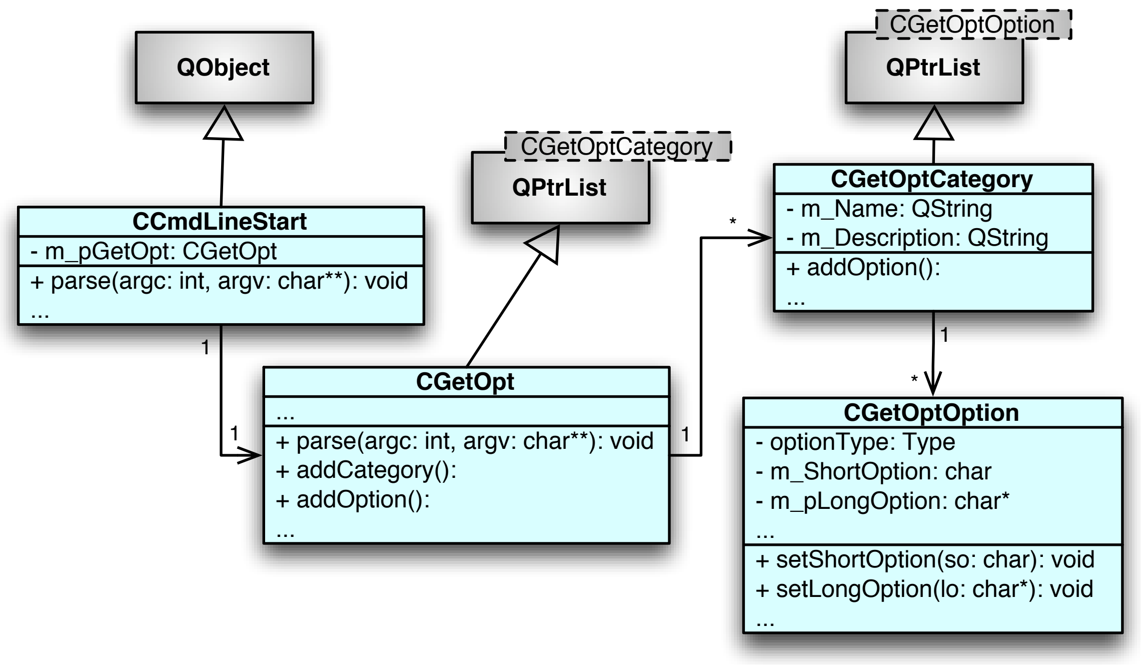


Figure 5.11: Class diagram of Command-Line Management classes

matrix together with an additional Out-of-FOV matrix of the same size.

If we have the application running on a four processor machine with a total of 16GB RAM available[4], the size of required memory temporarly raises above 4GB. Of course, within a 64bit operating system environment this would not cause any problem. But if the application is compiled in a 32bit environment, processes (including their threads) allocating more than 4GB address space end up in a *low memory condition*. As the movement correction has been planned as a platform independent application, memory allocation exceptions have been introduced to handle such a low memory situation. As soon as the application's wide memory allocation reaches the 4GB limit, any thread that tries to allocate additional memory will be suspended with a *wait condition* (cf. chapter 4). When another thread has finished its computations, it frees its temporary memory allocations and signals the waiting threads that they can retry to allocate the required memory and continue with their computations.

[4]this has been the case during development of the movement correction application.

Chapter 6

Validation

The following sections concentrate on validation of the results produced by the implemented movement correction.

6.1 Sinogram Sorting

After successful movement correction, the LORs are sorted into a sinogram file. In order to verify the correct functioning of the developed sorting routine, its results were compared with sinograms produced with `lm_sorter`, a sinogram sorting tool also developed at the PET center Rossendorf [Jus00]. The created sinograms of both applications were quantitatively compared by stripping the non-sensitive sub header data and using the `cmp` binary comparison tool to verify that the remaining data is equal. The test was repeated for several data sets and in no case any difference were found.

The `lm_sorter` application was validated in research with the PET research center Jülich in 2000, therefore the routines creating the sinograms in `lmmc` are assumed to be correct as well.

6.2 Movement Correction

To be able to verify the correct functioning of the newly implemented movement correction algorithm, test measurements with a *Hoffman brain phantom* were performed. This type of phantom is a device which is used to simulate brain investigations. For the tests the phantom was filled with a mixture of water (H_2O) and 224 MBq of the ^{18}F-FDG tracer substance.

In three successive tests, the phantom was placed at the patient bed and was kept at rest for the first half of data acquisition time. In five consecutive steps it was then moved during the second half of the emission scan. In order to verify and track the produced movement, a 6D-body (cf. figure 2.9) was fixed on the phantom. The movements were recorded to a separate `lmmc`-readable tracking file.

In the first performed test the Hoffman phantom was moved in axial direction (z), in the second test in transaxial direction (y), and in the third test it was rotated along the z axes of the scanner coordinate system until it reached a predefined location. For each test measurement three images were created. The first one with data acquired during the time the phantom remained in its rest position and the second with the data from the second half of the acquisition without any applied movement correction. The third image was again created with the data from the second half of the acquisition but with application of the movement correction. All created data sets were reconstructed with the standard routines available on the PET acquisition computer. The correct functioning of `lmmc` was finally determined by comparing the uncorrected and corrected images with the image of the phantom in rest position.

6.2.1 Axial Movement

The movements performed during the first test measurement are illustrated in figure 6.1.

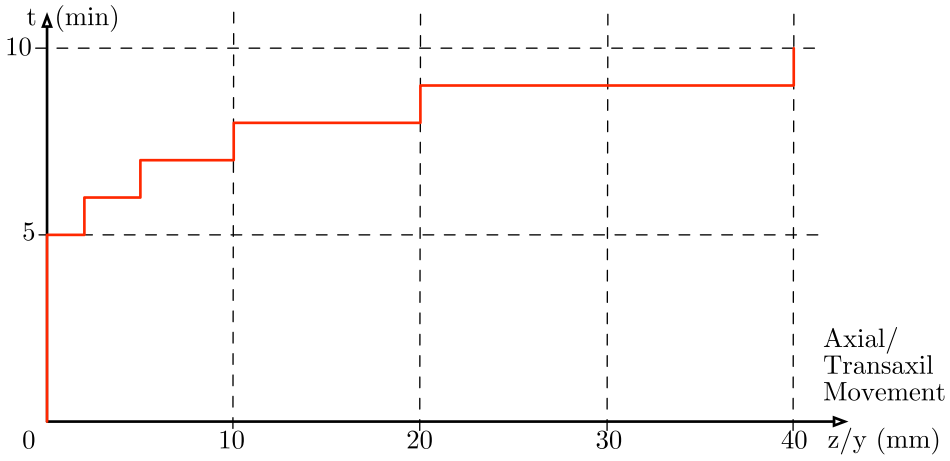


Figure 6.1: The manually produced movements of a *Hoffman phantom* as function of time during the test measurements with movements in axial (z) and transaxial (y) direction. The phantom was stepwise moved during a period of 10 minutes by a maximum of 40 mm.

In a total acquisition time of 10 minutes the phantom was moved 40 mm in axial direction (z). The reconstructed images are shown in figure 6.2. The top row of the figure shows sagittal planes of the three reconstructed volumes of the data set in rest position (left panel), of the uncorrected data set (middle), and the movement corrected data set (right). The middle row shows the difference between the uncorrected and corrected image, and the image in rest position. In the bottom row logarithmic intensity-correlation histograms are displayed. In these plots the x-axis corresponds to the logarithm of the intensity of a voxel[1] in the volume of the phantom in rest position, whereas the y-axis corresponds to the logarithm of the intensity of a voxel in the uncorrected, respectively corrected volume. The histogram is computed by looping over all voxels of the volumes to be compared and incrementing the histogram bins according to the

[1]Short for *volume pixel*, the smallest distinguishable box-shaped part of a three-dimensional image.

corresponding voxel intensities. Note, that a comparison of two identical data sets results in a logarithmic intensity-correlation histogram that follows a diagonal line with a slope of one.

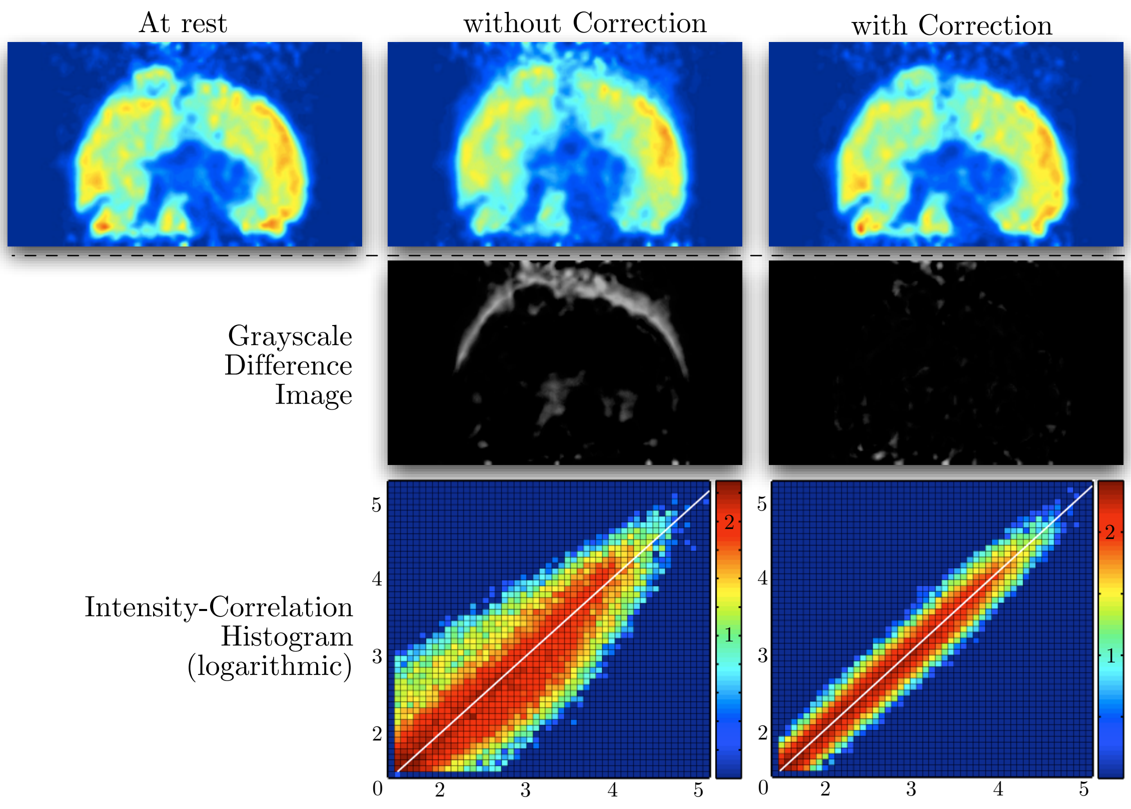

At rest
without Correction
with Correction
Grayscale Difference Image
Intensity-Correlation Histogram (logarithmic)
0
2
3
4
5
1
2

Figure 6.2: Results of the test measurement with movement in axial direction (z). The three images displayed in the top row show sagittal planes of the reconstructed volumes of the data set in rest position (left panel), of the uncorrected data set (middle), and the movement corrected data set (right). The middle row shows the difference between the uncorrected and corrected image in rest position. In the bottom row logarithmic intensity-correlation histograms are displayed. The image quality has noticeably improved by application of `lmmc`.

A comparison of the uncorrected and corrected data sets suggests that application of the movement correction does indeed result in a significant improvement of the image quality. Especially the image of the uncorrected data set shows a blurring at the top of the image, which represents the loss of information. Comparing the uncorrected and corrected image, this is much improved in the corrected data set, which hardly differs in focus from the image in rest position. This is also confirmed by the difference images and the provided intensity-correlation histograms. However, the remaining differences are mainly caused by the random nature of the coincidence registration and tracer decay, but also partly by the remaining discretization limitations of the movement correction algorithms.

6.2.2 Transaxial Movement

In analogy to the test measurement with movement in axial direction, the phantom remained in rest position during the first half of the data acquisition, and was then stepwise moved as illustrated in figure 6.1, but in transaxial (y) direction. The total acquisition time was 10 minutes with a maximum transaxial displacement of the phantom of 40 mm.

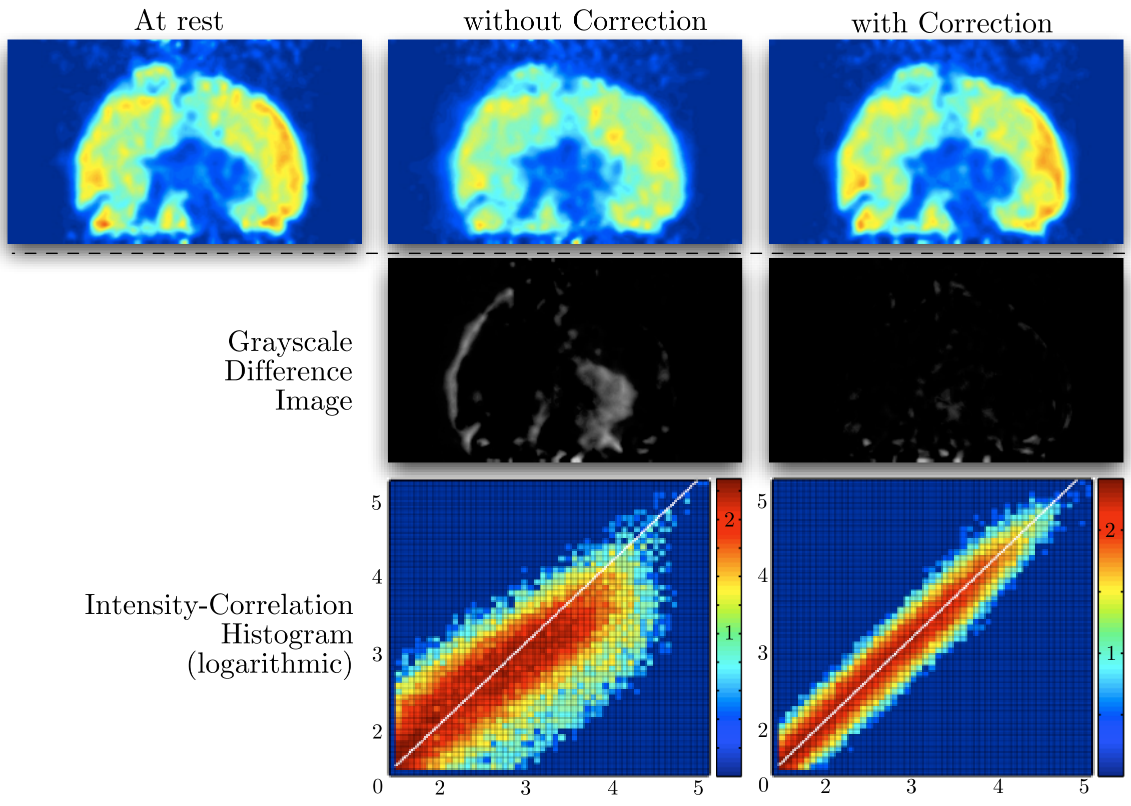


Figure 6.3: Similar to figure 6.2 but for test measurement with movement in transaxial direction (y direction).

Figure 6.3 shows the reconstructed images. The representation of the data is similar to case with axial movement (cf. figure 6.2), which is described in detail in section 6.2.1. Here the occurred blurring is visible at the left side of the uncorrected data sets, which is also confirmed by the difference images at the middle row of the figure. However, in this case the data shows that `lmmc` is able to correct movements occurring during data acquisition, thus its application results in a significant improvement of the image quality and minimizes the information loss.

6.2.3 Rotational Movement

During this test measurement the *Hoffman phantom* was stepwise rotated along the z axis during the data acquisition. In analogy to the previously described tests, it remained in rest position during the first half of the acquisition to allow the accumulation of enough data for the unmoved data set.

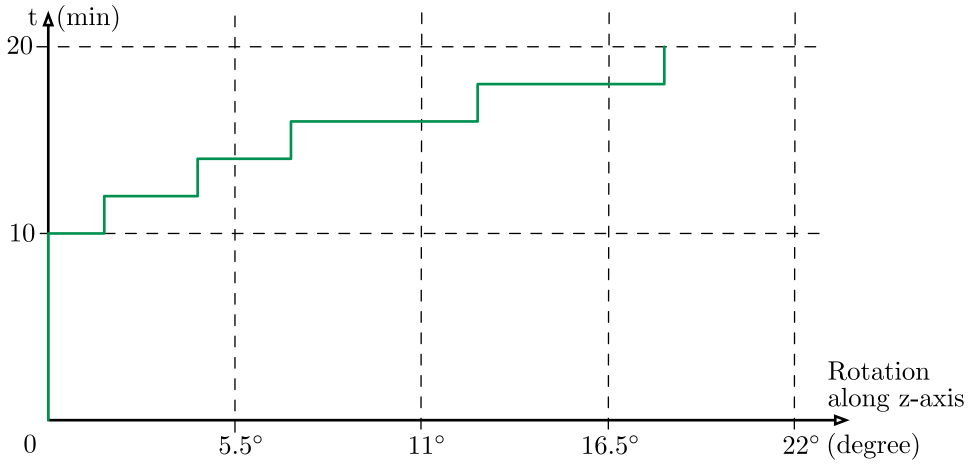


Figure 6.4: Similar to figure 6.1 but for test measurement with rotation along the z-axis.

Figure 6.4 illustrates the applied movements, whereas the total acquisition time was 20 minutes and the maximum angle of rotation was approximately 18°. Results are shown in figure 6.5, where compared to figure 6.2 and 6.3, the sagittal slices are replaced by transaxial slices.

Again, application of the movement correction results in an almost ideal compensation of the movement artifacts, where this observation is also supported by the generated intensity-correlation histograms.

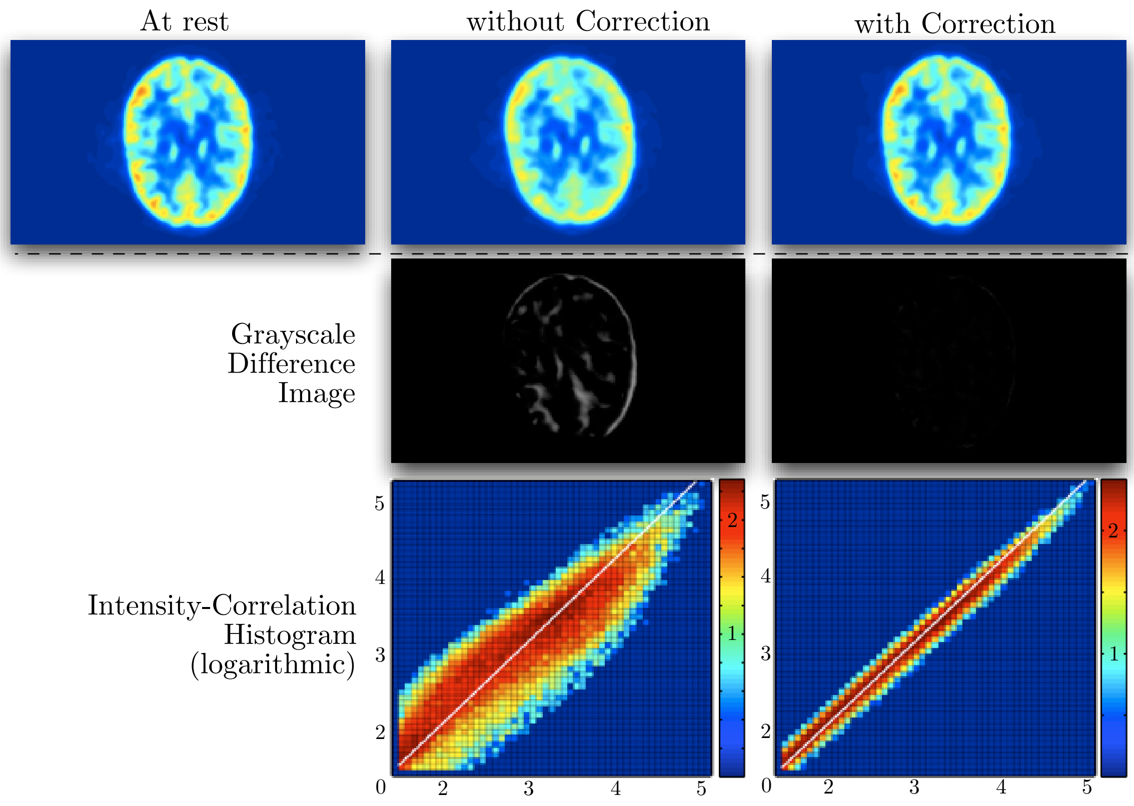

At rest
without Correction
with Correction
Grayscale Difference Image
Intensity-Correlation Histogram (logarithmic)
5
4
3
2
0
2
3
4
5
2
1

Figure 6.5: Results of the test measurement with rotation along z-axis. The three images at the top show transaxial planes of the reconstructed volumes of the data set in rest position (left panel), of the uncorrected data set (middle), and the movement corrected data set (right). The images in the middle row show the difference between the uncorrected, respectively corrected image, and the image in reposition is shown. In addition, logarithmic intensity-correlation histograms are displayed in the bottom row. Similar to the previous tests, also here the image quality is highly improved by the use of `lmmc`.

6.2.4 In Vivo

Finally the movement correction implementation was tested with acquisition data from a volunteer patient, who agreed to assist in this test prior to a regular whole body examination. The patient was advised to remain still during the first three minutes of the acquisition. Then he was asked to turn his head and to remain still in that new position for the rest of the acquisition, which lasted another three more minutes.

During the entire acquisition period the 6D body discussed in section 2.9 was attached to the patient's head allowing to capture the occurred movements. Figure 6.6 shows the tracked motion of all six degrees of freedom. It shows that the movement included all degrees of freedom but was dominated by a rotation along the scanners z-axis (*dtrack ang1* in figure).

Two data sets were produced, one with application of the movement correction and one without the correction. Figure 6.7 shows the reconstructed images of these data sets, where

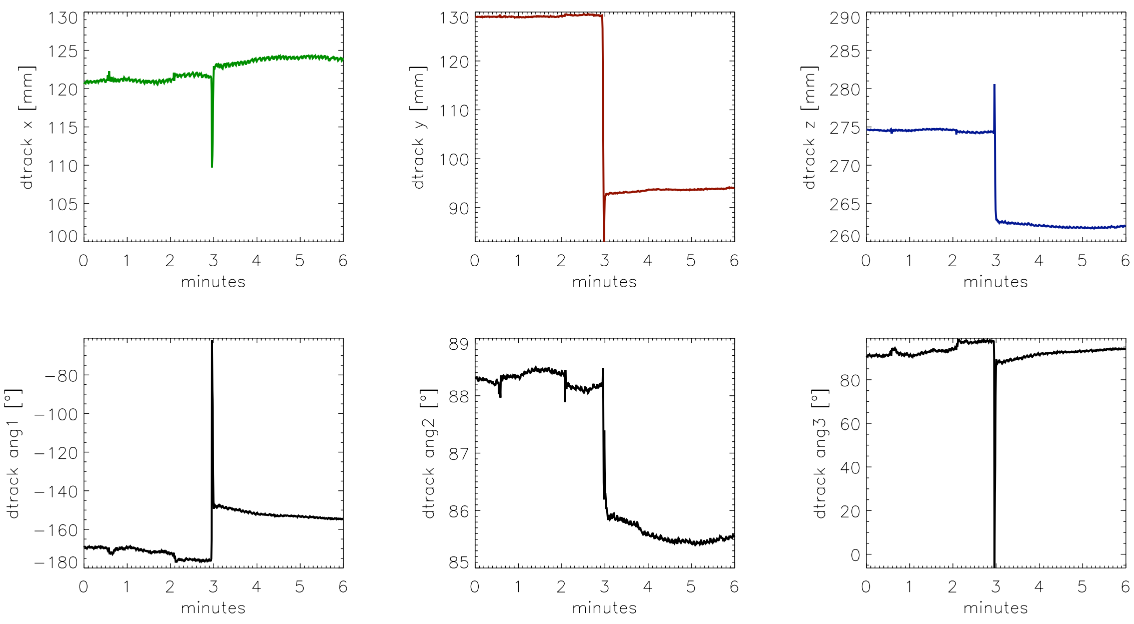


Figure 6.6: The movement during an *in vivo* study. The patient was advised to turn the head after three minutes of data acquisition. The graphs show the tracking information obtained from the stereoscopic tracking system available at the PET center with coordinates as defined in figure 2.1.

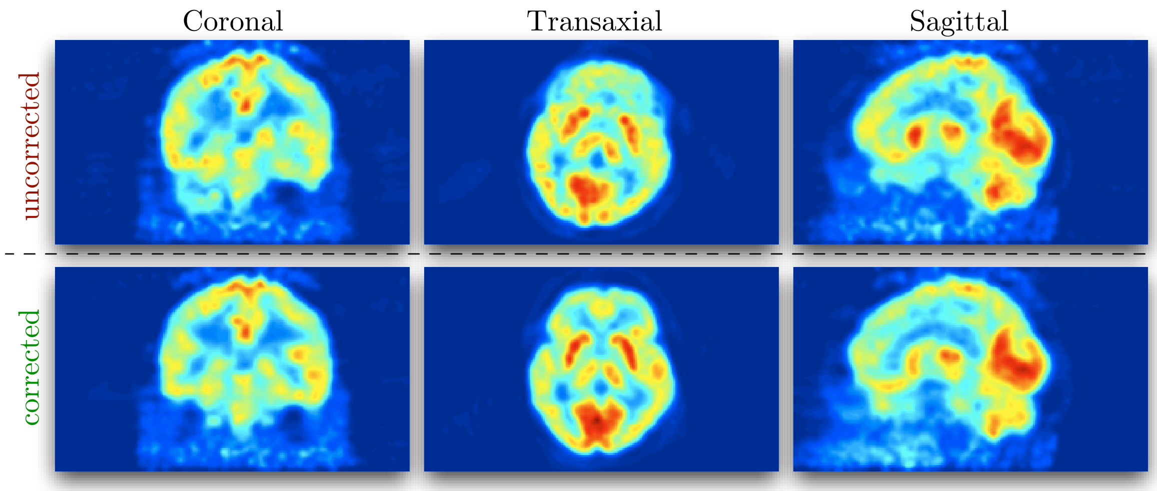


Figure 6.7: Reconstructed images of an *in vivo* study where the patient was advised to turn the head during the data acquisition. The upper row shows the images where artifacts and blurring caused by the patient's movement are especially visible within the transaxial view. In contrast, the lower row shows the same acquisition data, but after movement correction with `lmmc`. The improvement is obvious.

three different slices of each image volume are shown. The blurring caused by the movement of the patient is perfectly visible within the transaxial slice of the uncorrected data set. Again, its corrected counterpart shows a significant improvement of image quality, as well as the recovery of important brain structures.

6.3 Performance Comparison

After having verified the accuracy of the implemented movement correction algorithms, performance tests were performed. Computation times of the sequential implementation (`trans_lm`) and the new parallel implementation (`lmmc`) were recorded during four successive tests (*t1*-*t4*), as listed in table 6.1.

	Frames	**OFC**	**LDC**	**Program**	**Correction + Sorting**[2]	
t1	1	1/1/1	normal	`trans_lm`	9:48:25h + 0:30:51h	1
				`lmmc`	2:10:18h	2
			enhanced3D	`trans_lm`	25:01:47h + 2:39:50h	3
				`lmmc`	3:10:25h	4
t2	1	4/4/1	normal	`trans_lm`	2:50:37h + 0:32:44h	5
				`lmmc`	2:05:29h	6
			enhanced3D	`trans_lm`	17:58:23h + 2:45:11h	7
				`lmmc`	2:57:20h	8
t3	1	8/8/1	normal	`trans_lm`	2:29:57h + 0:32:3h	9
				`lmmc`	2:04:37h	10
			enhanced3D	`trans_lm`	17:53:53h + 2:51:55h	11
				`lmmc`	2:58:08h	12
t4	21[3]	1/1/1	normal	`lmmc`	2:34:34h	13
			enhanced3D	`lmmc`	3:23:56h	14
		4/4/1	normal	`lmmc`	1:17:20h	15
			enhanced3D	`lmmc`	1:36:05h	16

Table 6.1: Results of the performance tests (*t1*-*t4*). The performance of the sequential implementation (`trans_lm`) was compared to our implemented application (`lmmc`). Several different options were used during the tests to show the impact on the performance.

The tests were performed with listmode data of a 60 *min* PET study. As the mashing settings of the Out-of-FOV correction, as well as the type of the LOR discretization algorithms have an deep impact on the performance, *t1*-*t3* concentrated on varying the settings of these corrections, thus allowing to draw conclusions on their impact on the movement correction.

[2]in contrast to `lmmc`, the `trans_lm` application is not able to sort the corrected LORs in a sinogram - this task was therefore done by using `lm_sorter`.

[3]because `trans_lm` does not support multi-frame studies, these tests have been done with `lmmc` only.

Test *t4* concentrated on analyzing the performance of the movement correction of a multi-frame study. As the sequential `trans_lm` implementation does not directly support this type of studies, the tests were performed with the parallel implementation only. Furthermore, as `trans_lm` does not directly sort the corrected data into a sinogram, the sorting has been done with the `lm_sorter` command-line tool which was also developed at the PET center.

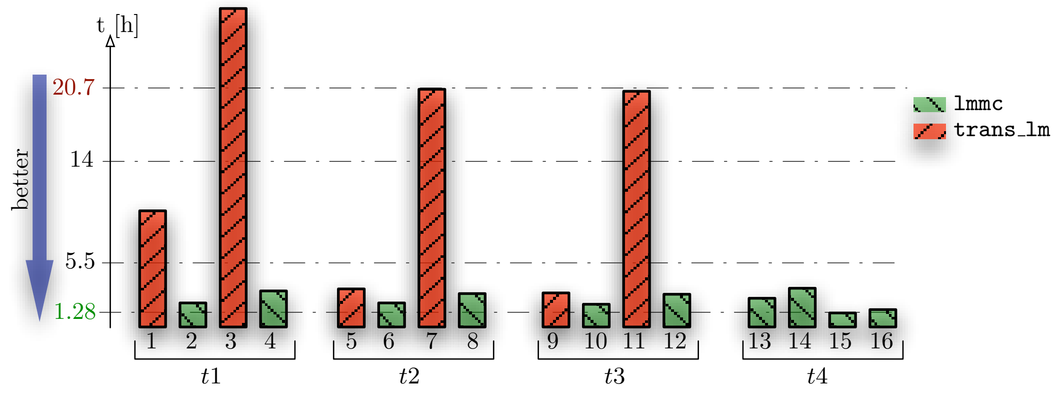


Figure 6.8: Plot illustrating the results of the performance test listed in table 6.1.

Figure 6.8 shows a graph in which all results of the performed tests are shown. To evaluate the performance improvement, a significant difference in the computation time is especially obvious between test 3 and 4. Here, in contrast to the $> 27h$ lasting computation of `trans_lm`, the same type of computation only took around 3 hours with `lmmc`. This results in a speed up of factor ≈ 9 due to the parallel execution of the Out-of-FOV correction.

However, since routinely performed PET examinations are generally having multiple frames, the *t*4 tests are more suitable for evaluating the usability of `lmmc` in routine operation. In addition, analysis and tests showed that the command-line options "OFC=4/4/1" and "LDC=enhanced3D" are the most suitable combination. This takes us to test 16 where the acquisition data of an one hour routine PET examination was corrected by `lmmc` in only $\approx 1\frac{1}{2}$ hours (**1:1.5** proportion).

Even if `trans_lm` is not able to handle multi-frame studies, the comparison between test 7 and its multi-frame complement number 16 shows, that due to the parallel implementation of each frame and its OFC children, the movement correction was sped up by a factor of ≈ 21.

6.4 Summary

The discussed validations and comparisons against the previously existing implementation of the movement correction (`trans_lm`) have shown the correct functioning of `lmmc`, and that the implementation is able to compensate the short comings of this implementation as discussed in section 3.1.

Especially the comparisons show that in fields of the increased performance the general usage of a movement correction in PET is greatly improved. With help of this application, the PET center Rossendorf accounted for the use of a movement correction within routine examinations as reasonable. This has not only been accounted due to the improved performance, but also due to the elimination of the following shortcomings of the previous movement correction implementation:

- No graphical interface for an intuitive use of the movement correction principles was available. (`trans_lm`)
- No multi-frame studies were supported. (`trans_lm`)
- The full movement correction implied the sequential application of several different tools and thus caused additional complexity for the process. (`trans_lm`, `lm_sorter`)
- No parallel sorting of multiple frames was possible. (`lm_sorter`)
- The non object-oriented implementation and the use of different types of programming languages caused the source code to be hardly maintainable and error prone. (`trans_lm`, `lm_sorter`)
- Algorithms were mainly implemented in a non reusable fashion. (`trans_lm`, `lm_sorter`)
- Data interfaces to other applications were not available. The import and export of acquisition data was limited to the individual application. (`trans_lm`, `lm_sorter`)

Chapter 7

Future Developments

This chapter briefly reviews possible directions for future developments of `lmmc`:

- Although not part of this thesis, the possibility to distribute computations on several different machines was accounted for. During the development of `lmmc`, all parallel computing related elements (cf. chapter 4) were implemented in a way to enable the export of computations over a network interface via XML streams (XML-RPC). Therefore, in future developments of the movement correction this technique should be used to increase the performance.

- To enable other PET facilities to use `lmmc`, other scanner and motion tracking system combinations can be supported. All main data structures and defines are parametrized. This allows to easily adapt `lmmc` to scanner and motion tracking system specific details.

- Other data formats like 64bit based listmode formats and the *Interfile* format[1] can be implemented in a future version of the application. This would allow to directly load and save data in these formats without conversion by other tools. Again, this would be of interest for application with other PET scanners.

- Additional tools embedded into the graphical user interface would improve the usability for quality control purposes. An elementary implementation of such graphical tools has already been developed but needs to be enhanced to provide a *quality control facility* for the application. This would allow the checking the accuracy of the motion tracking system in given time intervals.

- With a more sophisticated thread distribution, movement correction algorithm performance can probably be further enhanced. This would require the implementation of a centralized thread management entity in the *thread dispatcher*, verifying how many processors are currently available and assigning priorities to individual threads.

[1] cf. `http://www.keston.com/Interfile/interfile.htm`

- The intrinsic spatial movement correction can be parallelized by further analysis of the algorithms. This would allow to speed up the movement correction of a single frame study or to distribute computational intensive parts to other machines.

- The currently memory intensive implementation can be enhanced to manage the large amount of required memory in a separate memory management facility. In this facility, the matrices can be stored in a compressed format, allowing to reduce the total memory usage, but introducing more overhead. This would enable the use of the application on computer systems with low memory.

- Existing open-source image reconstruction software [LTJZ03] can be integrated into the application, allowing to directly generate the final images without having to use the reconstruction software of the PET scanner.

- The multithreaded C^{++} ECAT6/7 file format implementation can be moved to an own software development project. This would allow the use in other applications which require access to ECAT files in a multithreaded or object-oriented environment.

Chapter 8

Summary

Reducing the influence of movements during patient investigations is a persistent topic in Positron-Emission-Tomography. Therefore, a new movement correction technique has recently been developed at the PET center Rossendorf/Germany. In contrast to other approaches, this *coincidence based correction* provides the possibility to apply the correction directly to the raw coincidence data (*listmode*) of the tomograph. It turned out, however, that due to the large amount of data and the complexity of the involved algorithms, the use of the initial implementation resulted in processing times which were unacceptable for routine operation. Therefore, the goal of this thesis was the development and implementation of a parallel computing optimized movement correction method to overcome these restrictions.

To achieve this goal, a dependency analysis of the existing correction algorithms was performed and independent areas of computations were identified. The results of this analysis were transfered into an object-oriented software engineering process. After the specification of implementation boundaries, the identified use-cases and the application work-flows were described with the UML to assure *reusability* and *extensibility*. The resulting modules were implemented using the C^{++} programming language. The parallelized parts were embedded into separate threads to allow multiprocessor systems to distribute them onto multiple processors. Throughout the implementation, runtime optimizations for time critical regions were performed. By using the platform independent GUI and application framework "Qt", it was possible to keep the application portable and to include a user friendly graphical interface which is suitable for use especially by the technicians of the PET facility. In addition, a module to maintain and perform calibrations of the motion tracking system was implemented as an integral part of the application.

Performance evaluations gave the following results: due to the parallel optimization the processing times were reduced by a factor of $\approx$13. The data acquisition versus computation ratio could thus be reduced from approximately 1:20 to 1:1.5 (1h data acquisition : 1.5h movement correction). This, together with the implemented intuitive GUI enables the use of coincidence based movement correction in routine patient investigations, thus providing improved tomograph imaging.

Appendix A

User Documentation

A.1 System Requirements

The implemented application (named `lmmc`) has been developed in C^{++} using the freely available platform independent Qt framework. In addition to the availability of the Qt framework, the application uses the free available mathematical GNU scientific library (GSL).

Platform	*Operating System*	*Qt version* [Qt03]	*GSL version* [GDT03]
Sparc^{TM}	Solaris 2.8/2.9	3.2.3	1.4
PowerPC^{TM}	MacOSX 10.3.1 (Panther)	3.2.3	1.4
x86	Linux 2.4.22	3.2.3	1.4
x86	Microsoft WindowsXP	3.2.3	1.4

Table A.1: The movement correction natively supports different platforms and operating systems. The table shows the tested platforms and operating systems combinations, as well as the used Qt and GSL library versions.

Fortunately, versions of both exists for all major operating systems. This allows to use the movement correction application on all modern platforms and operating systems. The platform and operating system combinations on which the application has been tested are listed in table A.1.

However, in order to fully benefit from the multithreaded implementation, systems should have at least 2 processors with a minimum of 2 gigabyte RAM available. In case of this thesis, the tests were performed on a 64bit UltraSparc-III^{TM}v480 Sun 4x900MHz multiprocessor system running under Solaris^{TM}2.9, equipped with a total of 16 gigabyte RAM available.

A.2 Command-Line Options

The use of the `lmmc` via command-line options was implemented to support batch-processing. The available command-line options are listed in listing A.1.

Listing A.1: `lmmc` command-line options

```
LMMC - ListMode Movement Correction v0.12 (11.11.2003) [SPARC]
Copyright (C) 2003 by Jens Langner

Usage: lmmc <COMMAND> [OPTIONS]...

Commands:
<no command> - start LMMC with graphical user interface.
correct      - do a batch listmode correction.
sort         - just sort the specified listmode files into the sinogram.
transdata    - command to do transformation matrix computations.

Listmode correction and sorting:
  -L, --lmsfile=STRING    use information from listmode study file (*.lms) for processing
  -l, --lmfile=STRING     use listmode files (*.lm) [file1,file2,...]
  -f, --frmfile=STRING    use frame defintion file (*.frm)
  -F, --frm=STRING        direct frame specification in sec [0-900,...]
  -s, --sglfile=STRING    use singles file (*.sgl)

Correction only:
  -t, --tfmfile=STRING    use transformation matrix file (*.tfm)
  -N, --normmode=0        normalization mode
  -n, --normfile=STRING   normalization file specification (*.N)
  -m, --ofc=4/4/1         Out-of-FOV correction with mash [RElement/Angle/RingComb]
  -c, --ldc=2             LOR discretization correction
                          [0=disabled, 1=normal, 2=enhanced3D]

Output sinogram:
  -o, --outfile=STRING    output sinogram to file (*.S)
  -O, --outtype=0         sinogram file type
  -k, --mskfile=STRING    use mask file for specifing sinogram study relevant header
                          data. You can either specify a (*.msk) or ECAT compatible file.
  -a, --acp=2             specify used angular compression (multiple of 2)
  -D, --double[=yes/no]   use double precision instead of float during computations

Thread distribution:
  -T, --sft=0             max. simultaneous running Frame threads [0=auto]
  -V, --sot=0             max. simultaneous running Out-of-FOV threads [0=auto]

Transformation computation:
  -r, --trkfile=STRING    use tracking information file (*.trk)
  -C, --ccfile=STRING     use cross calibration transformation file (*.cc)
  -M, --tfile=STRING      filename of the output transformation file (*.tfm)
  -b, --refbody=1         number of body within tracking data to take as reference body
  -S, --stime=0           time within tracking file to take as the start time
  -R, --rtime=-1          time within tracking file to take as the reference time

Miscellaneous:
  -h, --help[=yes/no]     display this help message
  -v, --verbose[=yes/no]  be a bit more verbose in commandline mode
  -d, --debug=0           set debug level [0=disabled]
  -q, --quiet=0           be quiet while processing
```

A.3 Graphical User Interface

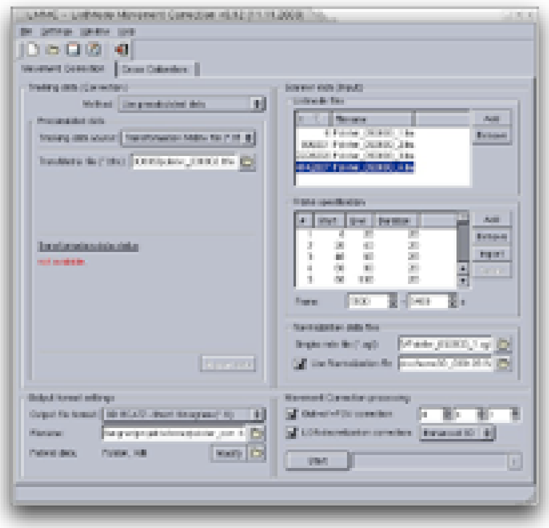


Figure A.1: Main Window of `lmmc`

The graphical user interface is started by calling `lmmc` without any command-line options. It is divided into two main working areas. The first one consists of graphical elements to perform movement corrections, the second one is responsible for administrating the cross-calibration (cf. section 2.1.1). The main window is therefore separated with *tab widgets*, so that a clear distinction is possible. Figure A.1 shows a snapshot of the main window

The following sections will concentrate on describing all main components of the graphical user interface together with their function. Snapshots are going to illustrate the components wherever applicable.

A.3.1 Main Components

A close look at the top elements of the GUI shows three main components of the application window. Figure A.2 shows the main menu at the topmost position of the window which can be

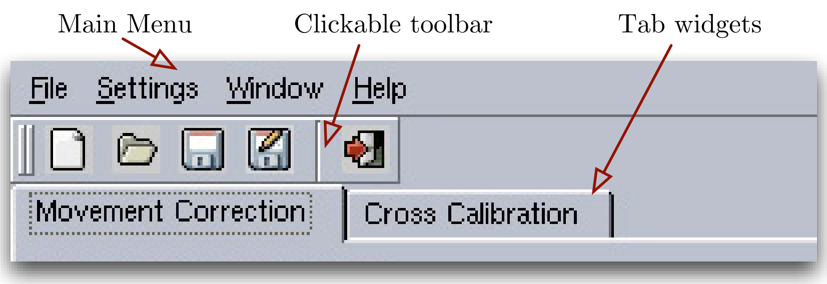
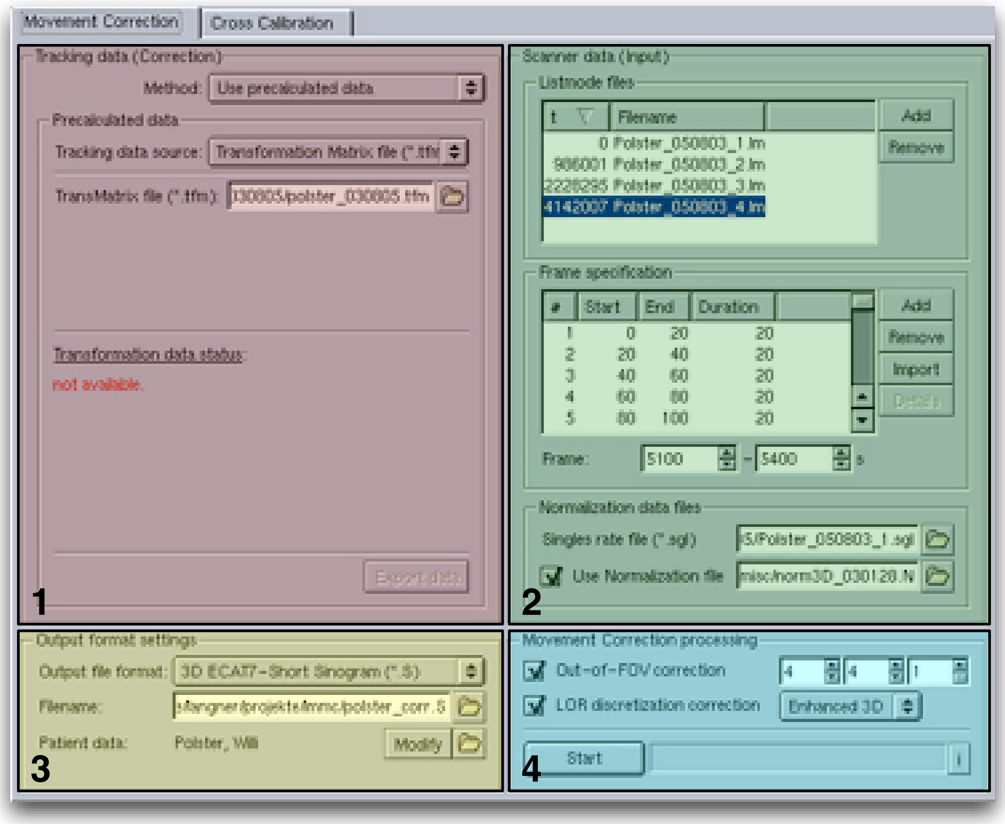


Figure A.2: The application window of `lmmc` has three main components. A main application menu, a click-able toolbar and two tab widgets for either the movement correction or cross calibration.

used to load and save `lmmc` based study files (`*.lms`), as well as to modify the default settings of the application. In addition, most of the functionality encapsulated within the main menu is also available through a separate toolbar underneath the menu. Finally, the tab widgets separating the different sections of the application are placed below the toolbar and can be used to switch between the GUI elements of the movement correction and the cross-calibration.

A.3.2 Movement Correction Components

The graphical components, which are responsible for setting up the movement correction are divided into four different graphical areas, as shown in figure A.3.



Tracking data (Correction)

Method: Use precalculated data

Precalculated data

Tracking data source: Transformation Matrix file (*.tfm

TransMatrix file (*.tfm): 030803/polster_030805.tfm

Transformation data status:

not available.

Export data

Scanner data (Input)

Listmode files

t	Filename
0	Polster_050803_1.lm
906001	Polster_050803_2.lm
2228295	Polster_050803_3.lm
4142007	Polster_050803_4.lm

Add Remove

Frame specification

#	Start	End	Duration
1	0	20	20
2	20	40	20
3	40	60	20
4	60	80	20
5	80	100	20

Add Remove Import Details

Frame: 5100 - 5400 s

Normalization data files

Singles rate file (*.sgl) 5/Polster_050803_1.sgl

[✓] Use Normalization file misc/norm3D_030128.N

Output format settings

Output file format: 3D ECAT7-Short Sinogram (*.S)

Filename: :/langner/projekte/lmmc/polster_corr.S

Patient data: Polster, Willi

Modify

Movement Correction processing

[✓] Out-of-FOV correction 4 4 1

[✓] LOR discretization correction Enhanced 3D

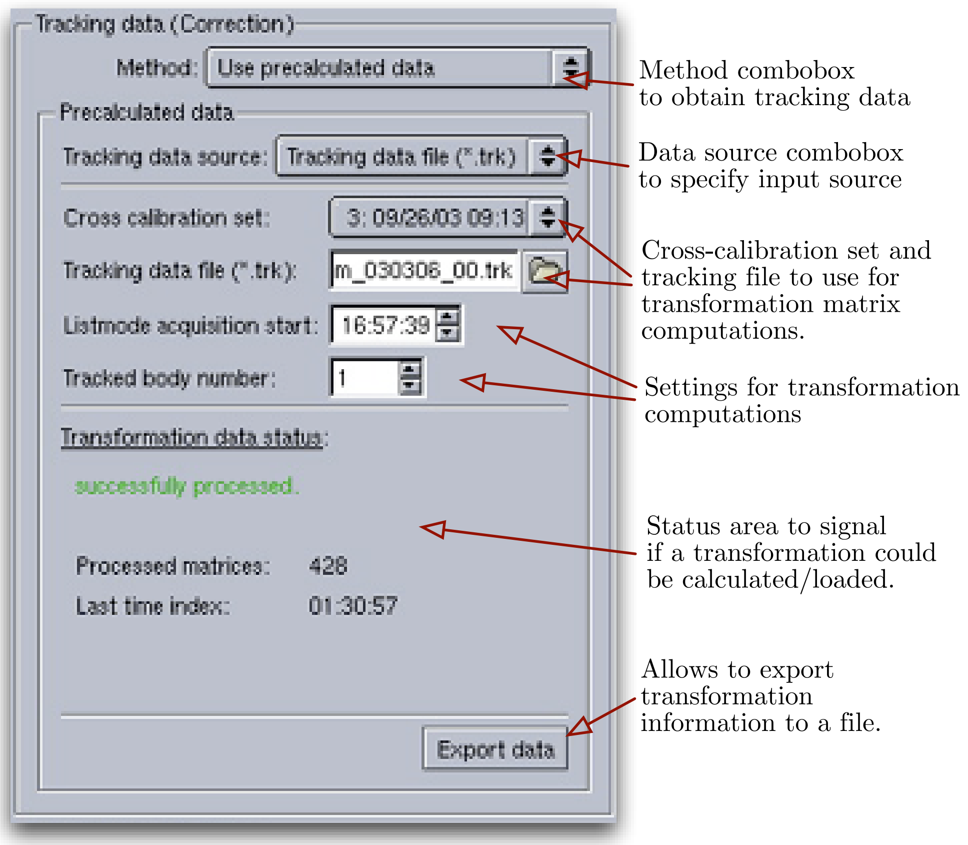
Start

Figure A.3: The GUI components of the movement correction are split into four areas; the succession 1-4 corresponds to the steps which have to be performed by the medical technician. In area 1 the motion tracking relevant data are specified. Area 2 is used to specify the acquisition data of the PET scanner. Area 3 specifies the desired output format and destination. Area 4 is used to set up computational options and to start the computations.

Each of these areas represent a single step in the preparation for processing the movement correction. They have been placed within the GUI such that the user has to perform four different steps to start the processing of the movement correction.

A.3.2.1 Tracking Data GUI Components (1)

Tracking data information can be specified with different methods within `lmmc`. Figure A.4 shows one of these methods where the tracking data GUI elements are set to obtain the motion tracking information from *precalculated data*. This means, that either an already existing transformation matrix file (`*.tfm`) can be specified or a raw motion tracking data file (`*.trk`) to calculate the necessary transformation matrices prior to the movement correction. As soon as the movement


Tracking data (Correction)

Method: Use precalculated data

Precalculated data

Tracking data source: Tracking data file (*.trk)

Cross calibration set: 3: 09/26/03 09:13

Tracking data file (*.trk): m_030306_00.trk

Listmode acquisition start: 16:57:39

Tracked body number: 1

Transformation data status:

successfully processed.

Processed matrices: 428

Last time index: 01:30:57

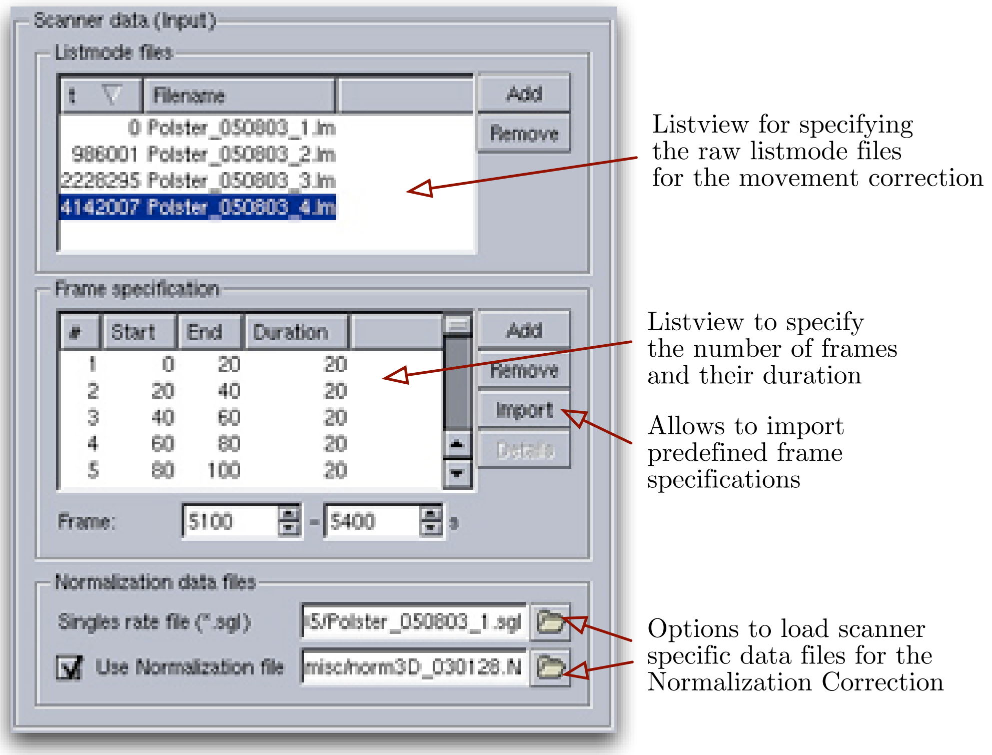
Export data
Method combobox to obtain tracking data
Data source combobox to specify input source
Cross-calibration set and tracking file to use for transformation matrix computations.
Settings for transformation computations
Status area to signal if a transformation could be calculated/loaded.
Allows to export transformation information to a file.

Figure A.4: The tracking data GUI elements consist of several comboboxes and filename specifying text fields. The use of the *precalculated data* methods is illustrated where a motion tracking information file (`*.trk`) is specified. The data contained in this file is then used together with the specified options, e.g. *cross calibration set*, to compute the required transformation information.

correction is initiated, the application computes the necessary transformation matrices from the provided motion tracking data, as discussed in section 2.3. Afterwards, the computed tracking information can be exported to a transformation matrix file which allows to use it for later runs by selecting the `(*.tfm)` mode in the *tracking data source* combobox.

A.3.2.2 PET Data GUI Components (2)

The right top area of the application can be used to specify the *listmode* files to be processed Within a listview, an unlimited number of files can be specified where `lmmc` does maintain the correct order of the listed files by itself. In addition, multiframe studies can be specified by adding the frame boundaries manually or by importing it from a predefined frame specification file `(*.frm)`. `lmmc` does ensure the correct sorting within the listview - starting at zero time and stepwise increasing until the last specified frame. The bottom elements of the PET data specification area are used to specify the *singles rate file* and the *normalization file* provided


Scanner data (Input)
Listmode files
Add
Remove
Frame specification
Add
Remove
Import
Details
Normalization data files
Listview for specifying the raw listmode files for the movement correction
Listview to specify the number of frames and their duration
Allows to import predefined frame specifications
Options to load scanner specific data files for the the Normalization Correction

t	Filename
0	Polster_050803_1.lm
986001	Polster_050803_2.lm
2228295	Polster_050803_3.lm
4142007	Polster_050803_4.lm

#	Start	End	Duration
1	0	20	20
2	20	40	20
3	40	60	20
4	60	80	20
5	80	100	20

Frame: 5100 - 5400 s

Singles rate file (*.sgl) i5/Polster_050803_1.sgl

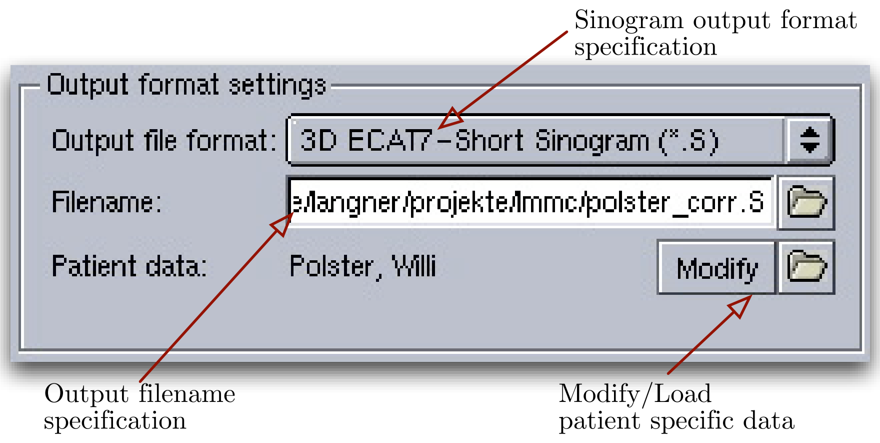
[✓] Use Normalization file misc/norm3D_030128.N

Figure A.5: Graphical elements for specifying the scanner dependent input data are combined within an own graphical area of the GUI. The top listview is used to specify the listmode data obtained from the tomographic acquisition. In a second listview the specification of the desired frames can be either manually specified or loaded from a predefined frame specification file (`*.frm`). Furthermore, the bottom text fields are used to specify the *singles rate file* and the *normalization file* to perform a *Normalization Correction* (see section 2.2.1.1).

by the PET scanner. Both are required to calculate the normalization matrix that is used to perform the *Normalization Correction* discussed in section 2.2.1.1.

A.3.2.3 Output Format GUI Components (3)

The format and filename in which the movement corrected LORs are saved can be specified with the graphical elements shown in figure A.6. Currently only the ECAT7 sinogram format is supported with three different data types (`byte/short/float`). In addition, an *automode* is provided which will automatically determine the correct data type prior to sinogram sorting. As most of the general output formats support the inclusion of patient specific data within so called *headers*, an additional component to either load patient data from other files or to specify the data manually is provided at the bottom of the output format group.

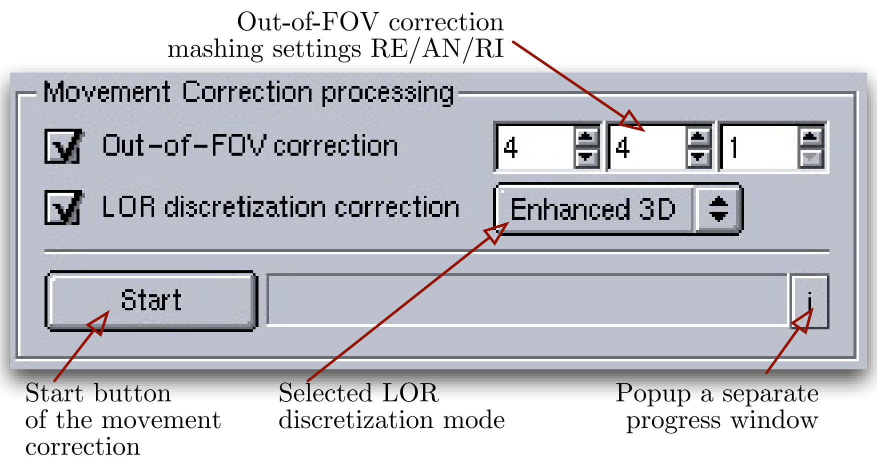

Sinogram output format specification
Output filename specification
Modify/Load patient specific data
Output format settings
Modify

Output file format: 3D ECAT7 - Short Sinogram (*.S)
Filename: e/langner/projekte/lmmc/polster_corr.S
Patient data: Polster, Willi

Figure A.6: The output format and filename can be specified within an own graphical component. A combobox allows to specify the format, which is currently limited to ECAT7 formats only. Patient specific header data can be specified either by manually modifying the current data set or by loading patient data from other file sources, such as e.g. *transmission files.*

A.3.2.4 Processing GUI Components (4)

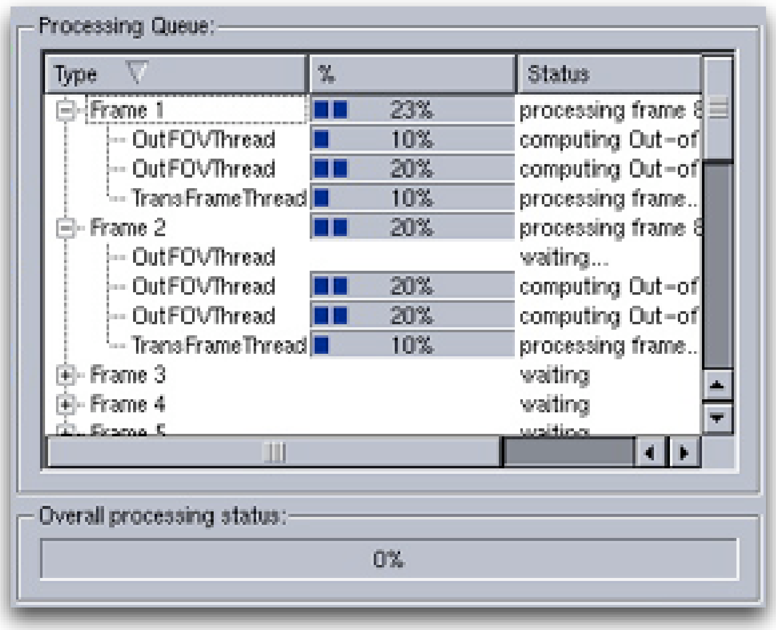
The graphical elements to start the movement correction is shown in figure A.7. Numerical boxes


Out-of-FOV correction mashing settings RE/AN/RI
Movement Correction processing
Start
i
Start button of the movement correction
Selected LOR discretization mode
Popup a separate progress window

☑ Out-of-FOV correction 4 4 1
☑ LOR discretization correction Enhanced 3D

Figure A.7: Graphical elements responsible for specifying the computational settings for the Out-of-FOV and LOR discretization correction.

are used to specify the mashing settings for the Out-of-FOV correction. And the desired LOR discretization correction mode (*normal/enhanced3D*) can be set with the middlemost combobox. By pressing the *Start* push button, the application checks if all necessary information needed to starting the movement correction was provided. If this is found o be the case, all main GUI elements are ghosted while `lmmc` is processing. Furthermore, a small button at the bottom right corner of the component allows to open a progress window in which the progress of a computation is illustrated as shown in figure A.8. This window shows the current status and


Processing Queue:

Type	%	Status
Frame 1	23%	processing frame 8
OutFOVThread	10%	computing Out-of
OutFOVThread	20%	computing Out-of
TransFrameThread	10%	processing frame..
Frame 2	20%	processing frame 8
OutFOVThread		waiting...
OutFOVThread	20%	computing Out-of
OutFOVThread	20%	computing Out-of
TransFrameThread	10%	processing frame..
Frame 3		waiting
Frame 4		waiting
Frame 5		waiting

Overall processing status:

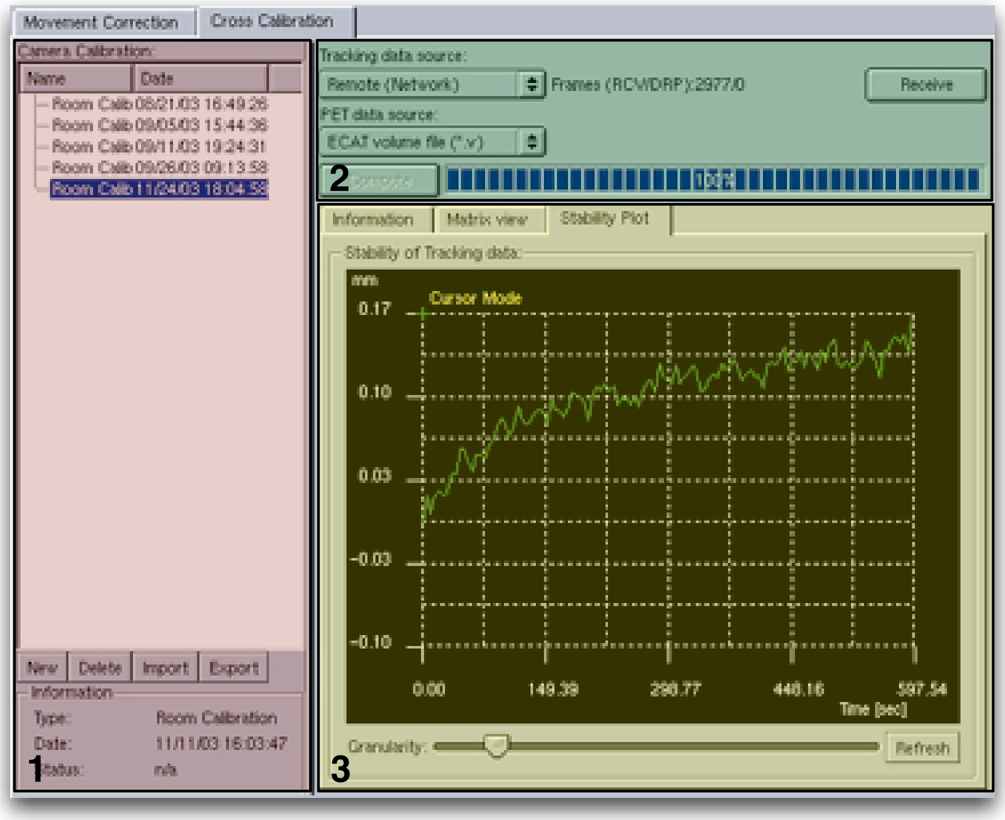
0%

Figure A.8: Progress window of `lmmc`. It shows the current status and percentile progress of each involved thread of the movement correction computations.

percentile progress of each thread, respectively *frame*. It allows the user of `lmmc` to track the progress of the computations as well as breaking the computations by pushing the push button in figure A.7 a second time.

A.3.3 Cross-Calibration Components

The cross-calibration tab has been split into three different areas, cf. figure A.9. In area one



Room Calib 08/21/03 16:49:26
Room Calib 09/05/03 15:44:38
Room Calib 09/11/03 19:24:31
Room Calib 09/26/03 09:13:58
Room Calib 11/24/03 18:04:58

Type: Room Calibration
Date: 11/11/03 16:03:47
Status: n/a

Tracking data source: Remote (Network) Frames (RCV/DRP):2977/0
PET data source: ECAT volume file (*.v)
100%

Figure A.9: View at the cross-calibration GUI components of `lmmc`. Within three main areas the user can define several calibration specific settings. In area one a listview of the stored calibration data is shown together with toolbuttons to import/export the data. Area two can be used to specify the data source from which the tracking and PET scanner cross-calibration dependent data can be obtained. Within area three statistical GUI elements are combined into different separating tab widgets.

elements are provided to maintain a database of performed room and body calibrations as well as toolbuttons to import and export calibration specific data to standardized files. Area two allows to specify the data source of the calibration data for both, the motion tracking system and the PET scanner, as well as to start the cross-calibration computation. Area number three is used to output statistical data concerning computation and results of the cross-calibration. Within three separate tab widgets, all computational data is presented to the scientist administrating the cross-calibration.

A.3.3.1 Data Storage Components (1)

Each performed cross-calibration is stored within an application wide configuration file. These calibrations are then loaded upon application start and added to the listview, as shown in figure A.10. Toolbuttons at the bottom of the listview allow to add and delete calibrations from the

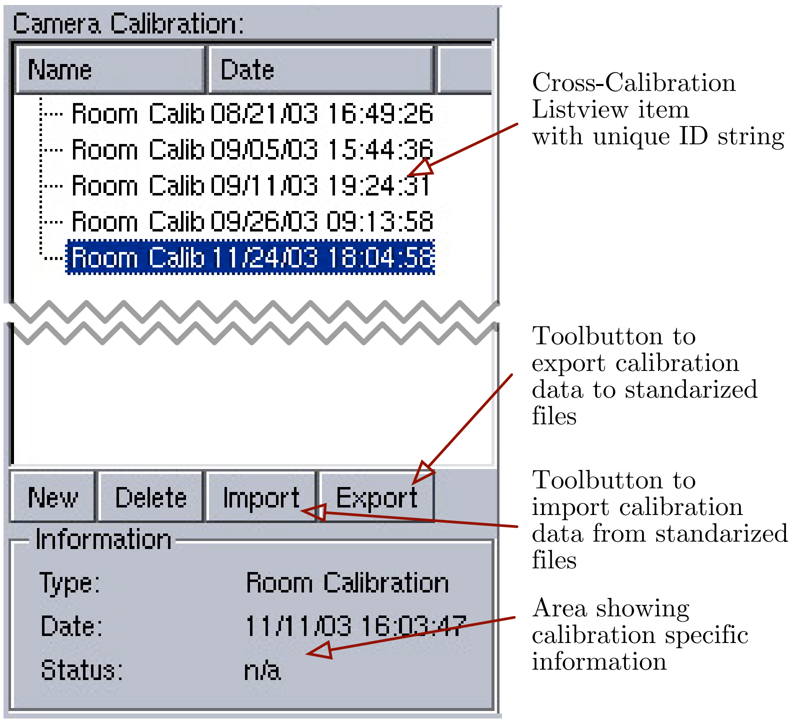

Camera Calibration:
Name
Date
Room Calib 08/21/03 16:49:26
Room Calib 09/05/03 15:44:36
Room Calib 09/11/03 19:24:31
Room Calib 09/26/03 09:13:58
Room Calib 11/24/03 18:04:58
New
Delete
Import
Export
Information
Type: Room Calibration
Date: 11/11/03 16:03:47
Status: n/a
Cross-Calibration Listview item with unique ID string
Toolbutton to export calibration data to standarized files
Toolbutton to import calibration data from standarized files
Area showing calibration specific information

Figure A.10: A separate GUI component is used to display the currently available cross-calibrations. Together with toolbuttons to add and delete a specific calibration from the list, import and export buttons allow to exchange information with other applications. A status information group always displays the calibration specific information.

maintained *calibration data storage*. Import and export of a calibration is performed by using the named toolbuttons, which requests the file name and directory for this operation. In addition to the listview a textual information group at the bottom of the component shows the relevant information of the currently selected calibration.

A.3.3.2 Data Source Components (2)

The data from which the cross-calibration is computed can be specified by using the graphical elements shown in figure A.11. Two comboboxes are used to specify the data sources of the

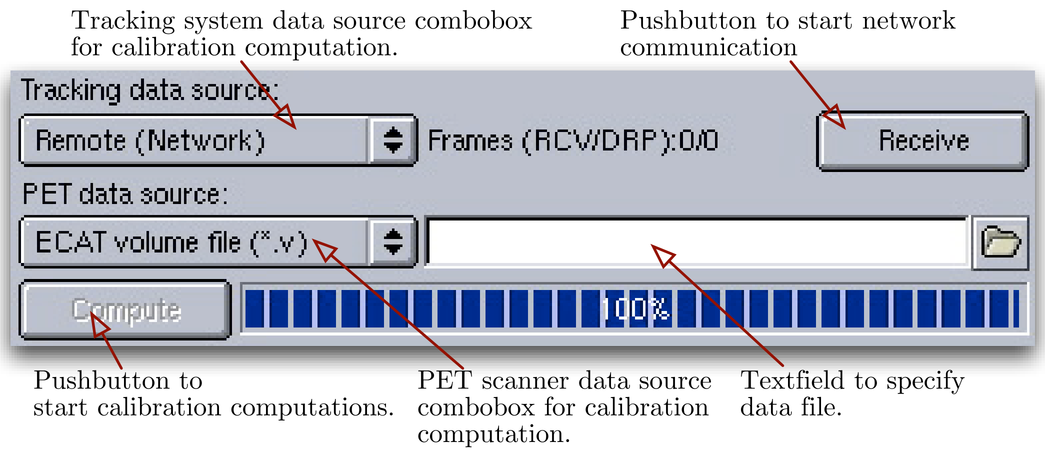

Tracking system data source combobox for calibration computation.
Pushbutton to start network communication
Tracking data source:
Remote (Network)
Frames (RCV/DRP):0/0
Receive
PET data source:
ECAT volume file (*.v)
Compute
100%
Pushbutton to start calibration computations.
PET scanner data source combobox for calibration computation.
Textfield to specify data file.

Figure A.11: Within the upper GUI elements of the showed cross-calibration component, the different data source can be specified from which the computation will retrieve their data prior to the calculations.

tracking system and PET scanner. For specification of the tracking data source the user can select *remote (network)* as the data source. The *receive* pushbutton is used to start a network communication to the motion tracking system and directly receives the tracking information required to compute the cross-calibration data.

A.3.3.3 Statistical Components (3)

During the tracking data acquisition for the cross-calibration computations, statistical data is displayed in the component shown in figure A.12. The component contains three areas that are

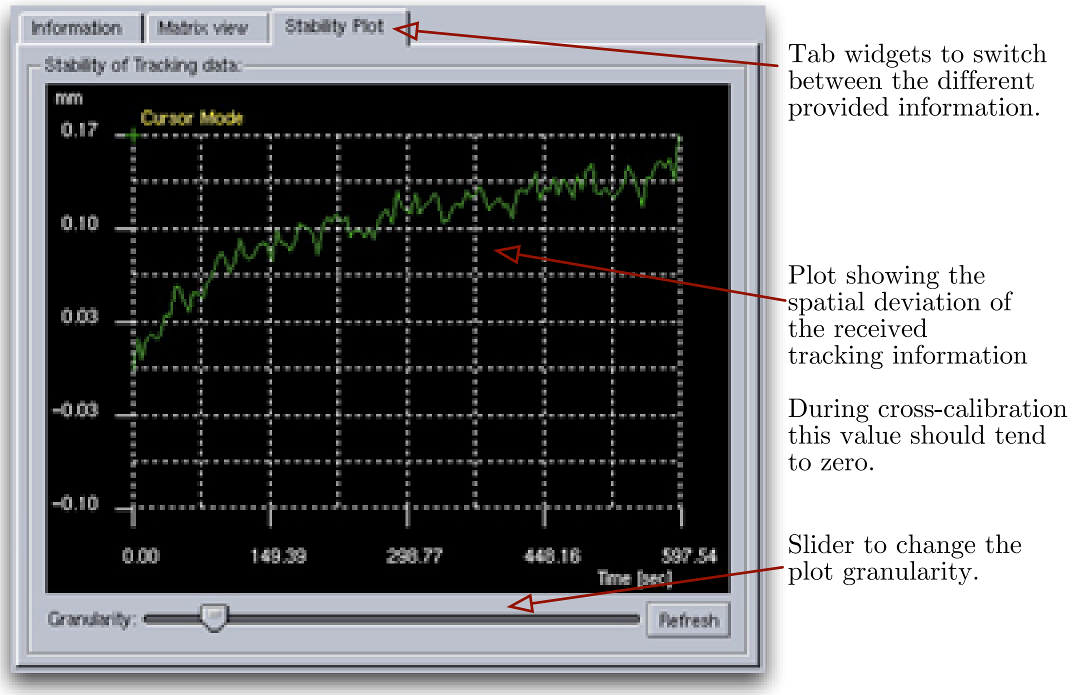
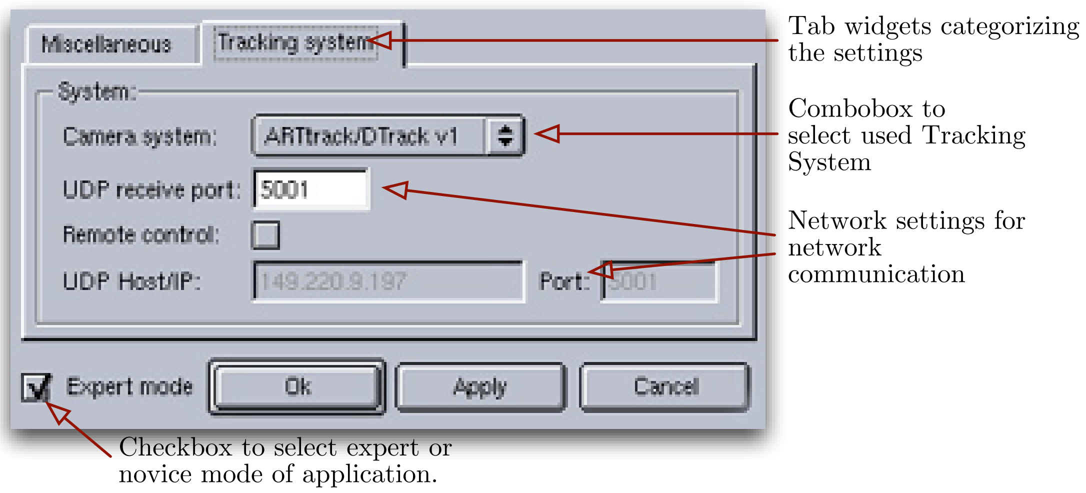

Information
Matrix view
Stability Plot
Stability of Tracking data:
mm
Cursor Mode
0.17
0.10
0.03
-0.03
-0.10
0.00
149.39
298.77
448.16
597.54
Time [sec]
Granularity:
Refresh
Tab widgets to switch between the different provided information.
Plot showing the spatial deviation of the received tracking information
During cross-calibration this value should tend to zero.
Slider to change the plot granularity.

Figure A.12: A separate component within the cross-calibration tab is used to provide statistical GUI elements. In addition to a general *information* field and a view of the calculated cross-calibration matrix, an interactive 2D plot is used to illustrate the spatial deviation of the motion information, thus allowing to draw conclusions on the stability of the received signal.

separates with tab widgets. Each of these widgets can be used to get additional information either during data acquisition or after having computation of the cross-calibration. In figure A.12, the implemented plot widget is shown that displays the stability of the motion tracking information received during the network based data acquisition. It shows the relative deviation of the tracking information since start of the data acquisition. However, in case of a motion tracking data acquisition for computing the cross-calibration this deviation should not vary to much. The slider widget at the bottom of the component is used to change the granularity of the time resolution of the plot.

A.3.4 Application Settings Components

Figure A.13 shows the application wide settings window that is opened from the *settings* item of the application's main menu. Two tab widgets provide the user with several graphical elements to change settings, which are either application or user dependent. The figure shows the active


Miscellaneous
Tracking system
Ok
Apply
Cancel
Tab widgets categorizing the settings
Combobox to select used Tracking System
Network settings for network communication
Checkbox to select expert or novice mode of application.

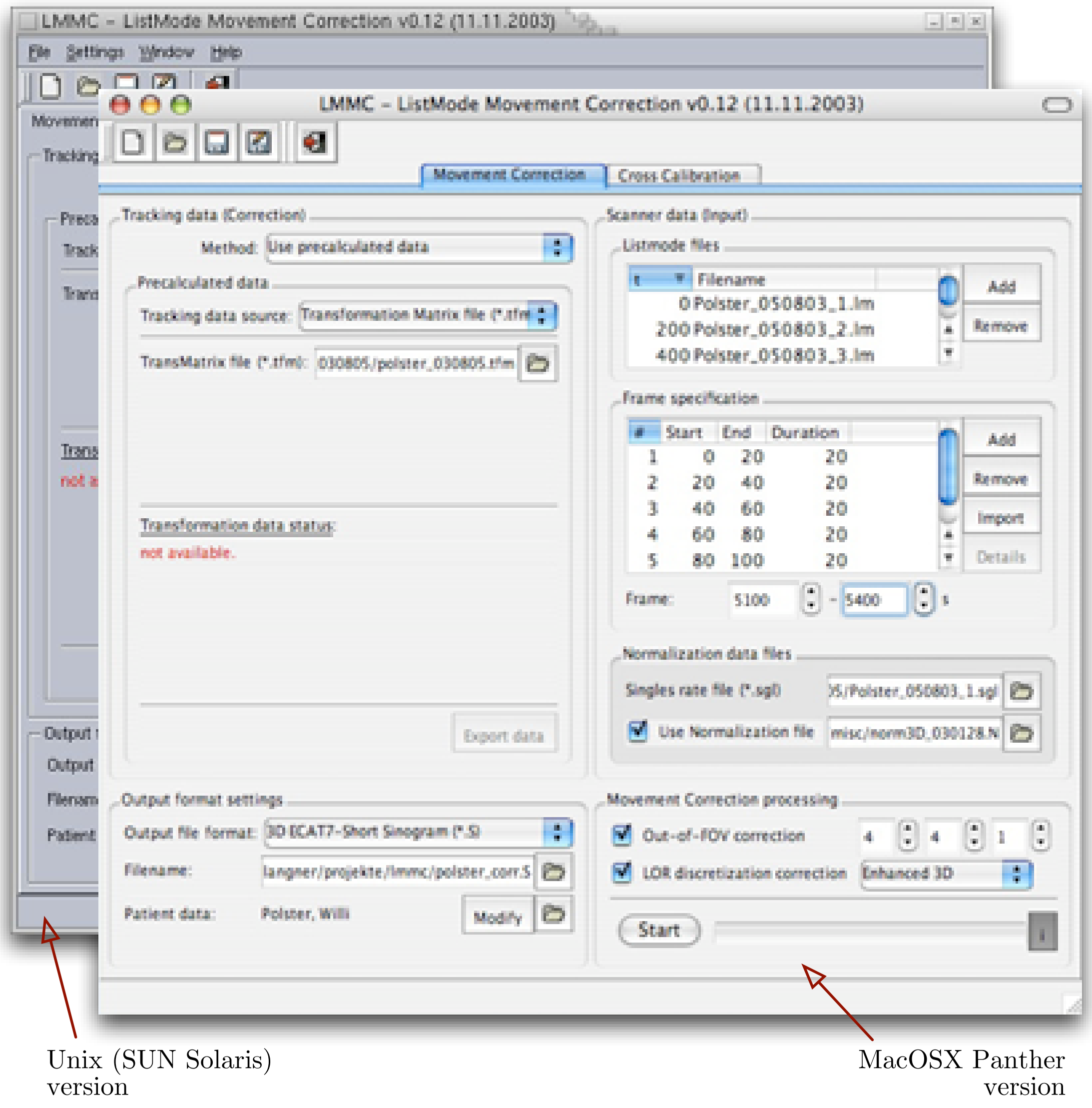
System:
Camera system: ARTtrack/DTrack v1
UDP receive port: 5001
Remote control: ☐
UDP Host/IP: 149.220.9.197 Port: 5001
☑ Expert mode

Figure A.13: The settings window dialog provided by `lmmc`. It controls application and user wide settings which are saved to different XML files upon closing the dialog. Tab widgets distinguish between different settings. The *tracking system* tab is shown that allows to set the application wide network settings for retrieving the motion tracking information. In addition, an *expert mode* checkbox allows to show and hide different GUI elements of the main window.

tracking system tab with elements to set the type of the tracking system as well as the settings for remote communication. A checkbox is included to switch between an *expert* and *novice* mode. In contrast to the expert mode, the novice mode hides several GUI elements of the whole application which are not necessary for routine operation; this can simplify the usage of the GUI for many users who are not so familiar with all the available functionality of the application.

A.4 Cross-Platform GUI Layout

As a cross-platform application, the source code of `lmmc` can be compiled *out of the box* on many platforms (see section A.1). This implies the *Look & Feel* of the graphical user interface. Therefore, figure A.14 shows `lmmc` running on two different operating system with having the same graphical layout, thus providing the same graphical elements and functionality.



LMMC – ListMode Movement Correction v0.12 (11.11.2003)

Tracking data (Correction)

Method: Use precalculated data

Precalculated data

Tracking data source: Transformation Matrix file (*.tfm)

TransMatrix file (*.tfm): 030805/polster_030805.tfm

Transformation data status:
not available.

Scanner data (Input)

Listmode files

t	Filename
0	Polster_050803_1.lm
200	Polster_050803_2.lm
400	Polster_050803_3.lm

Frame specification

#	Start	End	Duration
1	0	20	20
2	20	40	20
3	40	60	20
4	60	80	20
5	80	100	20

Frame: 5100 - 5400 s

Normalization data files

Singles rate file (*.sgl))5/Polster_050803_1.sgl

[✓] Use Normalization file misc/norm3D_030128.N

Output format settings

Output file format: 3D ECAT7-Short Sinogram (*.S)

Filename: langner/projekte/lmmc/polster_corr.S

Patient data: Polster, Willi

Movement Correction processing

[✓] Out-of-FOV correction 4 4 1

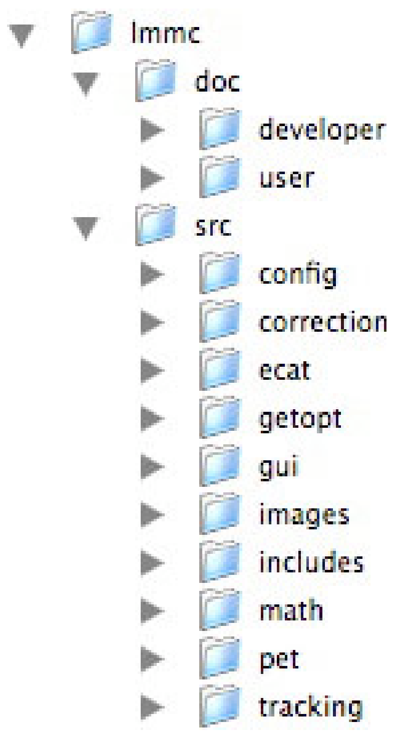
[✓] LOR discretization correction Enhanced 3D

Figure A.14: Two snapshots of the main view of the `lmmc` application. Usage of the cross-platform Qt framework allowed not only to maintain a single source code base for different platforms, but also to design the GUI layout in a platform independent way. Here the MacOSX^{TM}compiled version is shown together with the version compiled under Solaris^{TM}. The layout of both GUI look and act equally so that the same *Look & Feel* is ensured over all supported platforms (cf. section A.1)

Appendix B

Source Code Structure

The following list gives an overview of the chosen directory structure for organizing the source code within the development environment. Aside from the C^{++} source code of the classes, which reside in the `src` directory, a documentation directory `doc` includes user and developer specific documentation. The developer specific documentation has been automatically generated by a source code processing tool called *doxygen*[1].



lmmc	Main directory
doc	Documentation
developer	Developer documentation
user	User documentation
src	Main sourcecode
config	Configuration Management Classes
correction	Movement Correction Classes
ecat	ECAT File I/O Classes
getopt	Command-Line option Classes
gui	Graphical User Interface Classes
images	Images for GUI
includes	Includes with parameterized data
math	Mathematical Interface Classes
pet	PET dependant Calibration Classes
tracking	Tracking System Calibration Classes

The following sections will list all main directories of the source code directory, listing each source code file together with its classes and a short description.

[1]cf. http://www.doxygen.org/

B.1 C^{++} classes in module - `src`

These are the main C^{++} class files located within the main module `src`.

Files	*Classes*	*Description*
main.cpp	-	Entry point and preprocessing of command line parameters.
CCmdLineStart.cpp CCmdLineStart.h	`CCmdListStart`	Main command-line parsing class of the application.
CDebug.cpp CDebug.h	`CDebug`	Debugging class allowing to output runtime information.

B.2 C^{++} classes in module - `config`

The following classes manage the configuration data. This is done in separate XML trees providing import and export functions. The configuration is split into a user specific configuration allowing each user to have his own environment.

Files	*Classes*	*Description*
CLMMCConfig.cpp CLMMCConfig.h	`CLMMCConfig`	Main configuration class managing the sub-configuration classes.
CCalibrationConfig.cpp CCalibrationConfig.h	`CCalibrationConfig`	Calibration specific configuration data.
CCorrectionConfig.cpp CCorrectionConfig.h	`CCorrectionConfig`	Configuration data depending on the movement correction.
CTrackingConfig.cpp CTrackingConfig.h	`CTrackingConfig`	Tracking system specific configuration.
CMiscUserConfig.cpp CMiscUserConfig.h	`CMiscUserConfig`	Class for managing the individual application settings for an user.

B.3 C++ classes in module - `correction`

The classes implementing the movement correction algorithms are combined in an own `correction` module in the source code hierarchy. This includes the intrinsic LOR processing classes to process the movement corrections for any LOR.

Files	*Classes*	*Description*
CConvTablesFacade.cpp CConvTablesFacade.h	`CConvTablesFacade`	Class that accommodates calculated data matrices.
CLORProcessor.cpp CLORProcessor.h	`CLORProcessor`	The main movement correction processing class.
CPreProcessThread.cpp CPreProcessThread.h	`CPreProcessThread`	Thread class that is used to preprocess data matrices prior to the correction.
CThread.cpp CThread.h	`CThread`	Base class of any thread within lmmc.
CThreadDispatcher.cpp CThreadDispatcher.h	`CThreadDispatcher`	Main dispatcher for distributing the computations.
CListModeFile.cpp CListModeFile.h	`CListModeFile`	Entity class for a single listmode file.
CListModeFilePool.cpp CListModeFilePool.h	`CListModeFilePool`	Container class for managing several listmode files at once.
CDeadTimeCorrMatrix.cpp CDeadTimeCorrMatrix.h	`CDeadTimeCorrMatrix`	Storage class for the dead time matrix data.
CNormMatrix.cpp CNormMatrix.h	`CNormMatrix`	Storage class for normalization matrix data.
CSinglesFile.cpp CSinglesFile.h	`CSinglesFile`	Entity class for single singles file.
CSinoBuilderThread.cpp CSinoBuilderThread.h	`CSinoBuilderThread`	Thread class for sorting the data in the final sinogram.
CSinoFrame.cpp CSinoFrame.h	`CSinoFrame`	Entity class representing a single frame.
CSinoFramePool.cpp CSinoFramePool.h	`CSinoFramePool`	Container class managing several frames at once.
CSinoMatrix.cpp CSinoMatrix.h	`CSinoMatrix`	Storage class representing a single matrix within a sinogram.
CSinoSegment.cpp CSinoSegment.h	`CSinoSegment`	Storage class managing a single segment of a sinogram.
COutFOVCorrThread.cpp COutFOVCorrThread.h	`COutFOVCorrThread`	Thread class managing all Out-of-FOV corrections of a frame.
COutFOVMatrix.cpp COutFOVMatrix.h	`COutFOVMatrix`	Storage class managing a single Out-of-FOV matrix.
CTransMatrix.cpp CTransMatrix.h	`CTransMatrix`	Storage class managing a single transformation information.
CTransMatrixPool.cpp CTransMatrixPool.h	`CTransMatrixPool`	Contain class managing several transformation information at once.

B.4 C++ classes in module - `ecat`

The output file I/O functionality has been implemented in an encapsulated library interface. Still part of the development sources of this movement correction application, they are located within the `ecat` module.

Files	*Classes*	*Description*
CECATDirectory.cpp CECATDirectory.h	`CECATDirectory`	Class representing the matrix directory within an ECAT file.
CECATDirectoryItem.cpp CECATDirectoryItem.h	`CECATDirectoryItem`	Entity class representing a single matrix entry within the directory.
CECATFile.cpp CECATFile.h	`CECATFile`	Main ECAT file interface class.
CECATMainHeader.h	`CECATMainHeader`	Template class representing a version independent main header of an ECAT file.
CECATSubHeader.h	`CECATSubHeader`	Template class representing a version independent sub header of an ECAT file.
CECAT7MainHeader.cpp CECAT7MainHeader.h	`CECAT7MainHeader`	Data class representing a main header version 7.
CECAT7SubHeaderAttenCorr.cpp CECAT7SubHeaderAttenCorr.h	`CECAT7SubHeaderAttenC`	Data class representing an attenuation correction sub header version 7.
CECAT7SubHeaderImage.cpp CECAT7SubHeaderImage.h	`CECAT7SubHeaderImage`	Data class representing an Image sub header version 7.
CECAT7SubHeaderNorm.cpp CECAT7SubHeaderNorm.h	`CECAT7SubHeaderNorm`	Data class representing a 2D-Normalization sub header version 7.
CECAT7SubHeaderNorm3D.cpp CECAT7SubHeaderNorm3D.h	`CECAT7SubHeaderNorm3D`	Data class representing a 3D-Normalization sub header version 7.
CECAT7SubHeaderPolarMap.cpp CECAT7SubHeaderPolarMap.h	`CECAT7SubHeaderPolarM`	Data class representing a PolarMap sub header version 7.
CECAT7SubHeaderScan.cpp CECAT7SubHeaderScan.h	`CECAT7SubHeaderScan`	Data class representing a 2D-Sinogram sub header version 7.
CECAT7SubHeaderScan3D.cpp CECAT7SubHeaderScan3D.h	`CECAT7SubHeaderScan3D`	Data class representing a 3D-Sinogram sub header version 7.

B.5 C++ classes in module - `getopt`

Command-Line options are generally managed with a so called `getopt()` C-function. As the implemented application is using C++ as the main programming language an own, object-oriented variant of the `getopt` functionality has been implemented in a separate module.

Files	*Classes*	*Description*
CGetOpt.cpp CGetOpt.h	`CGetOpt`	Command-Line options management class.

B.6 C^{++} classes in module - `gui`

All class files of the graphical user interface are encapsulated in the **`gui`** module. Depending on the functionality of each class they are separated in own sub-modules. Starting at the main GUI classes the following list represents the graphical entry point classes of the application.

Files	*Classes*	*Description*
CMainMenuBar.cpp CMainMenuBar.h	`CMainMenuBar`	GUI class managing the menu bar of the application.
CMainToolBar.cpp CMainToolBar.h	`CMainToolBar`	GUI class managing the tool bar of the application.
CMainWidget.cpp CMainWidget.h	`CMainWidget`	GUI class representing the main widget.
CMainWindow.cpp CMainWindow.h	`CMainWindow`	GUI class representing the main window.

B.6.1 C^{++} classes in submodule - `gui/calibration`

The following list contains all GUI classes used by the calibration functionality of the application.

Files	*Classes*	*Description*
CCalibrateWidget.cpp CCalibrateWidget.h	`CCalibrateWidget`	Main GUI tabwidget of calibration.
CBodyCalibrationItem.h	`CBodyCalibrationItem`	Body calibration QListviewItem.
CRoomCalibrationItem.h	`CRoomCalibrationItem`	Room calibration QListviewItem.

B.6.2 C^{++} classes in submodule - `gui/plot`

Some elementary 2D graphing classes have been implemented to allow the drawing of statistical plots.

Files	*Classes*	*Description*
CPlot2DWidget.cpp CPlot2DWidget.h	`CPlot2DWidget`	Main 2D based plotting class.
CPlotBase.cpp CPlotBase.h	`CPlotBase`	Base class for the plotting functionality.
CPlotCursor.cpp CPlotCursor.h	`CPlotCursor`	Class representing a cursor in a single plotting instance.
CPlotTrace.cpp CPlotTrace.h	`CPlotTrace`	Class allowing to attach *traces* on a 2D plot object.

B.6.3 C++ classes in submodule - `gui/correction`

All graphical elements of the movement correction functionality have been encapsulated in a separate directory. These classes include widgets and dialog classes to provide a graphical user interface to control the movement correction.

Files	*Classes*	*Description*
CMoveCorrWidget.cpp CMoveCorrWidget.h	`CMoveCorrWidget`	Main GUI tabwidget of the movement correction.
COutputFormatCorrWidget.cpp COutputFormatCorrWidget.h	`COutputFormatCWidget`	Subwidget managing settings on the output formats.
CProcessingCorrWidget.cpp CProcessingCorrWidget.h	`CProcessingCorrWidget`	Subwidget managing settings on the movement correction.
CScannerDataCorrWidget.cpp CScannerDataCorrWidget.h	`CScannerDataCorrWidget`	Subwidget managing settings of the input PET scanner data.
CListModeFileItem.h	`CListModeFileItem`	QListviewItem inherited class storing a listmode file.
CSinoFrameItem.h	`CSinoFrameItem`	QListviewItem inherited class storing a single sinogram frame.
CTrackingDataCorrWidget.cpp CTrackingDataCorrWidget.h	`CTrackingDataCWidget`	Subwidget managing settings of the input tracking system data.
CProgressInfoDialog.cpp CProgressInfoDialog.h	`CProgressInfoDialog`	Dialog class managing the progress info dialog.
CProgressListItem.h	`CProgressListItem`	Base class for other ProgressItem classes.
CFrameProgressItem.h	`CFrameProgressItem`	QListviewItem inherited class to signal a frame based progress.
CThreadProgressItem.h	`CThreadProgressItem`	QListviewItem inherited class to signal a thread based progress.

B.6.4 C++ classes within submodule - `gui/settings`

The preferences of the application are by XML managing configuration classes but are also manageable through a separate dialog in the graphical user interface.

Files	*Classes*	*Description*
CSettingsDialog.cpp CSettingsDialog.h	`CSettingsDialog`	Class providing a separate dialog for configuring the preferences.

B.7 C^{++} classes in module - `math`

The mathematical functionality has been implemented in these *template classes*:

Files	*Classes*	*Description*
C2DMatrix.h	`C2DMatrix`	Direct access 2D-matrix template class.
C3DMatrix.h	`C3DMatrix`	Direct access 3D-matrix template class.
C4DMatrix.h	`C4DMatrix`	Direct access 4D-matrix template class.
CVector.h	`CVector`	Indirect access matrix template class representing a 2D vector.
CVectorArray.h	`CVectorArray`	Indirect access matrix template class representing an array of 2D vector.
CVectorMatrix.h	`CVectorMatrix`	Indirect access matrix template class representing an array of 2D vector of the same length.

B.8 C^{++} classes in module - `tracking`

The coordinate calibration management of the application has been separated within own management classes. These classes manage the cross-calibration of the PET and tracking system and the tracking system's own body and room calibration.

Files	*Classes*	*Description*
C3DObject.h	`C3DObject`	Base class for all tracking classes managing multidimensional objects.
C6DBody.cpp C6DBody.h	`C6DBody`	Class representing a single 6DBody tracking information.
C3DMarker.cpp C3DMarker.h	`C3DMarker`	Class representing a single 3DMarker tracking information.
CBodyCalibration.cpp CBodyCalibration.h	`CBodyCalibration`	Body calibration management class.
CRoomCalibration.cpp CRoomCalibration.h	`CRoomCalibration`	Room calibration management class.
CCameraFrame.cpp CCameraFrame.h	`CCameraFrame`	Storage class representing the data of a single tracking system frame.
CCoordTransMatrix.cpp CCoordTransMatrix.h	`CCoordTransMatrix`	Storage class representing a coordinate transformation.
CTrackData.cpp CTrackData.h	`CTrackData`	Motion tracking information storage class.

B.9 C++ classes in module - `pet`

The `pet` module contains classes to manage and process the PET data needed for the cross-calibration.

Files	*Classes*	*Description*
CPETData.cpp CPETData.h	$\boxed{\texttt{CPETData}}$	Class providing a storage for PET calibration data.

Theses

- Using the described application, spatial corrections of raw PET acquisition data which account for the effects of patient movements during the investigation, can be performed efficiently.
- The optimized and multithreading enabled implementation of the movement correction allows to process the correction of coincidences with high speed and accuracy.
- The graphical user interface of the application is suitable for use by the medical technicians. Design and implementation of graphical elements ensures an intuitive usage.
- A seperate expert mode in the GUI and the comprehensive implementation of command-line options provides the necessary functionality for experts working on improving the movement correction.
- Administration and calibration of the motion tracking system can be performed from with the application. In addition, a network implementation allows to obtain tracking data directly from the motion tracking system.
- Direct support of standarized file formats like the input *listmode* format and the output of the corrected data in *ECAT7* sinogram files allows the usage of existing image reconstruction software.
- XML based import and export capabilities ensure future compatibility of the application. The possibility to save all movement correction relevant data in XML files allows to store the results of the movement correction together with the routinely archived patient data.
- The platform independent implementation in C^{++} and the use of the cross-platform Qt framework ensures future extensibility.
- Due to its optimized implementation the movement correction is used at the PET center, Rossendorf within routine performed head examinations where even small patient movements have an impact on the final tomographic image quality.

List of Figures

List of Tables

Listings

Abbreviations

`lmmc`	<u>L</u>ist<u>M</u>ode <u>M</u>ovement <u>C</u>orrection
ANSI	American National Standards Institute
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
CCD	Charge-Coupled Device
CG	Center of Gravity
CLI	Command Line Interface
CPU	Central Processing Unit
CT	X-ray computed tomography
ECG	Electrocardiogram
FOV	Field of View
FWHM	Full-Width Half-Maximum
GSL	GNU Scientific Library
GTK	Gimp Toolkit
GUI	Graphical User Interface
I/O	Input/Output
IDL	Interactive Data Language
LDC	LOR discretization correction
LOR	Line of Response
MRI	Magnetic Resonance Imaging
OFC	Out-of-FOV Correction
OOP	Object-Oriented Programming
PET	Positron-Emission-Tomography
POSIX	Portable Operating System Interface
ROI	Region of Interest
RPC	Remote Procedure Call
SMP	Symmetric Multi Processing
UDP	User Datagram Protocol
UML	Unified Modeling Language
XML	eXtensible Markup Language

Bibliography

[Amd67] G. M. Amdahl. Validity of single-processor approach to achieving large-scale computing capability. *Proceedings of AFIPS Conference, Reston, VA.*, pages 483–485, 1967. 41

[ART02] A.R.T. - Advanced Realtime Tracking GmbH. *ARTtrack1 and DTrack technical documentation*, 2002. `http://www.ar-tracking.de/`. 37

[Büh03] P. Bühler. An accurate method for correction of movement in pet. Technical report, PET research center Rossendorf, 2003. vi, 13, 22, 27

[CGN95] M. Casey, H. Gadagkar, and D. Newport. A component based method for normalization in volume pet. *3rd int. Conf. on Three-dimensional image reconstruction in radiology and nuclear medicine*, pages 67–71, 1995. 19

[Col85] P. Collard. A file organization for image processing. *Proceedings of the Digital Equipment Users Society*, pages 163–164, 1985. 35

[Con03] World Wide Web Consortium. eXtensible Markup Language (XML), 2003. `http://www.w3.org/XML/`. 57

[Dav55] C. M. Davisson. *Interaction of gamma radiation with matter.* North-Holland Publishing Company, Amsterdam, 1955. 6

[ECA99] CTI PET Systems, Inc. *ECAT Software Operating Instructions*, 1999. 36

[GDT03] M. Galassi, J. Davis, and J. Theiler. GNU Scientific Library Version 1.4, 2003. `http://sources.redhat.com/gsl/`. 53, 79

[GHJV03] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 26. edition, 2003. 62

[Gro03] Object Management Group. Unified Modeling Language Version 2.0, 2003. `http://www.omg.org/uml/`. 51

[HHPK81] E. J. Hoffman, S.-C. Huan, M. E. Phelps, and D. E. Kuhl. *Quantification in Positron Emission Computed Tomography: 4.Effect of Accidental Coincidences.* Journal of Computer Assisted Tomography, 1981. 5

[Hou72] G. N. Hounsfield. *A method of and apparatus for examination of a body by radiation such as X or gamma radiation.* The Patent Office, London, 1972. v

[Jus00] U. Just. Sammlung der Einzelereignisse einer PET-Kamera, Listmodmessung und Sortieren nach beliebigen Kriterien. Msc thesis, Hochschule für Technik und Wirtschaft Dresden (FH), 2000. 36, 65

[Keh01] F. Kehren. *Vollständige iterative Rekonstruktion von dreidimensionalen Positronen-Emissions-Tomogrammen unter Einsatz einer speicherresidenten Systemmatrix auf Single- und Multiprozessor-Systemen.* PhD thesis, Forschungszentrum Jülich, 2001. 4, 5, 6, 9, 12, 103

[Lan02] J. Langner. Parallel programming support within the linux operating system. Technical report, Queensland University of Technology, Brisbane, 2002. `http://www.jens-langner.de/ftp/ParComp.pdf`. 41

[LH99] C. S. Levin and E. J. Hoffmann. *Calculation of positron range and its effect on the fundamental limit of positron emission tomography system spatial resolution.* Physics in Medicine and Biology, 1999. 3

[LTJZ03] C. Labbe, K. Thielemans, M.W. Jacobson, and A. Zverovich. Software for Tomographic Reconstruction (STIR) Version 1.1, 2003. `http://stir.irsl.org/`. 76

[Nic98] T. Nichols. HR+ List Mode Format, 1998. Technical note. 35

[Oes98] B. Oestereich. *Objektorientierte Softwareentwicklung: Analyse und Design mit der UML.* Oldenbourg, fourth edition, 1998. 51

[Pie99] U. Pietrzyk. *Positron Emission Tomography: Physical Background and Applications.* Shaker-Verlag Aachen, 1999. 7, 105

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C.* Cambridge University Press, second edition, 1992. 15, 23

[Qt03] Trolltech Inc. *Qt: A multiplatform, C++ application development framework*, 2003. `http://www.trolltech.com/`. 79

[Sie96] Siemens AG. *Siemens ECAT EXACT HR+ Operating Instructions*, März 1996. Part Number 9300030-000. 10, 11

[VBTM03] Peter E. Valk, Dale L. Bailey, David W. Townsend, and Michael N. Maisey. *Positron Emission Tomography.* Springer-Verlag London, 2003. 1, 5

[WWH88] K. Wienhard, R. Wagner, and W.-D. Heiss. *Grundlagen und Anwendungen der Positronen-Emissions-Tomographie.* Springer-Verlag Berlin, 1988. 6