Development and Performance of a Sparsity-Exploiting Algorithm for Few-View Single Photon Emission Computed Tomogrpahy (SPECT) Reconstruction

Paul Arthur Wolf
Marquette University

DEVELOPMENT AND PERFORMANCE OF A SPARSITY-EXPLOITING ALGORITHM FOR FEW-VIEW SINGLE PHOTON EMISSION COMPUTED TOMOGRAPHY (SPECT) RECONSTRUCTION

by

Paul Arthur Wolf

A Thesis submitted to the Faculty of the Graduate School,
Marquette University,
in Partial Fulfillment of the Requirements for
the Degree of Master of Science

Milwaukee, Wisconsin

December 2012

ABSTRACT
DEVELOPMENT AND PERFORMANCE OF A SPARSITY-EXPLOITING ALGORITHM FOR FEW-VIEW SINGLE PHOTON EMISSION COMPUTED TOMOGRAPHY (SPECT) RECONSTRUCTION

Paul Arthur Wolf

Marquette University, 2012

Single Photon Emission Computed Tomography (SPECT) provides noninvasive images of the distribution of radiotracer molecules. Dynamic Single Photon Emission Computed Tomography provides information about tracer uptake and washout from a series of time-sequence images. Stationary ring-like multi-camera systems are being developed to provide rapid dynamic acquisitions with high temporal sampling. Reducing the number of cameras reduces the cost of such systems but also reduces the number of views acquired, limiting the angular sampling of the system. Novel few-view image reconstruction methods may be beneficial and are being investigated for the application of dynamic SPECT.

A sparsity-exploiting algorithm intended for few-view Single Photon Emission Computed Tomography (SPECT) reconstruction is proposed and characterized. The reconstruction algorithm phenomenologically models the object as piecewise constant subject to a blurring operation. To validate that the reconstruction algorithm closely approximates the true object when the object model is known and the system is modeled exactly, projection data were generated from an object assuming this model and using the system matrix. Monte Carlo simulations were performed to provide more realistic data of a phantom with varying smoothness across the field of view. For all simulations, reconstructions were performed across a sweep of the two primary design parameters: the blurring parameter and the weighting of the total variation (TV) minimization term. A range of noise and angular sampling conditions were also investigated. Maximum-Likelihood Expectation Maximization (MLEM) reconstructions were performed to provide a reference image. Spatial resolution, accuracy, and signal-to-noise ratio were calculated and compared for all reconstructions. The results demonstrate that the reconstruction algorithm very closely approximates the true object under ideal conditions. While this reconstruction technique assumes a specific blurring model, the results suggest that the algorithm may provide high reconstruction accuracy even when the true blurring parameter is unknown. In general, increased values of the blurring parameter and TV weighting parameters reduced noise and streaking artifacts, while decreasing spatial resolution. The reconstructed images demonstrate that the reconstruction algorithm introduces low-frequency artifacts in the presence of noise, but eliminates streak artifacts due to angular undersampling. Further, as the number of views was decreased from 60 to 9 the accuracy of images reconstructed using the proposed algorithm varied by less than 3%. Overall, the results demonstrate preliminary feasibility of a sparsity-exploiting reconstruction algorithm which may be beneficial for few-view SPECT.

ACKNOWLEDGMENTS

Paul Arthur Wolf

I would like to express my heartfelt thanks and graciously express my true appreciation for my thesis advisor Dr. Taly Gilat Schmidt. Her patience and guidance have been instrumental in this effort. I would also like to acknowledge Dr. Emil Sidky and Dr. Anne Clough for their valuable direction on this project. Truly this work could not have been completed with out the invaluable contributions of each. I would like also to thank the Marquette University Graduate School and the Department of Biomedical Engineering for the opportunity to complete this study and further my education.

Finally, I would like to thank my family and friends for their love and constant support. In particular, I would like to thank Dan and Lorrie Wolf and Abby Harris.

This work is supported by NIH, Grant No. 1 R15 CA143713-01A1. Simulations were performed on the MUgrid computing cluster (NSF OCI-0923037 and NSF CBET-0521602).

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Statement of the Problem

To provide dynamic imaging with high temporal sampling, Single Photon Emission Computed Tomography (SPECT) systems may sample only a few angular positions over the 360 degree range required for reconstruction. Few-view, sparsity-exploiting reconstruction algorithms have been applied to CT [1–3]. These algorithms are based on the exploitation of gradient-magnitude sparsity, an idea which follows from a model that X-Ray attenuation coefficient maps are approximately piecewise constant. Such algorithms are not directly applicable to SPECT because SPECT objects are distributions of radioactivity which may have rapid but smooth variation between regions. Additionally, SPECT systems have relatively low efficiency, which results in data with a higher level of noise than is typical for CT [4]. A reconstruction algorithm appropriate for few-view reconstruction that takes into account the unique challenges of SPECT is required to provide SPECT reconstructions from a small number of views.

Small-animal SPECT systems are being developed with multiple cameras and without gantry rotation to provide rapid dynamic acquisitions [5], [6]. Reducing the number of views required for reconstruction would require fewer gamma cameras, facilitating a reduction in system cost. Algorithms for few-view reconstruction have been developed for CT which exploit gradient-magnitude sparsity by minimizing image total-variation (TV) [1–3]. TV-based few-view reconstruction methods for CT assume the X-Ray attenuation coefficient map is approximately piecewise constant. SPECT

reconstruction benefits from the inclusion of methods promoting further regularity [7], [8].

1.2 Objective of the Study

The overall goal of the project was to develop a novel few-view SPECT reconstruction algorithm and to characterize the performance of the developed algorithm. This goal can be separated into two specific aims.

Aim 1: Development of Few-View SPECT Reconstruction Algorithm

Develop a sparsity-exploiting, few-view SPECT reconstruction algorithm based on related few-view algorithms developed for Computed Tomography (CT).

Aim 2: Characterization of Few-View SPECT Reconstruction Algorithm

Characterize the performance of the developed algorithm for different sampling, noise and data inconsistency conditions and quantify the effect on image quality. Describe the effect of algorithm parameter settings.

1.3 Chapter Organization

Chapter 2 reviews the fundamental physics of SPECT imaging and the theory behind SPECT image reconstruction. Compressed sensing theory is introduced in the context of few-view tomographic reconstruction.

Chapter 3 describes a sparsity-exploiting reconstruction algorithm developed for few-view SPECT. The algorithm was validated and characterized using different data and parametric values.

Chapter 4 summarizes the results of thesis study and describes areas of future work.

CHAPTER 2: FOUNDATIONS OF SPECT IMAGING

2.1 SPECT Imaging Fundamentals

Tomographic imaging yields images of transverse slabs of a volume. In both transmission tomography, in which radiation is passed through an object and detected on the other side, and emission tomography, in which radiation is emitted from within an object and detected outside of the object, transverse images can be obtained from the mathematical synthesis of projection images acquired at multiple views around the object. Tomographic slices (slabs) display the internal morphology of an object in the absence of overlying and underlying structures. In this way, tomographic images are advantageous when compared to corresponding projection images [4]. An example of transmission tomography is X-ray computed tomography (CT). CT requires X-rays (photons) to be passed through an object at a large number of angles. The resulting tomographic image is a map of the efficiency with which photons are attenuated within the materials. In the arena of medical imaging, CT is considered a technique that is good at providing structural information about a patient. Functional CT scans, from which physiology can be determined, are also possible with the usage of contrast agents [9].

Whereas transmission tomography maps attenuation, emission tomography maps the distribution of radioactivity within an object. Emission imaging techniques require self-luminous (radioactive) objects [10]. Clinically, a chemical or compound containing a radioactive isotope (radio-pharmaceutical) is administered intravenously, orally or by inhalation. The radio-pharmaceutical distributes in the subject and is taken up according to the physiology of the patient. One or more position sensitive radiation detectors are

used to acquire projections outside patient. Imaging modalities that use this technique are considered under the umbrella of nuclear medicine. Emission tomographic imaging requires projections to be acquired at several angular positions distributed around the object. The distribution of the radionuclide depends upon the physiology of the test subject, thus nuclear medicine imaging is a method of functional imaging [4]. CT and nuclear medicine use ionizing radiation to image biological tissue.

Emission tomography includes two commonly used nuclear medicine modalities. Single Photon Emission Computed Tomography (SPECT) is distinguished itself from Positron Emission Tomography (PET) by the radionuclides used and the resultant emissions. SPECT uses radionuclides that emit gamma rays whereas radionuclides used in PET emit positrons. Radionuclides used in PET emit positrons that annihilate with a nearby electron, emitting two photons in opposite directions. Only photons detected at opposite angular positions at very nearly the same time are used in reconstruction. This requires complex discriminating electronics and at least two detectors, though typically many more detectors are used. The half-lives of commonly used positron-emitting radionuclides range from just over a minute (75 sec. for ^{82}Rb) to 110 minutes (for ^{18}F). Half-lives on this order require cyclotrons to be available locally. However, PET has the advantage that several biologically abundant elements (C, N, O) have isotopes that decay by positron-emission (^{11}C, ^{13}N, ^{15}O) [4]. The most commonly used radio-pharmaceutical for PET is ^{18}F-fluorodeoxyglucose (^{18}F-FDG), which mimics glucose [4].

SPECT radionuclides emit gamma radiation. One or more radiation detecting cameras are rotated about the patient to acquire the necessary projection data at various angular positions. Gamma-ray emitting radionuclides used in nuclear medicine generally

have longer half-lives than their positron emitting counterparts. The half-lives of gamma-ray emitting radionuclides used in nuclear medicine range from several hours (6 hr. for ^{99m}Tc) to several days (60 days for ^{125}I). This eliminates the need for a radionuclide production facility on site. Further, SPECT does not require coincident pairs to be detected, eliminating the requirement for multiple detectors. However, to increase the data production rate, multiple cameras can be used.

In the following sections, SPECT is considered in more depth.

2.1.1 Physiological localization

SPECT, as with all nuclear medicine, requires a radio-pharmaceutical to be introduced into the body. This can be done by injection, orally, or inhalation. The method depends upon the application and on the radio-pharmaceutical itself. The most commonly used isotope in SPECT is ^{99m}Tc, which can be bound to a number of complexes to create useful radio-pharmaceuticals [10]. These are generally administered intravenously [4]. Once the radio-pharmaceutical is administered, it is distributed throughout the body by one or more physiological mechanisms. These mechanisms again depend upon the application and on the radio-pharmaceutical. Typical mechanisms of localization include capillary blockade, perfusion and passive diffusion [4].

Capillary blockade occurs when particles larger than red blood cells are injected intravenously and become trapped in the narrow pathways of capillaries and arterioles. This is typical of pulmonary perfusion studies using ^{99m}Tc-MAA (micro-aggregated albumin) [11]. In such studies, approximately 85% of the ^{99m}Tc-MAA becomes trapped

in the micro-circulation of the lung. This permits measurement of regional pulmonary blood flow [11]. Perfusion describes the delivery of arterial blood to tissue. This is used in SPECT as the radio-pharmaceutical binds to blood cells and distributes around the body. This is an important diagnostic element in blood flow studies, e.g. hepatic and renal studies, myocardial perfusion studies [4] Myocardial perfusion studies can be used to diagnose coronary disease and may be used to plan treatment [12]. Passive diffusion is the movement of a substance from higher concentration to lower concentration. This mode of localization is used in brain studies because a healthy blood-brain barrier will not permit diffusion of radiopharmaceuticals whereas a compromised blood-brain barrier will.

2.1.2 Radiation Emission

The quantity of nuclear transformation occurring in a radioactive material is called activity. Activity is defined as the number of radioactive atoms undergoing nuclear transformation per unit time and is described by the following equation:

$$A = \frac{-\partial N}{\partial t} \tag{1}$$

where A is activity, N is the number of atoms undergoing nuclear transformation and t is time. Activity is expressed in units of Becquerels (Bq) or Curies (Ci). One Bq is defined as 1 disintegration per second; one Ci is defined as 3.7×10^{10} Bq.

Gamma radiation is emitted isotropically from radionuclides used in SPECT. That is, gamma rays are equally likely to be emitted in any direction. In the case of SPECT, as with all nuclear medicine, photons may exit the subject or object on

trajectories that do not intersect the detectors and cannot be detected because of the imaging geometry. The number of photons detected has implications on the quality of resulting images. In general, image quality increases with the number of detected photons. The number of photons detected is reduced by atomic interactions which alter a photon's path. The three significant interactions in nuclear medicine are Rayleigh scattering, Compton interactions and photoelectric absorption.

Rayleigh scattering occurs when an incident photon excites an atom. The electric field of the photon causes all of the electrons in an atom to resonate. The energy transferred to the electrons is immediately radiated, resulting in a photon with the same energy but in a different direction. The scattering atom is not damaged by this interaction. Detection of photons that have undergone Rayleigh scattering is a source of noise in medical imaging, however, Rayleigh scattering is relatively unlikely in the SPECT energy range (70-360 keV) [12].

Compton interactions are the most likely to occur in the SPECT energy range. A Compton interaction occurs when an incident photon collides with an electron in an atom. Some energy is transferred from the photon to the electron. The electron is ejected and the photon is scattered with a reduction in energy. Energy in this interaction is conserved, as described by the following equation:

$$E_{\text{photon}} = E_{scattered\ photon} + E_{e^-} + E_{binding}\,. \qquad (2)$$

Binding energy, $E_{binding}$, is small enough to be considered trivial. This interaction damages the atom, causing ionization. Ionization is the ejection of an electron from an atom. This results in an electron, which is negatively charged, and the correspondingly positively charged atom, termed an ion. Ionizing radiation causes both byproducts of ionization

(electron and ion) to be highly chemically reactive. This can easily damage DNA, causing cellular mutations. Additionally, ionizing radiation can directly damage DNA. Detection of photons that have undergone Compton interactions are a source of noise in medical imaging.

Photoelectric absorption is another type of atomic interaction that ionizes the atom. An incident photon transfers all of its energy to an orbital electron, which is ejected from the atom. The energy of the ejected electron is

$$E_{e^-} = E_{\text{photon}} - E_{binding} \ , \tag{3}$$

For this interaction to occur, the energy of the incident photon must be greater than the binding energy. The electron most likely to be ejected is the one whose binding energy is the closest to the energy of the incident photon. The probability of photoelectric absorption increases significantly with the atomic number of the atom with which it interacts [4].

Scattered photons yielded from Rayleigh and Compton interactions can introduce uncertain information in the data and have a negative effect on the tomographic reconstruction process [12]. Attenuation of photons can introduce quantitative errors and artifacts in SPECT images. Several strategies can be employed to correct for the effects of both scatter and attenuation. For instance, scatter can be modeled in the reconstruction algorithm and attenuation can be mapped and compensated for using a CT scan [13].

2.1.3 Radiation Detection

The projections used to generate a SPECT image correspond to planar images acquired in nuclear medicine. The process of acquiring a planar image requires two

components, the collimator and the detector.

SPECT uses radionuclides that emit gamma-radiation isotropically. In order to form a coherent image, a relationship between the incidence of photons upon the detector and the point of origin of each photon must be established. A collimator mounted between the object and the detector establishes this relationship by permitting only photons from certain directions to reach the detector, while absorbing photons from other directions. Thus, collimators are constructed of high atomic number, high-density materials. Lead and tungsten are common [4]. The high-density material includes holes which effectively filter emission rays, allowing only gamma-rays traveling in certain directions to pass through. These holes are of four basic geometries: parallel-hole, converging, diverging and pinhole. These four geometries are represented in Fig. 1.

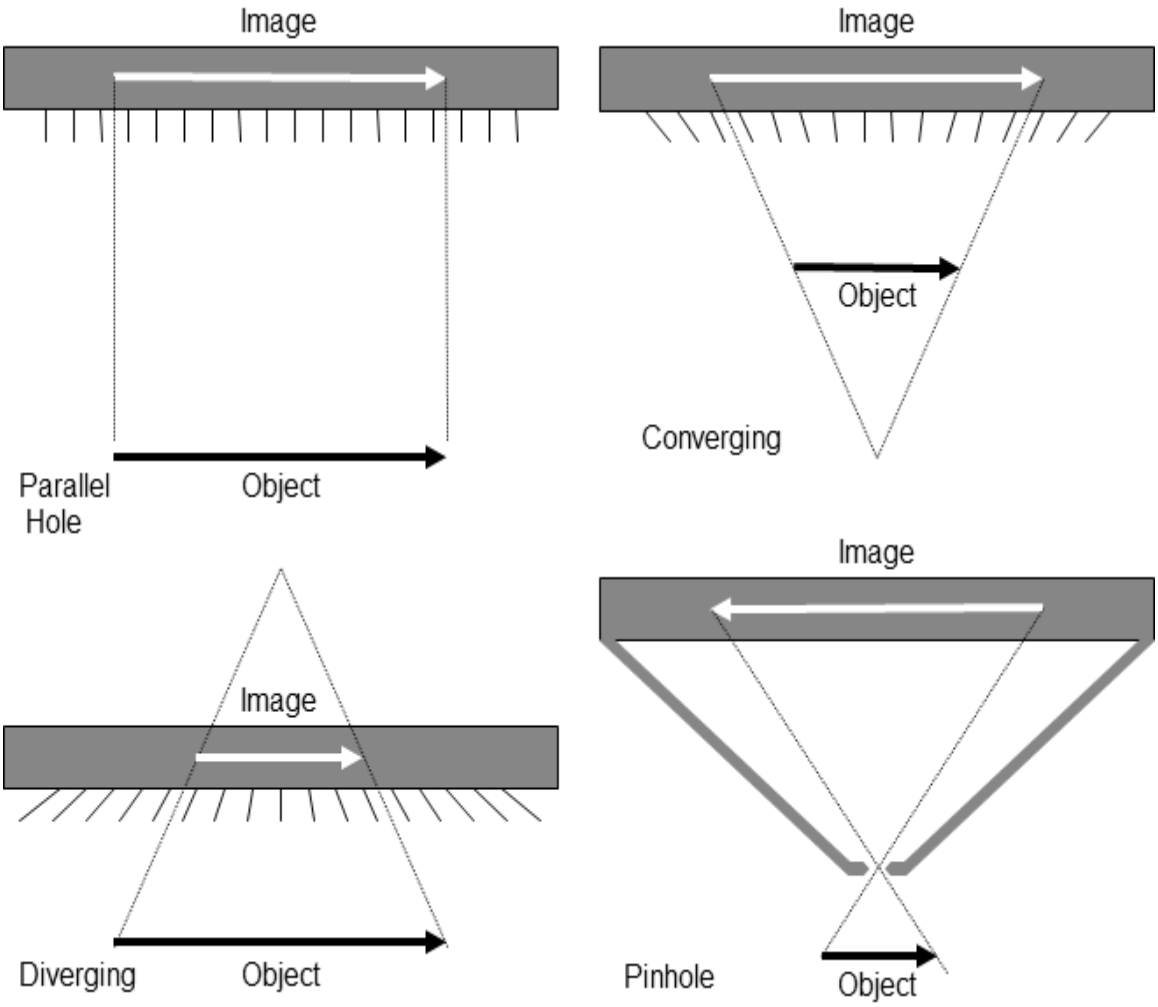

Figure 1. Four common collimator geometries.

The most commonly used collimator geometry is parallel-hole. Parallel-hole collimators use holes oriented parallel to each other. There is no magnification in this case. Spatial resolution degrades as distance between the collimator and the object increases. A converging collimator has holes focused at a point in front of the detector. The magnification of the detected object increases with the distance between collimator and object. Diverging collimators have holes focused at a point behind the detector. This has the opposite effect as the converging collimator; the minification of the detected object

increases with the distance between the collimator and object. Pinhole collimators include a small number of small holes oriented at the apex of leaded cones. The magnification/minificaton of the projected object is determined by both the distance between the object and collimator and the distance between the collimator and detector. Pinhole collimators are used commonly in pediatric nuclear medicine and small animal SPECT imaging. Work has been done to optimize multi-pinhole geometries [14],[15].

Photons pass through the holes of the collimator fall incident upon the detector. The most commonly used detector in SPECT is the Anger scintillation camera [4]. The Anger camera is ideally suited for ^{99m}Tc, which is used commonly in SPECT. The active scintillating component of the Anger camera is a thallium-activated NaI crystal. This crystal provides high photoelectric absorption probability for gamma rays and a relatively high gamma ray-to-light conversion efficiency [4]. The crystal is coupled to photomultiplier tubes (PMTs), which amplify and convert the scintillated light to an electrical current signal. The signal from each scintillation is processed separately. Images are acquired in either frame mode or list mode. Frame mode counts the number of photons incident upon each pixel of the camera for a set period, be it a set amount of time or a set number of counts. No timing information is stored besides the acquisition time. An image is created from the number of counts detected by each detector element. List mode stores the position of each interaction and the detection time. Using this timing and position information, the count data can be binned into images from list mode data. Alternatively, reconstruction directly from list mode data is possible [16]. List mode requires more complicated timing electronics and is widely used in SPECT acquisitions.

SPECT data are acquired on an arc of 360 degrees around the patient. Because projection images at angles 180 degrees apart from each other are ideally mirror images, a 180 degree arc is sufficient. However, because attenuation has a strong deleterious effect on the acquired data and projections are blurred by the collimator, 360 degree scans are used for non-cardiac SPECT studies [4]. To acquire data over the arc, one or more Anger cameras are rotated about the patient or object on a rotating gantry. Stationary camera systems have also been developed [17], [18].

Dynamic SPECT is a technique that provides information about tracer uptake and washout from a time series of images. This study of dynamic processes reflects physiologic functions, such as perfusion and metabolism, which change with time. Rotating dynamic SPECT systems which record a tomographic acquisition every 15 seconds and measure time-activity curves on the order of minutes have been developed [19], [20]. To increase temporal sampling, stationary systems have been developed [5], [6]. These systems, however, use a large number of cameras which, in some situations, may be cost prohibitive [21].

2.1.4 SPECT Image Reconstruction

Image reconstruction is equivalently considered as estimating an inverse problem. In the tomographic imaging context, the forward problem is the determination of a projection image produced by a given object and the inverse problem is the determination of an object from a given projection image. In the real case, in which there is noise and uncertainty, an object can yield an infinite number of images. This renders practical

inverse problems virtually impossible to solve for a unique solution. Rather, the goal of solving inverse problems is to yield an approximate object description. Thus, some inverse problems can be cast as estimation problems, taking into account the statistics of the data. The solutions to inverse problems can be divided into two general groups, direct solutions, which require one step, and implicit solutions, which require an iterative process. In order to discuss these methods, an understanding of estimation theory is needed.

2.1.4.a Estimation Theory

Estimation problems require four ingredients [22]. The first is a vector of parameters to be estimated. In the case of image reconstruction this corresponds to a set of coefficients of an expansion of basis functions that describe an object. Typically, rectangular pixels or voxels are used, but other basis functions can be used e.g. smoothly varying isotropic elements (blobs) [7], [23]. Second, the underlying randomness of the parameters must also be known. In the context of SPECT, which employs photon counting, a Poisson distribution can be assumed [22]. Third, it is essential to know the mapping from parameter space to the measurement space. When speaking of SPECT imaging, in which projection data are measured and the estimated parameters correspond to a description of the object, this mapping can be thought of as the solution to the forward problem or projection. Finally, estimation requires a procedure for mapping from the data to the estimate. The procedure can be analytical or iterative in nature [22]. Intuitively, an iterative procedure can be formulated as the optimization of some cost

function; often this is the distance between estimated data and measured data. For instance, for cost function $C(\boldsymbol{x},\boldsymbol{y})$, the formulation would be

$$\hat{\boldsymbol{x}} = argmin_x C(\boldsymbol{x},\boldsymbol{y}), \tag{4}$$

where, $\hat{\boldsymbol{x}}$ is the estimated solution, $\boldsymbol{x}$ is a set of coefficients that describes an object and $\boldsymbol{y}$ is a set of known properties that can be mapped to $\boldsymbol{x}$.

Parameters are said to be estimable from a data set if there is an unbiased estimator of it for all reasonable values. Consider the general case of a linear measurement system described by

$$\boldsymbol{g} = \boldsymbol{H}\boldsymbol{f} + n. \tag{5}$$

where $\boldsymbol{g}$ is the measured data, $\boldsymbol{H}$ is a measurement system operator, $\boldsymbol{f}$ is the true object and n is a member of the null space associated with the sampling operator such that $\boldsymbol{H}n = 0$. If there exist two objects $\boldsymbol{f}_1$ and $\boldsymbol{f}_2$ such that $\boldsymbol{H}\boldsymbol{f}_1 = \boldsymbol{H}\boldsymbol{f}_2$ but an estimated parameter of each object is unequal, the parameters of a general object $\boldsymbol{f}$ are not estimable. In the frame of tomography, if a system is estimable, there exists a unique object for a set of projections. The existence of null functions causes some parameters to be inestimable. Null functions of $\boldsymbol{H}$, $\boldsymbol{f}_{null}$, fulfill

$$\boldsymbol{H}\,\boldsymbol{f}_{null} = 0 \tag{6}$$

Such functions are said to belong to the null space of $\boldsymbol{H}$. Non-trivial null functions exist in all cases except where the rank of $\boldsymbol{H}$ matches the dimension of the space containing f. In the case of tomographic imaging, where the desired parameters are the coefficients of basis function expansions of an object, the parameters can be made estimable in the absence of noise by ensuring the basis functions have trivial null space.

2.1.4.b Direct Tomographic Reconstruction

Consider an estimable discrete, linear measurement system,

$$\boldsymbol{g} = \boldsymbol{H}\,\boldsymbol{f}. \tag{7}$$

$\boldsymbol{H}$ is the system matrix that describes the probability that a photon emitted from a certain location in the object vector, $\boldsymbol{f}$, contributes to the measured data vector, $\boldsymbol{g}$, at a certain location. $\boldsymbol{H}$ can be thought of as a set of linear equations that can be solved for the object. The object can be found by direct inversion of the system matrix, $\boldsymbol{H}$,

$$\boldsymbol{f} = \boldsymbol{H}^{-1}\boldsymbol{g}. \tag{8}$$

However, $\boldsymbol{H}$ is typically very large and the inversion is numerically unstable, particularly when noise is considered. Nonetheless, $\boldsymbol{f}$ can be approximated by pseudo inversion of the system matrix. Techniques which depend on inversion of the system matrix are generally difficult to implement and considered relatively slow [13]. If the system described by (7) is generalized to a continuous system, direct inversion of the system is theoretically possible. Analytical algorithms can be developed assuming continuous data and adapted to be used on realistic, sampled data. However, this assumption causes the system to be inestimable so these methods are approximate. This is the case for filtered back-projection (FBP), which is based on inversion of the Radon transform.

Filtered Back-projection (FBP)

The measured projection data in a tomographic imaging system is taken at a series of angles spanning the 360 degree arc about an object, thus it can be expressed in polar coordinates as $g(s,\phi)$, where s is the linear position of the projection data and ϕ is the angular position on the arc about the object. The object is described in two dimensions

by $f(x,y)$. The Radon transform of $f(x,y)$ is the set of projections at every angle and is defined by

$$g(s,\phi)=\int_{-\infty}^{\infty} f(s\cos(\phi)-u\sin(\phi), s\sin(\phi)+u\cos(\phi))du \qquad s=x\cos(\phi)+y\sin(\phi), \tag{9}$$

where u is a point on line s that intersects the detector and is perpendicular to ϕ [24], [25]. The ray-sums yielded by this relationship, $g(s,\phi)$, are analogous to the projection data. In the sense that the Radon transform generates projection data from the object, it is an approximate measurement system operator. The reverse of the projection operation is called back-projection. By inverting the Radon transform of an object, $f(x,y)$, from measured data, $g(s,\phi)$, an approximate object description can be obtained; a solution to the inverse problem is found. The inversion is derived for a continuously sampled case and then discretized for sampled data. FBP uses this discretized inversion of the Radon transform [13].

The projection operation accomplished using the Radon transform can be approximately inverted using continuous back-projection, defined by the following relationship

$$b(x,y)=\int_{0}^{\pi} g(s,\phi)d\phi, \tag{10}$$

where $b(x,y)$ is an approximation of the object that produced the projections, $g(s,\phi)$. This estimate can be described as taking the sum of all the ray-sums that intersect a given point. This process alone results in a blurred image of the object, but the application of a two-dimensional high-pass filter to the image serves to emphasize the edges and remove blurring. A ramp filter is ideal in the noiseless case in the sense that the exact object $\boldsymbol{f}$ is

recovered [26]. Other filters are used to suppress the amplification of high spatial frequency noise. In practice, filters are applied to projection data prior to back-projection as this is more computationally efficient [24].

FBP enjoys the benefit of short reconstruction times compared to the iterative algorithms presented below. FBP tends to emphasizes high frequency noise, due to the high-pass filtering. This limits the usefulness of low photon count images, like those expected using SPECT, reconstructed using FBP. Nonetheless, FBP has historically been applied to SPECT because of its computational efficiency [27]. Iterative algorithms can improve image quality compared to FBP by more accurately modeling the parameters of the physical imaging system and underlying statistics of photon detection [28].

2.1.4.c Implicit (Iterative) Tomographic Reconstruction

Iterative image reconstruction (IIR) algorithms find the solution, $\boldsymbol{f}$, to the estimable system presented in (7) by using an iterative procedure. These algorithms are often posed as optimization problems. Applying such a problem formulation to the system of (7) yields

$$\boldsymbol{f} = argmin_{\hat{f}}\, Q(\boldsymbol{g}, \boldsymbol{H}\hat{\boldsymbol{f}}), \tag{11}$$

where $\hat{\boldsymbol{f}}$ is the set of estimated parameters that describe $\boldsymbol{f}$ and Q is an objective functional that encourages agreement between $\boldsymbol{g}$ and $\boldsymbol{H}\hat{\boldsymbol{f}}$. $\boldsymbol{H}\hat{\boldsymbol{f}}$ is an estimation of the projection data produced by $\hat{\boldsymbol{f}}$. $\boldsymbol{H}\hat{\boldsymbol{f}}$ describes applying the system matrix to perform forward projection. The system matrix can be designed to take into account the physical effects of the system (e.g. blurring, attenuation) without greatly complicating the inverse

problem, allowing for better reconstructed image quality than FBP. It is desirable to use a well-designed system matrix, as it can improve the image quality of IIR algorithms [29]. Optimization problems such as (11) can be solved using any number of iterative methods appropriate to the objective functional [30], [31]. In general, such methods can be described as follows: an estimate of the data is compared to the measured data; this comparison is used to modify the estimate, generating a new estimate. IIR algorithms differ by the objective functional that is selected, Q, and the optimization method used to obtain the estimation.

2.2 Indirect SPECT Reconstruction

2.2.1 The Objective functional

The objective functional optimized in (11) has two important qualities. It should encourage agreement between $\boldsymbol{g}$ and $\boldsymbol{H}\hat{\boldsymbol{f}}$ and it should control the amplification of noise. The former can be considered as data fidelity and the latter as data regularity. The objective functional can include terms for both data fidelity and data regularity and can be written as

$$Q(\boldsymbol{g},\boldsymbol{H}\hat{\boldsymbol{f}}) = Q_{fid}(\boldsymbol{g},\boldsymbol{H}\hat{\boldsymbol{f}}) + \eta Q_{reg}(\hat{\boldsymbol{f}}), \tag{12}$$

where η is a problem parameter that controls the tradeoff between data agreement and regularity. The regularity term most often depends only on the estimate of the object, thus it is written as a factor of only the estimate of the object [10]. $Q(\boldsymbol{g},\boldsymbol{H}\hat{\boldsymbol{f}})$ can be minimized as a sum or its terms can be taken separately.

2.2.2 Data Fidelity Functionals

The purpose of the data fidelity functional is to encourage agreement between the estimated and measured data. It can be considered as a distance to be minimized. Thus, the simplest data fidelity function is the euclidean distance between the estimated data, $\boldsymbol{H}\hat{\boldsymbol{f}}$, and the measured data, $\boldsymbol{g}$. This is expressed as the square of the L_2 norm of the difference between $\boldsymbol{H}\hat{\boldsymbol{f}}$ and $\boldsymbol{g}$,

$$Q_{fid}(\boldsymbol{g},\boldsymbol{H}\hat{\boldsymbol{f}})=\|\boldsymbol{g}-\boldsymbol{H}\hat{\boldsymbol{f}}\|^2. \tag{13}$$

One choice to minimize this function is the least-squares algorithm. This algorithm can be derived from a Gaussian likelihood assumption [10]. Because SPECT can be modeled as a Poisson process, this data fidelity functional does not model the statistical properties of SPECT imaging.

A data fidelity function that incorporates the Poisson likelihood is the Kullback-Leibler (KL) distance (D_{KL}). Minimizing D_{KL} is equivalent to maximizing the log-likelihood of Poisson data [10]. D_{KL} is defined by

$$Q_{fid}(\boldsymbol{g},\boldsymbol{H}\hat{\boldsymbol{f}})=D_{KL}(\boldsymbol{g},\boldsymbol{H}\hat{\boldsymbol{f}})=\sum_{n=0}^{N}(\boldsymbol{H}\hat{\boldsymbol{f}})_n-g_n+g_n\ln(g_n/(\boldsymbol{H}\hat{\boldsymbol{f}})_n), \tag{14}$$

where N is the number of detector elements. D_{KL} is convex [31]. It can be minimized using any convergent convex minimization strategy. It is worth noting that D_{KL} is not a true distance because $D_{KL}(\boldsymbol{g},\boldsymbol{H}\hat{\boldsymbol{f}})\neq D_{KL}(\boldsymbol{H}\hat{\boldsymbol{f}},\boldsymbol{g})$. .

2.2.3 Data Regularity Functionals

Forcing exact agreement between the measured data and the estimated data when

either the measured data is corrupted by noise or the system matrix is not exact (or both) causes these inconsistencies to be amplified [22]. The regularization term penalizes values that are outliers to the data [31]. Thus, the regularizing term is sometimes called a penalty term. The inclusion of a penalty term that accounts for possible errors in the data or the data estimate results in the bi-criterion problem presented in (12) [31].

It is common to use regularizing functionals that are based on the Euclidean norm; for example,

$$Q_{reg}(\hat{\boldsymbol{f}}) = \|\boldsymbol{D}\hat{\boldsymbol{f}}\|^2, \tag{15}$$

where $\boldsymbol{D}$ is a discrete magnitude operator. This regularizing functional assumes that noise is rapidly varying and that the original signal is smooth. However, any abrupt changes intrinsic to the object (edges) will be compromised. A regularizing functional that attempts to preserve the abrupt changes at the edges is total-variation (TV). A TV regularizing functional is described by

$$Q_{reg}(\hat{\boldsymbol{f}}) = \|\boldsymbol{D}\hat{\boldsymbol{f}}\|_1. \tag{16}$$

This functional penalizes rapidly varying data but is more tolerant of singular changes. This regularizing functional has particular significance in its application to few-view reconstruction techniques because of its characterization as an L_1 norm. TV can potentially be used in some sparsity-exploiting reconstruction algorithms. This is explored further later.

IIR algorithms formulated as in (12) can use any convergent optimization technique. If the data fidelity term is convex, as (13) and (14) are, and the regularization term is convex, as (15) and (16) are, the objective function is convex and any convergent convex optimization technique can be used. If the system is estimable and the selected

algorithm is run to convergence, a unique and exact estimation of $\boldsymbol{f}$ will be the result. A popular iterative algorithm used to reconstruct SPECT images is maximum likelihood expectation maximization (MLEM).

2.2.4 Maximum Likelihood Expectation Maximization (MLEM)

The activity of a voxel, defined by (1), is the number of photons emitted per unit time from that voxel. This is a probabilistic process that is modeled as a Poisson distribution with mean b. The probability that b photons are emitted from voxel k in a specified period of time is

$$P(b)=\frac{e^{-b}b^{b}}{b!}. \tag{17}$$

SPECT reconstruction is the estimation of the expected value of the activity of a voxel. The goal is to determine the most likely estimate of the expected value of b, i.e. the maximum likelihood (ML) estimate of b. The likelihood is the probability of obtaining a particular set of data considering a given a particular probability distribution. In this way, ML methods take into account the statistical model of the data. The system matrix, $\boldsymbol{H}$, consists of the probabilities that an emission in a voxel is detected in a certain detector; the element H_{jk} is the probability that an emission from voxel k will be detected in detector element j. Because emission is a Poisson distributed process the likelihood function is

$$P(\boldsymbol{g}|\hat{\boldsymbol{f}};\boldsymbol{H})=\prod_{j} e^{\sum_{k=0} H_{jk} f'_{k}} \frac{\sum_{k}\left(H_{jk} f'_{k}\right)^{g_j}}{g_j!}\ . \tag{18}$$

The objective of ML methods is to find the vector $\hat{\boldsymbol{f}}$ that maximizes the likelihood function. One way of maximizing likelihood is the expectation maximization (EM) algorithm.

EM consists of two stages. The first stage computes an expected maximum likelihood assuming the log-likelihood of a set of estimated values. This is the expectation (E) step. The second stage is the maximization (M) step. The M step computes the values that maximize the expected likelihood determined in the E step. These computed values are then used to update the estimated values to be used in the next E step. Applying these steps causes the sequence to converge to the ML estimate, increasing the likelihood at each step [32]. One algorithm that uses EM to maximize the likelihood of a Poisson data model is Maximum Likelihood Expectation Maximization (MLEM). MLEM was first applied to emission tomographic reconstruction by Shepp and Vardi in 1977 [33].

The algorithm described by Shepp and Vardi uses the iterative procedure with a multiplicative update described by

$$\hat{f}_{k^{n+1}} = \hat{f}_k \sum_{j=0}^{N} \frac{g_j H_{jk}}{\left(Hf'^{n}\right)_j} \quad k = 1,2,\ldots, M\,, \tag{19}$$

where M is the number of voxels. MLEM maximizes the log-likelihood of a Poisson variable. This is equivalent to minimizing the D_{KL} between the data and the estimated data, $D_{KL}(\boldsymbol{g}, \boldsymbol{H}\hat{\boldsymbol{f}})$ [22]. Although efforts have been made to incorporate regularity [7], [8], MLEM typically does not include a provision for enforcing regularity to control noise amplification. This leads to noise deterioration in higher iterations which can result in a virtually useless image. Each step continues to approach a maximum likelihood, but

noise becomes emphasized, degrading the quality of the produced images. In practice, this problem can be avoided by stopping the iterative procedure before image quality is degraded. Thus, the image produced by the MLEM algorithm depends on the number of iterations and the initial value of the estimate. The stopping point is usually chosen on a subjective basis [22]. The MLEM algorithm is described by the following pseudo-code. $\boldsymbol{H}^T$ indicates back-projection and := denotes assignment.

Listing 1: Pseudocode of the MLEM algorithm

$f_0 := 1$; $v := 0$

repeat *main loop*

for $j = 0{:}N$; do : $v_j = g_j / (\boldsymbol{H}\tilde{\boldsymbol{f}}^n)_j$

for $k = 0{:}M$; $\tilde{f}_k^{n+1} = \tilde{f}_k^n (\boldsymbol{H}^T \boldsymbol{v})_k / (\boldsymbol{H}^T \boldsymbol{1})_k$

$n := n+1$

until {stopping criterion}

$\tilde{\boldsymbol{f}} := \tilde{\boldsymbol{f}}_n$

return $\tilde{\boldsymbol{f}}$

The implementation of the procedure described in (19) and by the pseudo-code above includes two applications of the system matrix. The first is the matrix multiplication of $\boldsymbol{H}$ with the estimate of the object, $\hat{\boldsymbol{f}}$. This is forward-projection. The other application is the sum of a product with a corresponding element of $\boldsymbol{H}$ for every detector element j performed for every voxel k. This corresponds to back-projection. Thus, for every iteration of the MLEM algorithm a forward- and back-projection are performed. The forward- and back-projection operations are computationally demanding and account for much of the complexity of the implementation of MLEM. Both of these operations, however, use the system matrix $\boldsymbol{H}$ which can model physical and statistical

parameters of the imaging system, allowing for improved reconstructions when compared to FBP [24]. The modeling of the system matrix and applying it in both a forward- and back-projection operation is discussed below. MLEM is popularly used for SPECT reconstruction because of its good handling of high levels of noise.

2.2.5 The System Matrix

The system matrix, $\boldsymbol{H}$, describes the probability that a photon emitted in a certain voxel is detected in a certain detector element. The application of the system matrix to a discrete object generates an estimation of the projection data yielded by an object; the application of the system matrix is forward projection. The adjoint is back-projection. The system matrix can be determined using Siddon's ray tracing algorithm [34].

Siddon's method assumes an object volume is an intersection of orthogonal sets of equally-spaced parallel lines, as opposed to considering each voxel an independent element. The intersections of a ray with the lines are calculated instead of the intersections of a ray with the voxels. This allows the length of a ray to be determined recursively from the first intersection. Siddon's method allows for the incorporation of prior knowledge (i.e. attenuation maps) into the model. The length of the segment of the ray included in a voxel is normalized to the voxel dimensions. This represents the element of $\boldsymbol{H}$, H_{jk}, which is the probability that an emission from voxel k is detected at detector element j [35]. The ray-driven approach in pinhole SPECT considers rays extending from the center of each detector element through object space, forming an inverse cone. This approach is depicted in fig. 1a.

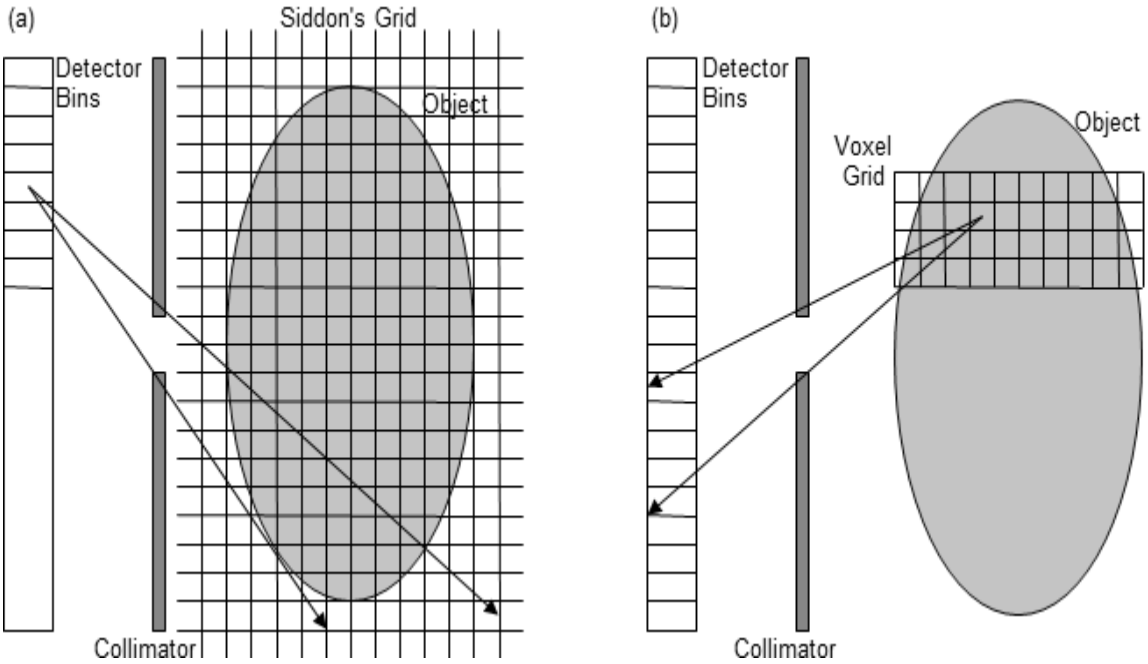

Figure 2. Illustrations of a ray-driven approach (a) and a voxel driven approach (b). Rays show cone boundaries.

An alternative method to generate the system matrix is the voxel driven approach [29]. The voxel-driven approach considers all possible emission directions for each voxel. The rays that emanate from the voxel, pass through the collimator and strike the detector are considered. Probabilities are assigned to the four nearest neighbor detector elements to the incident location. The probability is calculated corresponding to the distance from the incident point to the center of the detector element. This forms a cone of rays as shown in fig. 1b. In SPECT, the system matrix is very large. Methods have been developed to compute the necessary probabilities as needed instead of storing the entire matrix [29], [36].

2.2.6 Forward-projection

Forward projection is the estimation of projection data generated by an object. This can be thought of as applying the system matrix to a discrete representation of the object as follows:

$$\hat{g}_j = \sum_{k=0}^{M} H_{jk} f_k, \quad j = 1,2,\ldots, N. \tag{20}$$

This estimates the mean number of counts in each detector bin based on the geometric and physical properties of the system. This is written as $\boldsymbol{Hf}$.

2.2.7 Back-projection

Back-projection is the estimation of an object given projection data. This is done by taking the number of counts detected in a projection bin, weighting it by the probability that a count originated in a certain voxel and assigning that value to the voxel. This is summed for all detector elements. This procedure can be described as

$$\hat{f}_k = \sum_{j=0}^{N} H_{jk} g_j, \quad k = 1,2,\ldots, M. \tag{21}$$

where M is the number of voxels. This operation is adjoint to forward-projection; it is written as $\boldsymbol{H^T f}$.

2.3 Compressed Sensing

Compressed sensing (CS) is a mathematical theory that demonstrates that it is possible to accurately reconstruct a signal by L_1 minimization using seemingly undersampled data, provided the reconstructed signal is sparse [37], [38]. CS exploits the natural compressibility of images. This can be simply explained by considering the well-

known image compression paradigms widely used today. Compression formats aim to transform a signal into a vector of sparse coefficients. If the transform can be inverted, only the active coefficients need to be saved in order to accurately reconstruct the image. This compression procedure transforms an image into a sparse domain. CS asserts that if an image can transformed into a sparse domain, and has relatively few meaningful coefficients in that domain, only the sparse information is needed, allowing the use of seemingly undersampled data [39]. The use of undersampled data has an array of benefits. In CT, if fewer projections are needed to reconstruct images, radiation dose is reduced [40]. In magnetic resonance imaging (MRI), fewer samples enable a shorter acquisition, allowing for a faster frame-rate in dynamic scans [41]. In SPECT, using fewer projections has the potential to increase the temporal sampling of a dynamic SPECT scan.

There are conditions which must be met for a CS reconstruction to be feasible. The reconstructed signal must be sparse, and the artifacts inherent to undersampling must be incoherent with respect to the sparsifying transform, appearing as random noise. Coherence is the idea that an output can be estimated from an input to a given system. Incoherence is achieved when the effect of an input cannot be predicted. In the case of CS, artifacts due to undersampling would have a universal effect effect in the sparse domain, appearing as additive random noise [42]. These properties generally require the reconstruction procedure to be non-linear [39]. The sparsity requirement can be satisfied in any known linear transform domain, the inverse transform need not necessarily be known [1]. In a sparse representation of an image, significantly fewer pixels hold significant value to the image. To reconstruct this representation, fewer samples are

needed because less data needs to be determined. Incoherent sampling is a requirement of the first CS papers [37]. Subsequent work has relaxed this constraint, at the expense of requiring more data [43]. The non-linear reconstruction procedure is iterative in nature. This procedure requires the sparsity of the proposed solution to be enforced along with data consistency. The first CS papers show that by minimizing the L_1 norm of a sparse representation of a signal, subject to a data fidelity constraint, the signal can be reconstructed from undersampled data [37], [39]. L_1 minimization is a known heuristic for finding a sparse solution [31]. The general CS problem can be posed as

$$minimize \quad \|\Psi \boldsymbol{f}\|_1 \, subject\ to \, \boldsymbol{Hf} = \boldsymbol{g}, \tag{22}$$

where Ψ is a known linear sparsifying transform. In practice, error is allowed in the data agreement constraint to account for noise and imperfections in the system matrix. Equation (22) becomes

$$minimize \quad \|\Psi \boldsymbol{f}\|_1 \ \ subject\ to \, \|\boldsymbol{Hf} - \boldsymbol{g}\|_2 \le \varepsilon . \tag{23}$$

This is a constrained convex multiplication problem and can be solved by a variety of methods. Other constraints and penalty terms can be included in a CS framework. A CS-based or sparsity-exploiting algorithm will depend on the sparsifying transform, the optimization procedure used and any penalties and constraints added to equation (22). CS theory has been applied to both CT and MRI [3], [40], [44]. However, true CS methods cannot be applied for CT because the systems are not truly incoherent; however, algorithms for few-view reconstruction for CT have been developed based on this sparsity-exploiting theory [1–3], [45], [46]. Few-view, sparsity-exploiting CT reconstruction algorithms encourage sparsity by minimizing image total-variation (TV), using a discrete gradient magnitude operation as Ψ. These algorithms assume that the X-

Ray attenuation coefficient maps are approximately piecewise constant, an assumption that may not be valid for all SPECT objects.

CHAPTER 3: PERFORMANCE OF A SPARSITY-EXPLOITING ALGORITHM FOR FEW-VIEW SPECT RECONSTRUCTION

The feasibility of reconstructing from angularly undersampled, or few-view data, has recently been explored for CT [1–3], [45], [46]. These investigations are based on exploitation of gradient-magnitude sparsity, an idea promoted and theoretically investigated in the field of Compressed Sensing (CS). Few-view, sparsity-exploiting CT reconstruction algorithms promote gradient-magnitude sparsity by minimizing image total variation (TV). Success of these algorithms in allowing sampling reduction follows from an object model which is approximately piece-wise constant, a model that may not apply well for SPECT objects. The SPECT object function reflects the uptake of radio-labeled tracer in the body. This function is expected to have rapid, almost step-like, dependence at the borders of hot spots, but the transition is likely not as sharp as the X-ray attenuation coefficient map at tissue borders. The goal of this work is to modify the idea of exploiting gradient-magnitude sparsity to allow for softer transitions between regions of approximately constant values of radio-isotope concentration.

This chapter proposes an iterative algorithm for few-view SPECT reconstruction that allows for smoothed, step-like variation within the object by phenomenologically modeling the SPECT image as a blurred version of a piece-wise constant object and minimize TV of an intermediate piece-wise constant image to reconstruct a SPECT image. Using this model, a first-order primal-dual technique is implemented as an iterative procedure [47], [48]. The purpose of this study is to characterize the performance of the algorithm under varying sampling and noise conditions. Images reconstructed by Maximum-Likelihood Expectation Maximization (MLEM) serve as a

reference.

3.1 The Algorithm

The iterative image reconstruction algorithm (IIR) is designed by defining an optimization problem which implicitly specifies the object function based on a realistic data model and a model for object sparsity. In this preliminary investigation, the specified optimization problem is solved in order to characterize its solution and the solution's appropriateness for few-view/dynamic SPECT imaging. Future work will consider algorithm efficiency by designing IIR for approximate solution of the proposed optimization problem.

3.1.1 The SPECT Optimization Problem

The proposed SPECT optimization problem is formulated as an unconstrained minimization of an objective function which is the sum of a data fidelity term and an image regularity penalty. The design of both terms expresses the proper SPECT noise model and a modified version of gradient-magnitude object sparsity. We first describe how standard gradient-magnitude sparsity is incorporated into a SPECT optimization problem, and then we present our modified optimization which accounts for the smoother variations expected in a SPECT object function.

3.1.1.a Unconstrained Minimization for Gradient-Magnitude Sparsity Exploiting SPECT IIR

In expressing the SPECT data fidelity term, the data are modeled as a Poisson process the mean of which is described by the following linear system of equations:

$$\boldsymbol{g} = \boldsymbol{Hf}, \tag{24}$$

where $\boldsymbol{H}$ is the system matrix that describes the probability that a photon emitted from a certain location in the object vector, $\boldsymbol{f}$, contributes to the measured data vector, $\boldsymbol{g}$, at a certain location. Iterative tomographic image reconstruction techniques such as MLEM and Ordered Subset Expectation Maximization (OSEM) aim to maximize the log-likelihood of this Poisson random variable [13], [33], [49]. This is equivalent to minimizing the Kullback-Leibler (KL) data divergence (D_{KL}) [22]. For the present application of few-view SPECT, the data are acquired over too few views to provide a unique maximum likelihood image. In the limit of infinite photon counts and assuming that the mean model in equation (24) perfectly describes the imaging system, the underlying object function still cannot be determined because equation (24) is underdetermined.

In order to arrive at a reasonable solution, additional information or assumptions on the object function are needed. Recently, exploitation of gradient-magnitude sparsity has received much attention and has been implemented in IIR for few-view CT [1], [50]. This idea is an example of a general strategy under much recent investigation in CS, where sampling conditions are based on some form of identified sparsity in the image. In our application the strategy calls for narrowing the solution space to only images that exactly solve our linear model in equation (24). Among those images, the solution with the lowest TV is sought. In practice, this solution can be obtained approximately by combining a data fidelity term with a TV penalty, where the combination coefficient in

front of the TV penalty is vanishingly small. The TV-D_{KL} sum yields the following minimization:

$$minimize_f \{D_{KL}(\boldsymbol{g}, \boldsymbol{Hf}) + \gamma \|(|\boldsymbol{Df}|)\|_1 \}, \tag{25}$$

where $\boldsymbol{D}$ is a discrete gradient operator and γ is a weighting parameter. For sparsity-exploiting IIR, under ideal conditions, γ is chosen so that the data fidelity term far outweighs the TV-term. The role of the TV term is simply to break the degeneracy in the objective function among all solutions of equation (24).

The success of TV minimization for few-view CT IIR relies on the assumption that the X-ray attenuation coefficient map is approximately piecewise constant. Directly promoting sparsity of the gradient-magnitude image may not be as beneficial for SPECT, as in some cases tracer uptake may vary smoothly within objects, and borders of objects may show a smoothed step-like dependence. For example, some regions of the heart are supplied by a single coronary artery while other regions are supplied by multiple coronary arteries [51], [52]. Thus, cardiac perfusion studies may be one application for which the blurred piecewise constant model is appropriate. As another example, tumor vascularization is heterogeneous, with vascularization often varying from the tumor center to the periphery [53]. Therefore, our goal here is to find a sparsity-exploiting formulation which allows some degree of smoothness between regions with different uptake.

3.1.1.b Unconstrained Minimization for Sparsity Exploiting IIR Using a Blurred Piecewise Constant Object Model

In this work the TV minimization detailed in equation (24) is modified to allow for rapid but smooth variation by phenomenologically modeling objects as piecewise constant subject to a shift-invariant blurring operation.

The additional blurring operation can be incorporated into the framework developed above by minimizing the weighted sum of the TV of an intermediate piecewise constant object estimate and D_{KL} between the measured data and the projection data of the blurred object estimate. The modified TV-minimization problem becomes:

$$minimize_f \ \{D_{KL}(\mathbf{g}, \boldsymbol{Hu}) + \gamma \, \|(|\boldsymbol{Df}|)\|_1\} \ , \tag{26}$$

where $\boldsymbol{u}$ is the object estimate and $\boldsymbol{f}$ is an intermediate image with sparse gradient-magnitude. These are related by $\boldsymbol{u} = \boldsymbol{MGMf}$, where $\boldsymbol{M}$ is a support preserving image mask and $\boldsymbol{G}$ is a Gaussian blurring operation with standard deviation r. The operators $\boldsymbol{M}$ and $\boldsymbol{G}$ are symmetric so $\boldsymbol{M}^T = \boldsymbol{M}$ and $\boldsymbol{G}^T = \boldsymbol{G}$. The operator $\boldsymbol{G}$ extends data outside the physical support of the system assumed by $\boldsymbol{H}$ so the image mask $\boldsymbol{M}$ must be applied before and after $\boldsymbol{G}$. This optimization problem has two design parameters, γ, which is the weighting of the TV term, and r, which is the standard deviation of the Gaussian blurring kernel. The blurring parameter, r, represents smoothness in the underlying object, as opposed to blurring introduced by the imaging system. When $r = 0$, this formulation defaults to TV minimization problem in equation (25). If $\gamma = 0$, the formulation described by equation (26) minimizes D_{KL}, which is implicitly minimized in MLEM. The final image estimate is $\boldsymbol{u}$, the result of blurring and masking the intermediate piecewise constant object, $\boldsymbol{f}$. This construction enforces sparsity by minimizing the TV of $\boldsymbol{f}$ and encourages data match by minimizing D_{KL}.

3.1.2 Optimization Algorithm

Only recently have algorithms been developed that can be applied to large-scale, non-smooth convex optimization problems posed by equation (26). Sidky *et al.* adapts the Chambolle-Pock (CP) algorithm to solve the TV- D_{KL} sum described by equation (25) [48]. Applying the model as described above, this prototype can be modified to solve the optimization posed by equation (26). Pseudo-code describing this algorithm is written below.

Listing 2: Pseudocode of the proposed algorithm

$L := \|(\boldsymbol{HMGM},\boldsymbol{D})\|_2$; $\tau = \sigma = 0.9/L$; $\theta = 1$; $n = 0$

$\boldsymbol{f}_0 := \boldsymbol{f'}_0 := \boldsymbol{p_0} := \boldsymbol{q_0} := \mathbf{0}$

Repeat

$\boldsymbol{p}_{n+1} := 0.5(1 + \boldsymbol{p}_n + \sigma\, \boldsymbol{HMGMf'}_n - ((p_n + \sigma \boldsymbol{HMGMf'}_n - \mathbf{1})^2 + 4\sigma \boldsymbol{g})^{1/2}$

$\boldsymbol{q}_{n+1} := \gamma(\boldsymbol{q}_n + \sigma \boldsymbol{Df'}_n) / \max(\gamma, |\boldsymbol{q}_n + \sigma \boldsymbol{Df'}_n|)$

$\boldsymbol{f}_{n+1} := \boldsymbol{f}_n - \tau \boldsymbol{MGMH}^{\mathrm{T}} \boldsymbol{p}_{n+1} + \tau div(\boldsymbol{q}_{n+1})$

$\boldsymbol{f'}_{n+1} := \boldsymbol{f}_{n+1} + \theta(\boldsymbol{f}_{n+1} - \boldsymbol{f}_n)$

$n = n+1$

Until *stopping criterion*

This algorithm is a modification of Algorithm 5 described in previous work by Sidky *et al.* [48]. The convergence criterion described in that work was used here.

Simulation studies were conducted to characterize the performance of the proposed reconstruction technique over a range of angular sampling conditions, including cases in which the object does not match the phenomenological blurred piecewise constant model. The first simulation study used noiseless data generated from the system

forward model to validate that the reconstruction technique closely approximates the true object when the correct blurring and system models are used, and to investigate the effects of the design parameters r and γ. Another study reconstructed data generated by Monte Carlo simulation for a range of sampling and noise conditions and for varying values of algorithm parameters r and γ.

3.2 Inverse Crime Simulation Study

This study was designed to validate that the reconstruction technique approximates the true object when both the object model and system model are known exactly. The simulated object was generated from the object model and data were generated from the system forward model. Cases such as this, in which the data were produced directly from the model are referred to as the "inverse crime" [54]. This is investigated in the many-view (128 views) and few-view (9 views) cases. We also examine the effects of different blurring models on the gradient-magnitude sparsity of the intermediate object $\boldsymbol{f}$. The algorithm could enable further reductions in sampling if the blurring model increases the gradient-magnitude sparsity compared to the conventional TV minimization term. In order to investigate the performance of the reconstruction with inconsistent data, Poisson noise was added to the data and the study was repeated. We refer to this as the "noisy" case.

3.2.1 Methods

3.2.1.a Phantom

The intermediate piecewise constant object, $\boldsymbol{f}_{true}$, was defined on a 128 x 128 grid of 1-mm x 1-mm pixels, representing a 6-mm diameter disk embedded in a 76-mm diameter disk. The intensity of the small disk was 2000 arbitrary units and the intensity of the large disk was 200 arbitrary units. A Gaussian blurring kernel with standard deviation, $r_{true} = 0.75$ pixels was applied to this intermediate object to generate the ground-truth object. The intermediate object and the output of the blurring operation, $\boldsymbol{u}_{true}$ are shown in fig. 3.

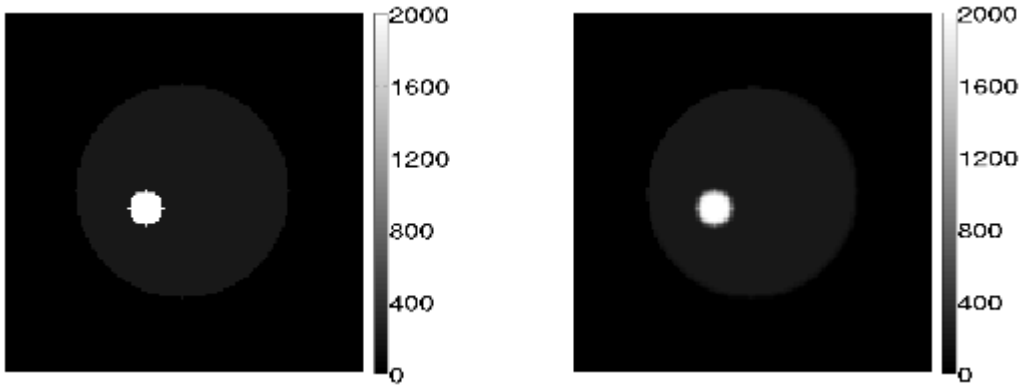

Figure 3. Piecewise Constant Object (left) and Phantom (right) used for simulations that generated data from the system matrix.

3.2.1.b Simulation

Projection data of the pixelized ground-truth object was generated from the system matrix. The system matrix was estimated using Siddon's raytracing algorithm for a single-pinhole SPECT system with 3 mm pinhole diameter, 1.0 mm system FWHM, 35 mm pinhole-to-object distance and 63.5 mm pinhole-to-detector distance [34]. Projection data were generated and reconstructed using 128 views, 60 views, 21 views, 15 views

and 9 views, uniformly distributed around 360 degrees.

A parametric sweep was performed to investigate the effects of the two parameters on the reconstructions: the TV weighting parameter, γ, and the standard deviation of the Gaussian blurring kernel, *r*. Reconstructions were performed with γ varying from 0.0001 to 1.0 and *r* varying from 0 to 2.0 pixels. For this case, r_{true} is known to be equal to 0.75. In practice, the amount of smoothness within the underlying object is unknown and may vary across the FOV. In this study, images are reconstructed using a range of *r* values to quantify the performance of the reconstruction technique for the expected case where the assumed *r* differs from r_{true}. To reduce the necessary sampling for accurate image reconstruction, a sparse representation of an image must exist. Our proposed reconstruction approach assumes that the gradient-magnitude of the intermediate image $\boldsymbol{f}$ has very few meaningful coefficients. However, using an incorrect blurring model in the reconstruction may negatively affect the sparsity of the intermediate object, $\boldsymbol{f}$, limiting the benefits of the algorithm. To investigate the effect of the assumed blurring model on the sparsity of the reconstructed intermediate object, $\boldsymbol{f}$, images were reconstructed from 9 and 128 views using a range of *r* values and sparsity evaluated as the number of coefficients greater than 10% of the maximum coefficient in the gradient-magnitude image of $\boldsymbol{f}$.

To investigate the performance of the reconstruction technique in the presence of noise, simulations varying the number of views and parameter values were repeated with Poisson noise added to the projections generated from the system model. All simulations modeled approximately 1052000 counts, thus the peak number of counts in the 128 view

projections was 298 while the peak number of counts in the 9 view projections was 3758. The noisy projection data were also reconstructed with MLEM in order to provide a reference reconstruction for comparison. As will be described in the next sub-section, the correlation coefficient (CC) of the reconstructed image with the true object is used as a metric of accuracy throughout this work. In order to select a comparable stopping iteration for MLEM reconstruction, the CC was calculated at each MLEM iteration and the final image selected as that with the highest CC value.

3.2.1.c Metrics

Evaluating the accuracy of the reconstructed object requires a measure of similarity or error between the reconstructed object and the true object. In SPECT imaging, including the GATE simulations proposed in section 3.3, the reconstructed activity is a scaled version of the true activity, with the scaling factor dependent on the geometric efficiency of the system. Our reconstruction methods correct for the spatially varying sensitivity of the SPECT system, as will be described in section 3.3.1.b. However, a global scaling correction factor is not applied because absolute quantification in SPECT is challenging and may confound the characterization of the algorithm. Therefore, our accuracy metric must provide a meaningful measure of similarity in cases where the scaling factor between the reconstructed and true object is unknown. In this work, reconstruction accuracy was quantified using the correlation coefficient (CC) of the reconstructed image estimate with the true object. CC is defined as

$$CC = \frac{\sum_{k=1}^{M} (u(k) - \bar{u})(u_{true}(k) - \overline{u_{true}})}{\{\sum_{k=1}^{M} (u(k) - \bar{u})^2 \sum_{k=1}^{M} (u_{true}(k) - \overline{u_{true}})^2\}^{1/2}} \quad , \tag{27}$$

where u_{true} is the true object, M is the number of voxels and $u(k)$ is the reconstructed object value at voxel k. This metric is commonly used in image registration and is the optimum similarity measure for images that vary by a linear factor [55]. This metric allows the quantification of the accuracy of the spatial distribution of the object, without requiring absolute quantitative accuracy. CC is equal to one when the reconstructed object matches the true object. We also quantified the change in CC over the range of studied parameters (r and γ), in order to quantify the sensitivity of the algorithm to parameter selections and to understand the performance of the algorithm when the assumed blurring parameter does not match the true object blur. Spatial resolution in the reconstructed images was quantified as the full-width at 10% of maximum (FW10M) of the central profile through the smaller disk. This measure was used instead of the more common full-width at half maximum (FWHM) because analysis of preliminary reconstructed images indicated that the FWHM was often accurate, even though the extent of the reconstructed object was greater than the true object. The FW10M more accurately quantified this blurring effect. The true object had a FW10M of 12 pixels. Signal-to-noise ratio (SNR) was calculated as the mean of a 3 pixel radius region in the background divided by the standard deviation of the same region.

3.2.2 Results

3.2.2.a Without Poisson Noise

We present results of the noiseless case in which the object was constructed from the object model and the projections were determined from the system matrix. The purpose of this study was to confirm that the reconstruction algorithm closely approximated the true object in the noiseless inverse crime case and to examine the effects of the design parameters r and γ as the number of views decreased. Both design parameters were varied and the number of angular samples reduced from 128 views to 9 views.

Reconstructions from 128 angular positions are shown in fig. 4, with profiles of selected reconstructions shown in fig. 5. Fig. 6a presents plots of CC over the range of studied γ and *r* values Reconstruction accuracy (CC) is high (CC > 0.980) for reconstruction using $\gamma < 1.0$, with CC varying by less than 2% for all *r* investigated. Using $\gamma = 0.0001$ or $\gamma = 0.01$ and using r_{true} ($r = 0.75$), the object is recovered nearly exactly with CC exceeding 0.999 in each case. The FW10M value of the true object is 12 pixels, which is correctly depicted by reconstructions using $r < 1.0$ and $\gamma < 0.1$. The profiles demonstrate decreased amplitude and increased object extent when $\gamma = 0.1$ or $\gamma = 1.0$, suggesting blurring of the object. For the lower γ cases, ring artifacts are visible when *r* is greater than 1.0.

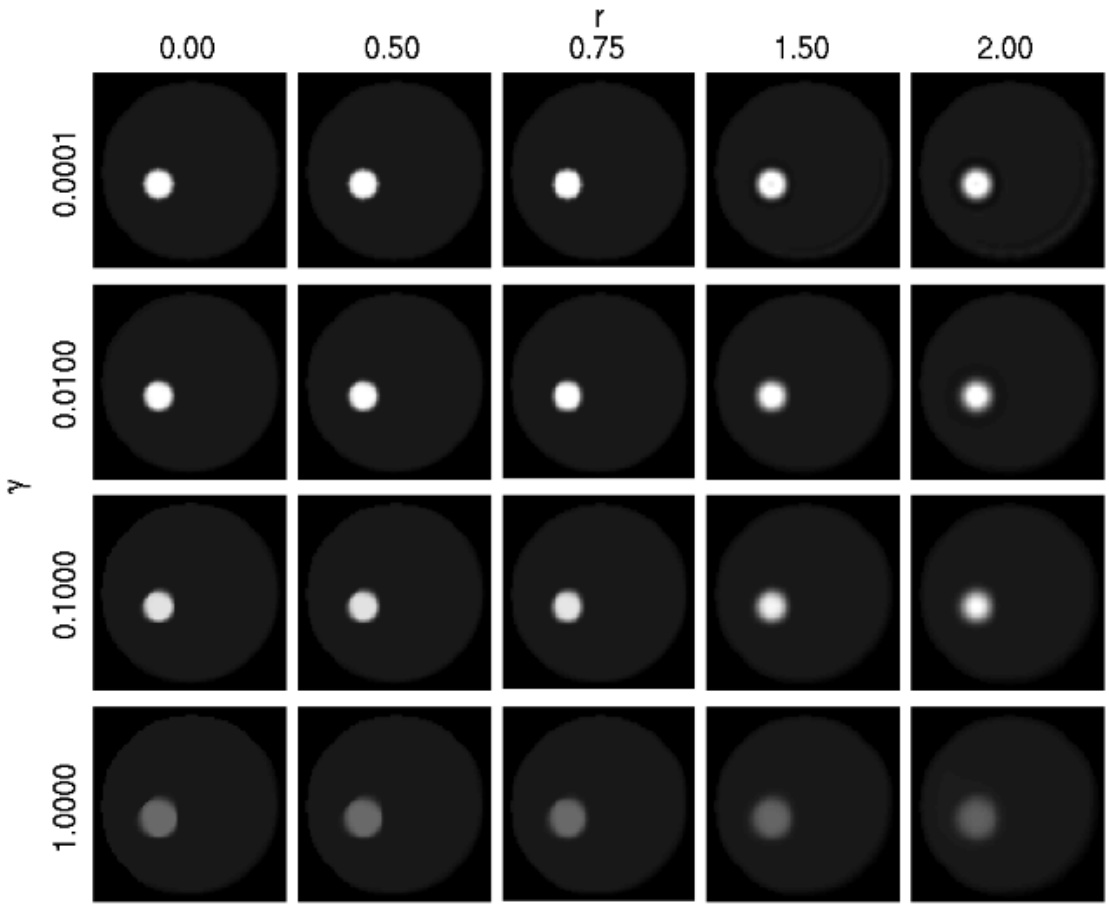

Figure 4. Images reconstructed from 128 views of noiseless inverse crime data using the proposed algorithm with varying values of r and γ.

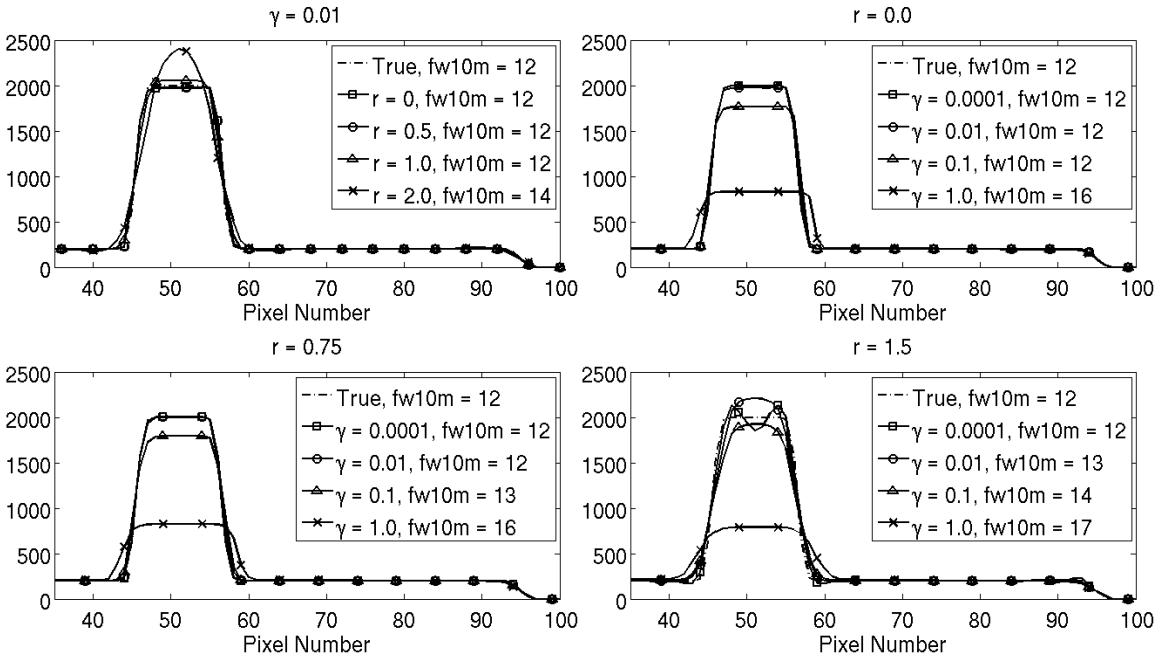

Figure 5. Central diagonal profiles through images reconstructed from noiseless inverse crime data from128 views using the proposed algorithm with varying values of *r* and γ.

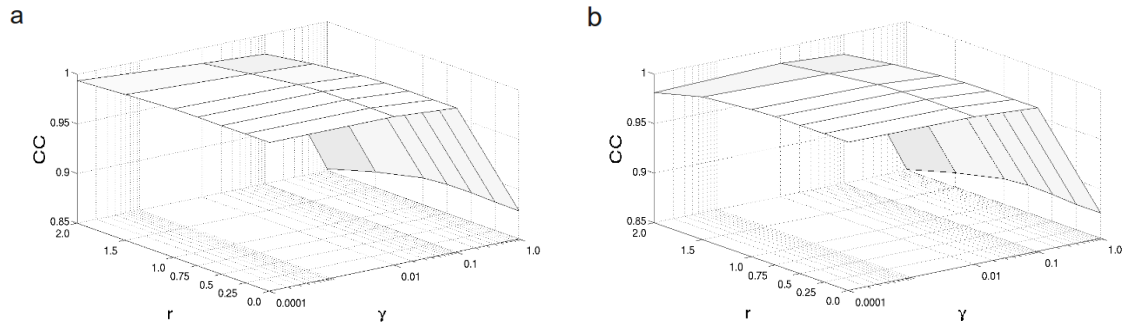

Figure 6. Plots depicting the CC over the range of studied *r* and γ parameters of images reconstructed from noiseless inverse crime data from 128 views (a) and 9 views (b).

The few-view case demonstrated similar trends, as shown in figs. 6b, 7 and 8. The object is nearly exactly recovered when γ = 0.01 and *r* = 0.75 is used. Using γ = 0.01, the CC varied by 1.5% (CC ranging from 0.984 - 0.999) across the range of studied *r* values. Over the parameter set studied, CC varied between 0.878 and 0.999 depending

on the value of γ used in reconstruction, with higher γ values resulting in lower CC. In addition to lower CC, images reconstructed with $\gamma = 1.0$ demonstrated reduced contrast and increased FW10M results, suggesting increased blurring. The FW10M value of the true object was 12, which was depicted in all reconstructions with $r \leq 1.0$ and $\gamma < 0.1$. As r increased beyond 1.0, the peak value increased and the profiles demonstrate larger spread, resulting in the lowest CC for reconstructions with $r = 2.0$. Overall, in both the 128 and 9-view case, CC values demonstrated a larger range over the set of γ values compared to r values, suggesting that the reconstruction technique is more sensitive to the selection of γ than r.

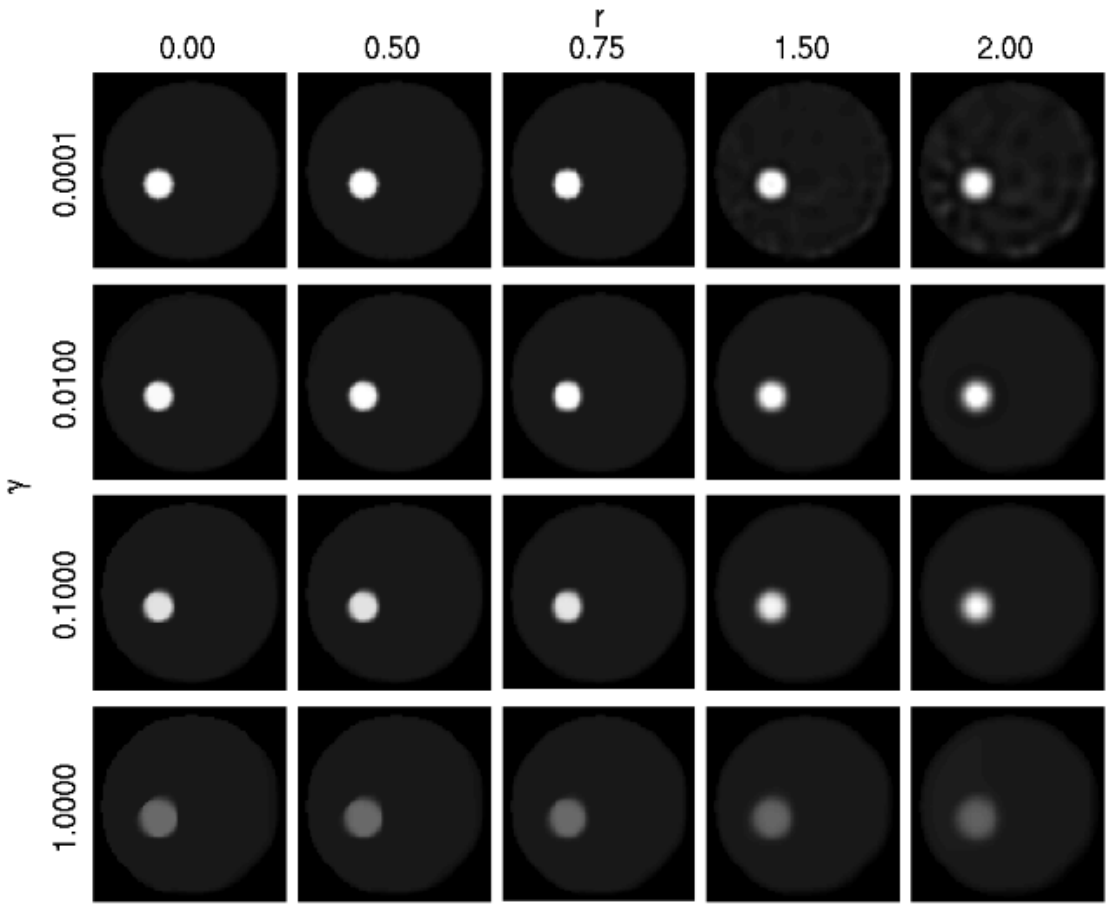

Figure 7. Images reconstructed from 9 views of noiseless inverse crime data using the proposed algorithm with varying values of r and γ.

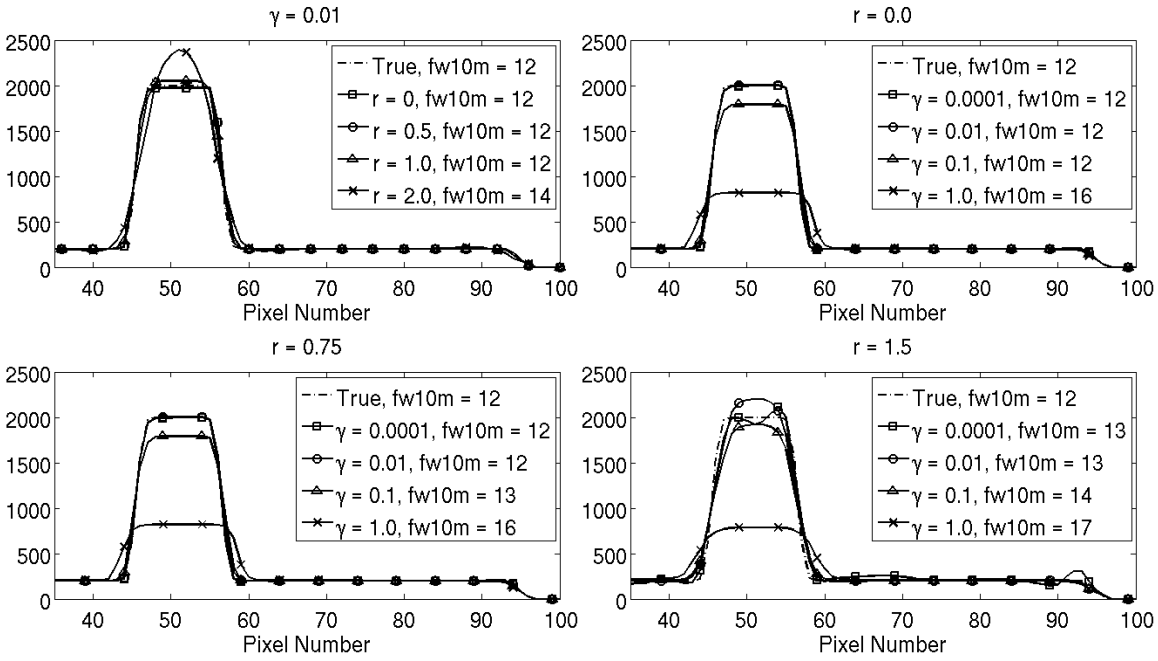

Figure 8. Central diagonal profiles through images reconstructed from noiseless inverse crime data from 9 views using the proposed algorithm with varying values of r and γ.

Evaluating Gradient Magnitude Sparsity of the Intermediate Image

This section evaluates the sparsity of the intermediate image $\boldsymbol{f}$ reconstructed from many-view and few-view data. Fig. 9 shows images of the intermediate image, $\boldsymbol{f}$, reconstructed from both 128 views and 9 views using different r and $\gamma = 0.0001$. Each image is captioned by its sparsity value (number of meaningful coefficients).

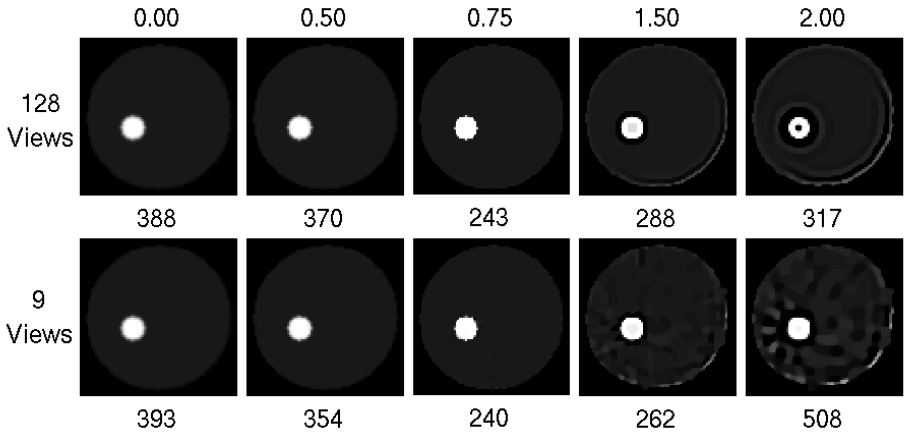

Figure 9. Intermediate images $\boldsymbol{f}$ and the number of meaningful sparsity coefficients reconstructed from128 and 9 noiseless inverse crime data using $\gamma = 0.0001$.

In both the many-view and few-view case, the image reconstructed with the true blurring model ($r = 0.75$) was the most sparse and as the r assumed by the algorithm diverged from r_{true} the images became less sparse. This indicates that using the correct blurring model may allow the greatest sampling reductions. Additionally, underestimating r leads to a gradual increase in the number of meaningful coefficients. In the few-view case, over-estimating r leads to a rapid increase in the number of meaningful coefficients, reflected by the fact that new structure enters the image. These artifacts survive the blurring with $\boldsymbol{G}$, leading to artifacts in the presented image $\boldsymbol{u}$.

3.2.2.b With Poisson Noise Added

We next considered data generated by the system matrix with the addition of Poisson noise. The purpose of this study was to examine the effects of noise on the reconstructions, using data from an otherwise inverse crime case, in which the object model and system model are known exactly. Fig. 10 shows images reconstructed from

128 views over the range of r and γ parameters, with profiles plotted in fig. 11. Fig. 12a shows the plot of the CC metric over the range of studied parameters. As in the noiseless case, the CC varied by less than 1.5% across the studied r values for $\gamma > 0.0001$. Unlike the noiseless case, when $\gamma = 0.0001$, the CC increased from 0.867 to 0.988 as r increased from 0.0 to 2.0, as the increased blurring provided additional regularity and noise reduction. Noise is also reduced as γ is increased, due to the increased weighting of the TV term. The highest CC value of 0.999 occurred when $r = 0.75$, the true value of r, and $\gamma = 0.01$. As in the noiseless case, contrast and spatial resolution decreased with increasing γ. The FW10M ranged from 12-14 for $\gamma < 1.0$, compared to a true value of 12.

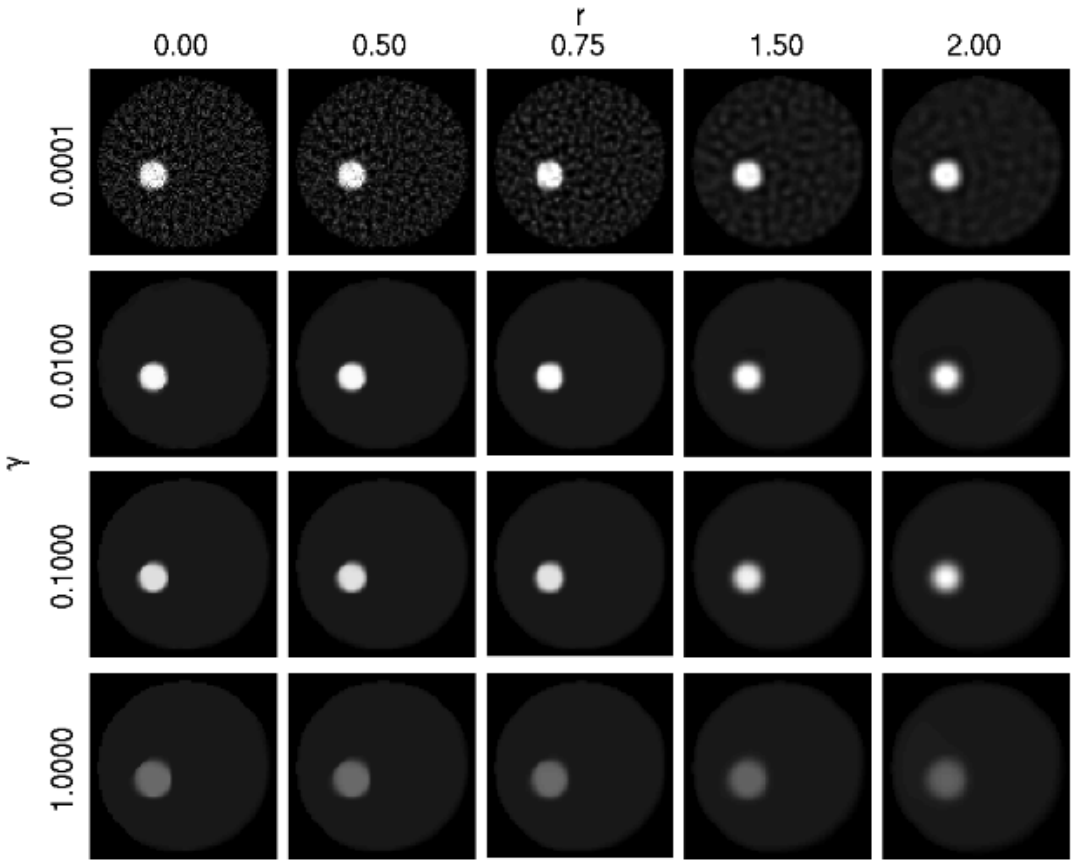

Figure 10. Images reconstructed from 128 views of noisy data using the proposed algorithm with varying values of r and γ. For these images, the projection data were generated by the system matrix.

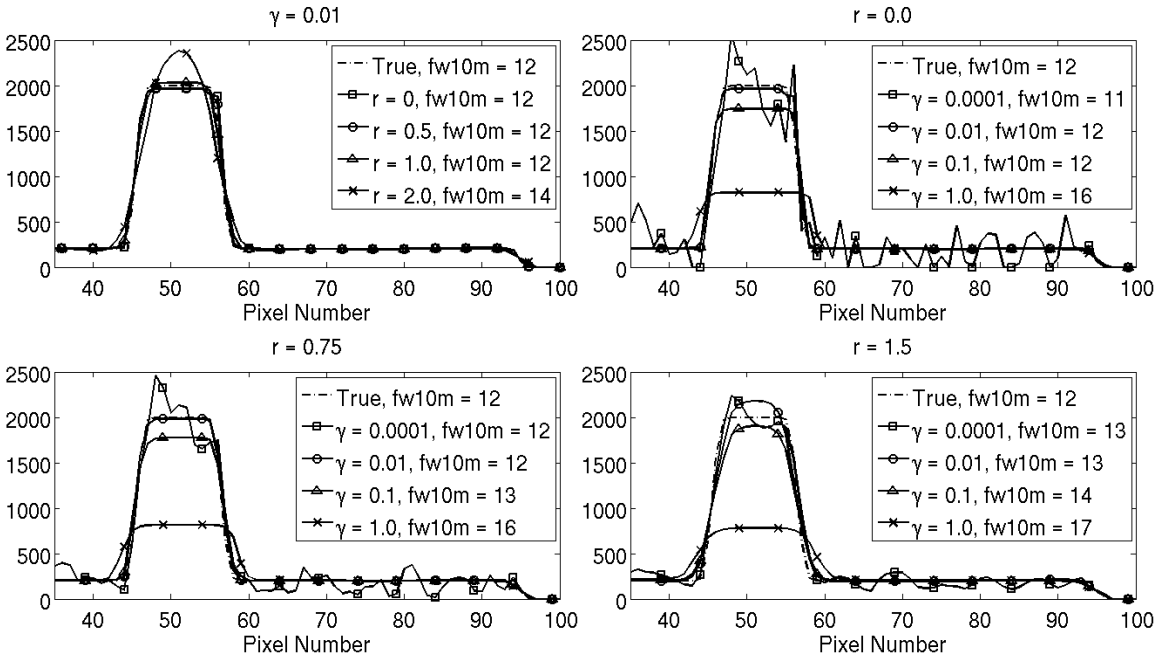

Figure 11. Central diagonal profiles through images reconstructed from noisy inverse crime data from128 views using the proposed algorithm with varying values of *r* and γ. For these images, the projection data were generated by the system matrix.

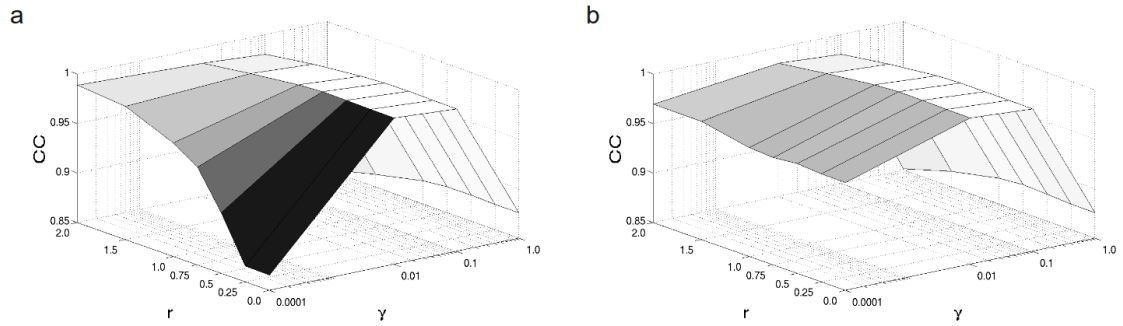

Figure 12. Plots depicting the CC over the range of studied *r* and γ parameters of images reconstructed from noisy data from 128 views (a) and 9 views (b). For these images, the projection data were generated by the system matrix.

Figs. 12b, 13 and 14 display the images, profiles and plots for noisy images reconstructed from nine views. Similar trends were observed as in the reconstructions from 128 views. Images reconstructed with low γ values (γ = 0.0001) demonstrated increased noise and streaking artifacts, which were reduced with increasing *r*. For

$\gamma = 0.01$, the highest CC occurred when the assumed blurring model match the true object ($r = 0.75$), with CC varying by less than 1.4% across the range of r values. The highest CC value of 0.997 was obtained with $\gamma = 0.01$ and $r = 0.75$. FW10M values were similar to the results using 128 views, with the exception of increased FW10M (14-15) when $\gamma = 0.0001$.

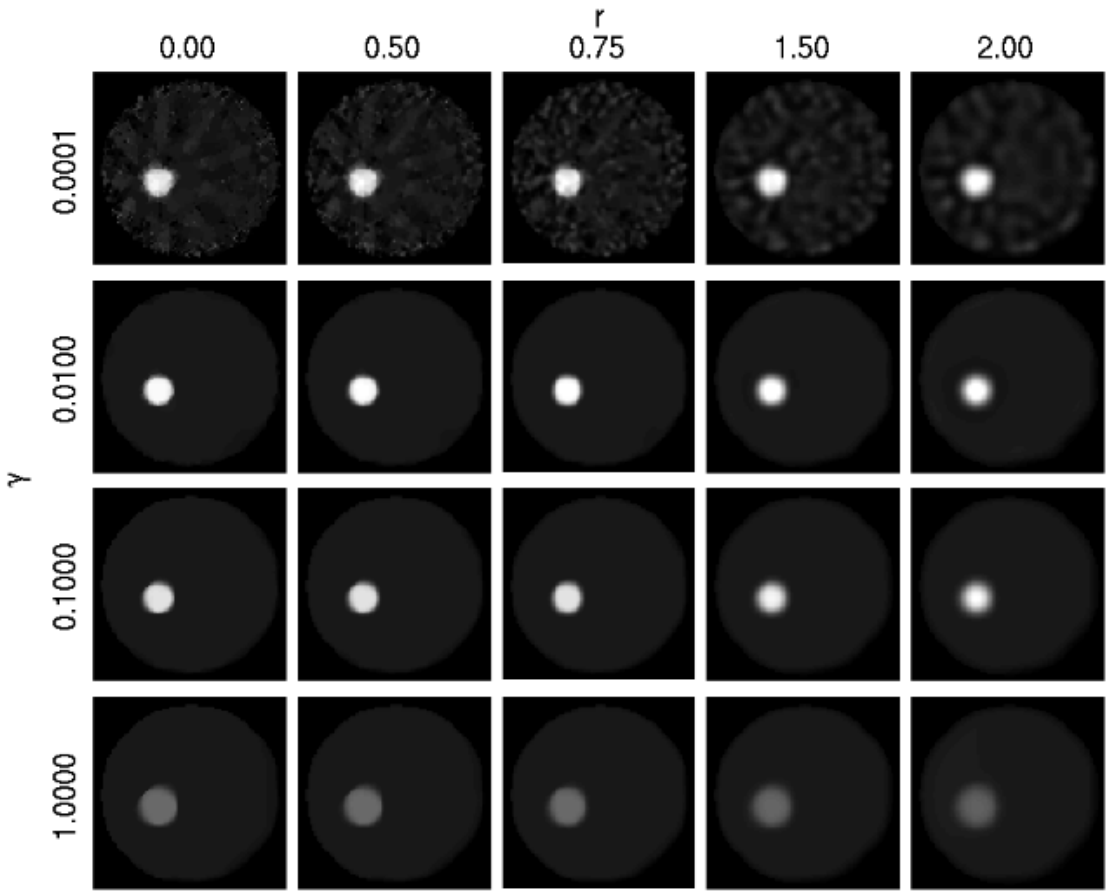

Figure 13. Images reconstructed from 9 views of noisy data using the proposed algorithm with varying values of r and γ. For these images, the projection data were generated by the system matrix.

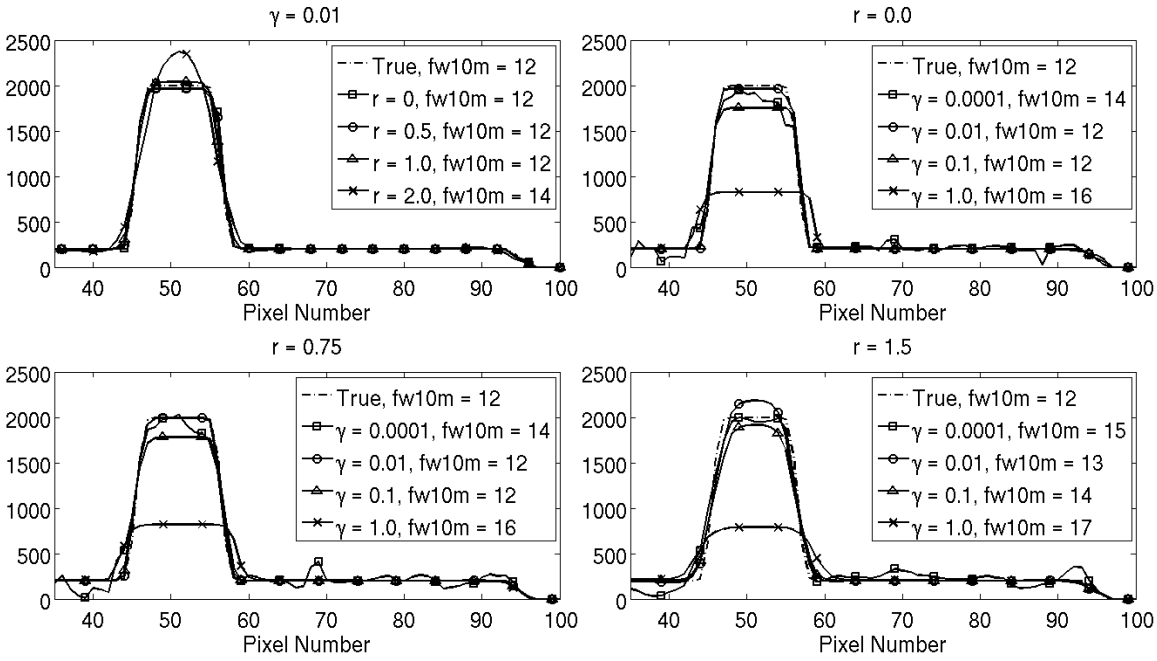

Figure 14. Central diagonal profiles through images reconstructed from noisy inverse crime data from 9 views using the proposed algorithm with varying values of r and γ. For these images, the projection data were generated by the system matrix.

Overall, as in the noiseless case, CC showed little variation with r but greater variation with γ, and both the 9- and 128-view reconstructions suggest that $\gamma = 0.01$ provides the highest CC. The lowest CC value in both the high- and few-view cases occurred with large values of r and γ ($r = 2.0$ and $\gamma = 1.0$).

Fig. 15 compares images reconstructed with the proposed reconstruction technique ($\gamma = 0.01$ and $r = 0.75$) and MLEM from data acquired with a varying number of angular views. Table I shows CC, SNR and FW10M values for each reconstruction technique and number of view. Images reconstructed using the proposed algorithm had CC values that were 2-4% higher for each case compared to MLEM. The greatest difference is noted for the 9 view case where the image reconstructed using the proposed algorithm yielded a CC of 0.994 while the MLEM image had a CC of 0.954. Streak

artifacts were present in the MLEM reconstructions, and were primarily absent in images reconstructed with the proposed algorithm. The noise level in MLEM reconstructions is higher, leading to lower SNR. Both algorithms provide similar FW10M values compared to the true value of 12.

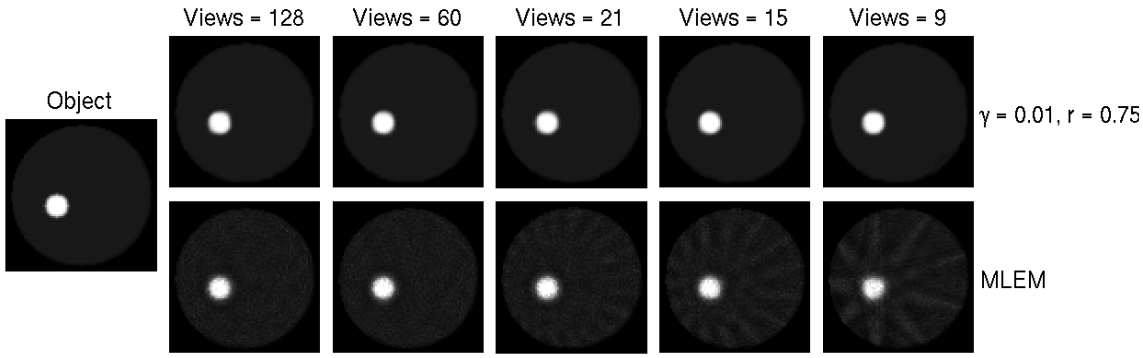

Figure 15. Images reconstructed from noisy projections using the proposed algorithm and MLEM for varying sampling cases. For these images, the projection data were generated by the system matrix.

Table I. Comparison of image quality metrics from images reconstructed from noisy projections generated by the system matrix.

		128 views	**60 views**	**21 views**	**15 views**	**9 views**
$\gamma = 0.010$ $r = 0.75$	**CC**	0.999	0.998	0.998	0.998	0.999
	SNR	314.7	736.5	154.14	615.16	61.17
	FW10M	12	12	12	12	12
MLEM	**CC**	0.986	0.987	0.984	0.981	0.973
	SNR	6.5	6.13	5.38	6.53	6.31
	FW10M	13	13	14	12	15

3.3 Monte Carlo Simulation Study

The purpose of this study was to characterize the performance of the reconstruction technique for the more realistic case where the object does not necessarily

match the model assumed in reconstruction, and the modeled system matrix is an approximation to the system that generated the data. These cases avoid the inverse crime problem. In addition, these simulations include realistic effects such as scatter, spatially-varying pinhole sensitivity and blurring from the pinhole aperture.

3.3.1 Methods

3.3.1.a Phantom

The object was defined on a 512 x 512 pixel grid of 0.25 x 0.25 mm pixels. The object consisted of a 28 mm-radius disk of background activity containing five contrast elements of varying size, shape, and intensity, as detailed in Table II and displayed in fig. 16. Two, two-dimensional Gaussian distributions with peak intensities 638 Bq and 319 Bq and standard deviations 4 mm and 8 mm truncated to have radius 4.4 mm were embedded in the larger disk. Also included in the phantom were a disk representing a cold region with radius 4.4 mm and one disk with radius 2.2 mm having constant intensity, as detailed in Table II. None of the elements in the phantom were generated by the smoothed piecewise constant model assumed by the reconstruction algorithm, thus representing a challenging reconstruction task.

Table II. GATE Phantom Specifications.

Element	Radius (mm)	Position (mm)	Intensity (relative)
A	28	(0,0)	Constant; 64 Bq/pixel
B	4.4	(-13,6)	Activity = 6.8 MBq; Peak = 638 Bq/pixel; Std Dev = 4 mm
C	4.4	(6,-13)	Activity = 0.49Mbq; Peak = 319 Bq/pixel; Std Dev = 8 mm
D	4.4	(0,15)	0
E	2.2	(18,4)	Constant; 640 Bq/pixel

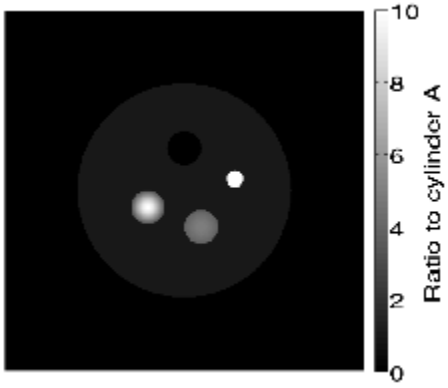

Figure 16. Voxelized phantom used in the GATE studies. The phantom contains contrast elements of varying shape and size as described in Table II.

3.3.1.b Simulations

Projections of the pixelized object were generated using Geant4 Application for Tomographic Emission (GATE) Monte Carlo simulation to model the stochastic emission of photons from a voxelized phantom and their stochastic transmission through the collimator and camera [56]. A three-camera system was simulated. Each collimator was simulated as a 20 mm thick tungsten plate having a 3 mm diameter pinhole with 1.5 mm channel length. A 128 mm x 1 mm NaI crystal was simulated and detected photons binned into 1 mm x 1 mm pixels. Compton scatter, Rayleigh scatter and photoelectric

absorption were included as possible interactions for 140 keV photons. Photons detected outside the 129.5 – 150.5 keV range were rejected as scatter. Electronic noise was not modeled. The system is described in fig. 17 and table III.

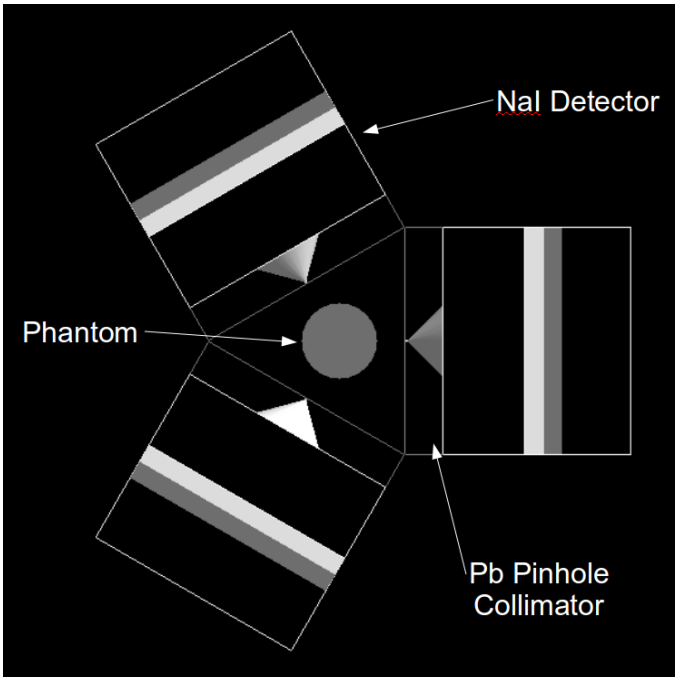

Figure 17. Diagram of Simulated SPECT system

Table III. Specifications of the simulated SPECT system

Camera Size	128 mm x 120 mm x 1 mm
Pinhole diameter	3 mm
Pinhole-to-object distance	35 mm
Pinhole-to-detector distance	63.5 mm

The sensitivity of pinhole collimators depends on the angle of the ray incident on the pinhole. In order to correct for the spatially-varying pinhole sensitivity during reconstruction, a sensitivity map was generated by simulating a flood source on the collimator surface [15]. The resulting projection represents the spatially-varying

sensitivity of the pinhole and was incorporated into the reconstruction algorithm. The sensitivity map was multiplied during each forward projection prior to the summing of data from each ray. Data were multiplied by the sensitivity map prior to backprojection.

Two distinct cases were simulated. In the first case, the total simulated scan time was held constant as the number of views decreased in order to examine the effects of angular undersampling independent of changes in noise. Scans comprising 60, 21, 15, and 9 views distributed over 360 degrees acquired during a 200 second scan were simulated. These data had approximately the same number of total counts in each simulation (~65000 counts). The noise level in SPECT imaging is dependent on the number of detected counts, so the reconstructed images should have similar noise statistics regardless of the number of view angles. The second simulated case held the acquisition time of each view constant across all angular sampling cases. By doing so, the scans that used fewer views had improved temporal sampling, but fewer counts. As the number of views decreased, so did the absolute intensities of the reconstructed images. Images were acquired over 10 seconds for each position of the three-camera gantry, thereby varying the total scan time from 200 seconds for 60 views to 30 seconds for 9 views. In this case, the simulated scan with the fewest views (9) had the fewest counts (~10000 counts) and, consequently, the highest noise level. This represents a more realistic approach for providing dynamic scans with high temporal sampling.

The simulated phantom cannot be described using a constant r across the spatial domain. Each disk has a definite edge and distinct profile. To investigate the effects of varying r in the case where its optimal value is unknown, data were reconstructed using the TV case ($r = 0.0$) and varying r from 0.25 to 2.0 pixels. The TV weighting parameter

was varied from 0.0001 to 1.0. The resulting images were evaluated on the basis of reconstruction accuracy with the CC metric as described in section 3.2.1.c. Spatial resolution was quantified by considering FW10M of a profile through the center of disk E. SNR was calculated as the mean of a 3 pixel-radius region in the background of disk A divided by the standard deviation of the same region. In each case, MLEM reconstructions are also presented as a reference, with the MLEM stopping iteration selected as the iteration with the highest CC value.

3.3.2 Results

In this section we present the results of the Monte Carlo simulations performed over a range of angular sampling schemes for two different cases: constant total scan time and constant scan time per view.

3.3.2.a Constant total scan time

Data reconstructed using the proposed algorithm from 60 views and a variety of r and γ values are presented in fig. 18, with profiles presented in fig. 19, and plots of CC in fig. 20.

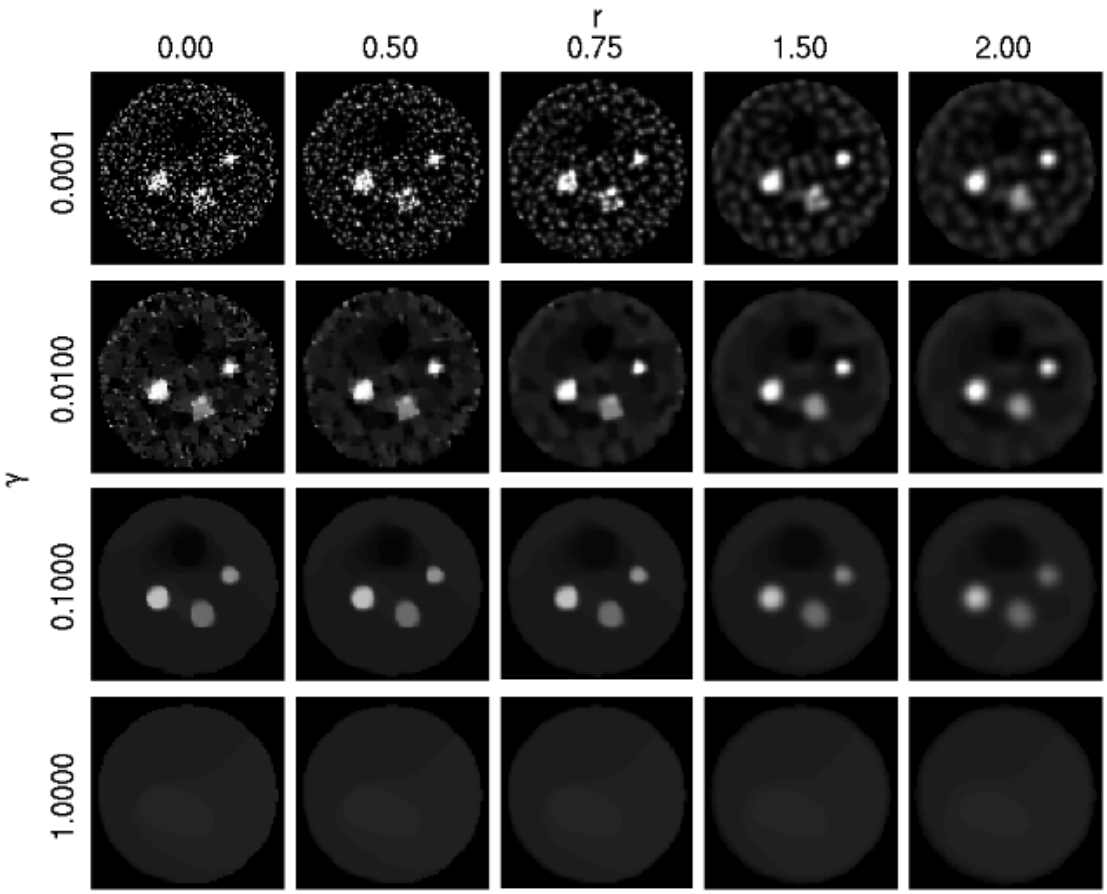

Figure 18. Images reconstructed from 60 views of GATE data simulated for 200 seconds using the proposed algorithm with varying values of r and γ.

When the TV weighting parameter was small ($\gamma = 0.0001$), the resulting image contained high frequency noise; when the TV weighting parameter was large ($\gamma = 1.0$), the object was blurred and contrast reduced. The remainder of the results will focus on $\gamma = 0.1$ and $\gamma = 0.01$. With $\gamma = 0.1$, the CC of the images with the true object varied by 3% across the studied r values, with CC equal to 0.942 at $r = 0.75$ and CC = 0.910 at $r = 2.0$. When $\gamma = 0.1$, the reconstructed profiles do not reach the true peak level for any values of r, as demonstrated in fig. 19. Using $\gamma = 0.01$, the profiles reach a higher peak but the CC

of these images varies by 11% from 0.946 when $r = 1.5$ to 0.836 at $r = 0.0$. As seen in fig. 19, the MLEM reconstructions also did not reach the peak values of the true object profiles, suggesting that this error may be caused by system blurring rather than the reconstruction algorithm. Using $\gamma = 0.01$, the r value that yields the optimal image (in terms of CC) is 1.5, compared to an optimal r value of 0.75 when $\gamma = 0.1$. The FW10M for the image reconstructed using $\gamma = 0.01$ and $r = 1.5$ was 7 pixels, compared to a FW10M of 8 pixels resulting from MLEM reconstruction, and a true value of 4. The FW10M for the images reconstructed with $\gamma = 0.1$ and $r = 0.75$ was 8 pixels.

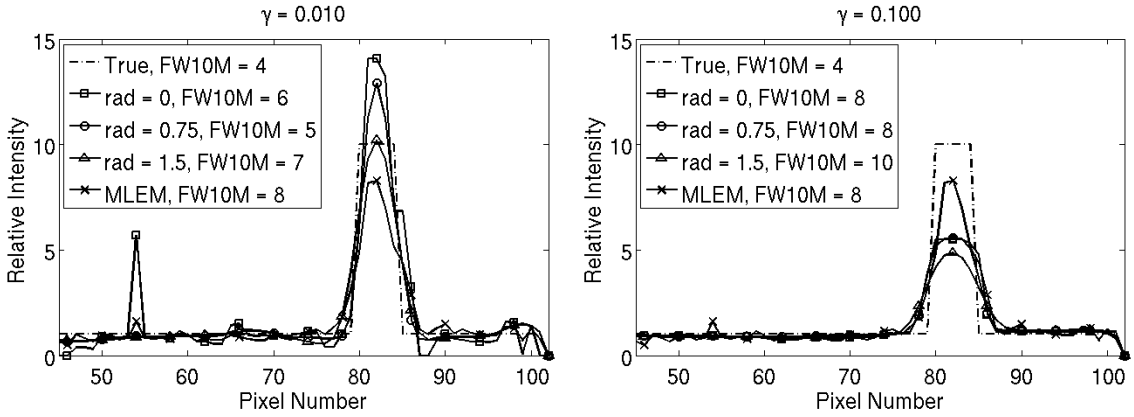

Figure 19. Central vertical profiles through images reconstructed from 60 views of GATE data simulated for 200 seconds using the proposed algorithm with varying values of r and γ.

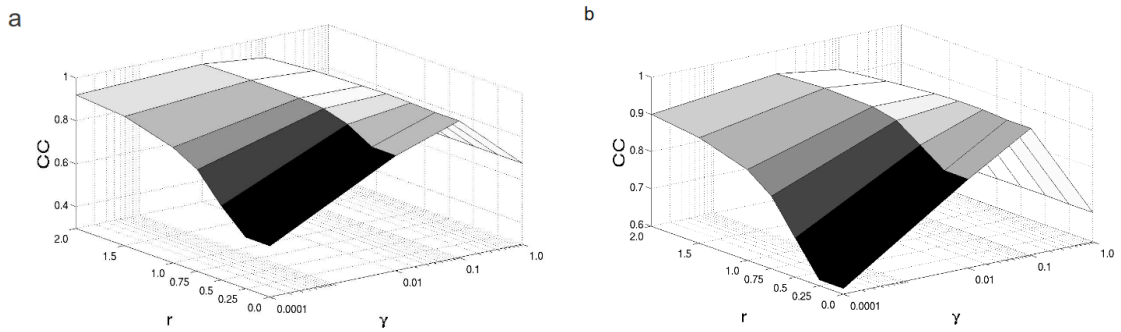

Figure 20. Plots depicting the CC over the range of studied *r* and γ parameters of images reconstructed from GATE data simulated for 200 seconds, using 60 views (a) and 9 views (b).

The images reconstructed from 9 views demonstrated behavior similar to images reconstructed using 60 views. Images are shown in fig. 21. Images reconstructed using γ = 0.01 contained more noise than images reconstructed using γ = 0.1. The CC of images reconstructed with γ = 0.01 vary by 11.3% from 0.946 (*r* = 1.5) to 0.839 (*r* = 0.0), depending on the value of *r*. Images using γ = 0.1 had a lesser dependence on *r*, varying by 3.1% from a 0.945 peak at *r* = 0.75 to 0.915 at *r* = 2.0. Images using γ = 0.01 and *r* = 1.5 have a FW10M value of 8, compared to the true FW10M value of 4 and a FW10M of 8 resulting from MLEM reconstruction. The FW10M of images reconstructed using γ = 0.1 and *r* = 0.75 was 9 pixels. Profiles are shown in fig. 22.

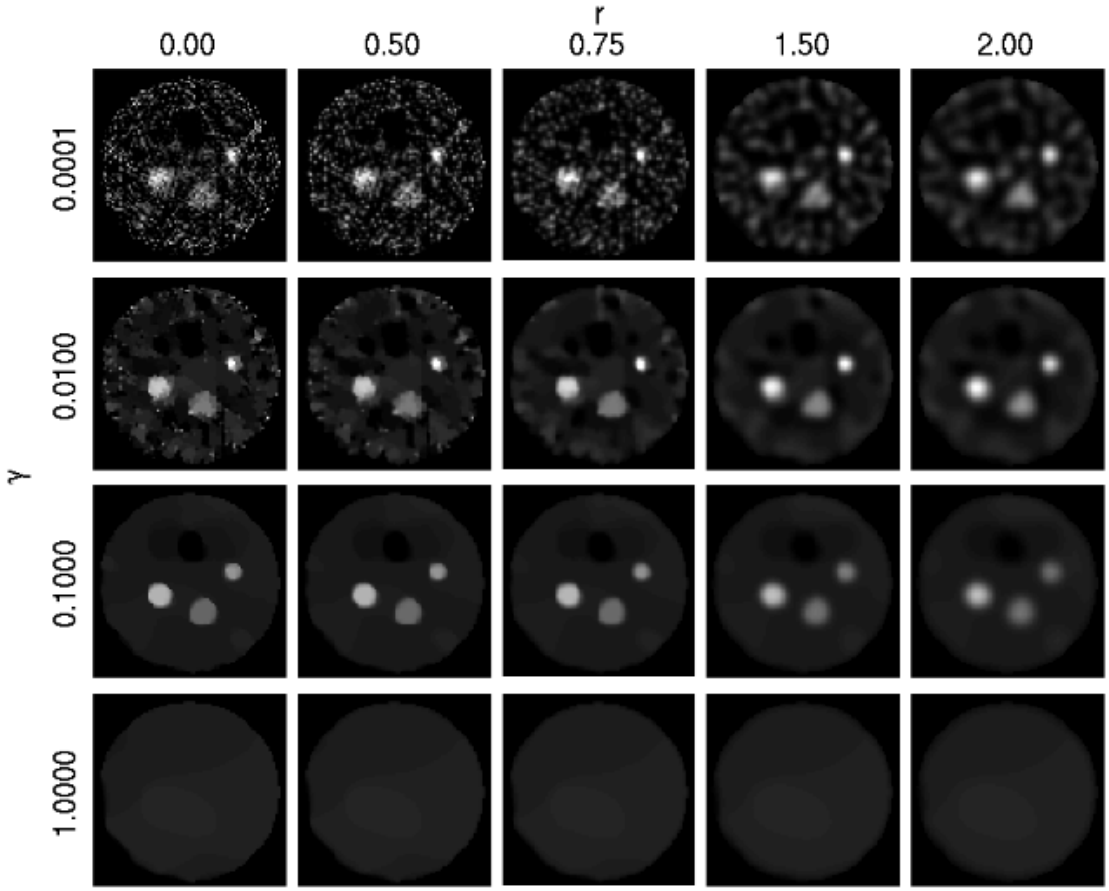

Figure 21. Images reconstructed from 9 views of GATE data simulated for 200 seconds using the proposed algorithm with varying values of r and γ.

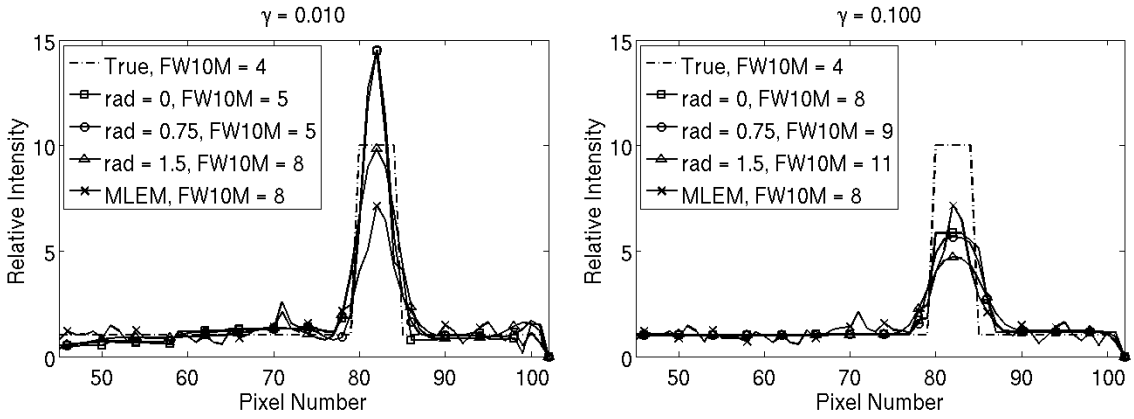

Figure 22. Central vertical profiles through images reconstructed from 9 views GATE data simulated for 200 seconds using the proposed algorithm with varying values of r and γ.

Fig. 23 compares images reconstructed with the proposed algorithm ($\gamma = 0.01$, $r = 1.5$ and $\gamma = 0.1$, $r = 0.75$) and MLEM from data acquired with a varying number of angular views. For images reconstructed using the proposed reconstruction technique with $\gamma = 0.01$ and $r = 1.50$, the CC of the images varied by less than 1% from 0.946 to 0.942 as the number of view decreases from 60 to 9. The CC varied similarly for images reconstructed using $\gamma = 0.1$ and $r = 0.75$. For comparison, the CC of images reconstructed by MLEM decreased 6.5% from 0.913 to 0.854 as the number of views degrease from 60 to 9. For this object, the proposed reconstruction algorithm using both $\gamma = 0.01$ and $\gamma = 0.1$ provided higher CC and lower SNR compared to MLEM for all angular cases, while providing similar FW10M values. Images reconstructed using the proposed algorithm contained low-frequency patchy artifacts in the background due to noise, while reducing the streak artifacts present in MLEM reconstructions from few-views.

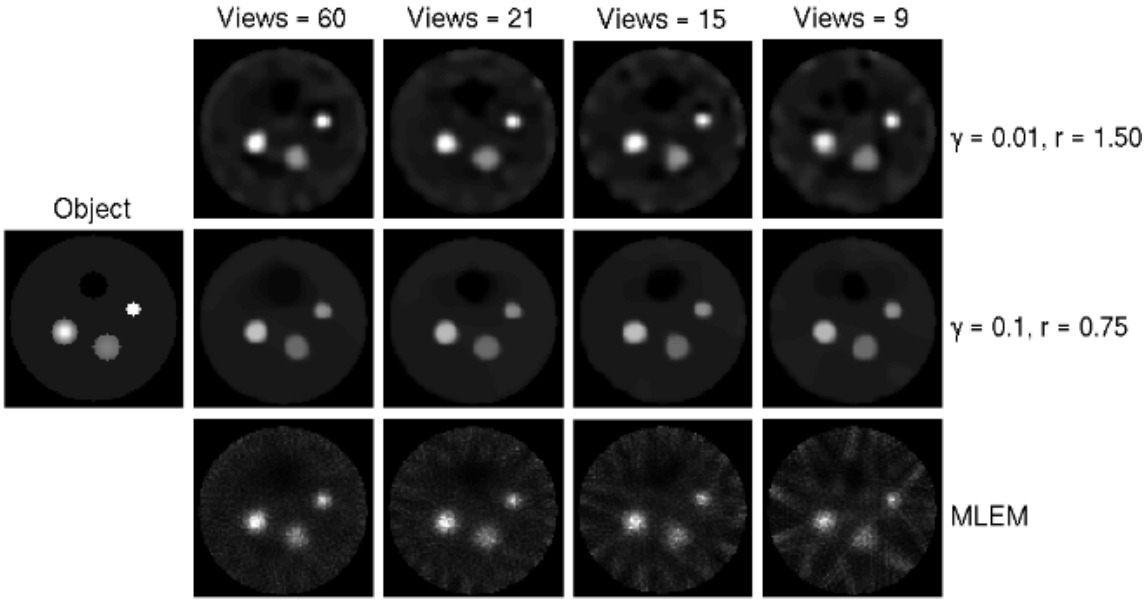

Figure 23. Reconstructions of GATE data simulated for 200s over different numbers of angles using the proposed algorithm and MLEM.

Table IV. Comparison of image quality metrics for images reconstructed from GATE data with the total scan time held constant as the number views decreased.

		60 views	**21 views**	**15 views**	**9 views**
$\gamma = 0.01$, $r = 1.50$	**CC**	0.946	0.945	0.942	0.946
	SNR	17.48	20.57	18.31	6.72
	FW10M	7	8	9	8
$\gamma = 0.10$, $r = 0.75$	**CC**	0.942	0.940	0.941	0.945
	SNR	21.83	24.55	20.95	81.18
	FW10M	8	8	9	9
MLEM	**CC**	0.913	0.901	0.889	0.854
	SNR	4.94	4.92	3.65	4.15
	FW10M	8	11	9	8

3.3.2.b Constant scan time per view

This set of simulations modeled a constant scan time per view (i.e., decreasing total scan time with decreasing number of views), representing the case where temporal sampling improves as the number of views decreases. Figs. 24-26 present images reconstructed from the 9 views with 10 seconds per view (compared to 66.67 seconds per view in figs. 17b, 21 and 22).

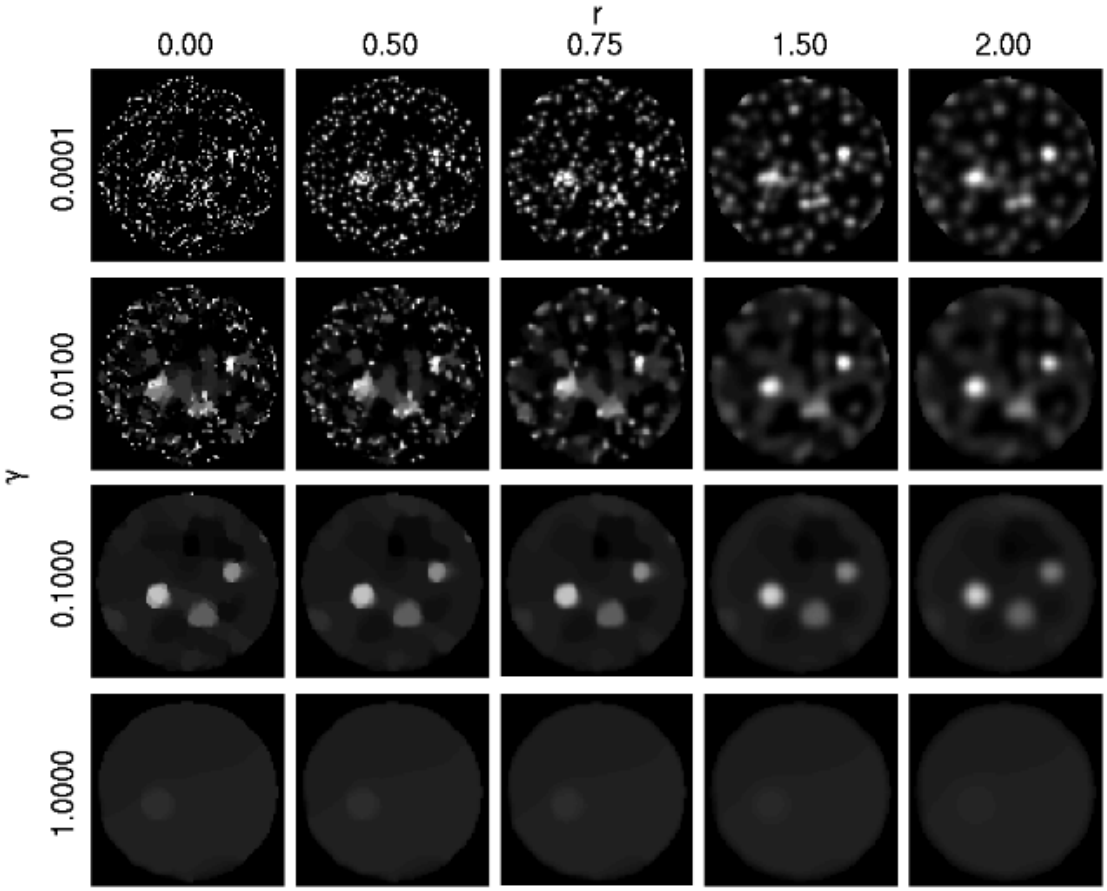

Figure 24. Images reconstructed from 9 views of GATE data simulated for 30 seconds using the proposed algorithm with varying values of r and γ.

Images reconstructed from nine views using $\gamma = 0.01$ had lower reconstruction accuracy (CC < 0.9) compared to the images reconstructed from a 200 second scan time presented in the previous section. Using $\gamma = 0.1$, a maximum CC value of 0.921 occurred when $r = 0.75$. Similar to the 200 second scans, the CC varied by less than 2% across the range of r values for $\gamma = 0.1$. However, unlike the 200 second scans, the 30 second scans showed a larger variation in CC (~40%) across the range of r values for γ not equal to

0.1. In addition to increasing CC, using $\gamma = 0.1$ resulted in reduced noise but increased blurring (higher FW10M) compared to $\gamma = 0.01$.

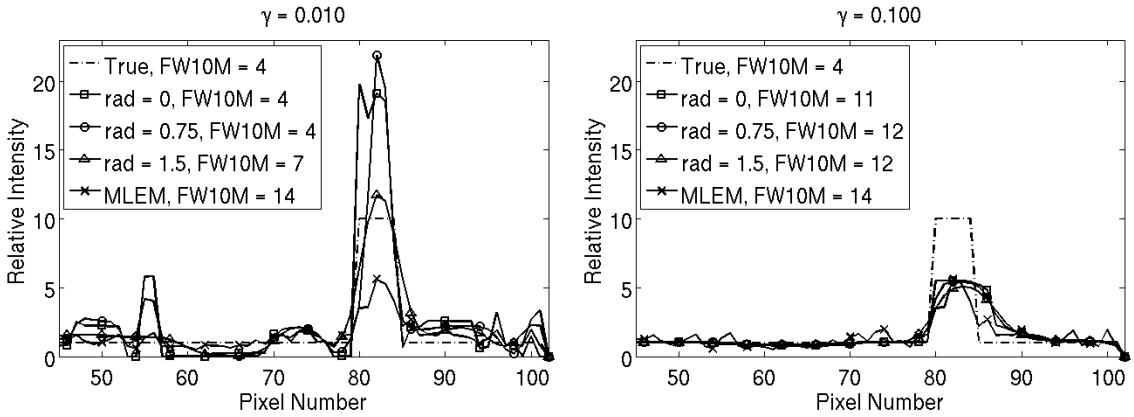

Figure 25. Central vertical profiles through images reconstructed from 9 views GATE data simulated for 30 seconds using the proposed algorithm with varying values of *r* and γ.

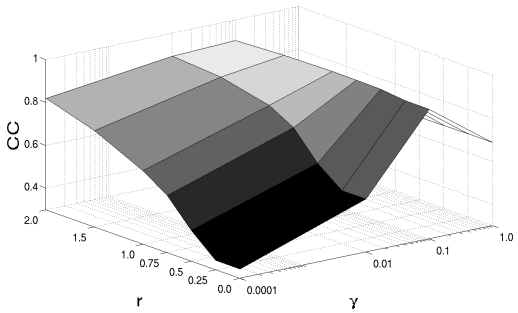

Figure 26. Plots depicting the CC over the range of studied *r* and γ parameters of images reconstructed from GATE data simulated for 9 views over 30 seconds.

Fig. 27 compares images reconstructed with the proposed algorithm ($\gamma = 0.01$, $r = 1.5$ and $\gamma = 0.01$, $r = 0.75$) and MLEM from data acquired with a varying number of angular views (9 to 60) and a constant 10 second acquisition time for each angular position of the three-camera system. Thus the total scan time was 200, 70, 50, and 30 seconds for 60, 21, 15, and 9 views, respectively. Associated image quality metrics are

presented in Table V. As scan time and angular sampling decreased, images reconstructed using the proposed algorithm with $\gamma = 0.01$ and $r = 1.50$ show decreased accuracy compared to scans with less noise and the same angular sampling presented in the previous section. When reconstructing from 21 views, 15 views and 9 views, higher CC is achieved using $\gamma = 0.1$ and $r = 0.75$, compared to using $\gamma = 0.01$ and $r = 1.50$. In both cases, the proposed reconstruction algorithm provides higher CC and SNR than MLEM. For reconstructions using $\gamma = 0.01$ and $r = 1.5$, CC varied by 7.5% from 0.946 to 0.875 as the number of views was reduced from 60 to 9. A variation of only 2.6% from 0.942 to 0.921 was measured for images reconstructed using $\gamma = 0.1$ and $r = 0.75$. These reductions were both less than the decrease measured for MLEM, which saw a decrease of 12.8%, from 0.915 to 0.798 as the number of views was reduced from 60 to 9.

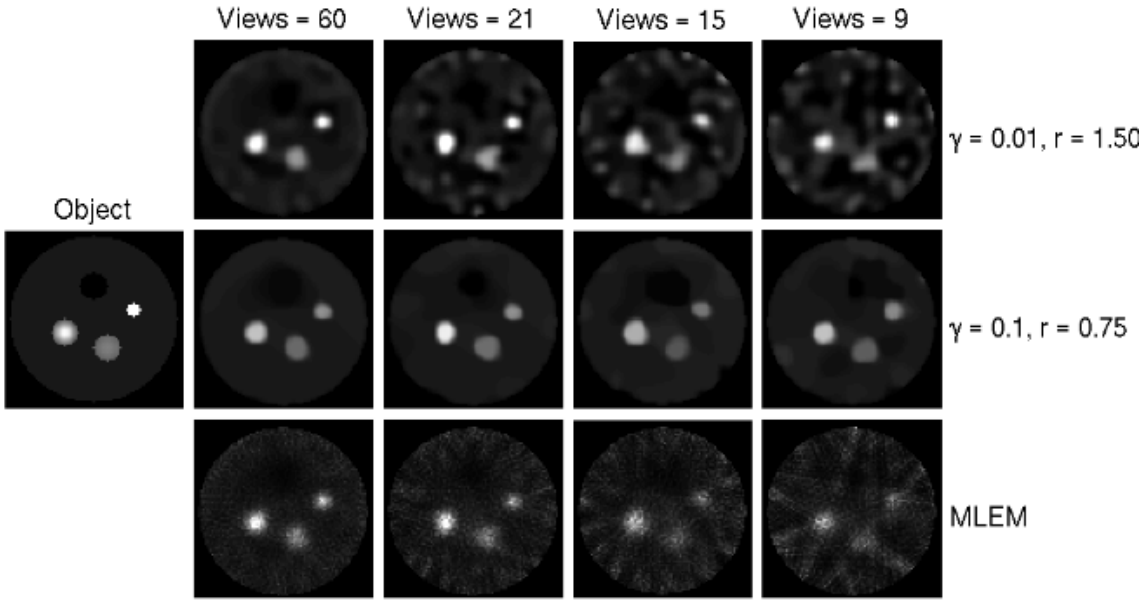

Figure 27. Images reconstructed using the proposed algorithm and MLEM from GATE data simulated with the same time per view for different numbers of views.

Table V. Comparison of image quality metrics for images reconstructed from GATE data with varying number of views and constant scan time per view.

		60 views	**21 views**	**15 views**	**9 views**
	CC	0.946	0.915	0.889	0.875
$\gamma = 0.01$, $r = 1.50$	**SNR**	17.48	9.96	1.29	10.82
	FW10M	7	6	9	7
	CC	0.942	0.937	0.908	0.921
$\gamma = 0.10$, $r = 0.75$	**SNR**	21.83	5149.86	24.76	52.09
	FW10M	8	8	10	12
	CC	0.915	0.885	0.829	0.798
MLEM	**SNR**	4.94	2.98	3.49	4.06
	FW10M	8	9	8	14

CHAPTER 4: DISCUSSION AND CONCLUSION

This chapter provides further discussion of the results and suggests future work.

4.1 Discussion

This work develops an algorithm for SPECT reconstruction from few-views. The performance of the algorithm is characterized using several different data sets and values for design parameters, r and γ. The presented simulations investigated the proposed reconstruction technique over a range of objects, noise conditions, and angular sampling schemes. Overall, the results demonstrate that blurring and noise regularization increased with increasing values of r, the standard deviation of the Gaussian blurring kernel, and γ, the TV weighting parameter. For example, in the high-view case with noiseless data generated from the system model, reconstructions using the lowest γ value studied ($\gamma = 0.0001$) yielded the most accurate images for a given value of r. When the data were made inconsistent by the addition of Poisson noise, the optimal studied γ value increased to $\gamma = 0.01$. These cases, in which data generated using the system matrix and the blurring model were used in reconstruction, indicate that accurate reconstruction is possible when the incorrect blurring model is used, as there was only a 2.5% decrease in the CC metric over all r studied when $\gamma = 0.1$ and $\gamma = 0.01$. However, in the few-view case, using an r larger than r_{true} causes the number of meaningful coefficients in the intermediate image f increases rapidly compared to using lower values of r. This indicates a less sparse image, limiting the effectiveness of exploiting gradient-magnitude

sparsity to reduce the number of views needed for reconstruction. When an approximately accurate blurring model is used, the intermediate image *f* is the most sparse in the gradient-magnitude sense. This may allow a greater reduction in the sampling necessary for reconstruction.

When noisy data were generated using GATE Monte Carlo simulations, larger *r* had a benefit when lower γ were used. For instance, in the 9 view case when data were simulated for 200 seconds and images were reconstructed with $\gamma = 0.01$, the CC varied by 11% over the range of studied *r* values, with a high *r* value ($r = 1.5$) yielding the most accurate reconstructions. When $\gamma = 0.1$ was used, a lower *r* ($r = 0.75$) yielded the most accurate reconstructions. Similarly, when the scan time was decreased in the few-view case, $r = 2.0$ yielded the most accurate reconstructions when $\gamma = 0.01$ was used; however, the highest overall CC in the few-view, decreasing scan time case was obtained with $\gamma = 0.1$ and $r = 0.75$. Reconstructions using both $\gamma = 0.01$, $r = 2.0$ and $\gamma = 0.1$, $r = 0.75$ have similar CC but different qualitative attributes (figs. 23 and 27) . The preferred parameter combination requires further study with observers. Overall, reconstructions from data generated using GATE simulations suggest that when the true blurring model is unknown and noise is present, lower values of γ ($\gamma = 0.01$ in this particular study) benefit from larger r values, while $\gamma = 0.1$ benefits from lower r values, with a smaller dependence on *r*. Since the inverse crime study demonstrated that smaller *r* values result in a more sparse intermediate image, the combination of $\gamma = 0.1$ and $r = 0.75$ may be advantageous for reconstruction from few-views.

The results also suggest that, when an appropriate value of the TV penalty term is included in the proposed reconstruction algorithm ($\gamma = 0.01$ or 0.1 for the cases studied), streaking artifacts are reduced compared to MLEM reconstructions. While images reconstructed with the proposed algorithm contain higher SNR, low frequency variations (patchy artifacts) were seen in high-noise simulation cases (fig. 24 and 27). Low frequency, patchy artifacts have been noted in CT TV reconstructions from noisy data, and future work is required to quantify the impact of these artifacts on the ability of observers to identify objects of diagnostic interest [57].

One limitation of the presented work is that the simulations modeled two dimensional objects and acquisition, whereas SPECT data are acquired in three dimensions. The principles and algorithm presented in this work can be generalized to a three dimensional case with the expansion of the system matrix and applying the blurring-masking function in three dimensions. Further, multi-pinhole collimation can be modeled in the system matrix in the three dimensional case. Additional studies are necessary to investigate this hypothesis. Future work should also be done to apply the algorithm to *in vivo* data to validate the assumption that SPECT objects may be presented as blurred piece-wise constant objects. The accuracy of the algorithm using few-views may enable high-temporal sampling that will have benefit for dynamic tracer uptake studies, such as [58]. The first pass tracer uptake in some cases occurs within seconds, in particular small animal lung perfusion studies require high temporal sampling to capture the dynamic features of tracer kinetics. With the inclusion of multi-pinhole collimators, it may be possible to perform such dynamic studies using a stationary three-camera system.

4.2 Conclusions

In this thesis, a sparsity-exploiting algorithm for tomographic reconstruction from few-view SPECT data is developed and characterized. The algorithm models the SPECT object as a blurred piecewise-constant object and exploits the sparsity of an intermediate deblurred image by minimizing image TV. An iterative process is implemented using a first-order primal-dual optimization.

The algorithm assumes a specific blurring model, but the results demonstrate that accurate reconstruction is possible when the true blurring model is unknown. In the low noise cases, the algorithm has limited sensitivity to the blurring model; in the high-noise case, reconstruction benefit from a blurring model with a larger radius. Reconstructed images demonstrate that the algorithm introduces low-frequency artifacts the presence of noise, but streak artifacts due to undersampling are eliminated. The effects of these artifacts on observers is a topic for future work. These results demonstrate the preliminary feasibility of the developed algorithm in the application of few-view SPECT.

REFERENCES

[1] G.-H. Chen, J. Tang, and S. Leng, “Prior image constrained compressed sensing (PICCS): A method to accurately reconstruct dynamic CT images from highly undersampled projection data sets,” *Med. Phys.*, vol. 35, no. 2, pp. 660–663, 2008.

[2] E. Y. Sidky and X. Pan, “Image reconstruction in circular cone-beam computed tomography by constrained, total-variation minimization.,” *Phys. Med. Biol.*, vol. 53, no. 17, pp. 4777–4807, Sep. 2008.

[3] E. Sidky, C. Kao, and X. Pan, “Accurate image reconstruction from few-views and limited-angle data in divergent-beam CT,” *J. X-ray Sci. Tech.*, vol. 14, pp. 119–139, 2006.

[4] J. T. Bushberg, J. A. Seibert, E. M. Leidholdt, and J. M. Boone, *The Essential Physics of Medical Imaging*, 2nd ed. Philadelphia: Lippincott Williams & Wilkins, 2002.

[5] F. J. Beekman, F. van der Have, B. Vastenhouw, A. J. a van der Linden, P. P. van Rijk, J. P. H. Burbach, and M. P. Smidt, “U-SPECT-I: a novel system for submillimeter-resolution tomography with radiolabeled molecules in mice.,” *J. Nucl. Med.*, vol. 46, no. 7, pp. 1194–1200, Jul. 2005.

[6] L. R. Furenlid, D. W. Wilson, Y.-C. Chen, H. Kim, P. J. Pietraski, M. J. Crawford, and H. H. Barrett, “FastSPECT II: A Second-Generation High-Resolution Dynamic SPECT Imager.,” *IEEE Trans. Nucl. Sci.*, vol. 51, no. 3, pp. 631–635, Jun. 2004.

[7] S. Matej and R. M. Lewitt, “Practical considerations for 3-D image reconstruction using spherically symmetric volume elements.,” *IEEE Trans. Med. Img.*, vol. 15, no. 1, pp. 68–78, 1996.

[8] B. Silverman, M. Jones, and J. Wilson, “A Smoothed EM Approach to Indirect Estimation Problems, With Particular, Reference to Stereology and Emission Tomography,” *J. R. Stat. Soc.*, vol. 52, no. 2, pp. 271–324, 1990.

[9] T.-Y. Lee, “Functional CT: physiological models,” *Trends Biotechnol.*, vol. 20, no. 8, pp. S3–S10, Aug. 2002.

[10] H. H. Barrett and W. Swindell, *Radiological Imaging: The Theory of Image Formation, Detection, and Processing*. New York, New York, USA: Academic Press, Inc., 1981.

[11] J. Petersson, A. Sánchez-Crespo, S. A. Larsson, and M. Mure, “Physiological imaging of the lung: single-photon-emission computed tomography (SPECT).,” *J. Appl. Physiol.*, vol. 102, no. 1, pp. 468–476, 2007.

[12] T. M. Bateman and S. J. Cullom, “Attenuation correction single-photon emission computed tomography myocardial perfusion imaging.,” *Seminars in Nuclear Medicine*, vol. 35, no. 1, pp. 37–51, Jan. 2005.

[13] S. Vandenberghe, Y. D’Asseler, R. Van de Walle, T. Kauppinen, M. Koole, L. Bouwens, K. Van Laere, I. Lemahieu, and R. a Dierckx, “Iterative reconstruction algorithms in nuclear medicine.,” *Comput. Med. Imaging Graphics*, vol. 25, no. 2, pp. 105–11, 2001.

[14] D. Ma, “Multi-pinhole dynamic SPECT imaging of the lung: simulation and system optimization,” Marquette University, 2010.

[15] C. Vanhove, M. Defrise, T. Lahoutte, and A. Bossuyt, “Three-pinhole collimator to improve axial spatial resolution and sensitivity in pinhole SPECT.,” *Eur. J. Nucl. Med. Mol. Imaging*, vol. 35, no. 2, pp. 407–15, Feb. 2008.

[16] L. Bouwens and R. V. de Walle, “LMIRA: list-mode iterative reconstruction algorithm for SPECT,” *IEEE Trans. Nucl. Sci.*, vol. 48, no. 4, pp. 1364–1370, 2001.

[17] F. J. Beekman and B. Vastenhouw, “Design and simulation of a high-resolution stationary SPECT system for small animals,” *Phys. Med. Biol.*, vol. 49, no. 19, pp. 4579–4592, Oct. 2004.

[18] Z. Liu, G. a Kastis, G. D. Stevenson, H. H. Barrett, L. R. Furenlid, M. a Kupinski, D. D. Patton, and D. W. Wilson, “Quantitative analysis of acute myocardial infarct in rat hearts with ischemia-reperfusion using a high-resolution stationary SPECT system.,” *J. Nucl. Med.*, vol. 43, no. 7, pp. 933–939, Jul. 2002.

[19] G. T. Gullberg, B. W. Reutter, A. Sitek, J. S. Maltz, and T. F. Budinger, “Dynamic single photon emission computed tomography--basic principles and cardiac applications.,” *Phys. Med. Biol.*, vol. 55, no. 20, pp. R111–91, Oct. 2010.

[20] A. Celler, J. Bong, S. Blinder, R. Attariwala, D. Noll, L. Hook, T. Farncombe, and R. Harrop, “Preliminary results of a clinical validation of the dSPECT method for determination of renal glomerular filtration rate (GFR),” in *IEEE Nuclear Science Symposium Conference Record*, 2001, vol. 2, pp. 1079–1082.

[21] G. C. Kagadis, G. Loudos, K. Katsanos, S. G. Langer, and G. C. Nikiforidis, “In vivo small animal imaging: Current status and future prospects,” *Med. Phys.*, vol. 37, no. 12, p. 6421, 2010.

[22] H. H. Barrett and K. J. Myers, *Foundations of Image Science*. Hoboken, NJ: John Wiley & Sons, Inc., 2004.

[23] a Yendiki and J. a Fessler, “A comparison of rotation- and blob-based system models for 3D SPECT with depth-dependent detector response,” *Physics in Medicine and Biology*, vol. 49, no. 11, pp. 2157–2168, Jun. 2004.

[24] P. P. Bruyant, “Analytic and iterative reconstruction algorithms in SPECT.,” *J. Nucl. Med.*, vol. 43, no. 10, pp. 1343–58, Oct. 2002.

[25] J. Radon, “On the Determination of Functions From Their Integral Values Along Certain Manifolds,” *IEEE transactions on medical imaging*, vol. MI-5, no. 4, pp. 170–176, Mar. 1986.

[26] R. A. Brooks and G. Di Chiro, “Principles of computer assisted tomography (CAT) in radiographic and radioisotopic imaging.,” *Phys. Med. Biol.*, vol. 21, no. 5, pp. 689–732, Sep. 1976.

[27] J. Li, R. J. Jaszczak, K. L. Greer, and R. E. Coleman, “A filtered backprojection algorithm for pinhole SPECT with a displaced centre of rotation.,” *Phys. Med. Biol.*, vol. 39, no. 1, pp. 165–76, Jan. 1994.

[28] E. S. Chornoboy, C. J. Chen, M. I. Miller, T. R. Miller, and D. L. Snyder, “An evaluation of maximum likelihood reconstruction for SPECT,” *IEEE Trans. Med. Img.*, vol. 9, no. 1, pp. 99–110, 1990.

[29] P.-C. Huang, C.-H. Hsu, I.-T. Hsiao, and K. M. Lin, “An accurate and efficient system model of iterative image reconstruction in high-resolution pinhole SPECT for small animal research,” *J. Instrum.*, vol. 4, no. 06, Jun. 2009.

[30] E. K. P. Chong and S. H. Zak, *An Introduction To Optimization*, Third. Hoboken, New Jersey, USA: John Wiley & Sons, Inc., 2008.

[31] S. Boyd and L. Vandenberghe, *Convex Optimization*, 7th ed. New York, New York, USA: , 2009.

[32] W. Navidi, “A graphical illustration of the EM algorithm,” *Am. Stat.*, vol. 51, no. 1, pp. 29–31, 1997.

[33] L. A. Shepp and Y. Vardi, “Maximum likelihood reconstruction for emission tomography.,” *IEEE Trans. Med. Img*, vol. 1, no. 2, pp. 113–22, Jan. 1982.

[34] R. L. Siddon, “Fast calculation of the exact radiological path for a three-dimensional CT array,” *Med. Phys.*, vol. 12, pp. 252–255, 1985.

[35] C. Wietholt, “Design and application of a dual-modality small animal SPECT and micro-CT system,” Marquette University, 2004.

[36] C. Wietholt, T. Hsiao, and C. T. Chen, “New ray-driven system matrix for small-animal pinhole-SPECT with detector blur, geometric response and edge penetration modeling,” in *IEEE Nuclear Science Symposium Conference Record*, 2006, vol. 6, pp. 3414–3419.

[37] E. J. Candes, J. Romberg, and T. Tao, “Robust uncertainty principles: exact signal reconstruction from highly incomplete frequency information,” *IEEE Trans. Inf. Theory*, vol. 52, no. 2, Feb. 2006.

[38] E. J. Candes, J. K. Romberg, and T. Tao, “Stable signal recovery from incomplete and inaccurate measurements,” *Comm. Pure Appl. Math.*, vol. 59, no. 8, pp. 1207–1223, Aug. 2006.

[39] D. L. Donoho, “Compressed sensing,” *IEEE Trans. Inf. Theory*, vol. 52, no. 4, pp. 1289–1306, Apr. 2006.

[40] G. H. Chen, J. Tang, and S. Leng, “Prior image constrained compressed sensing (PICCS),” *Proc. SPIE*, vol. 6856, p. 685618, 2008.

[41] M. Lustig and D. L. Donoho, “Compressed sensing MRI,” *IEEE Signal Process. Mag.*, vol. 25, no. 2, pp. 72–82, 2008.

[42] M. Lustig and D. Donoho, “Compressed sensing MRI,” *IEEE Signal Process. Mag.*, no. March 2008, pp. 72–82, 2008.

[43] E. Candes and J. Romberg, “Sparsity and incoherence in compressive sampling,” *Inverse problems*, pp. 1–20, 2007.

[44] M. Lustig, D. Donoho, and J. M. Pauly, “Sparse MRI: The application of compressed sensing for rapid MR imaging.,” *Magn. Reson. Med.*, vol. 58, no. 6, pp. 1182–95, Dec. 2007.

[45] X. Duan, L. Zhang, Y. Xing, Z. Chen, and J. Cheng, “Few-View Projection Reconstruction With an Iterative Reconstruction-Reprojection Algorithm and TV Constraint,” *IEEE Trans. Nucl. Sci.*, vol. 56, no. 3, pp. 1377–1382, 2009.

[46] L. Ritschl, F. Bergner, C. Fleischmann, and M. Kachelriess, “Improved total variation-based CT image reconstruction applied to clinical data.,” *Phys. Med. Biol.*, vol. 56, no. 6, pp. 1545–1561, 2011.

[47] A. Chambolle and T. Pock, “A first-order primal-dual algorithm for convex problems with applications to imaging,” *J. Math. Imag. Vis.*, vol. 40, no. 1, pp. 1–26, 2011.

[48] E. Y. Sidky, J. H. Jørgensen, and X. Pan, “Convex optimization problem prototyping for image reconstruction in computed tomography with the Chambolle–Pock algorithm,” *Phys. Med. Biol.*, vol. 57, no. 10, pp. 3065–3091, May 2012.

[49] H. Hudson and R. Larkin, “Accelerated image reconstruction using ordered subsets of projection data,” *IEEE Trans. Med. Img.*, vol. 13, no. 4, pp. 601–609, 1994.

[50] E. Y. Sidky, X. Pan, I. S. Reiser, R. M. Nishikawa, R. H. Moore, and D. B. Kopans, “Enhanced imaging of microcalcifications in digital breast tomosynthesis through improved image-reconstruction algorithms,” *Med. Phys.*, vol. 36, no. 11, pp. 4920–4932, 2009.

[51] P. Donato, P. Coelho, C. Santos, A. Bernardes, and F. Caseiro-Alves, “Correspondence between left ventricular 17 myocardial segments and coronary anatomy obtained by multi-detector computed tomography: an ex vivo contribution.,” *Surgical and radiologic anatomy*, vol. 34, no. 9, pp. 805–10, Nov. 2012.

[52] O. Pereztol-Valdés, J. Candell-Riera, C. Santana-Boado, J. Angel, S. Aguadé-Bruix, J. Castell-Conesa, E. V. Garcia, and J. Soler-Soler, “Correspondence between left ventricular 17 myocardial segments and coronary arteries.,” *European heart journal*, vol. 26, no. 24, pp. 2637–43, Dec. 2005.

[53] R. Jain, “Determinants of tumor blood flow: a review,” *Cancer Research*, vol. 48, pp. 2641–2658, 1988.

[54] J. Kaipio and E. Somersalo, *Statistical and Computational Inverse Problems*. New York: Springer Science+Business Media, LLC, 2005.

[55] D. L. Hill, P. G. Batchelor, M. Holden, and D. J. Hawkes, “Medical image registration,” *Phys. Med. Biol.*, vol. 46, no. 3, pp. R1–45, Mar. 2001.

[56] S. Jan, G. Santin, D. Strul, and S. Staelens, “GATE: a simulation toolkit for PET and SPECT,” *Phys. Med. Biol.*, vol. 49, p. 4543, 2004.

[57] J. Tang, B. E. Nett, and G.-H. Chen, “Performance comparison between total variation (TV)-based compressed sensing and statistical iterative reconstruction algorithms.,” *Phys. Med. Biol.*, vol. 54, no. 19, pp. 5781–5804, Oct. 2009.

[58] D. Ma, P. Wolf, A. Clough, and T. Schmidt, “The Performance of MLEM for Dynamic Imaging From Simulated Few-View, Multi-Pinhole SPECT,” *IEEE Trans. Nucl. Sci.*, 2012.

APPENDIX A: DEFINITION OF VARIABLES USED

Scalars

A	Activity, Bq or Ci
D_{KL}	Kullback-Leibler Distance
E	Energy
n	Iteration number
N	Number of detector elements
M	Number of voxels
r	Blurring algorithm design parameter; radius of Gaussian blur model used in reconstruction
ϕ	Angular position about an object
γ	TV weighting algorithm design parameter

Vectors and Operators

$\boldsymbol{D}$	Discrete gradient operator
$\boldsymbol{f}$	Object
$\hat{\boldsymbol{f}}$	Object estimate
$\boldsymbol{f'}$	Intermediate piecewise constant image
$\boldsymbol{g}$	Measured projection data vector
$\boldsymbol{G}$	Gaussian blur image operator
$\boldsymbol{H}$	System operator; system matrix
$\boldsymbol{M}$	Support preserving image mask operator
$\boldsymbol{u}$	Object estimate; piece-wise constant object subject to blurring operation
Ψ	Linear sparsifying operator

APPENDIX B: A WAVELET BASED ALGORITHM FOR FEW-VIEW SPECT RECONSTRUCTION

The contents of this appendix were published in the conference record of the 2011 IEEE Nuclear Science Symposium/Medical Imaging Conference, entitled *A Compressed Sensing Algorithm for Sparse-view Pinhole Single Photon Emission Computed Tomography.* Citations in this appendix refer to the references listed at the end of this appendix.

B.1 Introduction

Dynamic Single Photon Emission Computed Tomography (SPECT) provides information about tracer uptake and washout from a series of time-sequence images. Dynamic SPECT systems measuring time activity curves on the order of minutes have been developed [1][2]. However, to adequately sample the time-activity curve of some tracers, a temporal sampling on the order of seconds is required. Stationary multiple camera systems are being developed to provide rapid dynamic acquisitions [3][4]. To reduce cost, a limited number of cameras may be used, resulting in angularly undersampled data.

The image reconstruction theory of compressed sensing (CS) exploits sparsity in the object to potentially allow for a reduction in the data sampling. Thus, if some representation exists in which the coefficients of an image are sparse, the same image can be represented using less information. The object can then be accurately reconstructed

from undersampled data. Reconstruction from angularly undersampled data has been recently studied for CT [5][6].

We propose a novel reconstruction algorithm for sparse-view pinhole SPECT based on CS theory. The algorithm models Poisson noise statistics and uses the spline wavelet transform as the sparsifying transform to address the unique challenges of SPECT imaging.

Algorithm performance is evaluated using metrics for image fidelity and spatial accuracy. These results are compared to results obtained using Maximum-Likelihood Expectation Maximization (MLEM).

B.2 Methods

B.2.1 The Algorithm

CS algorithms solve a constrained optimization problem to recover the image. The L_1-norm of the sparse representation is minimized and constrained by data fidelity. For example, if the data fidelity constraint is the L_2-norm of the difference between the estimated and measured data, the CS optimization problem can be described as

$$min.\ (\|\Psi\sigma\|_1)\ s.t.\ \|H\hat{s}-y\|_2 < \varepsilon, \tag{B.1}$$

where H is the system matrix, $\hat{s}$ is the estimated image, y is the measured data, Ψ is a sparsifying transform, and s is the true object. Data fidelity is imposed by the constraint and sparsity is enforced by the objective function. This optimization problem is solved

by considering the images that satisfy the constraint, then selecting the image with the most sparse representation.

Previous CS algorithms for sparse-view tomographic reconstruction assume a piecewise constant object, using gradient magnitude as the sparsifying transform [6]. In SPECT imaging, the underlying objects represent a distribution of activity, which is not necessarily piecewise constant and may be smoothly varying.

Poisson noise due to photon counting statistics can be incorporated into the CS framework by minimizing the Kullback-Leibler distance (D_{KL}) to achieve data fidelity instead of the more commonly used L_2 norm, which assumes Gaussian noise. In our proposed implementation, D_{KL} is minimized by gradient descent. To account for the expected piecewise smooth nature of the tracer distribution, we propose the spline wavelet transform for the sparsifying transform, Ψ. In our proposed algorithm, sparsity is enforced using the Iterative Hard Thresholding algorithm [7].

The spline wavelet transform is characterized by having its synthesis functions be polynomial splines. If a function is piecewise smooth, the signal can be sparsely approximated by spline wavelets. The wavelet coefficients will be near zero where the function can be well approximated by a polynomial [8][9]. Assuming underlying SPECT objects are piecewise smooth, the spline wavelet transform will operate as a sparsifying transform.

Fig. B.1 shows a SPECT image of a rat-lung, the gradient magnitude image and the spline wavelet transform. The image is noisy and the underlying distribution is likely piecewise smooth. The sorted and normalized coefficients of the image, the gradient magnitude image and spline wavelet transform are shown in fig. B.2. The spline wavelet transform coefficients decay faster than the gradient magnitude coefficients, indicating that the spline wavelet transform yields a more sparse image.

Equation B.2 describes the implementation of the proposed algorithm where x is the sparse domain estimate of the object, λ is the gradient descent step size, and G_K is a nonlinear operator that retains the K-largest coefficients setting the remaining coefficients to zero [7].

$$x^{n+1} = G_K[x^n - \lambda\Psi(\nabla D_{KL}(y, H\hat{s}))] \tag{B.2}$$

The gradient of D_{KL} with respect to $\hat{s}$ (∇D_{KL}) is

$$\nabla_{\hat{s}_j} D_{KL} = \sum_{m=1}^{M} H_{m,j}(1 - y_m/(H\hat{s})_m), \quad j = 1, ..., N, \tag{B.3}$$

where N is the number of voxels in the image volume and M is the number of measurements. Note that this is equivalent to back projection of the parenthetical expression.

The proposed algorithm is also described by the following pseudo code. The symbol := denotes assignment, Ψ^{-1} indicates the inverse sparsifying transform, H^T indicates back projection, and $\hat{y}$ represents the estimate of the data.

1. $\lambda := 1$; $\lambda_{red} := 0.8$
2. $\hat{s}_0 := H^T y$; $n := 1$
3. repeat *main loop*
4. $\hat{y} := H\hat{s}_n$
5. for $i = 0, M$ do : $t_i = 1 - y_i/\hat{y}_i$
6. $g := H^T t$ *calculate gradient of* D_{KL}
7. repeat *adaptive gradient descent loop*
8. $\hat{s}_n := \Psi^{-1} G_K[\Psi(\hat{s}_{n-1} - \lambda g)]$
9. for $i = 0, N$ do : if $\hat{s}_{n_i} < 0$ then $\hat{s}_{n_i} = 0$ *enforce positivity*
10. $\lambda := \lambda * \lambda_{red}$
11. until $\{D_{KL}(y, H\hat{s}_n) < D_{KL}(y, H\hat{s}_{n-1})\}$
12. $n := n + 1$
13. until {stopping criterion}
14. $\hat{s} := \hat{s}_n$
15. return $\hat{s}$

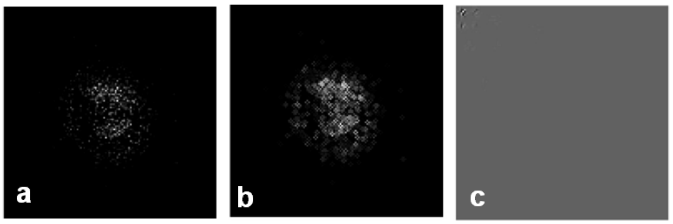

Fig. B.1. (a.) rat lung image, (b.) gradient magnitude image of rat lung image, (c.) spline wavelet transform of rat lung image

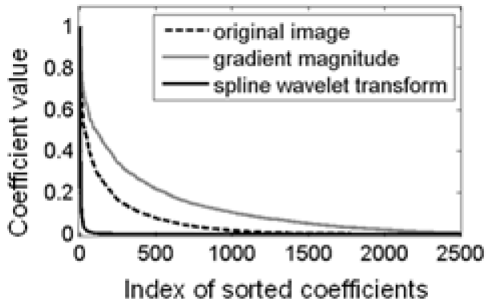

Fig. B.2. A comparison of the coefficients of the rat lung image and transformed images displayed in fig. B.1.

B.3 Simulation Study

Preliminary feasibility of the algorithm was studied through simulations. The simulated object consisted of two 2D Gaussian distributions sampled onto a 128 x 128-pixel grid. The Gaussian objects had a standard deviation of 4 pixels and maximum values of 1 and 2. Each object was truncated at two standard deviations.

Simulated projections were generated using the system matrix, resulting in 128-pixel projections at 128, 60, 15, 10, and 5 views distributed evenly over 360 degrees. Poisson noise was added to each projection dataset such that the total number of counts remained constant as the number of views decreased.

The sparsifying transform, Ψ, was a 7-stage discrete wavelet transform with an orthogonal spline wavelet kernel with 5 vanishing points. The simulated object was

transformed using the sparsifying transform, Ψ, and was determined to have 1976 non-zero coefficients. This was used as the value of the parameter K.

To quantify the accuracy of the proposed algorithm, three metrics were evaluated. The coefficient of variation (CV) between the estimated data and the projection data was calculated. This is a measure of image fidelity and is described by

$$CV = (\sum (H\hat{s}-y)^2)^{1/2}/ \text{ y} * 100. \quad \text{(B.4)}$$

Quantitative accuracy was quantified using the contrast error (CE),

$$CE = |C_{measured} - C_{true}| / C_{true}. \quad \text{(B.5)}$$

CE is a comparison between the reconstructed contrast and true contrast between two ROIs. CE is independent of number of reconstructed counts. To quantify spatial accuracy, the scaled peak cross-correlation with the true object was used. Images were scaled using

$$Scaling\ Factor = N_{object} / N_{image}, \quad \text{(B.6)}$$

where N is the number of counts in the FOV. This metric is independent of the number of reconstructed counts. Images with a higher peak cross-correlation more accurately depict the spatial distribution of an object.

B.4 Results

Fig. B.3 shows the images reconstructed from 128, 60, 15, 10, and 5 views using the proposed CS algorithm (a) and MLEM (b). Table B.I displays the described metrics

for each algorithm and each sampling case. Fig. B.4 and fig. B.5 show selected profiles plotted through the center of a selection of images displayed in fig. B.3.

B.5 Conclusions

Images reconstructed from ten views using both the proposed CS algorithm and MLEM depict the object contrast to < 2% error. Spatial accuracy varied by less than 5% as the number of views decreased.

The results of Table B.I suggest similar performance of the MLEM and proposed CS algorithm. We are currently investigating alternative sparsifying transforms and strategies to provide improved performance compared to MLEM.

Table B.I. Image Quality Metrics For Reconstructed Images

CS	**Number of Views**				
	128	60	15	10	5
CV	3.41	2.36	3.03	2.85	2.11
CE (%)	2.64	4.66	9.54	1.65	2.81
Peak XCorr	275.77	271.59	270.59	271.59	260.86
MLEM					
CV	1.98	2.03	1.78	1.67	1.33
CE (%)	1.86	0.42	14.48	0.93	6.36
Peak Xcorr	269.62	276.23	276.87	271.16	261.34

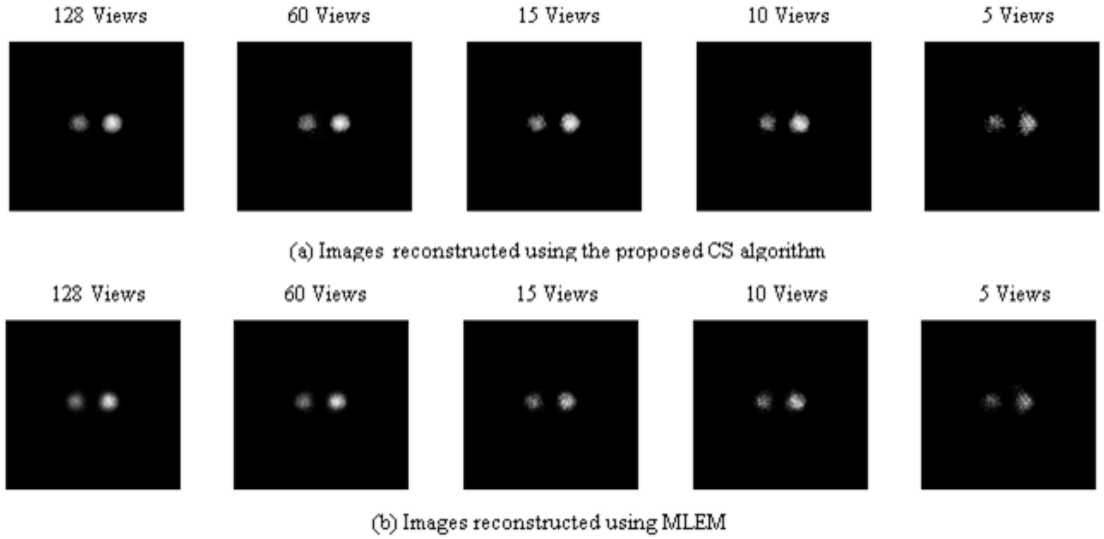

Fig. B.3. Images reconstructed using (a) the proposed algorithm and (b) MLEM

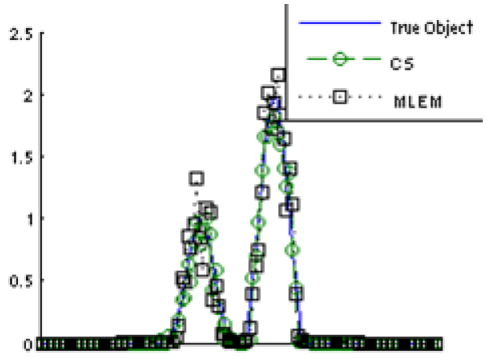

Fig. B.4. Profiles through images reconstructed with 60 views

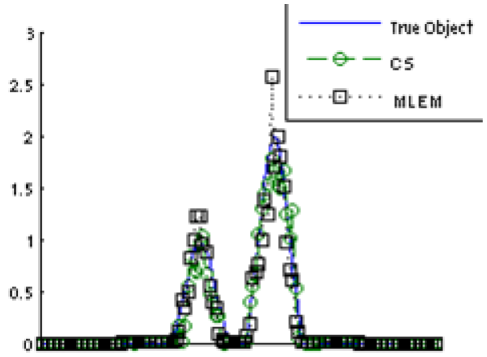

Fig. B.5. Profiles through images reconstructed with 10 views

References

[1] G. T. Gullberg, B. W. Reutter, A. Sitek, J. S. Maltz, and T. F. Budinger, “Dynamic single photon emission computed tomography--basic principles and cardiac applications.,” *Physics in medicine and biology*, vol. 55, no. 20, pp. R111-91, Oct. 2010.

[2] A. Celler et al., “Preliminary results of a clinical validation of the dSPECT method for determination of renal glomerular filtration rate (GFR),” in *Nuclear Science Symposium Conference Record, 2001 IEEE*, 2001, vol. 2, pp. 1079–1082.

[3] F. J. Beekman and B. Vastenhouw, “Design and simulation of a high-resolution stationary SPECT system for small animals,” *Physics in Medicine and Biology*, vol. 49, no. 19, pp. 4579-4592, Oct. 2004.

[4] L. R. Furenlid et al., “FastSPECT II: A Second-Generation High-Resolution Dynamic SPECT Imager.,” *IEEE transactions on nuclear science*, vol. 51, no. 3, pp. 631-635, Jun. 2004.

[5] G.-H. Chen, J. Tang, and S. Leng, “Prior image constrained compressed sensing (PICCS): A method to accurately reconstruct dynamic CT images from highly undersampled projection data sets,” *Medical Physics*, vol. 35, no. 2, p. 660, 2008.

[6] E. Y. Sidky and X. Pan, “Image reconstruction in circular cone-beam computed tomography by constrained, total-variation minimization.,” *Physics in medicine and biology*, vol. 53, no. 17, pp. 4777-807, Sep. 2008.

[7] T. Blumensath and M. E. Davies, “Iterative hard thresholding for compressed sensing,” *Applied and Computational Harmonic Analysis*, vol. 27, no. 3, pp. 265–274, Nov. 2009.

[8] M. Unser and T. Blu, “Wavelet theory demystified,” *Signal Processing, IEEE Transactions on*, vol. 51, no. 2, pp. 470–483, 2003.

[9] M. Unser, “Ten good reasons for using spline wavelets,” *Image Processing*, vol. 3169, pp. 422-431, 1997.

APPENDIX C: ALGORITHM PSEUDOCODE

This appendix lists thorough pseudocode describing the algorithm implemented in this thesis. The main program is listed first, followed by the sub-routines called.

Main:

```
Read in arguments from commandline
Initialize system geometry/system model
Allocate memory
Initialize blurring model (see sub-routine)
Calculate L (see sub-routine)
g = Load projection file
until stopping criterion
p n+1
    pn-1 = p
    mask ubar
    temp = forward project(ubar) (see sub-routine)
    for all projection positions
       p(position) = 0.5(1 + pn-1(position) + σ * temp(position) –
       ((p(position) + σ * temp(position) – 1)^2 + 4σ*g(position))^1/2
    end
q n+1
    temp = gradient(ubar) (see sub-routine)
    for all vector points
       q(point) = gamma * (q(point) + σ * temp(position))
    end
    mag = magnitude(q) (see sub-routine)
    for all voxels
       if mag(voxel) < gamma; mag(voxel) = gamma;
    end
    for all vector points
       q(point) = q(point) / mag(voxel of point)
    end
u n+1
    div = divergence(q)
    temp = backproject(p)
    un-1 = u
    for all voxels
       u(voxel) = u(voxel) – τ * temp(voxel) + τ * div(voxel)
       if u(voxel) < 0; u(voxel) = 0
    end
ubar n+1
    for all voxels
       ubar(voxel) = u(voxel) + θ(un-1(voxel) – u(voxel))
```

```
    end
    check convergence
repeat
```

forward project(object data):

```
mask(object data)
blur(object data) (see sub-routine)
mask(object data
for all angles
    calculate rotated geometry
    for all pinholes
       for all detector elements
          for all pinhole matrix points
             calculate total ray length
             compute first and last plane intersections, x and y
             compute other intersections, x and y
             sort intersections
             calculate length of intersection in each voxel
             temp += object data(voxel) * length(voxel)
          end
          out(detector) += temp
       end
    end
end
return out
```

backproject(projection data):

```
for all angles
    calculate rotated geometry
    for all pinholes
       for all detector elements
          for all pinhole matrix points
             calculate total ray length
             compute first and last plane intersections, x and y
             compute other intersections, x and y
             sort intersections
             calculate length of intersection in each voxel
             out(voxel) = projection data(detector) * length(voxel)
          end
       end
    end
    for all voxels
       new voxel value(voxel) += out(voxel)
    end
end
mask(new voxel value)
blur(new voxel value)
mask(new voxel value)
```

```
22: return new voxel value
```

Calculate L:

```
 1: x = non-zero object
 2: temp = forward project(x)
 3: y = backproject(temp)
 4: temp = gradient(x)
 5: z = divergence(temp)
 6: for all voxel values
 7:     x(voxel) = y(voxel) + z(voxel)
 8: end
 9: for all voxel values
10:     mag += x*x
11: end
12: mag = mag^1/2
13: for all voxel values
14:     x(voxel) = x(voxel)/mag
15: end
16: y = forward project(x)
17: for all voxels
18:     L += y(voxels) * y(voxels)
19: y = gradient(x)
20: mag = magnitude(y)
21: for all voxels
22:     L += mag(voxels) * mag(voxels)
23: end
24: return L
```

Gradient(object data):

```
 1: for all y voxels
 2:     for voxels x = 1:end
 3:        gradx(x-1,y) = object data(x,y) – object data(x-1,y)
 4:     end
 5:     gradx(end,y) = -data(end,y)
 6: end
 7: for all x voxels
 8:     for y = 1:end
 9:        grady(x,y-1) = object data(x,y) – object data(x,y-1)
10:     end
11:     grady(x,end) = -data(x,end)
12: end
13: return gradx,grady
```

divergence(gradx,grady):

```
1: for voxels y = 1:end
2:     for voxels x = 1:end
3:        div(x,y) = gradx(x-1,y) + grady(x,y-1) – gradx(x,y) –
          grady(x,y)
```

```
4:     end
5:     div(x,0) = gradx(x-1,0) – gradx(x,0) – grady(x,0)
6: end
7: div(0,0) = -gradx(0,0) – grady(0,0)
8: mask(div)
```

magnitude:

```
1: for all voxels
2:     mag(voxel) = gradx(voxel) * gradx(voxel) + grady(voxel) *
       grady(voxel)
3:     mag(voxel) = mag(voxel)^1/2
4: end
```

Initialize blurring model:

```
1: create 2-d Gaussian with radius r
2: fft(gaussian)
```

blur(object data):

```
1: x = fft(object data) * fft(gaussian)
2: return ifft(x)
```

APPENDIX D: USER'S GUIDE

CPSPECT is a command line faculty that implements several instances of the CP algorithm described here and in [48]. The full C++ code can be found in appendix X.

D.1 Basic usage

The developed program accepts arguments which must be presented in the following order:

```
./cp [algorithm number] [number of angles] [projection filename] [gamma (if alg >= 4)] [r (if alg >=6)]
```

D.1.1 Arguments

Algorithm Number: The software implements listing 3-5 of [48]. Algorithm number uses the same numbers and additionally includes two other algorithms that are based on this thesis. Algorithm 2 can also be implemented if the hard coded non-negativity constraint is removed.

3: least-squares (LS) with non-negativity constraint

4: LS+TV

5: KL+TV

6: LS+TV with blurring model

7: KL+TV with blurring model

Note that algorithm 7 is the algorithm that is characterized in this work. All algorithms have hard coded non-negativity constraints. These will be further discussed in a later section.

Number of Angles: The number of projection views distributed around 360 degrees

Projection Filename: Full path to the file containing the projections to be used in reconstruction. The software assumes the projection file is formatted as a view-by-view, row-by-row (i.e. view 1, row 1; view 1, row 2; …view 1, row X; view 2, row 1;...) 32-bit floating point, big-endian file. For the inverse crime case, where projection data are generated exactly from the object model, the value of this argument should be set to `ic`.

Gamma: When applicable, this is the TV weighting parameter. If using algorithm 3, this argument is not used.

R: When applicable, this is the blurring model radius, *r*. If an algorithm number less than 6 is used, this argument is ignored.

D.1.2 Output

The algorithm outputs several images. Intermediate versions of *ubar*, *u* and *u'* are saved. The intermediate piece-wise constant images are u*. The output images are labeled uprime*. If the *r* parameter is set to 0.0 or the blurring model is not used, u* are

the output images. The final images are named uFinal and uprimeFinal. The output images are formatted as row-by-row, 64-bit floating point, little-endian images. Additionally, an ASCII text file (output) is written. This documents the conditional primal dual gap among other things, which are clearly labeled in the file.

D.2 Modifying the code

The code developed may require modification based on the task at hand, for instance including or excluding the positivity constraints or altering the sensitivity correction details. Some frequent modifications are described below.

D.2.1 Specifying Geometry

It is important that the system modeled match the physical geometry as close a possible to eliminate artifacts and improve reconstruction. The full geometry of the system is specified in CPSPECT.cpp in the `init()` function. All lengths are specified in millimeters. There is code set up for one, two, four and nine pinholes as described in [14]. In the main program, the number of pinholes is set, this allows the number of pinholes to be added as an argument easily if need be. All z dimensions are set to 1, effectively making this two dimensional processing code. To generalize this code to three dimensions, the z dimensions will have to be specified and other changes made.

D.2.2 Non-negativity Constraints

A non-negativity constraint can be applied to each algorithm relatively easily. There is a function written to do this, the `enforcePositivity()` function of the `ReconData` class. In the code they are in CPSPECT.cpp: line 528 for algorithm 3, line 744 for algorithms 4 and 6, line 1110 for algorithms 5 and 7. The theory behind applying these is documented in [48].

D.2.3 Sensitivity Correction

When projection data is acquired from physical system or a simulation of a physical system (as in the GATE simulations in this work), the sensitivity of pinhole collimators depends on the angle of the ray incident on the pinhole. In order to correct for the spatially-varying pinhole sensitivity during reconstruction, a sensitivity map was generated by simulating a flood source on the collimator surface [15]. The resulting projection represents the spatially-varying sensitivity of the pinhole and was incorporated into the reconstruction algorithm. The sensitivity map was multiplied during each forward projection prior to the summing of data from each ray. Data were multiplied by the sensitivity map prior to backprojection. If the collimator has multiple pinholes, a sensitivity map must be simulated for each pinhole.

This scheme is applied in the C++ code: the sensitivity map is loaded in the `init()` function of the `ProjData` class (projdata.cpp, lines 336-349). This block of code calls the `load_Sensitivity_Map()` function that loads the sensitivity map, which is formatted in the same manner as the projection file (row-by-row, 32-bit floating

point, big-endian). The sensitivity correction is incorporated in to the system matrix in the file smpinholerd.cpp, lines 744-746 and lines 1094-1096. To reconstruct without sensitivity correction, these lines can be simply ignored (deleted or commented out). To make this more accessible, I would recommend adding a flag as an argument so the sensitivity correction does not have to be hard coded.

D.2.4 Stopping Criterion

The algorithms described in this work are guaranteed to converge and use the primal-dual gap as a measure of that convergence. However, often the output images are consistent after a certain number of iterations. Either of these criterion may be used as a stopping criterion. The stopping criteria can be changed by changing the main iterative loop for each algorithm. This can be found in CPSPECT.cpp at the following locations: line 501 for algorithm 3, line 695 for algorithms 4 and 6, line 1028 for algorithms 5 and 7. Again including this option in the argument list would be helpful.

D.2.5 Compiling the Code

CPSPECT was developed in NetBeans IDE. The makefiles were automatically generated by that program. The command `make all`, submitted from the source directory compiles both a “release” and “debug” configuration. The debug executable has gdb attached and therefore runs more slowly. Executables of both configurations are put in the `dist/` directory structure. To change the release configuration make target,

change the file nbproject/Makefile-Release. Lines 73-77 and 170 specify the make target.

APPENDIX E: C++ CODE

The complete C++ code implemented in the above thesis is listed below.

CPSPECT.cpp

```
/*
 * File:   CPSPECT.cpp
 * Author: wolfp
 *
 * Created on October 10, 2011, 1:32 PM
 */

#include "CPSPECT.h"
#include <string>
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <time.h>
#include <iomanip>
using namespace std;

CPSPECT::CPSPECT() {
    forward = NULL;
    back = NULL;
    systemMat = NULL;
}

CPSPECT::CPSPECT(const CPSPECT& orig) {
}

CPSPECT::~CPSPECT() {
    delete [] forward;
    delete [] back;
}

void CPSPECT::init(int numph, int numdet, int numangle, double FWHM) {

    //Dimensions of recon
    unsigned short xDim = 128;
    unsigned short yDim = 128;
    unsigned short zDim = 1;

    int locID = 0;
    int typeID = 1;

    // number of pinholes and detectors
    int pinnum = numph;
    int detnum = numdet;
    int ratio;

    unsigned short symmetry = 0;
```

```
unsigned short angleNum = numangle; //number of views
unsigned short pinHoleMatrix = 2; //Size of pinhole matrix
unsigned short viewTime = 0;

double halflife = 999999.0;

double yLength = 128.0; //physical size of detector
unsigned short yDetDim = 128; //number of detector bins
double zLength = 1.0;
unsigned short zDetDim = 1;

double p2dLength; // pinhole to detector length
double p2oLength; // pinhole to object length

double pinHoleDiameter = 2.0;
double psfFWHM = FWHM; //system blur
double pinHoleMatSizeY = 2.0;
double pinHoleMatSizeZ = 2.0;

double *yPHShift = new double[pinnum];
double *zPHShift = new double[pinnum];

// Compensation for off center detector
double yDetShift = 0.0;
double zDetShift = 0.0;
double xDetTilt = 0.0;
double yDetTilt = 0.0;
double zDetTilt = 0.0;

string fileNamePinholeModel = "";

//Initialize specific pinhole model
switch (pinnum) {
        //Single pinhole
    case 1:
        p2oLength = 35.0;
        p2dLength = 63.5;
        yPHShift[0] = 0.0;
        zPHShift[0] = 0.0;
        pinHoleDiameter = 1.5;
        break;
        //Two pinholes
    case 2:
        p2oLength = 44.5;
        p2dLength = 54.0;
        yPHShift[0] = 12.3;
        zPHShift[0] = 0.0;
        yPHShift[1] = -12.3;
        zPHShift[1] = 0.0;
        break;
        //Four pinholes
    case 4:
        p2oLength = 52.0;
        p2dLength = 46.5;
```

```
            yPHShift[0] = 10.7;
            zPHShift[0] = 12.8;
            yPHShift[1] = 10.7;
            zPHShift[1] = -12.8;
            yPHShift[2] = -10.7;
            zPHShift[2] = 12.8;
            yPHShift[3] = -10.7;
            zPHShift[3] = -12.8;
            break;
            //Nine pinholes
        case 9:
            p2oLength = 53.5;
            p2dLength = 45.0;
            yPHShift[0] = 21.3;
            zPHShift[0] = 0.0;
            yPHShift[1] = -21.3;
            zPHShift[1] = 0.0;
            yPHShift[2] = 15.0;
            zPHShift[2] = 15.0;
            yPHShift[3] = 15.0;
            zPHShift[3] = -15.0;
            yPHShift[4] = -15.0;
            zPHShift[4] = 15.0;
            yPHShift[5] = -15.0;
            zPHShift[5] = -15.0;
            yPHShift[6] = 0.0;
            zPHShift[6] = 15.0;
            yPHShift[7] = 0.0;
            zPHShift[7] = -15.0;
            yPHShift[8] = 0.0;
            zPHShift[8] = 0.0;
            break;
    }

    /* Initialize members of ReconData class */
    rData.init(xDim, yDim, zDim);
    rData.setProjPointer(&pData);
    rData.setLength((yLength * p2oLength / p2dLength));
    rData.initRotation();
    rData.init_backScalingFactors();

    /* Initialize members of ProjData class */
    pData.setReconPointer(&rData);
    pData.init(pinnum, detnum, yDim, zDim, yDetDim, zDetDim,
            angleNum, yLength, zLength, p2dLength, p2oLength,
            pinHoleMatrix, pinHoleDiameter, viewTime, halflife,
            psfFWHM, fileNamePinholeModel, pinHoleMatSizeY,
            pinHoleMatSizeZ, locID, typeID, yPHShift, zPHShift,
            yDetShift, zDetShift, xDetTilt, yDetTilt, zDetTilt,
symmetry);
    pData.initRayDrivenVariables();
    pData.initPinhole();
    pData.initPinholeFunctRound();
```

```
    if (pData.get_symmetry())
        ratio = 2;
    else
        ratio = 1;

    /* Initialize system matrix */
    systemMat = new SMPinholeRD(&pData, &rData, ratio);
    systemMat->buildMatrix();

    /* Initialize projectors */
    if (pData.get_symmetry())
        forward = new FPPinholeRaySym(&pData, &rData, systemMat);
    else
        forward = new FPPinholeRay(&pData, &rData, systemMat);

    if (pData.get_symmetry())
        back = new BPPinholeRaySym(&pData, &rData, systemMat);
    else
        back = new BPPinholeRay(&pData, &rData, systemMat);
}

bool CPSPECT::initReconstruction() {
    /* Sets up angles and scaling factors used by the projectors */
    int i = 0;
    int j = 0;

    angleNum = pData.get_angleNum();
    int order = angleNum;
    angleStep = angleNum / order;
    stepsize = pData.get_angleStepSize();
    angleSubsets = new int*[angleStep];
    radSubsets = new double*[angleStep];

    for (i = 0; i < angleStep; i++) {
        angleSubsets[i] = new int[order];
        radSubsets[i] = new double[order];
    }

    for (i = 0; i < angleStep; i++) {
        for (j = 0; j < order; j++) {
            angleSubsets[i][j] = i + j * angleStep;
        }
    }

    radSubsets[0][0] = 0;

    for (i = 0; i < angleStep; i++)
        for (j = 1; j < order; j++)
            radSubsets[i][j] = radSubsets[i][j - 1] + stepsize;

    if (angleStep == 1) {
        if (order >= 10) {
            radSubsets[0][10] = 120 * M_PIl / 180;
            radSubsets[0][20] = 2 * 120 * M_PIl / 180;
```

```
        }
        i = 0;
        for (j = 1; j < order; j++) {
            if (j != 10 && j != 20) {
                radSubsets[i][j] = radSubsets[i][j - 1] + stepsize;
            }
        }
    } else {
        for (i = 1; i < angleStep; i++) {
            radSubsets[i][0] = radSubsets[i - 1][0] + stepsize;
        }
    }

    for (i = 0; i < angleStep; i++)
        for (j = 1; j < order; j++)
            radSubsets[i][j] = radSubsets[i][j - 1] + stepsize;

    set_tbsf(order);
    set_scalingFactors(order);
}

void CPSPECT::set_scalingFactors(int order) {
    cout << "initiate scaling factors..." << endl;

    int i = 0;
    int a = 0;
    double theta = 0.0;

    rData.initRotation();

    // Set dataEst vector to 1
    for (i = 0; i < angleNum; i++)
        pData.set_dataEst(i, 1.0);


    rData.reset_objectEst();

    // Back project dataEst vector
    for (a = 0; a < order; a++) {
        theta = (radSubsets[0][a]);
        back->project(angleSubsets[0][a], theta, pData.get_dataEst(),
rData.get_objectEst());
    }


    rData.copy_backScalingFactors();
    rData.save_backScalingFactors("bsf.recon");
    pData.reset_dataEst();
}

void CPSPECT::set_tbsf(int order) {

    cout << "initiate lb..." << endl;
```

```
    int i = 0;

    for (i = 0; i < angleNum; i++)
        pData.set_dataEst(i, 1.0);


    rData.reset_objectEst();

    backProject(pData.get_dataEst(), rData.get_objectEst());


    rData.copy_tbackScalingFactors();
    rData.save_tbackScalingFactors("tbsf.recon");
    rData.reset_objectEst();

}

bool CPSPECT::reconstruct() {
    int order;

    initReconstruction();
    rData.init_recondata();

    order = angleNum;
}

double CPSPECT::calculateL(int alg) {
    //Implements listing 8 from Sidky,Jorgensen,Pan 2012

    rData.initVectors();

    int zdim = rData.get_zDim();
    int xdim = rData.get_xDim();
    int ydim = rData.get_yDim();

    double mag = 0;
    double val = 0;
    double err = 0;
    double L = 0;

    double** previousObj;
    previousObj = new double*[zdim];
    for (int i = 0; i < zdim; i++) {
        previousObj[i] = new double[xdim * ydim];
        memset(previousObj[i], 0, xdim * ydim * sizeof (double));
    }

    rData.setFirstObjectEst();

    for (int iter = 0; iter < 20; iter++) {
        mag = 0;
        err = 0;
        L = 0;
```

```
        //Save objectEstimate to compute error (difference between
iterations)
        for (int z = 0; z < zdim; z++) {
            for (int y = 0; y < ydim; y++) {
                for (int x = 0; x < xdim; x++) {
                    previousObj[z][x + (xdim * y)] =
rData.get_objectEst(z, y, x);
                }
            }
        }
        rData.saveSomeData(previousObj, "prevObj.recon");

        ////line 4

        // This is for the algorithms with TV penalties
        if (alg != 2) {
            rData.calculateGradient(rData.get_objectEst(),
rData.get_gradUbarVector());
            rData.calculateNegDivergence(rData.get_gradUbarVector(),
rData.get_objectUpdate());
        }
        //

        //Calculate Kx
        pData.reset_dataEst();
        forwardProject(rData.get_objectEst(), pData.get_dataEst());
        pData.save_dataEst("forward.proj");


        //Calculate K^TKx
        rData.reset_objectEst();
        backProject(pData.get_dataEst(), rData.get_objectEst());
        rData.saveSomeData(rData.get_objectEst(), "back.recon");

        // Only for algs with TV penalties
        if (alg != 2) {
            for (int z = 0; z < zdim; z++) {
                for (int x = 0; x < xdim; x++) {
                    for (int y = 0; y < ydim; y++) {
                        rData.set_objectEst(rData.get_objectEst(z, y,
x) + rData.get_objectUpdate(z, y, x), z, y, x);
                    }
                }
            }
        }
        //

        ////line 5
        //Calculate magnitude, used to normalize to unity
        for (int z = 0; z < zdim; z++) {
            for (int x = 0; x < xdim; x++) {
                for (int y = 0; y < ydim; y++) {
                    mag += rData.get_objectEst(z, y, x) *
rData.get_objectEst(z, y, x);
```

```
                }
            }
        }
        mag = sqrt(mag);

        // Normalize objectEstimate to unity (divide by magnitude),
calculate err
        for (int z = 0; z < zdim; z++) {
            for (int x = 0; x < xdim; x++) {
                for (int y = 0; y < ydim; y++) {
                    rData.set_objectEst(rData.get_objectEst(z, y, x) /
mag, z, y, x);
                    err += (rData.get_objectEst(z, y, x) -
previousObj[z][x + (xdim * y)]) * (rData.get_objectEst(z, y, x) -
previousObj[z][x + (xdim * y)]);
                }
            }
        }

        ////line6
        //Calculate L (they call it s) (apply A and take magnitude)
        forwardProject(rData.get_objectEst(), pData.get_dataEst());
        for (int a = 0; a < angleNum; a++) {
            for (int y = 0; y < pData.get_yDetectDim(); y++) {
                for (int z = 0; z < pData.get_zDetectDim(); z++) {
                    L += pData.get_dataEst(a, y, z) *
pData.get_dataEst(a, y, z);
                }
            }
        }

        // Only for TV penalty algs
        if (alg != 2) {
            rData.calculateGradient(rData.get_objectEst(),
rData.get_gradUbarVector());
            rData.calculateMagnitudeImage(rData.get_gradUbarVector(),
rData.get_magImage());
            for (int z = 0; z < zdim; z++) {
                for (int y = 0; y < ydim; y++) {
                    for (int x = 0; x < xdim; x++) {
                        val = rData.get_magImage(z, y, x) *
rData.get_magImage(z, y, x);
                        L = L + val;
                    }
                }
            }
        }
        L = sqrt(L);

        cout << "iter: " << iter << "\t" << "err: " << err << "\tL: "
<< L << endl;
    }

    rData.reset_objectEst();
```

```
    pData.reset_dataEst();

    for (int i = 0; i < zdim; i++) {
        delete []previousObj[i];
    }
    delete[] previousObj;
    return L;
}

void CPSPECT::leastSquares(double L, string filename) {
    //Implements listing 2 from Sidky,Jorgensen,Pan 2012
    rData.init_recondata();
    int iter = 1000000;

    double sigma = 1 / L;
    double tau = 1 / L;
    double theta = 1.0;
    double val = 0;
    double dotProd = 0;
    double dataEstNormSq = 0;
    double leastSq = 0;
    double SSE = 0;
    double maxEl = 0;
    double minEl = 0;
    fstream filestr;

    double iP;
    double mse;

    char * siter = "0";
    siter = new char[25];

    rData.reset_objectEst();
    rData.reset_objectUpdate();

    pData.reset_dataEst();
    pData.reset_prevDataEst();

    for (int z = 0; z < rData.get_zDim(); z++)
        for (int y = 0; y < rData.get_yDim(); y++)
            for (int x = 0; x < rData.get_xDim(); x++) {
                if ((x + y) % 2 == 0)
                    rData.set_objectEst(1.0, z, y, x);
                else
                    rData.set_objectEst(0.0, z, y, x);
                pData.set_projData(x, y, z, 1.0);
            }

    pData.reset_dataEst();
    rData.saveSomeData(rData.get_objectEst(), "initialObject.recon");
    forwardProject(rData.get_objectEst(), pData.get_dataEst());
    iP = 0;
    cout << iP << endl;
    for (int a = 0; a < pData.get_angleNum(); a++)
```

```
        for (int y = 0; y < pData.get_yDetectDim(); y++)
            for (int z = 0; z < pData.get_zDetectDim(); z++) {
                iP += pData.get_dataEst(a, y, z) *
pData.get_projData(a, y, z);
            }
    pData.save_dataEst("fpO.recon");
    cout << iP << endl;

    pData.save_dataEst("initialProj.proj");
    rData.reset_objectUpdate();
    backProject(pData.get_projData(), rData.get_objectUpdate());

    iP = 0;
    cout << iP << endl;
    for (int z = 0; z < rData.get_zDim(); z++)
        for (int y = 0; y < rData.get_yDim(); y++)
            for (int x = 0; x < rData.get_xDim(); x++) {
                iP += rData.get_objectUpdate(z, y, x) *
rData.get_objectEst(z, y, x);
            }
    rData.saveSomeData(rData.get_objectUpdate(), "bpO.recon");
    cout << iP << endl;

    cout << "stop" << endl;


    if (!filename.compare("ic")) {
        // InverseCrime Case

rData.load_objectEst("/home/wolfp/Desktop/inverseCrimeInput/2Sigmafaker
econGaussSmall");
        forwardProject(rData.get_objectEst(), pData.get_projData());
        pData.save_projData("projData.proj");
    } else {
        pData.load(filename);
    }

    rData.reset_objectEst(); //ubar
    rData.reset_objectUpdate(); //u

    pData.reset_dataEst(); //p
    pData.reset_prevDataEst(); //pn-1

    for (int n = 0; n < iter; n++) {
        //pn+1
        pData.set_prevDataEst();
        pData.reset_dataEst();
        forwardProject(rData.get_objectEst(), pData.get_dataEst());
        for (int a = 0; a < angleNum; a++)
            for (int y = 0; y < pData.get_yDetectDim(); y++)
                for (int z = 0; z < pData.get_zDetectDim(); z++) {
                    val = pData.get_prevDataEst(a, y, z) + sigma *
                            ((pData.get_dataEst(a, y, z) -
pData.get_projData(a, y, z)));
```

```
                    val = val / (1 + sigma);
                    pData.set_dataEst(a, y, z, val);
                }
        //un+1
        rData.set_prevObjectUpdate();
        rData.saveSomeData(rData.get_prevObjectUpdate(),
"prevObjectUpdate");
        rData.reset_objectEst();
        backProject(pData.get_dataEst(), rData.get_objectEst());
        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    val = rData.get_objectUpdate(z, y, x) -
                            (tau * rData.get_objectEst(z, y, x));
                    rData.set_objectUpdate(val, z, y, x);
                }

        //enforce positivity on un+1 for algorithm 3
        rData.enforcePositivity(rData.get_objectUpdate());
        rData.reset_objectEst();
        //ubarn+1
        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    val = rData.get_objectUpdate(z, y, x) + theta *
                            (rData.get_objectUpdate(z, y, x) -
rData.get_prevObjectUpdate(z, y, x));
                    rData.set_objectEst(val, z, y, x);
                }

        // save output
        sprintf(siter, "u%d", n);
        rData.saveSomeData(rData.get_objectUpdate(), siter);

        for (int z = 0; z < rData.get_zDim(); z++)
            rData.applyGaussianBlur(z, rData.get_prevObjectUpdate());
        sprintf(siter, "uprime%d", n);
        rData.saveSomeData(rData.get_prevObjectUpdate(), siter);


        //check convergence
        //PD(u',p')
        /* prevDataEst can be used here because it is reassigned at the
top
         * of the loop */
        dotProd = 0;
        dataEstNormSq = 0;
        leastSq = 0;
        SSE = 0;
        pData.reset_prevDataEst();
        forwardProject(rData.get_objectUpdate(),
pData.get_prevDataEst());

        mse = pData.CalMSE(pData.get_prevDataEst());
```

```
        for (int a = 0; a < angleNum; a++)
            for (int y = 0; y < pData.get_yDetectDim(); y++)
                for (int z = 0; z < pData.get_zDetectDim(); z++) {
                    dotProd += pData.get_dataEst(a, y, z) *
pData.get_projData(a, y, z);
                    dataEstNormSq += pData.get_dataEst(a, y, z) *
pData.get_dataEst(a, y, z);
                    leastSq += (pData.get_prevDataEst(a, y, z) -
pData.get_projData(a, y, z)) *
                            (pData.get_prevDataEst(a, y, z) -
pData.get_projData(a, y, z));
                }
        //PD(u',p')
        val = leastSq + (0.5 * dataEstNormSq) + dotProd;
        //Calculate A^Tp
        /* prevObjectUpdate can be used here.  It is reassigned before
         * its next use */
        rData.reset_someData(rData.get_prevObjectUpdate());
        backProject(pData.get_dataEst(), rData.get_prevObjectUpdate());
        // All elements should be 0...i'll report the maximal element
        maxEl = 0;
        minEl = 1000;
        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    if (rData.get_prevObjectUpdate(z, y, x) > maxEl)
                        maxEl = rData.get_prevObjectUpdate(z, y, x);
                    if (rData.get_prevObjectUpdate(z, y, x) < minEl)
                        minEl = rData.get_prevObjectUpdate(z, y, x);
                }

        cout << "iter: " << n << "\tPD(u',p') = " << val << "\tMSE = "
<< mse << "\tA^Tp max = " << maxEl << "\tA^Tp min = " << minEl << endl;

        filestr.open("output", fstream::out | fstream::app);
        filestr << "iter: " << n << "\tPD(u',p') = " << val << "\tA^Tp
max = " << maxEl << "\tA^Tp min = " << minEl << "\tMSE" << mse << endl;
        filestr.close();
    }
}

void CPSPECT::leastSquaresPlusTV(double L, string filename, double lam)
{
    //Implements listing 4 from Sidky,Jorgensen,Pan 2012
    rData.init_recondata();
    int n = 0;

    double iP = 0;
    double sigma = 1 / L;
    double tau = 1 / L;
    double theta = 1.0;
    double val = 0;
    double dotProd = 0;
```

```
    double dataEstNormSq = 0;
    double leastSq = 0;
    double SSE = 0;
    double l1norm = 0;
    double lambda = lam;
    double minEl, maxEl;
    double mse = 0;
    double PD = 100;
    fstream filestr;

    char * siter = "0";
    siter = new char[25];
    rData.initVectors();


    rData.reset_objectEst();

    pData.reset_dataEst();
    pData.reset_prevDataEst();

    for (int z = 0; z < rData.get_zDim(); z++)
        for (int y = 0; y < rData.get_yDim(); y++)
            for (int x = 0; x < rData.get_xDim(); x++) {
                if ((x + y) % 2 == 0)
                    rData.set_objectEst(1.0, z, y, x);
                else
                    rData.set_objectEst(0.0, z, y, x);
            }

    for (int a = 0; a < pData.get_angleNum(); a++)
        for (int z = 0; z < pData.get_zDetectDim(); z++)
            for (int y = 0; y < pData.get_yDetectDim(); y++)
                pData.set_projData(a, y, z, 1.0);

    //Checks that projectors are transpose
    pData.reset_dataEst();
    rData.saveSomeData(rData.get_objectEst(), "initialObject.recon");
    forwardProject(rData.get_objectEst(), pData.get_dataEst());
    iP = 0;
    cout << iP << endl;
    for (int a = 0; a < pData.get_angleNum(); a++)
        for (int y = 0; y < pData.get_yDetectDim(); y++)
            for (int z = 0; z < pData.get_zDetectDim(); z++) {
                iP += pData.get_dataEst(a, y, z) *
pData.get_projData(a, y, z);
            }
    pData.save_dataEst("fpO.recon");
    cout << iP << endl;

    rData.reset_objectUpdate();
    backProject(pData.get_projData(), rData.get_objectUpdate());

    iP = 0;
    cout << iP << endl;
```

```
    for (int z = 0; z < rData.get_zDim(); z++)
        for (int y = 0; y < rData.get_yDim(); y++)
            for (int x = 0; x < rData.get_xDim(); x++) {
                iP += rData.get_objectUpdate(z, y, x) *
rData.get_objectEst(z, y, x);
            }
    rData.saveSomeData(rData.get_objectUpdate(), "bpO.recon");
    cout << iP << endl;



    if (!filename.compare("ic")) {
        // InverseCrime Case
        if (rData.get_xDim() == 128) {

rData.load_objectEst("/home/wolfp/Desktop/inverseCrimeInput/2Sigmafaker
econGauss");
        } else if (rData.get_xDim() == 64) {
            if (blurFlag)

rData.load_objectEst("/home/wolfp/newPhantom/newPhantom64");
            else

rData.load_objectEst("/home/wolfp/newPhantom/newPhantom64");
        }
        forwardProject(rData.get_objectEst(), pData.get_projData());
        pData.save_projData("projData.proj");
    } else {
        pData.load(filename);
    }

    rData.reset_objectEst(); //ubar (output of bp input of fp)
    rData.reset_objectUpdate(); //u
    rData.reset_vectors(); //q and such

    pData.reset_dataEst(); //p (output of fp input of bp)
    pData.reset_prevDataEst(); //pn-1


    while (fabs(PD) > 0.00001) {
        //pn+1
        pData.set_prevDataEst();
        forwardProject(rData.get_objectEst(), pData.get_dataEst());
        for (int a = 0; a < angleNum; a++)
            for (int y = 0; y < pData.get_yDetectDim(); y++)
                for (int z = 0; z < pData.get_zDetectDim(); z++) {
                    val = pData.get_prevDataEst(a, y, z) + sigma *
                            ((pData.get_dataEst(a, y, z) -
pData.get_projData(a, y, z)));
                    val = val / (1 + (sigma));
                    pData.set_dataEst(a, y, z, val);
                }
        //qn+1
```

```
        rData.calculateGradient(rData.get_objectEst(),
rData.get_gradUbarVector());

        for (int buffer = 0; buffer < 2; buffer++) {
            for (int z = 0; z < rData.get_zDim(); z++)
                for (int y = 0; y < rData.get_yDim(); y++)
                    for (int x = 0; x < rData.get_xDim(); x++) {
                        val = rData.get_qVector(buffer, z, y, x)
                                + sigma *
rData.get_gradUbarVector(buffer, z, y, x);
                        rData.set_qVector(val, buffer, z, y, x);
                    }
        }

        rData.calculateMagnitudeImageLB(lambda, rData.get_qVector(),
rData.get_magImage());

        for (int buffer = 0; buffer < 2; buffer++) {
            for (int z = 0; z < rData.get_zDim(); z++)
                for (int y = 0; y < rData.get_yDim(); y++)
                    for (int x = 0; x < rData.get_xDim(); x++) {
                        val = lambda * rData.get_qVector(buffer, z, y,
x)
                                / rData.get_magImage(z, y, x);
                        rData.set_qVector(val, buffer, z, y, x);
                    }
        }

        //un+1
        rData.calculateNegDivergence(rData.get_qVector(),
rData.get_divQ());
        rData.set_prevObjectUpdate();
        backProject(pData.get_dataEst(), rData.get_objectEst());
        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    val = rData.get_objectUpdate(z, y, x) -
                            (tau * rData.get_objectEst(z, y, x)) - (tau
* rData.get_divQ(z, y, x));
                    rData.set_objectUpdate(val, z, y, x);
                }
        //enforce positivity on un+1 for algorithm 5
        //        rData.enforcePositivity(rData.get_objectUpdate());
        //ubarn+1
        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    val = rData.get_objectUpdate(z, y, x) + theta *
                            (rData.get_objectUpdate(z, y, x) -
rData.get_prevObjectUpdate(z, y, x));
                    rData.set_objectEst(val, z, y, x);
                }

        // save output
```

```
        if (n % 100 == 0) {
            sprintf(siter, "u%d", n);
            rData.saveSomeData(rData.get_objectUpdate(), siter);


            rData.set_prevObjectUpdate();
            if (blurFlag) {
                for (int z = 0; z < rData.get_zDim(); z++)
                    rData.applyGaussianBlur(z,
rData.get_prevObjectUpdate());
                rData.enforceFOV(rData.get_prevObjectUpdate());
                sprintf(siter, "uprime%d", n);
                rData.saveSomeData(rData.get_prevObjectUpdate(),
siter);
            }
        }

        //check convergence
        //PD(u',p')
        /* prevDataEst can be used here because it is reassigned at the
top
         * of the loop.  also ubarVector and magImage */
        dotProd = 0;
        dataEstNormSq = 0;
        leastSq = 0;
        SSE = 0;
        l1norm = 0;
        forwardProject(rData.get_objectUpdate(),
pData.get_prevDataEst());

        mse = pData.CalMSE(pData.get_prevDataEst());
        rData.calculateGradient(rData.get_objectUpdate(),
rData.get_gradUbarVector());
        rData.calculateMagnitudeImage(rData.get_gradUbarVector(),
rData.get_magImage());

        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    l1norm += fabs(rData.get_magImage(z, y, x));
                }

        for (int a = 0; a < angleNum; a++)
            for (int y = 0; y < pData.get_yDetectDim(); y++)
                for (int z = 0; z < pData.get_zDetectDim(); z++) {
                    dotProd += pData.get_dataEst(a, y, z) *
pData.get_projData(a, y, z);
                    dataEstNormSq += pData.get_dataEst(a, y, z) *
pData.get_dataEst(a, y, z);
                    leastSq += (pData.get_prevDataEst(a, y, z) -
pData.get_projData(a, y, z)) *
                            (pData.get_prevDataEst(a, y, z) -
pData.get_projData(a, y, z));
                }
```

```
        //PD(u',p')
        PD = 0.5 * leastSq + lambda * l1norm + 0.5 * dataEstNormSq +
dotProd;
        //A^Tp
        /* prevObjectUpdate can be used here.  It is reassigned before
         * its next use */
        backProject(pData.get_dataEst(), rData.get_prevObjectUpdate());
        // All elements should be 0...i'll report the maximal element
        maxEl = 0;
        minEl = 1000;
        maxEl = 0;
        minEl = 1000;
        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    if (-rData.get_prevObjectUpdate(z, y, x) -
rData.get_divQ(z, y, x) > maxEl)
                        maxEl = fabs(-rData.get_prevObjectUpdate(z, y,
x) - rData.get_divQ(z, y, x));
                    if (-rData.get_prevObjectUpdate(z, y, x) -
rData.get_divQ(z, y, x) < minEl)
                        minEl = -rData.get_prevObjectUpdate(z, y, x) -
rData.get_divQ(z, y, x);
                }
        cout << "iter: " << n << "\tPD(u',p') = " << fixed <<
setprecision(5) << PD << "\tMSE = " << mse << "\tA^Tp max = " << maxEl
<< "\tA^Tp min = " << minEl << endl;


        filestr.open("output", fstream::out | fstream::app);
        filestr << "iter: " << n << "\tPD(u',p') = " << fixed <<
setprecision(5) << PD << "\tA^Tp max = " << maxEl << "\tA^Tp min = " <<
minEl << "\tMSE = " << mse << endl;
        filestr.close();
        n++;

    }
}

void CPSPECT::systemCheck() {
    //Check system matrices and print them out.  This takes a long time
and is a
    //pain in the ass
    FILE * pFile;
    pFile = fopen("fpSystemMat", "wb");

    pData.reset_dataEst();
    rData.reset_objectEst();
    for (int i = 0; i < rData.get_xDim() * rData.get_yDim(); i++) {
        rData.set_objectEst(1.0, 0, i);
        pData.reset_dataEst();
        forwardProject(rData.get_objectEst(), pData.get_dataEst());
        fwrite(pData.get_dataEst(), sizeof (double), 128 * 128, pFile);
        rData.reset_objectEst();
```

```
        cout << i << endl;
    }
    fclose(pFile);

    pFile = fopen("bpSystemMat", "wb");

    pData.reset_dataEst();
    rData.reset_objectEst();
    for (int a = 0; a < pData.get_angleNum(); a++) {
        for (int i = 0; i < pData.get_zDetectDim() *
pData.get_yDetectDim(); i++) {
            pData.set_dataEst(a, i, 1.0);
            rData.reset_objectEst();
            backProject(pData.get_dataEst(), rData.get_objectEst());
            fwrite(pData.get_dataEst(), sizeof (double), 128 * 128,
pFile);
            pData.reset_dataEst();
            cout << a << "\t" << i << endl;
        }
    }
    fclose(pFile);

}

void CPSPECT::KLPlusTV(double L, string filename, double lam) {
    //Implements listing 5 from Sidky,Jorgensen,Pan 2012
    rData.init_recondata();

    int iter = 100000000;

    clock_t init, final;

    double PD = 100;
    double iP = 0;
    double root = 0;
    double sigma = 0.4 / L;
    double tau = 0.4 / L;
    double theta = 1.0;
    double val = 0;
    double dotProd = 0;
    double dataEstNormSq = 0;
    double leastSq = 0;
    double SSE = 0;
    double l1norm = 0;
    double lambda = lam;
    double minEl = 1000;
    double maxEl = 0;
    double mse = 0;
    double dkl = 0;
    fstream filestr;

    char * siter = "0";
    siter = new char[25];
    rData.initVectors();
```

```
    rData.reset_objectEst(); //ubar (output of bp input of fp)
    //    rData.reset_objectUpdate(); //u

    pData.reset_dataEst(); //p (output of fp input of bp)
    pData.reset_prevDataEst(); //pn-1

    for (int z = 0; z < rData.get_zDim(); z++)
        for (int y = 0; y < rData.get_yDim(); y++)
            for (int x = 0; x < rData.get_xDim(); x++) {
                if ((x + y) % 2 == 0)
                    rData.set_objectEst(1.0, z, y, x);
                else
                    rData.set_objectEst(0.0, z, y, x);
            }

    for (int a = 0; a < pData.get_angleNum(); a++)
        for (int z = 0; z < pData.get_zDetectDim(); z++)
            for (int y = 0; y < pData.get_yDetectDim(); y++)
                pData.set_projData(a, y, z, 1.0);

    //Check that everything is transpose

/*********************************************************************
********/
    rData.reset_vectors();

    for (int b = 0; b < 2; b++)
        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    if ((x + y) % 2 == 0) {
                        rData.set_objectEst(1, z, y, x);
                        rData.set_qVector(1, b, z, y, x);
                    } else {
                        rData.set_objectEst(y / ((x + 1)*(x + 1)), z,
y, x);
                        rData.set_qVector(y / ((x + 1)*(x + 1)), b, z,
y, x);
                    }
                }

    rData.calculateGradient(rData.get_objectEst(),
rData.get_gradUbarVector());

    iP = 0;
    cout << iP << endl;
    for (int b = 0; b < 2; b++)
        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    iP += rData.get_qVector(b, z, y, x) *
rData.get_gradUbarVector(b, z, y, x);
```

```
                }
    cout << iP << endl;


    rData.reset_someData(rData.get_divQ());

    rData.calculateNegDivergence(rData.get_qVector(),
rData.get_divQ());

    iP = 0;
    cout << iP << endl;
    for (int z = 0; z < rData.get_zDim(); z++)
        for (int y = 0; y < rData.get_yDim(); y++)
            for (int x = 0; x < rData.get_xDim(); x++) {
                iP += rData.get_divQ(z, y, x) * rData.get_objectEst(z,
y, x);
            }

    cout << iP << endl;

    pData.reset_dataEst();
    rData.saveSomeData(rData.get_objectEst(), "initialObject.recon");
    init = clock();
    forwardProject(rData.get_objectEst(), pData.get_dataEst());
    final = clock() - init;
    cout << "FP time:\t" << (double) final / ((double) CLOCKS_PER_SEC)
<< endl;

    iP = 0;
    cout << iP << endl;
    for (int a = 0; a < pData.get_angleNum(); a++)
        for (int y = 0; y < pData.get_yDetectDim(); y++)
            for (int z = 0; z < pData.get_zDetectDim(); z++) {
                iP += pData.get_dataEst(a, y, z) *
pData.get_projData(a, y, z);
            }
    pData.save_dataEst("fpO.recon");
    cout << iP << endl;

    pData.save_dataEst("initialProj.proj");
    rData.reset_objectUpdate();
    init = clock();
    backProject(pData.get_projData(), rData.get_objectUpdate());
    final = clock() - init;
    cout << "BP time:\t" << (double) final / ((double) CLOCKS_PER_SEC)
<< endl;
    iP = 0;
    cout << iP << endl;
    for (int z = 0; z < rData.get_zDim(); z++)
        for (int y = 0; y < rData.get_yDim(); y++)
            for (int x = 0; x < rData.get_xDim(); x++) {
                iP += rData.get_objectUpdate(z, y, x) *
rData.get_objectEst(z, y, x);
            }
```

```
    rData.saveSomeData(rData.get_objectUpdate(), "bp0.recon");
    cout << iP << endl;
/
**********************************************************************
*******/

    if (!filename.compare("ic")) {
        // InverseCrime Case
        if (rData.get_xDim() == 128) {
            rData.load_objectEst("/home/wolfp/newPhantom/PWC1HI");
        } else if (rData.get_xDim() == 64) {
            if (blurFlag) {
                rData.load_objectEst("/home/wolfp/newPhantom/PWC164");
                cout << "loaded" << endl;
            } else {

rData.load_objectEst("/home/wolfp/newPhantom/newPhantom64");
            }
        }

        forwardProject(rData.get_objectEst(), pData.get_projData());

        pData.save_projData("projData.proj");
        if (blurFlag) {
            rData.applyGaussianBlur(0, rData.get_objectEst());
            rData.enforceFOV(rData.get_objectEst());
            rData.saveSomeData(rData.get_objectEst(), "blurredobject");
        }
    } else {
        pData.load(filename);
    }

    rData.reset_objectEst(); //ubar (output of bp input of fp)
    rData.reset_objectUpdate(); //u
    rData.reset_vectors();

    pData.reset_dataEst(); //p (output of fp input of bp)
    pData.reset_prevDataEst(); //pn-1

    int n = 0;
    while (n < 200000) {//(fabs(PD) > 0.00001) {
        maxEl = 0;
        minEl = 1000;
        //pn+1
        pData.reset_data(pData.get_prevDataEst());
        pData.set_prevDataEst();


        rData.enforceFOV(rData.get_objectEst());

        pData.reset_dataEst();
        forwardProject(rData.get_objectEst(), pData.get_dataEst());
        for (int a = 0; a < angleNum; a++)
            for (int y = 0; y < pData.get_yDetectDim(); y++)
```

```
                for (int z = 0; z < pData.get_zDetectDim(); z++) {
                    val = 1 + pData.get_prevDataEst(a, y, z) + (sigma *
                            pData.get_dataEst(a, y, z));

                    root = (pData.get_prevDataEst(a, y, z) + sigma *
                            pData.get_dataEst(a, y, z) - 1)*
                            (pData.get_prevDataEst(a, y, z) + sigma *
                            pData.get_dataEst(a, y, z) - 1) +
                            (4 * sigma * pData.get_projData(a, y, z));
                    if (root < 0)
                        cout << "big trouble" << endl;

                    root = sqrt(root);
                    val = 0.5 * (val - root);
                    pData.set_dataEst(a, y, z, val);
                }
        //qn+1
        rData.calculateGradient(rData.get_objectEst(),
rData.get_gradUbarVector());
        for (int buffer = 0; buffer < 2; buffer++) {
            for (int z = 0; z < rData.get_zDim(); z++)
                for (int y = 0; y < rData.get_yDim(); y++)
                    for (int x = 0; x < rData.get_xDim(); x++) {
                        val = rData.get_qVector(buffer, z, y, x)
                                + sigma *
rData.get_gradUbarVector(buffer, z, y, x);
                        rData.set_qVector(val, buffer, z, y, x);
                    }
        }

        rData.calculateMagnitudeImageLB(lambda, rData.get_qVector(),
rData.get_magImage());

        rData.saveSomeData(rData.get_magImage(), "magImage");

        for (int buffer = 0; buffer < 2; buffer++) {
            for (int z = 0; z < rData.get_zDim(); z++)
                for (int y = 0; y < rData.get_yDim(); y++)
                    for (int x = 0; x < rData.get_xDim(); x++) {
                        val = lambda * rData.get_qVector(buffer, z, y,
x)
                                / rData.get_magImage(z, y, x);
                        rData.set_qVector(val, buffer, z, y, x);
                    }
        }

        //un+1
        rData.calculateNegDivergence(rData.get_qVector(),
rData.get_divQ());
        rData.set_prevObjectUpdate();
        rData.reset_objectEst();

        backProject(pData.get_dataEst(), rData.get_objectEst()); //A^T
p
```

```
        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    val = rData.get_objectUpdate(z, y, x) -
                            (tau * rData.get_objectEst(z, y, x)) - (tau
* rData.get_divQ(z, y, x));
                    rData.set_objectUpdate(val, z, y, x);
                }


        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    if (-rData.get_objectEst(z, y, x) -
rData.get_divQ(z, y, x) > maxEl)
                        maxEl = -rData.get_objectEst(z, y, x) -
rData.get_divQ(z, y, x);
                    if (-rData.get_objectEst(z, y, x) -
rData.get_divQ(z, y, x) < minEl)
                        minEl = -rData.get_objectEst(z, y, x) -
rData.get_divQ(z, y, x);
                }

        rData.enforceFOV(rData.get_objectUpdate());
        //enforce positivity on un+1
        rData.enforcePositivity(rData.get_objectUpdate());
        //ubarn+1
        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    val = rData.get_objectUpdate(z, y, x) + theta *
                            (rData.get_objectUpdate(z, y, x) -
rData.get_prevObjectUpdate(z, y, x));
                    rData.set_objectEst(val, z, y, x);
                }

        // save output
        //        if (n % 100 == 0) {
        sprintf(siter, "u%d", n);
        rData.saveSomeData(rData.get_objectUpdate(), siter);

        sprintf(siter, "ubar%d", n);
        rData.saveSomeData(rData.get_objectEst(), siter);

        rData.set_prevObjectUpdate();
        if (blurFlag) {
            for (int z = 0; z < rData.get_zDim(); z++)
                rData.applyGaussianBlur(z,
rData.get_prevObjectUpdate());
            rData.enforceFOV(rData.get_prevObjectUpdate());
            sprintf(siter, "uprime%d", n);
            rData.saveSomeData(rData.get_prevObjectUpdate(), siter);
        }
        //        }
```

```
        //check convergence
        //PD(u',p')
        /* prevDataEst can be used here because it is reassigned at the
top
         * of the loop.  also ubarVector and magImage */
        dotProd = 0;
        dataEstNormSq = 0;
        leastSq = 0;
        SSE = 0;

        forwardProject(rData.get_objectUpdate(),
pData.get_prevDataEst());

        mse = pData.CalMSE(pData.get_prevDataEst());
        dkl = pData.CalDKL(pData.get_projData(),
pData.get_prevDataEst());

        rData.enforceFOV(rData.get_objectUpdate());

        rData.calculateGradient(rData.get_objectUpdate(),
rData.get_gradUbarVector());
        rData.calculateMagnitudeImage(rData.get_gradUbarVector(),
rData.get_magImage());

        l1norm = 0;
        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    if (rData.get_magImage(z, y, x) >= 1E-8)
                        l1norm += fabs(rData.get_magImage(z, y, x));
                    else
                        l1norm += 0;


                }

        for (int a = 0; a < angleNum; a++)
            for (int y = 0; y < pData.get_yDetectDim(); y++)
                for (int z = 0; z < pData.get_zDetectDim(); z++) {
                    dotProd += pData.get_dataEst(a, y, z) *
pData.get_projData(a, y, z);
                    dataEstNormSq += pData.get_dataEst(a, y, z) *
pData.get_dataEst(a, y, z);
                    leastSq += (pData.get_prevDataEst(a, y, z) -
pData.get_projData(a, y, z)) *
                            (pData.get_prevDataEst(a, y, z) -
pData.get_projData(a, y, z));
                }
        //PD(u',p')
        //        cout<<pData.CalDKL(pData.get_projData(),
pData.get_prevDataEst())<<"\t"<<pData.klDual()<<endl;
        val = (pData.CalDKL(pData.get_projData(),
pData.get_prevDataEst()) + (lambda * l1norm)) - (pData.klDual());
```

```
        cout << "iter: " << n << "\tPD(u',p') = " << fixed <<
setprecision(6) << val << "\tMSE = " << mse << "\tDKL = " << dkl <<
"\tTV(noBlur) = " << l1norm << "\tA^Tp max = " << maxEl << "\tA^Tp min
= " << minEl << endl;


        filestr.open("output", fstream::out | fstream::app);
        filestr << "iter: " << n << "\tPD(u',p') = " << fixed <<
setprecision(6) << val << "\tA^Tp max = " << maxEl << "\tA^Tp min = "
<< minEl << "\tMSE = " << mse << "\tDKL = " << dkl << "\tTV = " <<
l1norm << endl;
        filestr.close();

        PD = val;
        n++;
    }
    sprintf(siter, "uFinal");
    rData.saveSomeData(rData.get_objectUpdate(), siter);


    rData.set_prevObjectUpdate();
    if (blurFlag) {
        for (int z = 0; z < rData.get_zDim(); z++)
            rData.applyGaussianBlur(z, rData.get_prevObjectUpdate());
        rData.enforceFOV(rData.get_prevObjectUpdate());
        sprintf(siter, "uprimeFinal");
        rData.saveSomeData(rData.get_prevObjectUpdate(), siter);
    }
}
```

CPSPECT.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/*
 * File:   CPSPECT.h
 * Author: wolfp
 *
 * Created on October 10, 2011, 1:32 PM
 */

#ifndef _CPSPECT_H
#define     _CPSPECT_H
#include "projdata.h"
#include "backprojector.h"
#include "forwardprojector.h"
#include "point3d.h"
#include "projdata.h"
#include "projectorenum.h"
#include "systemmatrix.h"
#include "bppinholeray.h"
#include "bppinholeraysym.h"
#include "fppinholeray.h"
#include "fppinholeraysym.h"
#include "smpinholerd.h"
```

```
class CPSPECT {
public:
    CPSPECT();
    CPSPECT(const CPSPECT& orig);
    virtual ~CPSPECT();

    /** The projection data set */
    ProjData pData;
    /** The reconstruction data set */
    ReconData rData;

    void init(int numph, int numdet, int numangle, double psfFWHM
/*input arguments from cmd line*/);
    bool reconstruct();
    bool initReconstruction();
    void set_tbsf(int order);
    void set_scalingFactors(int order);

    double calculateL(int alg);
    void leastSquares(double L, string filename);
    void leastSquaresPlusTV(double L, string filename, double lam);
    void systemCheck();
    void KLPlusTV(double L, string filename, double lam);

    void setBlurFlag(int flag) {
        blurFlag = flag;
    }

    // Perform forward projection in a easy to use way

    void forwardProject() {
        //not for use with blurring model
        int s = 0;
        rData.enforceFOV();
        // NEED BEFORE EVERY FP
        pData.reset_dataEst();
        //Forward project (apply A)
        for (int a = 0; a < angleNum; a++) {
            forward->project(angleSubsets[s][a], -(radSubsets[s][a]));
        }
    };

    //includes blurring and masking
    void forwardProject(double** from, double** to) {
        int s = 0;
        //create scratch space to blur
        double **scratch;
        scratch = new double*[rData.get_zDim()];
        for (int i = 0; i < rData.get_zDim(); i++) {
            scratch[i] = new double[rData.get_xDim() *
rData.get_yDim()];
        }
```

```
        for (int i = 0; i < rData.get_zDim(); i++) {
            memset(scratch[i], 0, rData.get_xDim() * rData.get_yDim() *
sizeof (double));
        }


        for (int z = 0; z < rData.get_zDim(); z++)
            for (int y = 0; y < rData.get_yDim(); y++)
                for (int x = 0; x < rData.get_xDim(); x++) {
                    scratch[z][x + (y * rData.get_xDim())] =
rData.get_data(z, y, x, from);
                }

        rData.enforceFOV(scratch);

        if (blurFlag) {
            for (int z = 0; z < rData.get_zDim(); z++)
                rData.applyGaussianBlur(z, scratch);
        }

        rData.saveSomeData(scratch, "blurred.recon");
        //
        rData.enforceFOV(scratch);

        // NEED BEFORE EVERY FP
        pData.reset_data(to);
        //Forward project (apply A)
        for (int a = 0; a < angleNum; a++) {
            forward->project(angleSubsets[s][a], -(radSubsets[s][a]),
scratch, to);
        }

        for (int i = 0; i < rData.get_zDim(); i++)
            delete[] scratch[i];
        delete[] scratch;
    };

    //Backprojection
    void backProject() {
        //not for use with backprojection
        int s = 0;
        //NEED BEFORE EVERY BP
        rData.reset_objectEst();
        //Backproject (apply A^t)
        for (int a = 0; a < angleNum; a++) {
            back->project(angleSubsets[s][a], (radSubsets[s][a]));
        }
        rData.enforceFOV();
    };

    //includes blurring and maskign
    void backProject(double** from, double** to) {
        int s = 0;
```

```
        //NEED BEFORE EVERY BP
        rData.reset_someData(to);
        //Backproject (apply A^t)
        for (int a = 0; a < angleNum; a++) {
            back->project(angleSubsets[s][a], (radSubsets[s][a]), from,
to);
        }

        rData.enforceFOV(to);

        if (blurFlag) {
            for (int z = 0; z < rData.get_zDim(); z++)
                rData.applyGaussianBlur(z, to);
        }

        rData.enforceFOV(to);

    }

private:
    ForwardProjector *forward;
    BackProjector *back;
    SystemMatrix *systemMat;

    int angleStep;
    double stepsize;
    int angleNum;
    int **angleSubsets;
    int detectnum;
    double **radSubsets;
    double **sparseSum;
    double **stationarySparseSum;
    double **gradient;
    double **enforcedGradient;
    double **temp;
    int K;
    double *gamma;
    int waveOrder;

    int blurFlag;

    struct element {
        double value;
        int xloc, yloc, imloc;
    };

};

#endif      /* _CPSPECT_H */
```

backprojector.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
```

```
/
***************************************************************************
****
                          backprojector.cpp  -  description
                             -------------------
    begin                : Thu Jan 06 09:11:48 CST 2005
    copyright            : (C) 2005 by Christian Wietholt
    email                : cwietholt@nhri.org.tw

***************************************************************************
****/

/
***************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

***************************************************************************
****/
#include "backprojector.h"

#include "recondata.h"
#include "projdata.h"

/**
 * Constructors/Destructors
 */
BackProjector::BackProjector() {
}

BackProjector::BackProjector(ProjData *p, ReconData *r) {
    proj = p;
    recon = r;
    xDim = recon->get_xDim();
    yDim = recon->get_yDim();
    zDim = recon->get_zDim();
    angles = proj->get_angleNum();
    p2oLength = proj->get_p2oLength();
    p2dLength = proj->get_p2dLength();
    yDetectDim = proj->get_yDetectDim();
    zDetectDim = proj->get_zDetectDim();
    xPixelSize = recon->get_xPixelSize();
    yPixelSize = recon->get_yPixelSize();
```

```
    zPixelSize = recon->get_zPixelSize();
    elementSizeY = proj->get_elementSizeY();
    elementSizeZ = proj->get_elementSizeZ();
}

BackProjector::BackProjector(ProjData *p, ReconData *r, SystemMatrix
*s) {
    proj = p;
    recon = r;
    system = s;
    xDim = recon->get_xDim();
    yDim = recon->get_yDim();
    zDim = recon->get_zDim();
    angles = proj->get_angleNum();
    p2oLength = proj->get_p2oLength();
    p2dLength = proj->get_p2dLength();
    yDetectDim = proj->get_yDetectDim();
    zDetectDim = proj->get_zDetectDim();
    xPixelSize = recon->get_xPixelSize();
    yPixelSize = recon->get_yPixelSize();
    zPixelSize = recon->get_zPixelSize();
    elementSizeY = proj->get_elementSizeY();
    elementSizeZ = proj->get_elementSizeZ();
}

BackProjector::~BackProjector() {
}
/**
 * Methods
 */
```

backprojector.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************
****
                          backprojector.h  -  description
                             -------------------
    begin                : Thu Jan 06 09:11:48 CST 2005
    copyright            : (C) 2005 by Christian Wietholt
    email                : cwietholt@nhri.org.tw

***************************************************************
****/

/
***************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
```

```
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

*************************************************************************
****/
#ifndef BACKPROJECTOR_H
#define BACKPROJECTOR_H

#include "point3d.h"
#include "systemmatrix.h"

class ProjData;
class ReconData;
//class ReconProgress;
class SystemMatrix;

/**
 * Class BackProjector
 *
 */
class BackProjector {
    /**
     * Public stuff
     */
public:
    /**
     * Fields
     */
    /**
     *
     */
    /**
     * Constructors
     */
    BackProjector();
    BackProjector(ProjData *p, ReconData *r);
    BackProjector(ProjData *p, ReconData *r, SystemMatrix *s);

    virtual ~BackProjector();
    /**
     * Accessor Methods
     */

    /**
     * Operations
     */
    virtual void project(int, double) {
    };

    virtual void project(int, double, double**, double**) {
```

```
    };
    /**
     * Protected stuff
     */
protected:
    /**
     * Fields
     */
    ProjData *proj;
    ReconData *recon;
    SystemMatrix *system;

    /** The rotated starting point. */
    Point3D startPointRot;
    /** The rotated increment to the next point in that row (X)  */
    Point3D incrementXRot;
    /** The rotated increment to the next point in that column (Y)  */
    Point3D incrementYRot;

    Point3D incrementZ;

    double p2oLength;
    double p2dLength;
    double elementSizeY;
    double elementSizeZ;
    double xPixelSize;
    double yPixelSize;
    double zPixelSize;

    unsigned short xDim;
    unsigned short yDim;
    unsigned short zDim;
    unsigned short yDetectDim;
    unsigned short zDetectDim;
    unsigned short angles;
    /**
     *
     */
    /**
     * Constructors
     */
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
    /**
     * Private stuff
     */
private:
    /**
     * Fields
     */
```

```
    /**
     *
     */
    /**
     * Constructors
     */
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
};
#endif //BACKPROJECTOR_H
```

bppinholray.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************************
****
                          bppinholeray.cpp  -  description
                             -------------------
    begin                : Mon Apr 06 16:24:48 CST 2006
    copyright            : (C) 2006 by Christian Wietholt
    email                : cwietholt@nhri.org.tw

***************************************************************************
****/

/
***************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

***************************************************************************
****/
#include "bppinholeray.h"
#include "projdata.h"
#include "recondata.h"
#include "raytbl3dinc.h"
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <iomanip>
```

```
#include <cstdlib>
#include <sstream>
using namespace std;

/**
 * Constructors/Destructors
 */
BPPinholeRay::BPPinholeRay() {
}

BPPinholeRay::BPPinholeRay(ProjData *p, ReconData *r, SystemMatrix
*s) : BackProjector(p, r, s) {


}

/**
 * Methods
 */
void BPPinholeRay::project(int a, double theta) {
    // Does back projection
    double **from;
    double** to;

    from = proj->get_dataEst();
    to = recon->get_objectEst();

    recon->reset_rotData();
    recon->reset_brotData();
    //Do raytracing
    system->buildMatrix(-theta, a, to, from, 0);

    //Sum output of raytrace
    for (int z = 0; z < zDim; z++)
        for (int y = 0; y < yDim; y++)
            for (int x = 0; x < xDim; x++)
                recon->add_data(z, y, x, recon->get_rotData(z, y, x),
to);
}

void BPPinholeRay::project(int a, double theta, double **from, double**
to) {

    recon->reset_rotData();
    recon->reset_brotData();
    //Do raytracing
    system->buildMatrix(-theta, a, to, from, 0);

    //Sum output of raytracing
    for (int z = 0; z < zDim; z++)
        for (int y = 0; y < yDim; y++)
            for (int x = 0; x < xDim; x++) {
                recon->add_data(z, y, x, recon->get_rotData(z, y, x),
to);
```

```
            }
}
```

bppinholeray.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
*************************************************************************
****
                          bppinholeray.h  -  description
                             -------------------
    begin                : Mon Apr 24 16:23:48 CST 2006
    copyright            : (C) 2006 by Christian Wietholt
    email                : cwietholt@nhri.org.tw

*************************************************************************
****/

/
*************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

*************************************************************************
****/
#ifndef BPPINHOLERAY_H
#define BPPINHOLERAY_H

#include "backprojector.h"

class ProjData;
class ReconData;

/**
 * Class BPPinholeRay
 *
 */
class BPPinholeRay : public BackProjector {
    /**
     * Public stuff
     */
public:
    /**
     * Fields
     */
```

```
    /**
     *
     */
    /**
     * Constructors
     */
    BPPinholeRay();
    BPPinholeRay(ProjData *p, ReconData *r, SystemMatrix *s);
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
    virtual void project(int a, double scale);

    virtual void project(int a, double scale, double** from, double**
to);

    /**
     * Protected stuff
     */
protected:
    /**
     * Fields
     */
    /**
     *
     */
    /**
     * Constructors
     */
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
    /**
     * Private stuff
     */
private:
    /**
     * Fields
     */
    /**
     *
     */
    /**
     * Constructors
     */
    /**
     * Accessor Methods
     */
```

```
    /**
     * Operations
     */
};
#endif //BPPINHOLERAY_H
```

forwardprojector.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************************
****
                          forwardprojector.cpp  -  description
                             -------------------
    begin                : Thu Jan 06 09:11:48 CST 2005
    copyright            : (C) 2005 by Christian Wietholt
    email                : cwietholt@nhri.org.tw

***************************************************************************
****/

/
***************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

***************************************************************************
****/

#include "forwardprojector.h"

#include "recondata.h"
#include "projdata.h"

/**
 * Constructors/Destructors
 */

ForwardProjector::ForwardProjector() {
}

ForwardProjector::ForwardProjector(ProjData *p, ReconData *r) {
    proj = p;
    recon = r;
    xDim = recon->get_xDim();
```

```
    yDim = recon->get_yDim();
    zDim = recon->get_zDim();
    angles = proj->get_angleNum();
    p2oLength = proj->get_p2oLength();
    p2dLength = proj->get_p2dLength();
    yDetectDim = proj->get_yDetectDim();
    zDetectDim = proj->get_zDetectDim();
    xPixelSize = recon->get_xPixelSize();
    yPixelSize = recon->get_yPixelSize();
    zPixelSize = recon->get_zPixelSize();
    elementSizeY = proj->get_elementSizeY();
    elementSizeZ = proj->get_elementSizeZ();
}

ForwardProjector::ForwardProjector(ProjData *p, ReconData *r,
SystemMatrix *s) {
    proj = p;
    recon = r;
    system = s;
    xDim = recon->get_xDim();
    yDim = recon->get_yDim();
    zDim = recon->get_zDim();
    angles = proj->get_angleNum();
    p2oLength = proj->get_p2oLength();
    p2dLength = proj->get_p2dLength();
    yDetectDim = proj->get_yDetectDim();
    zDetectDim = proj->get_zDetectDim();
    xPixelSize = recon->get_xPixelSize();
    yPixelSize = recon->get_yPixelSize();
    zPixelSize = recon->get_zPixelSize();
    elementSizeY = proj->get_elementSizeY();
    elementSizeZ = proj->get_elementSizeZ();
};

ForwardProjector::~ForwardProjector() {
}
/**
 * Methods
 */
```

forwardprojector.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************************
****
                          forwardprojector.h  -  description
                             -------------------
    begin                : Thu Jan 06 09:11:48 CST 2005
    copyright            : (C) 2005 by Christian Wietholt
    email                : cwietholt@nhri.org.tw

***************************************************************************
****/
```

```
/
*************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

*************************************************************************
****/

#ifndef FORWARDPROJECTOR_H
#define FORWARDPROJECTOR_H

#include "point3d.h"
#include "systemmatrix.h"

/**
 * Class ForwardProjector
 *
 */
class ForwardProjector {
    /**
     * Public stuff
     */
public:
    /**
     * Fields
     */
    /**
     *
     */
    /**
     * Constructors
     */
    ForwardProjector();
    ForwardProjector(ProjData *p, ReconData *r);
    ForwardProjector(ProjData *p, ReconData *r, SystemMatrix *s);

    virtual ~ForwardProjector();
    /**
     * Accessor Methods
     */

    /**
     * Operations
```

```
     */
    virtual void project(int, double) {
    };

    virtual void project(int, double, double**, double**) {
    };
    /**
     * Protected stuff
     */
protected:
    /**
     * Fields
     */
    ProjData *proj;
    ReconData *recon;
    SystemMatrix *system;

    /** The rotated starting point. */
    Point3D startPointRot;
    /** The rotated increment to the next point in that row (X)  */
    Point3D incrementXRot;
    /** The rotated increment to the next point in that column (Y)  */
    Point3D incrementYRot;

    Point3D incrementZ;

    double p2oLength;
    double p2dLength;
    double elementSizeY;
    double elementSizeZ;
    double xPixelSize;
    double yPixelSize;
    double zPixelSize;

    unsigned short xDim;
    unsigned short yDim;
    unsigned short zDim;
    unsigned short yDetectDim;
    unsigned short zDetectDim;
    unsigned short angles;
    /**
     *
     */
    /**
     * Constructors
     */
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
    /**
     * Private stuff
```

```
     */
private:
    /**
     * Fields
     */
    /**
     *
     */
    /**
     * Constructors
     */
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
};
#endif //FORWARDPROJECTOR_H
```

fppinholeray.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************************
                          fppinholeray.cpp  -  description
                             -------------------
    begin                : Mon Apr 21 09:17:48 CST 2006
    copyright            : (C) 2006 by Christian Wietholt
    email                : cwietholt@nhri.org.tw

***************************************************************************
****/

/
***************************************************************************
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

***************************************************************************
****/
#include "fppinholeray.h"
#include "point3d.h"
#include "projdata.h"
```

```
#include "recondata.h"
#include "raytbl3dinc.h"
#include <iostream>
#include <fstream>
#include <stdlib.h>
#include <iomanip>
#include <cstdlib>
#include <time.h>
#include <sstream>
#include <vector>
using namespace std;

/**
 * Constructors/Destructors
 */
FPPinholeRay::FPPinholeRay() {
}

FPPinholeRay::FPPinholeRay(ProjData *p, ReconData *r, SystemMatrix
*s) : ForwardProjector(p, r, s) {
}

FPPinholeRay::~FPPinholeRay() {
}
/**
 * Methods
 */

void FPPinholeRay::project(int a, double theta) {

    double **from;
    double **to;
    int pinHoleMatrix = 0;

    from = recon->get_objectEst();
    to = proj->get_dataEst();

    pinHoleMatrix = proj->get_pinHoleMatrix();

    // initialize the object rotation framework
    // get data from rotated object using recon->get_rotData(z, y, x)
    recon->reset_rotData();
    for (int z = 0; z < recon->get_zDim(); z++) {
        for (int y = 0; y < recon->get_yDim(); y++) {
            for (int x = 0; x < recon-> get_xDim(); x++) {
                recon->set_rotData(z, y, x, recon->get_data(z, y, x,
from));
            }
        }
    }

    // Do raytrace
    system->buildMatrix(theta, a, to,from, 1);
}
```

```
void FPPinholeRay::project(int a, double theta, double **from, double
**to) {

    int pinHoleMatrix = 0;

    pinHoleMatrix = proj->get_pinHoleMatrix();

    // initialize the object rotation framework
    // get data from rotated object using recon->get_rotData(z, y, x)
    recon->reset_rotData();
    for (int z = 0; z < recon->get_zDim(); z++) {
        for (int y = 0; y < recon->get_yDim(); y++) {
            for (int x = 0; x < recon-> get_xDim(); x++) {
                recon->set_rotData(z, y, x, recon->get_data(z, y, x,
from));
            }
        }
    }

    system->buildMatrix(theta, a, to,from, 1);
}
```

fppinholeray.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************************
                          fppinholeray.h  -  description
                             -------------------
    begin                : Mon Apr 21 09:16:48 CST 2006
    copyright            : (C) 2006 by Christian Wietholt
    email                : cwietholt@nhri.org.tw

***************************************************************************/

/
***************************************************************************
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

***************************************************************************/
```

```
#ifndef FPPINHOLERAY_H
#define FPPINHOLERAY_H

#include "forwardprojector.h"

class ProjData;
class ReconData;

/**
 * Class FPPinholeRay
 * Implementation of a forward projector with a pinhole geometry using
a infinitisimal small aperture.
 */
class FPPinholeRay : public ForwardProjector {
    /**
     * Public stuff
     */
public:
    /**
     * Fields
     */
    /**
     *
     */
    /**
     * Constructors
     */
    FPPinholeRay();

    FPPinholeRay(ProjData *p, ReconData *r, SystemMatrix *s);

    ~FPPinholeRay();
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
    virtual void project(int a, double scale);

    virtual void project(int a, double scale, double **from, double
**to);

    /**
     * Protected stuff
     */
protected:
    /**
     * Fields
     */
    /**
     *
     */
    /**
```

```
     * Constructors
     */
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
    /**
     * Private stuff
     */
private:
    /**
     * Fields
     */
    /**
     *
     */
    /**
     * Constructors
     */
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */

};
#endif //FPPINHOLERAY_H
```

line3d.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
// Line3d.cpp: implementation of the CLine3d class.
//
//////////////////////////////////////////////////////////////////////

#include "line3d.h"

#include "point3d.h"
#include "math.h"


//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CLine3d::CLine3d() {
    x0 = 0;
    y0 = 0;
    z0 = 0;
    x1 = 0;
    y1 = 0;
    z1 = 0;
}
```

```
CLine3d::CLine3d(double x0, double y0, double z0, double x1, double y1,
double z1) {
    CLine3d::x0 = x0;
    CLine3d::y0 = y0;
    CLine3d::z0 = z0;
    CLine3d::x1 = x1;
    CLine3d::y1 = y1;
    CLine3d::z1 = z1;
}

void CLine3d::operator=(const CLine3d &other) {
    x0 = other.x0;
    y0 = other.y0;
    z0 = other.z0;
    x1 = other.x1;
    y1 = other.y1;
    z1 = other.z1;

}

CLine3d::~CLine3d() {

}

CLine3d::CLine3d(Point3D p0, Point3D p1) {
    x0 = p0.x;
    y0 = p0.y;
    z0 = p0.z;
    x1 = p1.x;
    y1 = p1.y;
    z1 = p1.z;
}

void CLine3d::Create(double p0x, double p0y, double p0z, double p1x,
double p1y, double p1z) {
    x0 = p0x;
    y0 = p0y;
    z0 = p0z;
    x1 = p1x;
    y1 = p1y;
    z1 = p1z;
}

void CLine3d::Translate(double tx, double ty, double tz) {
    x0 += tx;
    y0 += ty;
    z0 += tz;
    x1 += tx;
    y1 += ty;
    z1 += tz;


}
```

```
void CLine3d::RotateX(double theta) {

    double dYtemp = 0;
    double dZtemp = 0;
    double dCosTheta = 0;
    double dSinTheta = 0;

    dCosTheta = cos(theta);
    dSinTheta = sin(theta);

    dYtemp = dCosTheta * y0 + dSinTheta * z0;
    dZtemp = -dSinTheta * y0 + dCosTheta * z0;

    y0 = dYtemp;
    z0 = dZtemp;

    dYtemp = dCosTheta * y1 + dSinTheta*z1;
    dZtemp = -dSinTheta * y1 + dCosTheta*z1;

    y1 = dYtemp;
    z1 = dZtemp;
}

void CLine3d::RotateY(double theta) {

    double dXtemp = 0;
    double dZtemp = 0;
    double dCosTheta = 0;
    double dSinTheta = 0;

    dCosTheta = cos(theta);
    dSinTheta = sin(theta);

    dXtemp = dCosTheta * x0 + dSinTheta * z0;
    dZtemp = -dSinTheta * x0 + dCosTheta * z0;

    x0 = dXtemp;
    z0 = dZtemp;

    dXtemp = dCosTheta * x1 + dSinTheta * z1;
    dZtemp = -dSinTheta * x1 + dCosTheta * z1;

    x1 = dXtemp;
    z1 = dZtemp;
}

void CLine3d::RotateZ(double theta) {

    double dXtemp = 0;
    double dYtemp = 0;
    double dCosTheta = 0;
    double dSinTheta = 0;
```

```
    dCosTheta = cos(theta);
    dSinTheta = sin(theta);

    dXtemp = dCosTheta * x0 + dSinTheta * y0;
    dYtemp = -dSinTheta * x0 + dCosTheta * y0;

    x0 = dXtemp;
    y0 = dYtemp;

    dXtemp = dCosTheta * x1 + dSinTheta * y1;
    dYtemp = -dSinTheta * x1 + dCosTheta * y1;

    x1 = dXtemp;
    y1 = dYtemp;
}

double CLine3d::Length() {
    return (sqrt((x1 - x0) * (x1 - x0) +
            (y1 - y0) * (y1 - y0) +
            (z1 - z0) * (z1 - z0)));
}

double CLine3d::DotProduct(CLine3d OtherLine) {
    CLine3d tline1 = CLine3d(x0, y0, z0, x1, y1, z1);
    CLine3d tline2 = OtherLine;

    tline1.Normalize();
    tline2.Normalize();

    return (tline1.x1 * tline2.x1 +

            tline1.y1 * tline2.y1 +
            tline1.z1 * tline2.z1);

}

Point3D CLine3d::CrossProduct(CLine3d OtherLine) {
    CLine3d l1 = CLine3d(x0, y0, z0, x1, y1, z1);
    CLine3d l2 = OtherLine;
    l1.Normalize();
    l2.Normalize();
    Point3D ReturnPoint;
    ReturnPoint.x = l1.y1 * l2.z1 - l1.z1 * l2.y1;
    ReturnPoint.y = l1.z1 * l2.x1 - l1.x1 * l2.z1;
    ReturnPoint.z = l1.x1 * l2.y1 - l1.y1 * l2.x1;
    return (ReturnPoint);


}

void CLine3d::Normalize() {
    double l = Length();
    x1 = (x1 - x0) / l;
    y1 = (y1 - y0) / l;
```

```
    z1 = (z1 - z0) / l;
    x0 = 0;
    y0 = 0;
    z0 = 0;
}

/*
CString CLine3d::ReturnString(){
        char      szMystr[50];
        sprintf (szMystr,"LINE Dist: %f",Length());
  return ((CString)szMystr);
}
 */

Point3D CLine3d::GetP0() {
    return (Point3D(x0, y0, z0));
}

Point3D CLine3d::GetP1() {
    return (Point3D(x1, y1, z1));
}
```

line3d.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************************
****
                          line3d.h  -  description
                             -------------------
    begin                : Mon Mar 18 2002
    copyright            : (C) 2002 by Christopher Hanger
    email                : christian.wietholt@marquette.edu

***************************************************************************
****/

/
***************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

***************************************************************************
****/
```

```
#ifndef CLINE3D_H
#define CLINE3D_H

class Point3D;

class CLine3d {
public:

    CLine3d();
    CLine3d(double x0, double y0, double z0, double x1, double y1,
double z1);
    CLine3d(Point3D p0, Point3D p1);
    virtual ~CLine3d();

    void Create(double p0x, double p0y, double p0z, double p1x, double
p1y, double p1z);
    Point3D CrossProduct(CLine3d OtherLine);

    Point3D GetP0();
    Point3D GetP1();
    void Normalize();
    double DotProduct(CLine3d OtherLine);
    double Length();

    void Translate(double tx, double ty, double tz);
    //      CString ReturnString();

    void RotateX(double theta);
    void RotateY(double theta);
    void RotateZ(double theta);

    void operator=(const CLine3d &other);

    double x0;
    double y0;
    double z0;
    double x1;
    double y1;
    double z1;
};

#endif  //CLINE3D_H
```

main.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */

#include <stdlib.h>
#include <iostream>
#include <cstdlib>

#include "CPSPECT.h"
using namespace std;

/*
 * Implements CP algorithm proposed by E. Sidky
```

```
 */
int main(int argc, char** argv) {

    int pinnum = 1;
    int detnum = 1;
    int anglenum = 64;
    int alg = 2;
    double lambda = 1E-4;
    double psfFWHM = 1.0;
    double blur = 1.0;
    double L = 287;
    cout << argc << endl;

    if (argc < 4) {
        cout << "Usage: ./cp [algorithm number] [number of angles] "
                "[projection file name] "
                "[labmda (if alg >= 4)] [radius of blur (if alg >=6)]"
<< endl;
        return 1;
    }

    alg = atoi(argv[1]);
    anglenum = atoi(argv[2]);
    string filename(argv[3]);
    if (argc >= 5)
        lambda = atof(argv[4]);
    if (argc >= 6)
        blur = atof(argv[5]);

    CPSPECT *recon = new CPSPECT();

    recon->init(pinnum, detnum, anglenum, psfFWHM);
    recon->initReconstruction();

    if ((alg > 5) && (blur != 0)) {
        recon->pData.initGaussianBlur(blur);
        recon->rData.initGaussianBlur(blur);
        recon->setBlurFlag(1);
    }

    L = recon->calculateL(alg);

    cout << "L: " << L << endl;

    if (alg == 2)
        recon->leastSquares(L, filename);
    else if (alg == 4)
        recon->leastSquaresPlusTV(L, filename, lambda);
    else if (alg == 5)
        recon->KLPlusTV(L, filename, lambda);
    else if (alg == 6) {
        if (blur != 0) {
            recon->pData.initGaussianBlur(blur);
            recon->rData.initGaussianBlur(blur);
```

```
            recon->setBlurFlag(1);
        }
        recon->leastSquaresPlusTV(L, filename, lambda);
    } else if (alg == 7) {
        if (blur != 0) {
            recon->pData.initGaussianBlur(blur);
            recon->rData.initGaussianBlur(blur);
            recon->setBlurFlag(1);
        }
        recon->KLPlusTV(L, filename, lambda);
    }

    return 0;
}
```

point3d.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
**************************************************************************
****
                          projdata.cpp  -  description
                             -------------------
    begin                : Mon Mar 18 2002
    copyright            : (C) 2002 by Christopher Hanger
    email                : christian.wietholt@marquette.edu

**************************************************************************
****/

/
**************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

**************************************************************************
****/

#include "point3d.h"
#include "line3d.h"
#include <iostream>
#include <math.h>
using namespace std;
//////////////////////////////////////////////////////////////////////
// Construction/Destruction
```

```
//////////////////////////////////////////////////////////////////////

Point3D::Point3D() {
    x = 0;
    y = 0;
    z = 0;
}

Point3D::Point3D(double xx, double yy, double zz) {
    x = xx;
    y = yy;
    z = zz;
    s = 0;
}

Point3D::Point3D(double xx, double yy, double zz, double ss) {
    x = xx;
    y = yy;
    z = zz;
    s = ss;
}

void Point3D::Create(double xx, double yy, double zz) {
    x = xx;
    y = yy;
    z = zz;
    s = 0;
}

void Point3D::Create(double xx, double yy, double zz, double ss) {
    x = xx;
    y = yy;
    z = zz;
    s = ss;
}

Point3D::~Point3D() {

}

void Point3D::RotateX(double theta) {
    double dYtemp = 0;
    double dZtemp = 0;
    double dCosTheta = 0;
    double dSinTheta = 0;

    dCosTheta = cos(theta);
    dSinTheta = sin(theta);

    dYtemp = dCosTheta * y + dSinTheta * z;
    dZtemp = -dSinTheta * y + dCosTheta * z;

    y = dYtemp;
    z = dZtemp;
```

```
}

void Point3D::RotateY(double theta) {
    double dXtemp = 0;
    double dZtemp = 0;
    double dCosTheta = 0;
    double dSinTheta = 0;

    dCosTheta = cos(theta);
    dSinTheta = sin(theta);

    dXtemp = dCosTheta * x + dSinTheta * z;
    dZtemp = -dSinTheta * x + dCosTheta * z;

    x = dXtemp;
    z = dZtemp;
}

void Point3D::RotateZ(double theta) {
    double dXtemp = 0;
    double dYtemp = 0;
    double dCosTheta = 0;
    double dSinTheta = 0;

    dCosTheta = cos(theta);


    dSinTheta = sin(theta);


    dXtemp = dCosTheta * x + dSinTheta * y;
    dYtemp = -dSinTheta * x + dCosTheta * y;

    x = dXtemp;
    y = dYtemp;

}

void Point3D::RotateVector(Point3D Mu, double dAlpha) {

/*****************************************************************
*****************************

*****************************************************************
*****************************
    To use this function, take the point you wanna rotate.
    Put it in the XZ plane.
    Set (negative)Y to the offset down the line or whatever

*****************************************************************
*****************************
```

```
**************************************************************************
*****************************/

    double Hphi = 0;
    double sinphi = 0;
    double cosphi = 0;
    double dSinAlpha = 0;
    double dCosAlpha = 0;
    double dXtemp = 0;
    double dYtemp = 0;
    double dZtemp = 0;

    //Find the normal of Mu
    double Htheta = sqrt(Mu.x * Mu.x + Mu.y * Mu.y);
    double sintheta = 0;
    double costheta = 0;
    if (Htheta > 0) {
        sintheta = Mu.x / Htheta;
        costheta = Mu.y / Htheta;
        Mu.x = 0;
        Mu.y = Htheta;
    } else {
        sintheta = 0;
        costheta = 1;
    }

    Hphi = sqrt(Mu.z * Mu.z + Mu.y * Mu.y);
    sinphi = Mu.z / Hphi;
    cosphi = -Mu.y / Hphi;


    //**********************N O W ,    R O T A T E   T H E   M U T H A
****************************
    //Rotate Alpha (in the xz plane) first...
    ///Rotate Y alpha degrees...
    dSinAlpha = sin(dAlpha * 3.141592653589 / 180);
    dCosAlpha = cos(dAlpha * 3.141592653589 / 180);

    dXtemp = dCosAlpha * x + dSinAlpha * z;
    dZtemp = -dSinAlpha * x + dCosAlpha * z;

    x = dXtemp;
    z = dZtemp;

    //Rotate x...
    dYtemp = cosphi * y + sinphi*z;
    dZtemp = -sinphi * y + cosphi * z;
    y = dYtemp;
    z = dZtemp;

    //Rotate z...
    dXtemp = costheta * x + sintheta * y;
    dYtemp = -sintheta * x + costheta * y;
```

```
    x = dXtemp;
    y = dYtemp;

}

double Point3D::DistanceFrom(Point3D p) {
    return (sqrt((p.x - x) * (p.x - x) + (p.y - y) * (p.y - y) + (p.z -
z) * (p.z - z)));
}


/*
Point3D Point3D::ProjectOntoPlane(Point3D P0, Point3D P1, Point3D
P2,Point3D P3){
        CLine3d V1 (P3, Point3D (x, y, z));
        CLine3d V2 (P3, P0);
        CLine3d V3 (P3, P2);
        double theta = V1.DotProduct(V2) * V1.Length() / V2.Length();
        double phi = V1.DotProduct(V3) * V1.Length() / V3.Length();

        Point3D P4 = P3 + (P0-P3) * theta;
        Point3D P5 = P2 + (P1-P2) * theta;
        Point3D PF = P4 + (P5-P4) * phi;
        return (PF);
}
 */
```

point3d.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
**************************************************************************
****
                          point3d.h  -  description
                             -------------------
    begin                : Mon Mar 18 2002
    copyright            : (C) 2002 by Christopher Hanger
    email                : christian.wietholt@marquette.edu

**************************************************************************
****/

/
**************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
```

```
 *
*

**********************************************************************
****/

#ifndef POINT3D_H
#define POINT3D_H

class Point3D {
public:

    Point3D();
    Point3D(double, double, double);
    Point3D(double, double, double, double);

    ~Point3D();

    void Create(double, double, double);
    void Create(double, double, double, double);

    //      Point3D ProjectOntoPlane(Point3D P0, Point3D P1, Point3D
P2,Point3D P3);

    double DistanceFrom(Point3D p);
    void RotateVector(Point3D Mu, double dAlpha);

    void Translate(double tx, double ty, double tz) {
        x += tx;
        y += ty;
        z += tz;
    };

    void Translate(Point3D t) {
        x += t.x;
        y += t.y;
        z += t.z;
    };

    void RotateX(double theta);
    void RotateY(double theta);
    void RotateZ(double theta);

    bool operator ==(const Point3D &other) {
        if (x != other.x) return (false);
        if (y != other.y) return (false);
        if (z != other.z) return (false);
        return (true);
    };

    void
    operator =(const Point3D &other) {
        x = other.x;
        y = other.y;
```

```
        z = other.z;
        s = other.s;
    };

    void
    operator +=(const Point3D &other) {
        x += other.x;
        y += other.y;
        z += other.z;
    };

    Point3D
    operator +(const Point3D &other) {
        return (Point3D(x + other.x, y + other.y, z + other.z));
    };

    Point3D
    operator -(const Point3D &other) {
        return (Point3D(x - other.x, y - other.y, z - other.z));
    };

    Point3D
    operator *(double dNum) {
        return (Point3D(x * dNum, y * dNum, z * dNum));
    };

    Point3D
    operator *(int dNum) {
        return (Point3D(x * dNum, y * dNum, z * dNum));
    };

    Point3D
    operator /(double dNum) {
        return (Point3D(x / dNum, y / dNum, z / dNum));
    };

    double alphaVal() {
        return s;
    }

    //      CString ReturnString();

    double x;
    double y;
    double z;
    double s;

};

#endif //  POINT3D_H
```

projdata.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
```

```
/
***************************************************************************
****
                          projdata.cpp  -  description
                             -------------------
    begin                : Fri Mar 8 2002
    copyright            : (C) 2002 by Christian Wietholt
    email                : christian.wietholt@marquette.edu

***************************************************************************
****/

/
***************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

***************************************************************************
****/

#include <math.h>
#include <algorithm>

#ifndef M_PIl
#endif

#include "projdata.h"
#include "systemmatrix.h"
#include <cstring>
#include <string>
#include <iostream>
#include <fstream>
#include <endian.h>
using namespace std;

ProjData::ProjData() {
    projMin = 999999;
    projMax = 0;
    pinHoleFunct = NULL;
}

ProjData::~ProjData() {
    delete []DetectPosY;
```

```
    delete []DetectPosZ;
    delete []projData;
    delete []dataEst;
    delete []pinHoleFunct;
    delete []pinHoleRayPos;
    delete []pinHoleRayHexaPos;
    delete []ang;
    delete []ax;

    fftwl_free(outData);
    fftwl_free(realData);
    fftwl_free(inData);
    fftwl_free(filterKernel);
    fftwl_destroy_plan(pAA);
    fftwl_destroy_plan(pBB);
}

/*
 * Methods
 */

/* Initializers */

bool ProjData::init(int pn, int detnum, unsigned short y, unsigned
short z, unsigned short yD, unsigned short zD, unsigned short a, double
yL, double zL, double p2d, double p2o, unsigned short pMat, double
pDia, unsigned short vT, double hl, double psf, string fnPhM, double
pMatSY, double pMatSZ, int locID, int typeID, double *yPHS,
        double *zPHS, double yDS, double zDS, double xDT, double yDT,
double zDT,
        unsigned short sym) {
    /** This function will initialize the data arrays for the
projection to be created.  */

    char * siter = "0";
    siter = new char[75];

    bool success = false;
    long i = 0;
    int detectPosOversize = 2;

    symmetry = sym;
    pinnum = pn;
    xDetCenterPos = 0.0;
    yDetStart2Center = 0.0;
    zDetStart2Center = 0.0;

    yDetectShift = yDS;
    zDetectShift = zDS;

    xDetectTilt = xDT;
    yDetectTilt = yDT;
    zDetectTilt = zDT;
```

```
    yDetectDim = yD;
    zDetectDim = zD;

    yDetectDimHalf = yD / 2;
    zDetectDimHalf = zD / 2;
    yDetectDimHalfP = yD / 2 + 1;
    zDetectDimHalfP = zD / 2 + 1;
    detectnum = detnum;

    angleNum = a;
    yDetectLength = yL;
    zDetectLength = zL;
    detectEle = yDetectDim * zDetectDim;
    detectEleQuarter = yDetectDimHalf * zDetectDimHalf;
    detectEleQuarterM = (yDetectDimHalf * zDetectDimHalf) - 1;
    p2dLength = p2d;
    p2oLength = p2o;
    yPhShift = yPHS;
    zPhShift = zPHS;
    pinHoleMatrix = pMat;
    pinHoleMatrixSizeY = pMatSY;
    pinHoleMatrixSizeZ = pMatSZ;
    pinHoleDiameter = pDia;
    pinHoleRadius = pinHoleDiameter / 2.0f;
    pinHoleType = typeID;
    detectorBlurrFWHM = psf;
    detectorBlurrRadius = psf / 2.0;
    fileNamePinholeModel = fnPhM;

    if (typeID == INFILE) {
        switch (locID) {
            case 0:
                if (symmetry)
                    get_PinHoleFunctPtr =
&ProjData::get_PinHoleFunctCenterSym;
                else
                    get_PinHoleFunctPtr =
&ProjData::get_PinHoleFunctCenter;
                break;
            case 1:
                get_PinHoleFunctPtr =
&ProjData::get_PinHoleFunctCorner;
                break;
        }
    } else
        get_PinHoleFunctPtr = &ProjData::get_PinHoleFunctNoFile;


    if (pinnum == 1 && yDetectDim == 128 && zDetectDim == 1) {
        sensitiveMap = new double*[1];
        for (int i = 0; i < 1; i++) {
            sensitiveMap[i] = new double[zDetectDim * yDetectDim];
        }
        for (int i = 0; i < 1; i++) {
```

```
            memset(sensitiveMap[i], 0, zDetectDim * yDetectDim * sizeof
(double));
        }

        for (int i = 0; i < pinnum; i++) {
            sprintf(siter, "/home/wolfp/newProjMLEM/1Dsrc/%dph/ph
%d.proj", pinnum, i);
            load_Sensitivity_Map(siter, i);
        }
    }

    viewTime = vT;
    halflife = hl;


    // get enough memory for the positions
    DetectPosY = new double[yDetectDim * detectPosOversize];
    DetectPosZ = new double[zDetectDim * detectPosOversize];

    //calculate pixel size
    elementSizeY = yDetectLength / yDetectDim;
    elementSizeZ = zDetectLength / zDetectDim;

    //calculate the rotation step size in radians
    angleStepSize = 2 * M_PIl / angleNum;

    // for both methods
    yDetStart2Center = (-(yDetectLength / 2)) + (elementSizeY / 2);
    zDetStart2Center = (-(zDetectLength / 2)) + (elementSizeZ / 2);

    for (i = 0; i < yDetectDim * detectPosOversize; i++)
        DetectPosY[i] = yDetStart2Center + i * elementSizeY;
    for (i = 0; i < zDetectDim * detectPosOversize; i++)
        DetectPosZ[i] = zDetStart2Center + i * elementSizeZ;

    success = initData();

    // the area of a detector pixel
    pixelArea = elementSizeY * elementSizeZ;
    scalingFactor = pixelArea / recon->get_VoxelVolume();


    return success;
}

void ProjData::initRayDrivenVariables() {
    // for Ray Driven projections
    xDetCenterPos = -(p2oLength + p2dLength);

    startDetectPoint = new Point3D(0.0,
            yDetStart2Center,
            zDetStart2Center);

    startDetectPoint->RotateX(xDetectTilt * M_PIl / 180.0);
```

```
    startDetectPoint->RotateY(yDetectTilt * M_PIl / 180.0);
    startDetectPoint->RotateZ(zDetectTilt * M_PIl / 180.0);

    startDetectPoint->Translate(xDetCenterPos, yDetectShift,
zDetectShift);

    incrementDetectY = new Point3D(0.0, elementSizeY, 0.0);
    incrementDetectY->RotateX(xDetectTilt * M_PIl / 180.0);
    incrementDetectY->RotateY(yDetectTilt * M_PIl / 180.0);
    incrementDetectY->RotateZ(zDetectTilt * M_PIl / 180.0);

    incrementDetectZ = new Point3D(0.0, 0.0, elementSizeZ);
    incrementDetectZ->RotateX(xDetectTilt * M_PIl / 180.0);
    incrementDetectZ->RotateY(yDetectTilt * M_PIl / 180.0);
    incrementDetectZ->RotateZ(zDetectTilt * M_PIl / 180.0);

    // done with Ray Driven stuff
}

bool ProjData::initData() {
    int i = 0;

    bool success = false;

    projData = new double*[angleNum];
    dataEst = new double*[angleNum];
    prevDataEst = new double*[angleNum];
    for (i = 0; i < angleNum; i++) {
        projData[i] = new double[yDetectDim * zDetectDim];
        dataEst[i] = new double[yDetectDim * zDetectDim];
        prevDataEst[i] = new double[yDetectDim * zDetectDim];
        success = true;
    }
    if (success) {
        for (i = 0; i < angleNum; i++) {
            memset(projData[i], 0, yDetectDim * zDetectDim * sizeof
(double));
            memset(dataEst[i], 0, yDetectDim * zDetectDim * sizeof
(double));
            memset(prevDataEst[i], 0, yDetectDim * zDetectDim * sizeof
(double));
        }
    }
    elementNum = yDetectDim * zDetectDim * angleNum;
    return success;
}

void ProjData::initPinhole() {
    int i = 0;
    if (pinHoleFunct == NULL) {
        if (pinHoleMatrixSizeY == 0.0) {
            pinHoleElementSize = pinHoleDiameter / pinHoleMatrix;
            yPHStart2Center = -(pinHoleDiameter / 2.0) +
(pinHoleElementSize / 2.0);
```

```
            zPHStart2Center = -(pinHoleDiameter / 2.0) +
(pinHoleElementSize / 2.0);
        } else {
            pinHoleElementSize = pinHoleMatrixSizeY / pinHoleMatrix;
            yPHStart2Center = -(pinHoleMatrixSizeY / 2.0) +
(pinHoleElementSize / 2.0);
            zPHStart2Center = -(pinHoleMatrixSizeZ / 2.0) +
(pinHoleElementSize / 2.0);
        }

        startPinHolePoint = new Point3D *[pinnum];
        for (i = 0; i < pinnum; i++) {
            startPinHolePoint[i] = new Point3D(-p2oLength,
yPHStart2Center, zPHStart2Center);
        }


        incrementPinHoleY = new Point3D(0.0, pinHoleElementSize, 0.0);
        incrementPinHoleZ = new Point3D(0.0, 0.0, pinHoleElementSize);
    }
}

void ProjData::allocPinholeFunct(int num) {
    int i = 0;
    int j = 0;
    pinHoleFunct = new double**[num];
    for (j = 0; j < num; j++) {
        pinHoleFunct[j] = new double*[pinHoleMatrix];
        for (i = 0; i < pinHoleMatrix; i++) {
            pinHoleFunct[j][i] = new double[pinHoleMatrix];
        }
    }
}

void ProjData::initPinholeFunctRound() {
    int g = 0;
    int h = 0;
    int count = 0;
    int i = 0;

    double distance = 0.0;
    double ylenSq = 0.0;
    double zlenSq = 0.0;

    Point3D doPointZ;
    Point3D doPointZY;

    allocPinholeFunct(pinnum);

    count = 0;
    for (i = 0; i < pinnum; i++) {
        doPointZ = *startPinHolePoint[i];
        // loop through all y pinhole matrix points
        for (g = 0; g < pinHoleMatrix; g++) {
```

```
            doPointZY = doPointZ;
            // loop through all x pinhole matrix points
            for (h = 0; h < pinHoleMatrix; h++) {
                ylenSq = doPointZY.y * doPointZY.y;
                zlenSq = doPointZY.z * doPointZY.z;
                distance = sqrt(ylenSq + zlenSq);
                if (distance < pinHoleDiameter / 2.0) {
                    pinHoleFunct[i][g][h] = 1.0;
                    count++;
                } else {
                    pinHoleFunct[i][g][h] = 0.0;
                }
                doPointZY += *incrementPinHoleY;
            }
            doPointZ += *incrementPinHoleZ;
        }

        shiftPinhole(i);
    }

}

void ProjData::init_forwardScalingFactors() {
    int i = 0;

    fsfData = new double*[angleNum];

    for (i = 0; i < angleNum; i++) {
        fsfData[i] = new double[yDetectDim * zDetectDim];
    }

    for (i = 0; i < angleNum; i++)
        memset(fsfData[i], 0, yDetectDim * zDetectDim * sizeof
(double));
}

/* Accessors */
// Pretty self-explanatory

unsigned short ProjData::get_symmetry() {
    return symmetry;
}

double ProjData::get_DetectPosYN(int i) {
    return DetectPosY[0] + i * elementSizeY;
}

double ProjData::get_DetectPosZN(int i) {
    return DetectPosZ[0] + i * elementSizeZ;
}

void ProjData::get_ProjLine(double *line, unsigned short yPos, unsigned
short zPos) {
    /** writes the data of the desired line to the Pointer */
```

```
    int i = 0;

    for (i = 0; i < yDetectDim; i++)
        line[yDetectDim * zPos + i] = projData[zPos][yPos * yDetectDim
+ i];
}

double ProjData::get_ScalingFactor() {
    return scalingFactor;
};

void ProjData::reset_dataEst() {
    for (int a = 0; a < angleNum; a++)
        memset(dataEst[a], 0, yDetectDim * zDetectDim * sizeof
(double));
}

void ProjData::reset_data(double** data) {
    for (int a = 0; a < angleNum; a++)
        memset(data[a], 0, yDetectDim * zDetectDim * sizeof (double));
}

void ProjData::reset_prevDataEst() {
    for (int a = 0; a < angleNum; a++)
        memset(prevDataEst[a], 0, yDetectDim * zDetectDim * sizeof
(double));
}

void ProjData::set_dataEst(int a, double val) {
    int i = 0;
    for (i = 0; i < yDetectDim * zDetectDim; i++)
        dataEst[a][i] = val;
}

void ProjData::set_dataEst(int a, int y, int z, double val) {
    dataEst[a][y + (yDetectDim * z)] = val;
}

void ProjData::set_dataEst(int a, int i, double val) {
    dataEst[a][i] = val;
}

void ProjData::set_projData(int a, int y, int z, double val) {
    projData[a][y + (yDetectDim * z)] = val;
}

void ProjData::set_prevDataEst() {
    for (int a = 0; a < angleNum; a++)
        for (int i = 0; i < yDetectDim * zDetectDim; i++)
            prevDataEst[a][i] = dataEst[a][i];
}

void ProjData::set_angles(double max_angle) {
    int i;
```

```
    double rotate;

    /* Rotate by 1/20 of one discrete angle */
    rotate = max_angle / ((double) angleNum * 2.0l * 10.0l);

    /* ------------ Precompute sin, cos, tan ---------------- */
    for (i = 0; i < angleNum; i++) {
        ang[i].rad = max_angle * (double) i / angleNum + rotate;
        ang[i].sin = sin(ang[i].rad);
        ang[i].cos = cos(ang[i].rad);
        ang[i].tan = tan(ang[i].rad);
    }

    ax[0] = 0;
    ax[1] = M_PI / 2.;
    ax[2] = M_PI;
    ax[3] = 3. * M_PI / 2.;
    ax[4] = 2. * M_PI;
}

/* File ops*/
// also pretty clear

void ProjData::save_dataEst(const string & filename) {
    int i = 0;
    int j = 0;

    fstream file;

    file.open(filename.c_str(), fstream::out | fstream::binary);

    for (i = 0; i < angleNum; i++) {
        for (j = 0; j < yDetectDim * zDetectDim; j++) {
            file.write((const char*) & dataEst[i][j], sizeof (double));

        }
    }
    file.close();
}

void ProjData::save_projData(const string & filename) {
    int i = 0;
    int j = 0;

    fstream file;
    file.open(filename.c_str(), fstream::out | fstream::binary);

    for (i = 0; i < angleNum; i++) {
        for (j = 0; j < yDetectDim * zDetectDim; j++) {
            file.write((const char*) & projData[i][j], sizeof
(double));
        }
    }
    file.close();
```

```
}

void ProjData::save_someProjData(const string & filename, double**
data) {
    int i = 0;
    int j = 0;

    fstream file;
    file.open(filename.c_str(), fstream::out | fstream::binary);

    for (i = 0; i < angleNum; i++) {
        for (j = 0; j < yDetectDim * zDetectDim; j++) {
            file.write((const char*) & data[i][j], sizeof (double));
        }
    }
    file.close();
}

void ProjData::save_forwardScalingFactors(const string & filename) {
    int i = 0;
    int j = 0;

    fstream file;
    file.open(filename.c_str(), fstream::out | fstream::binary);

    for (i = 0; i < angleNum; i++) {
        for (j = 0; j < yDetectDim * zDetectDim; j++) {
            file.write((const char*) & fsfData[i][j], sizeof (double));
        }
    }
    file.close();
}

void ProjData::save_pinholeFunct(const string &filename, int num) {
    int i = 0;
    int j = 0;
    int k = 0;

    fstream file;
    file.open(filename.c_str(), fstream::out | fstream::binary);

    for (k = 0; k < num; k++) {
        for (j = 0; j < pinHoleMatrix; j++) {
            for (i = 0; i < pinHoleMatrix; i++) {
                file.write((const char*) & pinHoleFunct[k][i], sizeof
(double));
            }
        }
    }
    file.close();
}

void ProjData::save_forward(const string &filename, int s) {
```

```
    int i = 0;
    int j = 0;

    fstream file;
    string pfilename = filename;
    char *sTest;

    sprintf(sTest, "%d", s);
    pfilename.append(sTest);
    file.open(pfilename.c_str(), fstream::out | fstream::binary);
    //      QDataStream stream(&file);
    //need following line for qtv4.6
    //
stream.setdoubleingPointPrecision(QDataStream::SinglePrecision);

    for (i = 0; i < angleNum; i++) {
        for (j = 0; j < yDetectDim * zDetectDim; j++) {
            //                stream<< dataEst[i][j];
            //            file << dataEst[i][j];
            file.write((const char*) & dataEst[i][j], sizeof (double));
        }
    }
    file.close();
}

void ProjData::save_onenewproj(const string &filename, int a) {
    int j = 0;

    //      QFile file(filename);
    fstream file;
    string pfilename = filename;
    //        QString sTest;
    //    char *sTest;
    //      sTest.setNum(s);
    //    sprintf(sTest, "%d", a);
    //    file.rename(filename + sTest);
    //    pfilename.append(sTest);
    //    file.open(QIODevice::WriteOnly);
    file.open(pfilename.c_str(), fstream::out | fstream::binary);

    for (j = 0; j < yDetectDim * zDetectDim; j++) {
        file.write((const char*) & dataEst[a][j], sizeof (double));
    }

    file.close();
}

bool ProjData::load(const string &filename) {
    bool success = true;
    int i = 0;
    long j = 0;
    totalcounts = 0.0;
    float b;
```

```
    fstream file;
    file.open(filename.c_str(), fstream::in | fstream::binary);
    if (file.fail()) {
        cerr << "Could not open file " << filename << endl;
        return 0;
    }

    for (i = 0; i < angleNum; i++) {
        for (j = 0; j < yDetectDim * zDetectDim; j++) {
            file.read((char*) & /*projData[i][j]*/b, 4);

            /* Accounts for endianness...*.proj files are written as
big-endian.
             * Intel machines are native little endian. This reads in
big-endian
             * data as little-endian if the native format is little
endian */
            if (__BYTE_ORDER == __LITTLE_ENDIAN) {
                float a;
                unsigned char *dst = (unsigned char *) & a;
                unsigned char *src = (unsigned char *) & b;
//projData[i][j];

                dst[0] = src[3];
                dst[1] = src[2];
                dst[2] = src[1];
                dst[3] = src[0];

                projData[i][j] = (double) a;
            }

            totalcounts += projData[i][j];
            projMin = MIN(projMin, projData[i][j]);
            projMax = MAX(projMax, projData[i][j]);
        }

    }
    file.close();
    return success;

}

bool ProjData::load_Sensitivity_Map(const string &filename, int ph) {
    bool success = true;
    int i = 0;
    long j = 0;
    float b;

    fstream file;
    file.open(filename.c_str(), fstream::in | fstream::binary);
    if (file.fail()) {
        cerr << "Could not open file " << filename << endl;
        return 0;
    }
```

```
    for (j = 0; j < yDetectDim * zDetectDim; j++) {
        file.read((char*) & /*projData[i][j]*/b, 4);

        /* Accounts for endianness...*.proj files are written as big-
endian.
         * Intel machines are native little endian. This reads in big-
endian
         * data as little-endian if the native format is little endian
*/
        if (__BYTE_ORDER == __LITTLE_ENDIAN) {
            float a;
            unsigned char *dst = (unsigned char *) & a;
            unsigned char *src = (unsigned char *) & b; //projData[i]
[j];

            dst[0] = src[3];
            dst[1] = src[2];
            dst[2] = src[1];
            dst[3] = src[0];

            sensitiveMap[ph][j] = (double) a;
            if (sensitiveMap[ph][j] == 0)
                sensitiveMap[ph][j] = 1;
        }
    }
    file.close();
    return success;

}

/* Copy data */
// Pretty clear

void ProjData::copy_dataEst_to_projData(double gamma) {
    int j = 0;
    int i = 0;
    for (i = 0; i < angleNum; i++) {
        for (j = 0; j < yDetectDim * zDetectDim; j++) {
            dataEst[i][j] = gamma * projData[i][j];
        }
    }
}

void ProjData::copyProjData2dataEst() {
    int i = 0;
    int j = 0;

    for (j = 0; j < angleNum; j++)
        for (i = 0; i < yDetectDim * zDetectDim; i++) {
            dataEst[j][i] = projData[j][i];
        }
}
```

```
void ProjData::copy_forwardScalingFactors() {
    int i = 0;
    int j = 0;

    for (i = 0; i < angleNum; i++)
        for (j = 0; j < yDetectDim * zDetectDim; j++)
            fsfData[i][j] = dataEst[i][j];

}

/* Operations*/

//Calculate rmse of dataEst

double ProjData::CalMSE() {
    double mse;
    double errorsum;

    int i = 0;
    int j = 0;
    int k = 0;
    mse = 0.0;
    errorsum = 0.0;

    // RMSE for non-zero
    double rmse = 0.0;
    double NonZeroCountNew = 0.0;
    double NonZeroMeasured = 0.0;
    double MaxProjData = 0.0;
    double sumProjValue = 0.0;

    for (i = 0; i < angleNum; i++) {
        for (j = 0; j < yDetectDim; j++) {
            for (k = 0; k < zDetectDim; k++) {
                if (dataEst[i][j * zDetectDim + k] != 0) {
                    errorsum += ((dataEst[i][j * zDetectDim + k] -
projData[i][j * zDetectDim + k])*(dataEst[i][j * zDetectDim + k] -
projData[i][j * zDetectDim + k]));
                    NonZeroCountNew++;
                }

                if (projData[i][j * zDetectDim + k] != 0) {
                    NonZeroMeasured++;
                    sumProjValue = sumProjValue + projData[i][j *
zDetectDim + k];
                }
            }
        }
    }
    if (sumProjValue != 0)
        rmse = sqrt((errorsum) / NonZeroCountNew) / (sumProjValue /
NonZeroMeasured)*100;

    else
```

```
        rmse = 10000;

    return rmse;
}

// Calculates rmse of to

double ProjData::CalMSE(float** to) {
    double mse;
    double errorsum;

    int i = 0;
    int j = 0;
    int k = 0;
    mse = 0.0;
    errorsum = 0.0;

    // RMSE for non-zero
    double rmse = 0.0;
    double NonZeroCountNew = 0.0;
    double NonZeroMeasured = 0.0;
    double MaxProjData = 0.0;
    double sumProjValue = 0.0;

    for (i = 0; i < angleNum; i++) {
        for (j = 0; j < yDetectDim; j++) {
            for (k = 0; k < zDetectDim; k++) {
                if (to[i][j * zDetectDim + k] != 0) {
                    errorsum += ((to[i][j * zDetectDim + k] -
projData[i][j * zDetectDim + k])*(to[i][j * zDetectDim + k] -
projData[i][j * zDetectDim + k]));
                    NonZeroCountNew++;
                }

                if (projData[i][j * zDetectDim + k] != 0) {
                    NonZeroMeasured++;
                    sumProjValue = sumProjValue + projData[i][j *
zDetectDim + k];
                }
            }
        }
    }
    if (sumProjValue != 0)
        rmse = sqrt((errorsum) / NonZeroCountNew) / (sumProjValue /
NonZeroMeasured)*100;

    else
        rmse = 10000;

    return rmse;
}

// double version
```

```
double ProjData::CalMSE(double** to) {
    double mse;
    double errorsum;

    int i = 0;
    int j = 0;
    int k = 0;
    mse = 0.0;
    errorsum = 0.0;

    // RMSE for non-zero
    double rmse = 0.0;
    double NonZeroCountNew = 0.0;
    double NonZeroMeasured = 0.0;
    double MaxProjData = 0.0;
    double sumProjValue = 0.0;

    for (i = 0; i < angleNum; i++) {
        for (j = 0; j < yDetectDim; j++) {
            for (k = 0; k < zDetectDim; k++) {
                if (to[i][j * zDetectDim + k] != 0) {
                    errorsum += ((to[i][j * zDetectDim + k] -
projData[i][j * zDetectDim + k])*(to[i][j * zDetectDim + k] -
projData[i][j * zDetectDim + k]));
                    NonZeroCountNew++;
                }

                if (projData[i][j * zDetectDim + k] != 0) {
                    NonZeroMeasured++;
                    sumProjValue = sumProjValue + projData[i][j *
zDetectDim + k];
                }
            }
        }
    }
    if (sumProjValue != 0)
        rmse = sqrt((errorsum) / NonZeroCountNew) / (sumProjValue /
NonZeroMeasured)*100;

    else
        rmse = 10000;

    return rmse;
}

// Calculates the legendre transform of Dkl...necessary for KLTV
algorithms

double ProjData::klDual() {
    double val = 0;
    double yt = 0;
    double eps = 1E-8;

    for (int a = 0; a < angleNum; a++) {
```

```
        for (int z = 0; z < zDetectDim; z++) {
            for (int y = 0; y < yDetectDim; y++) {
                if ((1 - get_dataEst(a, y, z)) >= eps)
                    yt = (1 - get_dataEst(a, y, z));
                else
                    yt = 1;
                val += get_projData(a, y, z) * log(yt);
            }
        }
    }

    return val;
}

// Calculates DKL(from,to) as in Barrett and Myers

double ProjData::CalDKL(double** from, double** to) {
    double dkl = 0;
    double yt = 0;
    double yf = 0;
    double eps = 1E-8;

    for (int i = 0; i < angleNum; i++) {
        for (int j = 0; j < yDetectDim * zDetectDim; j++) {
            if (from[i][j] >= eps)
                yf = from[i][j];
            else
                yf = 1;

            if (to[i][j] >= eps)
                yt = to[i][j];
            else
                yt = 1;

            dkl += to[i][j] - from[i][j] + from[i][j] * log(yf) -
from[i][j] * log(yt);
        }
    }

    return dkl;

}

double ProjData::CalDKL(float** from, double** to) {
    double dkl = 0;
    double yt = 0;
    double yf = 0;
    double eps = 1E-8;


    for (int i = 0; i < angleNum; i++) {
        for (int j = 0; j < yDetectDim * zDetectDim; j++) {
            if (from[i][j] >= eps)
                yf = from[i][j];
```

```
            else
                yf = 1;

            if (to[i][j] >= eps)
                yt = to[i][j];
            else
                yt = 1;

            dkl += to[i][j] - from[i][j] + from[i][j] * log(yf) -
from[i][j] * log(yt);
        }
    }

    return dkl;
}

double ProjData::CalDKL(float** from, float** to) {
    double ln = 0;
    double dkl = 0;
    double yt = 0;
    double yf = 0;
    double eps = 1E-8;


    for (int i = 0; i < angleNum; i++) {
        for (int j = 0; j < yDetectDim * zDetectDim; j++) {
            if (from[i][j] >= eps)
                yf = from[i][j];
            else
                yf = 1;

            if (to[i][j] >= eps)
                yt = to[i][j];
            else
                yt = 1;

            dkl += to[i][j] - from[i][j] + from[i][j] * log(yf) -
from[i][j] * log(yt);
        }
    }

    return dkl;

}

double ProjData::CalDKL(double** from, float** to) {
    double ln = 0;
    double dkl = 0;
    double yt = 0;
    double yf = 0;
    double eps = 1E-8;


    for (int i = 0; i < angleNum; i++) {
```

```
        for (int j = 0; j < yDetectDim * zDetectDim; j++) {
            if (from[i][j] >= eps)
                yf = from[i][j];
            else
                yf = 1;

            if (to[i][j] >= eps)
                yt = to[i][j];
            else
                yt = 1;

            dkl += to[i][j] - from[i][j] + from[i][j] * log(yf) -
from[i][j] * log(yt);
        }
    }

    return dkl;

}

//DKL(projData,DataEst)

double ProjData::CalDKL() {
    double ln = 0;
    double dkl = 0;

    for (int i = 0; i < angleNum; i++) {
        for (int j = 0; j < yDetectDim * zDetectDim; j++) {
            if (projData[i][j] == 0) {
                ln = 0;
            } else if (dataEst[i][j] <= /*0.00001*/0) {
                ln = projData[i][j] * log(projData[i][j] / 1E-31);
            } else
                ln = projData[i][j] * log(projData[i][j] / dataEst[i]
[j]);
            //            cout << ln << endl;
            dkl = dkl + dataEst[i][j] - projData[i][j] + ln;
        }
    }

    return dkl;

}

double ProjData::compute_RMS(int a) {
    int i = 0;
    double rms = 0;

    for (i = 0; i < yDetectDim * zDetectDim; i++)
        rms += (dataEst[a][i] - projData[a][i]) * (dataEst[a][i] -
projData[a][i]);

    return rms;
}
```

```
double ProjData::sumBins() {
    double sum = 0;
    int i = 0;
    int j = 0;

    for (i = 0; i < angleNum; i++)
        for (j = 0; j < yDetectDim * zDetectDim; j++)
            sum += projData[i][j];
    return sum;
}

void ProjData::shiftX(int amount) {
    /** This function shifts each projection image in X direction
     *      by the amount of pixels given to the function. It adds
     *      zeros to the shifted data.
     */
    int i = 0;
    int j = 0;
    int k = 0;

    double **dataShift;

    dataShift = new double*[angleNum];
    for (i = 0; i < angleNum; i++)
        dataShift[i] = new double[yDetectDim * zDetectDim];

    for (i = 0; i < angleNum; i++)
        for (j = 0; j < yDetectDim * zDetectDim; j++)
            dataShift[i][j] = 0;

    if (amount >= 0) {
        for (k = 0; k < angleNum; k++)
            for (j = 0; j < zDetectDim; j++)
                for (i = 0; i < yDetectDim - amount; i++)
                    dataShift[k][j * yDetectDim + i + amount] =
projData[k][j * yDetectDim + i];
    } else {
        for (k = 0; k < angleNum; k++)
            for (j = 0; j < zDetectDim; j++)
                for (i = -amount; i < yDetectDim; i++)
                    dataShift[k][j * yDetectDim + i + amount] =
projData[k][j * yDetectDim + i];
    }
    for (i = 0; i < angleNum; i++)
        for (j = 0; j < yDetectDim * zDetectDim; j++)
            projData[i][j] = dataShift[i][j];


    delete []dataShift;
}

void ProjData::correctDecay(double time, double halfLife) {
    /** Corrects the data slice by slice with
```

```
     * the function 1/(e^(-halfLife*time*slice))
     */
    int i = 0;
    int j = 0;
    double correctionFactor = 0.0;
    double lambda = 0.0;

    projMin = 99999999;
    projMax = 0;
    lambda = M_LN2 / halfLife;

    for (i = 0; i < angleNum; i++) {
        correctionFactor = pow(M_E, (-lambda * time * (angleNum - i)));
        for (j = 0; j < yDetectDim * zDetectDim; j++) {
            projData[i][j] = projData[i][j] / correctionFactor;
            projMin = MIN(projMin, projData[i][j]);
            projMax = MAX(projMax, projData[i][j]);
        }
    }
    //      emit signalProjChanged();
}

// Getts fftw ready for convolution for gaussian filter kernel

void ProjData::initGaussianBlur(double radius) {
    realData = (long double*) fftwl_malloc(sizeof (long double) * 2 *
yDetectDim * zDetectDim);
    inData = (fftwl_complex*) fftwl_malloc(sizeof (fftwl_complex) * (2
* yDetectDimHalf + 1) * zDetectDim);
    outData = (fftwl_complex*) fftwl_malloc(sizeof (fftwl_complex) * (2
* yDetectDimHalf + 1) * zDetectDim);
    filterKernel = (fftwl_complex*) fftwl_malloc(sizeof (fftwl_complex)
* (2 * yDetectDimHalf + 1) * zDetectDim);

    memset(realData, 0, 2 * yDetectDim * zDetectDim * sizeof (long
double));
    memset(inData, 0, (2 * yDetectDimHalf + 1) * zDetectDim * sizeof
(fftwl_complex));
    memset(outData, 0, (2 * yDetectDimHalf + 1) * zDetectDim * sizeof
(fftwl_complex));
    memset(filterKernel, 0, (2 * yDetectDimHalf + 1) * zDetectDim *
sizeof (fftwl_complex));

    pAA = fftwl_plan_dft_r2c_1d(2 * yDetectDim, realData, outData,
FFTW_MEASURE);
    pBB = fftwl_plan_dft_c2r_1d(2 * yDetectDim, inData, realData,
FFTW_MEASURE);

    filterScaling = 1.0 / (double) (2 * yDetectDim * zDetectDim);
    createGaussianFilterKernel(radius);
}

// Makes fft of 2-d gaussian for use with blurring model
void ProjData::createGaussianFilterKernel(double radius) {
```

```
    int i = 0;
    int j = 0;
    int ij = 0;

    double detectBlurGaussValue = 0.0;
    double probGauss = 0.0;
    double probSum = 0.0;
    double ylenSq = 0.0;
    double zlenSq = 0.0;

    detectBlurGaussValue = -(1 / (2 * radius * radius));


    for (i = 0; i < zDetectDim; i++)
        for (j = 0; j < 2 * yDetectDim; j++) {
            ij = j + i * 2 * yDetectDim;
            ylenSq = (j - (yDetectDim / 2) - (yDetectDim / 2)) * (j -
(yDetectDim / 2) - (yDetectDim / 2));
            zlenSq = (i - zDetectDim / 2) * (i - zDetectDim / 2);
            probGauss = exp(detectBlurGaussValue * (ylenSq + zlenSq));
            if (j > yDetectDim / 2 && j < 1.5 * yDetectDim) {
                realData[ij] = probGauss;
                probSum += probGauss;
            } else
                realData[ij] = 0;
        }

    i = 0;
    for (j = 0; j < 2 * yDetectDim; j++) {
        ij = j + i * 2 * yDetectDim;
        realData[ij] = realData[ij] / probSum;
    }

    fftwl_execute(pAA);

    for (i = 0; i < zDetectDim; i++)
        i = 0;
    for (j = 0; j < (2 * yDetectDimHalf + 1); j++) {
        ij = j + i * (2 * yDetectDimHalf + 1);
        c_re(filterKernel[ij]) = c_re(outData[ij]);
        c_im(filterKernel[ij]) = c_im(outData[ij]);

        c_re(inData[ij]) = c_re(outData[ij]);
        c_im(inData[ij]) = c_im(outData[ij]);
    }

    fftwl_execute(pBB);
}

//fft to and multiply with fft of gaussian, transform back and be happy
void ProjData::applyGaussianBlur(int a, double** to) {
    int i = 0;
    int j = 0;
    int ij = 0;
```

```
    int ii = 0;
    int jj = 0;
    int iijj = 0;

    double** tester;
    tester = new double*[angleNum];
    for (i = 0; i < angleNum; i++)
        tester[i] = new double[2 * yDetectDim * zDetectDim];
    for (i = 0; i < zDetectDim; i++)
        memset(tester[i], 0, 2 * yDetectDim * zDetectDim * sizeof
(double));

    //set realData to data and zero pad
    for (i = 0; i < zDetectDim; i++)
        for (j = 0; j < 2 * yDetectDim; j++) {
            ij = j + i * 2 * yDetectDim;
            realData[ij] = to[a][ij];

            if (j > yDetectDim / 2 && j < 3 * yDetectDim / 2) {
                realData[ij] = to[a][(j - yDetectDim / 2) + i * 2 *
yDetectDim];

            } else
                realData[ij] = 0;
        }
    //forward transform
    fftwl_execute(pAA);

    //multiply filter kernel and complex output data
    for (i = 0; i < zDetectDim; i++)
        for (j = 0; j < (2 * yDetectDimHalf + 1); j++) {
            ij = j + i * (2 * yDetectDimHalf + 1);
            c_re(inData[ij]) = (c_re(outData[ij]) *
c_re(filterKernel[ij])
                    - c_im(outData[ij]) * c_im(filterKernel[ij]));
            c_im(inData[ij]) = (c_re(outData[ij]) *
c_im(filterKernel[ij])
                    + c_im(outData[ij]) * c_re(filterKernel[ij]));
        }
    //inverse transform
    fftwl_execute(pBB);

    //set data to interim matrix
    for (i = zDetectDimHalf, ii = 0; i < zDetectDim; i++, ii++)
        for (j = 2 * yDetectDimHalf, jj = 0; j < 2 * yDetectDim; j++,
jj++) {
            ij = j + i * 2 * yDetectDim;
            iijj = jj + ii * 2 * yDetectDim;
            tester[0][iijj] = realData[ij] * filterScaling;
            tester[0][ij] = realData[iijj] * filterScaling;

            ij = jj + i * 2 * yDetectDim;
            iijj = j + ii * 2 * yDetectDim;
            tester[0][iijj] = realData[ij] * filterScaling;
```

```
            tester[0][ij] = realData[iijj] * filterScaling;
        }

    //remove zero padding
    for (i = 0; i < zDetectDim; i++)
        for (j = 0.5 * yDetectDim; j < 1.5 * yDetectDim; j++) {
            to[a][(j - (yDetectDim / 2)) + i * yDetectDim] = tester[0]
[j + i * 2 * yDetectDim];
        }

    for (i = 0; i < angleNum; i++)
        delete[] tester[i];
    delete[] tester;
}

void ProjData::scale_dataEst() {
    int i = 0;
    int j = 0;
    int k = 0;

    for (i = 0; i < angleNum; i++)
        for (j = 0; j < yDetectDim; j++)
            for (k = 0; k < zDetectDim; k++)
                if (fsfData[i][j + k * yDetectDim] <= 0.0001)
                    dataEst[i][j + k * yDetectDim] /= 1.0;
                else
                    dataEst[i][j + k * yDetectDim] /= fsfData[i][j + k
* yDetectDim];
}

void ProjData::scale_projData() {
    int i = 0;
    int j = 0;
    int k = 0;

    for (i = 0; i < angleNum; i++)
        for (j = 0; j < yDetectDim; j++)
            for (k = 0; k < zDetectDim; k++)
                if (fsfData[i][j + k * yDetectDim] <= 0.0001)
                    projData[i][j + k * yDetectDim] /= 1.0;
                else
                    projData[i][j + k * yDetectDim] /= fsfData[i][j + k
* yDetectDim];
}

void ProjData::scale_dataEst(int angle) {
    int j = 0;
    int k = 0;

    for (j = 0; j < yDetectDim; j++)
        for (k = 0; k < zDetectDim; k++)
            if (fsfData[angle][j + k * yDetectDim] <= 0.0001)
                dataEst[angle][j + k * yDetectDim] /= 1.0;
            else
```

```
                dataEst[angle][j + k * yDetectDim] /= fsfData[angle][j
+ k * yDetectDim];
}
```

projdata.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */

/
***************************************************************************
                          projdata.h  -  description
                             -------------------
    begin                : Fri Mar 8 2002
    copyright            : (C) 2002 by Christian Wietholt
    email                : christian.wietholt@marquette.edu

***************************************************************************
****/

/
***************************************************************************
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

***************************************************************************
****/

#ifndef PROJDATA_H
#define PROJDATA_H

#ifndef M_E
#define M_E 2.71828182845904523536028747135266625L  /* e */
#endif
#ifndef M_LN2
#define M_LN2 0.69314718055994530941723212145817766L  /* log_e 2 */
#endif

#define c_re(c) ((c)[0])
#define c_im(c) ((c)[1])

#include "fftw3.h"
#include "math.h"

#include "point3d.h"
```

```
#include "projectorenum.h"
#include "recondata.h"
#include "defs.h"
#include <cstring>
#include <string>
#include <iostream>
using namespace std;

/**This class includes all the function necessary to deal with
the projection data and to calculate the data from the created
distribution. It will be inherited by the SimSPECTDoc class.
 *@author Christian Wietholt
 */

class ProjData {

public:
    ProjData();
    ~ProjData();
    /** This function will initialize the data arrays for the
projection to be created.  */
    bool init(int pn, int detnum, unsigned short y, unsigned short z,
unsigned short yD, unsigned short zD, unsigned short a,
            double yL, double zL, double p2d, double p2o,
            unsigned short pMat, double pDia, unsigned short vT, double
hl, double psf,
            string fnPhM, double pMatSY, double pMatSZ, int locID, int
typeID, double *yPHS,
            double *zPHS, double yDS, double zDS, double xDT, double
yDT, double zDT, unsigned short sym);
    /** Deals with all the memory allocation stuff for the projection
data.  */
    bool initProjData();
    void initRayDrivenVariables();
    void initPinhole();
    void initPinholeFunctRound();
    void allocPinholeFunct(int num);

    /** This function shifts each projection image in X direction by
     *      the amount of pixels given to the function. It adds zeros to
     *      the shifted data.
     */
    virtual void shiftX(int amount);
    /** Corrects the data slice by slice with
     * the function 1/(e^(-halfLife*time*slice))
     */
    void correctDecay(double time, double halfLife);
    void reset_dataEst();
    void reset_data(double** data);
    void reset_prevDataEst();
    void set_prevDataEst();

    double CalMSE();
    double CalMSE(double** to);
```

```
    double CalMSE(float** to);

    double CalDKL();
    double CalDKL(double** from, double** to);
    double CalDKL(double** from, float** to);
    double CalDKL(float** from, double** to);
    double CalDKL(float** from, float** to);

    float get_projData(int ang, int ycoor, int zcoor) {
        return projData[ang][ycoor + (zcoor * yDetectDim)];
    };

    double** get_projData() {
        return projData;
    }

    void get_ProjLine(double *line, unsigned short yPos, unsigned short
zPos);

    unsigned short get_yDetectDim() {
        return yDetectDim;
    };

    unsigned short get_zDetectDim() {
        return zDetectDim;
    };

    unsigned short get_pinHoleMatrix() {
        return pinHoleMatrix;
    };

    unsigned short get_angleNum() {
        return angleNum;
    };

    int get_ElementNum() {
        return elementNum;
    };

    int get_PinHoleType() {
        return pinHoleType;
    }

    double get_projMin() {
        return projMin;
    }

    double get_projMax() {
        return projMax;
    }

    double get_data(int a, int yCoor, int zCoor, double** data) {
        return data[a][yCoor + (zCoor * yDetectDim)];
    };
```

```
    double get_dataEst(int a, int yCoor, int zCoor) {
        return dataEst[a][yCoor + (zCoor * yDetectDim)];
    };

    double** get_dataEst() {
        return dataEst;
    };

    double get_sensitiveMap(unsigned short pinhole, unsigned short z,
unsigned short y) {
        return sensitiveMap[pinhole][y + z * yDetectDim ];
    };

    double** get_sensitiveMap() {
        return sensitiveMap;
    };

    double get_prevDataEst(int a, int yCoor, int zCoor) {
        return prevDataEst[a][yCoor + (zCoor * yDetectDim)];
    };

    double** get_prevDataEst() {
        return prevDataEst;
    };

    double get_p2oLength() {
        return p2oLength;
    };

    double get_p2dLength() {
        return p2dLength;
    };

    double get_elementSizeY() {
        return elementSizeY;
    };

    double get_elementSizeZ() {
        return elementSizeZ;
    };

    double get_DetectPosY(int i) {
        return DetectPosY[i];
    };

    double get_DetectPosZ(int i) {
        return DetectPosZ[i];
    };
    double get_ScalingFactor();

    double get_PinHoleRadius() {
        return pinHoleRadius;
    };
```

```
double get_DetectorBlurrRadius() {
    return detectorBlurrRadius;
};
double get_DetectPosYN(int i);
double get_DetectPosZN(int i);

double get_totalcounts() {
    return totalcounts;
};

unsigned short get_detectnum() {
    return detectnum;
};
double (ProjData::*get_PinHoleFunctPtr)(int, int, int);

double get_angleStepSize() {
    return angleStepSize;
};

double get_PinHoleFunctCorner(int g, int h, int pos) {
    return pinHoleFunct[pos][g][h];
};

double get_PinHoleFunctCenter(int g, int h, int pos) {
    return pinHoleFunct[(detectEle - 1) - pos][g][h];
};

double get_PinHoleFunctCenterSym(int g, int h, int pos) {
    return pinHoleFunct[detectEleQuarterM - pos][g][h];
};

double get_PinHoleFunctNoFile(int g, int h, int pos) {
    return pinHoleFunct[pos][g][h];
};

int get_pinholenum() {
    return pinnum;
}

unsigned short get_symmetry(); //{return symmetry;};

Point3D get_startDetectPoint() {
    return *startDetectPoint;
};

Point3D get_startPinHolePoint(int p) {
    return *startPinHolePoint[p];
};

Point3D get_incrementPinHoleY() {
    return *incrementPinHoleY;
};
```

```
    Point3D get_incrementPinHoleZ() {
        return *incrementPinHoleZ;
    };

    Point3D get_incrementDetectY() {
        return *incrementDetectY;
    };

    Point3D get_incrementDetectZ() {
        return *incrementDetectZ;
    };

    void add_dataEst(int a, int yCoor, int zCoor, double inten) {
        dataEst[a][yCoor + (zCoor * yDetectDim)] += inten;
    };

    void add_data(int a, int yCoor, int zCoor, double inten, double
**data) {
        data[a][yCoor + (zCoor * yDetectDim)] += inten;
    };

    double klDual();

    void setReconPointer(ReconData *r) {
        recon = r;
    };
    void save_oneproj(const string &filename, int a);
    void save_onenewproj(const string &filename, int a);
    void save_projData(const string &filename);
    void save_someProjData(const string &filename, double** data);
    void save_dataEst(const string &filename);
    void save_pinholeFunct(const string &filename, int num);
    void compareDivide(int a);
    void compareSubtract(int a);

    void compareProjDataVdataEst();
    void compareProjDataVdataEst(double compare);

    void copyProjData2dataEst();
    bool load(const string &filename);
    bool load_Sensitivity_Map(const string &filename,int ph);
    void save_forward(const string &filename, int s);
    double sumBins();
    double compute_RMS(int a);
    void initGaussianBlur(double radius);
    void applyGaussianBlur(int a, double** to);
    void createGaussianFilterKernel(double radius);
    void set_dataEst(int a, double val);
    void set_projData(int a, int y, int z, double val);
    void set_dataEst(int a, int y, int z, double val);
    void set_dataEst(int a, int i, double val);
    void copy_dataEst_to_projData(double gamma);
    /** Stuff for fan beam projections */
    SETANG *ang;
```

```
    double *ax;
    FAN fan;
    void set_angles(double max_angle);

    void init_forwardScalingFactors();
    void copy_forwardScalingFactors();
    void save_forwardScalingFactors(const string &filename);

    void scale_projData();
    void scale_dataEst();
    void scale_dataEst(int angle);

    void enforceFOV(double** theData) {

        for (int a = 0; a < angleNum; a++) {
            for (int z = 0; z < zDetectDim; z++) {
                for (int y = 0; y < 5; y++) {
                    theData[a][z * yDetectDim + (yDetectDim - 1 - y)] =
0;
                    theData[a][z * yDetectDim + y] = 0;
                }
            }
        }
    }
    double totalcounts;


protected: // protected attributes

    unsigned short yDetectDim;
    unsigned short zDetectDim;
    unsigned short angleNum;
    unsigned short pinHoleMatrix;
    unsigned short viewTime;

    int elementNum;
    int pinnum;
    /** This is for storing some scaling factors for the projection
images */
    double **fsfData;
    /** Here the data loaded from the file is stored.  */
    double** projData;

    double** sensitiveMap;

    /** Here the data created by the projection algorithm is stored.
*/
    double** dataEst;
    double** prevDataEst;

    double ***pinHoleFunct;
    /** the halflife of the isotope used */
    double halflife;
    /** the minimum and maximum in the projection data */
```

```
    double projMin;
    double projMax;
    double yDetectLength;
    double zDetectLength;
    double p2dLength;
    double p2oLength;
    /** the size of one detector element in y direction*/
    double elementSizeY;
    /** the size of one detector element in z direction*/
    double elementSizeZ;
    /** The amount of each angular rotation in radians */
    double angleStepSize;
    /** the size of one pinhole grid element */
    double pinHoleElementSize;
    /** the diameter of the pinhole in millimeter */
    double pinHoleDiameter;
    /** the radius of the pinhole in millimeter */
    double pinHoleRadius;
    /** The positions in y direction of the detector elements. */
    double* DetectPosY;
    /** The positions in z direction of the detector elements. */
    double* DetectPosZ;
    /** the pinhole function */
    double detectorBlurrFWHM;
    double detectorBlurrRadius;
    double scalingFactor;

    /** the position of all rays through the pinhole */
    Point3D *pinHoleRayPos;
    Point3D *pinHoleRayHexaPos;

    /** center of pixel in upper left corner */
    Point3D **startPinHolePoint;
    Point3D *incrementPinHoleY;
    Point3D *incrementPinHoleZ;

    Point3D *startDetectPoint;
    Point3D *incrementDetectY;
    Point3D *incrementDetectYRot;
    Point3D *incrementDetectZ;

protected: // protected methods

    template<typename T> T MIN(const T &x, const T &y) {
        return ( (x) < (y) ? (x) : (y));
    }

    template<typename T> T MAX(const T &x, const T &y) {
        return ( (x) > (y) ? (x) : (y));
    }

private:
    bool initData();
```

```
    void shiftPinhole(int i) {
        startPinHolePoint[i]->Translate(0.0, yPhShift[i], zPhShift[i]);
    };

    unsigned short symmetry;
    int detectnum;
    int yDetectDimHalfP;
    int zDetectDimHalfP;
    int yDetectDimHalf;
    int zDetectDimHalf;
    int detectEle;
    int detectEleQuarter;
    int detectEleQuarterM;
    int pinHoleType;
    double *yPhShift;
    double *zPhShift;
    double yPHStart2Center;
    double zPHStart2Center;

    double yDetectShift;
    double zDetectShift;
    double xDetectTilt;
    double yDetectTilt;
    double zDetectTilt;
    double xDetCenterPos;
    double yDetStart2Center;
    double zDetStart2Center;
    double pixelArea;
    double filterScaling;
    double pinHoleMatrixSizeY;
    double pinHoleMatrixSizeZ;

    long double *realData;

    fftwl_complex *inData;
    fftwl_complex *outData;
    fftwl_complex *filterKernel;

    fftwl_plan pAA;
    fftwl_plan pBB;
    ReconData *recon;

    string fileNamePinholeModel;

};
#endif
```

raytbl.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
**********************************************************************
****
 *   Copyright (C) 2005 by Christian Wietholt
*
```

```
 *   chris@gamma.mipl.cgu.edu.tw
*
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*
 *   This program is distributed in the hope that it will be useful,
*
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
*
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*
 *   GNU General Public License for more details.
*
 *
*
 *   You should have received a copy of the GNU General Public License
*
 *   along with this program; if not, write to the
*
 *   Free Software Foundation, Inc.,
*
 *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
*

**********************************************************************
****/


#include <stdlib.h>

#include "raytbl.h"
#include <iostream>
using namespace std;

RayTBL::RayTBL(short xdim, short ydim) {
    bool success = true;
    xDim = xdim;
    yDim = ydim;
    size = 0;

    if ((x = new short[(int) (xDim * 2)]) == NULL)
        success = false;
    if ((y = new short[(int) (yDim * 2)]) == NULL)
        success = false;
    if ((length = new double[(int) (xDim * 2)]) == NULL)
```

```
        success = false;

    if (!success) {
        cout << "Memory Allocation Error" << endl;
        exit(0);
    } else
        reset();
}

void RayTBL::reset() {
    memset(x, 0, sizeof (short) *(int) (2 * xDim));
    memset(y, 0, sizeof (short) *(int) (2 * yDim));
    memset(length, 0, sizeof (double) *(int) (2 * xDim));
}

void RayTBL::setValues(int pos, short xv, short yv, double lv, int i,
int j, int g) {
    i = i;
    j = j;
    g = g;
    length[pos] = lv;
    x[pos] = xv;
    y[pos] = yv;
}

RayTBL::~RayTBL() {
    delete[]x;
    delete[]y;
    delete[]length;
}
```

raytbl.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
**********************************************************************
****
 *   Copyright (C) 2005 by Christian Wietholt
*
 *   chris@gamma.mipl.cgu.edu.tw
*
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*
 *   This program is distributed in the hope that it will be useful,
*
```

```
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
*
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*
 *   GNU General Public License for more details.
*
 *
*
 *   You should have received a copy of the GNU General Public License
*
 *   along with this program; if not, write to the
*
 *   Free Software Foundation, Inc.,
*
 *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
*

**********************************************************************
****/
#ifndef RAYTBL_H
#define RAYTBL_H

#include <string.h>

/**
@author Christian Wietholt
 */
class RayTBL {
public:
    RayTBL(short xdim, short ydim);

    ~RayTBL();

    void reset();
    void setValues(int pos, short xv, short yv, double lv, int i, int
j, int g);

    short *x;
    short *y;
    double *length;
    short size;
    short xDim;
    short yDim;


};

#endif
```

raytbl3d.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
**********************************************************************
****
```

```
 *   Copyright (C) 2006 by Christian Wietholt
*
 *   chris@gamma.mipl.cgu.edu.tw
*
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*
 *   This program is distributed in the hope that it will be useful,
*
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
*
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*
 *   GNU General Public License for more details.
*
 *
*
 *   You should have received a copy of the GNU General Public License
*
 *   along with this program; if not, write to the
*
 *   Free Software Foundation, Inc.,
*
 *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
*

**********************************************************************
****/

#include <stdlib.h>
#include <iostream>
using namespace std;

#include "raytbl3d.h"

RayTBL3D::RayTBL3D(unsigned short xdim, unsigned short size) {
    bool success = true;
    if (size == 0)
        arrSize = xdim * 2;
    else
        arrSize = size;

    if ((xyz = new unsigned int[arrSize]) == NULL)
        success = false;
    if ((length = new double[arrSize]) == NULL)
```

```
        success = false;

    if (!success) {
        cout << "Memory Allocation Error" << endl;
        exit(0);
    } else
        reset();
}

void RayTBL3D::reset() {
    memset(xyz, 0, sizeof (unsigned int) * arrSize);
    memset(length, 0, sizeof (double) * arrSize);
}

RayTBL3D::~RayTBL3D() {
    delete[]xyz;
    delete[]length;
}
```

raytbl3d.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
**********************************************************************
****
 *   Copyright (C) 2006 by Christian Wietholt
*
 *   chris@gamma.mipl.cgu.edu.tw
*
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*
 *   This program is distributed in the hope that it will be useful,
*
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
*
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*
 *   GNU General Public License for more details.
*
 *
*
 *   You should have received a copy of the GNU General Public License
*
 *   along with this program; if not, write to the
*
```

```
 *   Free Software Foundation, Inc.,
*
 *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
*

**********************************************************************
****/
#ifndef RAYTBL3D_H
#define RAYTBL3D_H

#include <string.h>

/**
@author Christian Wietholt
 */
class RayTBL3D {
public:
    RayTBL3D(unsigned short xdim, unsigned short size);

    ~RayTBL3D();

    void reset();

    unsigned int *xyz;
    double *length;
    unsigned short arrSize;
};

#endif
```

raytbl3dinc.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
**********************************************************************
****
 *   Copyright (C) 2006 by Christian Wietholt
*
 *   chris@gamma.mipl.cgu.edu.tw
*
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*
 *   This program is distributed in the hope that it will be useful,
*
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
*
```

```
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*
 *   GNU General Public License for more details.
*
 *
*
 *   You should have received a copy of the GNU General Public License
*
 *   along with this program; if not, write to the
*
 *   Free Software Foundation, Inc.,
*
 *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
*

**********************************************************************
****/

#include "purge.h"
#include "raytbl3dinc.h"

RayTBL3DInc::RayTBL3DInc() {
}

RayTBL3DInc::~RayTBL3DInc() {
    length.clear();
    xyzInc.clear();
    xyzSgn.clear();
}
```

raytbl3dinc.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
**********************************************************************
****
 *   Copyright (C) 2006 by Christian Wietholt
*
 *   chris@gamma.mipl.cgu.edu.tw
*
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*
 *   This program is distributed in the hope that it will be useful,
*
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
*
```

```
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*
 *   GNU General Public License for more details.
*
 *
*
 *   You should have received a copy of the GNU General Public License
*
 *   along with this program; if not, write to the
*
 *   Free Software Foundation, Inc.,
*
 *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
*

**************************************************************************
****/
#ifndef RAYTBL3DINC_H
#define RAYTBL3DINC_H
#include <vector>
#include <string.h>

typedef std::vector<double> LengthContainer;
typedef LengthContainer::iterator LengthIter;
typedef LengthContainer::reverse_iterator rLengthIter;

typedef std::vector<unsigned char> IncContainer;
typedef IncContainer::iterator IncIter;
typedef IncContainer::reverse_iterator rIncIter;

/**
@author Christian Wietholt
 */
class RayTBL3DInc {
public:
    RayTBL3DInc();

    ~RayTBL3DInc();

    void clear_all() {
        length.clear();
        xyzInc.clear();
        xyzSgn.clear();
    };

    LengthContainer length;
    unsigned short xStart;
    unsigned short yStart;
    unsigned short zStart;
    IncContainer xyzInc;
    IncContainer xyzSgn;
};
```

```
#endif
```

recondata.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************************
***
                          recondata.cpp  -  description
                             -------------------
    begin                : Fri Jan 5 2005
    copyright            : (C) 2005 by Christian Wietholt
    email                : christian.wietholt@marquette.edu

***************************************************************************
****/

/
***************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

***************************************************************************
****/

#include "recondata.h"
#include "projdata.h"
#include <iostream>
#include <fstream>
#include <sstream>
#include <stdlib.h>
using namespace std;
/**
 * Constructors/Destructors
 */

/**
 * Methods
 */

/* Initializers */

// Initializes cylinder that is fov of the system
void ReconData::init_recondata() {
    int i = 0;
```

```
    double iz = 0.0;
    int j = 0;
    double jy = 0.0;
    int k = 0;
    double kx = 0.0;
    long counts = 0;
    double tcounts = 0.0;
    reset_data();

    iz = planeZ[/*42*/0];
    for (i = /*42*/0; i < /*87*/zDim; i++) {
        jy = planeY[0];
        for (j = 0; j < yDim; j++) {
            kx = planeX[0];
            for (k = 0; k < xDim; k++) {
                if (((jy - 0.0)*(jy - 0.0)+(kx - 0.0)*(kx - 0.0)) <
441) {
                    counts++;
                    data[i][j * xDim + k] = 1;
                } else {
                    data[i][j * xDim + k] = 0;
                }
                kx = kx + xPixelSize;
            }
            jy = jy + yPixelSize;
        }
        iz = iz + zPixelSize;
    }
}

bool ReconData::init(int x, int y, int z) {
    /** This function allocates memory to store the generated density
distribution and
    initializes it with the background intensity */
    int i = 0;
    bool success = true;

    withAttn = false;

    xDim = x;
    yDim = y;
    zDim = z;

    // get memory for the distribution data
    data = new double*[zDim];
    objectEst = new double*[zDim];
    objectUpdate = new double*[zDim];
    prevObjectUpdate = new double*[zDim];
    divQ = new double*[zDim];
    magImage = new double*[zDim];
    for (i = 0; i < zDim; i++) {
        objectEst[i] = new double[xDim * yDim];
        objectUpdate[i] = new double[xDim * yDim];
        prevObjectUpdate[i] = new double[xDim * yDim];
```

```
        data[i] = new double[xDim * yDim];
        divQ[i] = new double[xDim * yDim];
        magImage[i] = new double[xDim * yDim];
    }

    planeX = new double[xDim + 2];
    planeY = new double[yDim + 2];
    planeZ = new double[zDim + 2];

    voxelPosX = new double[xDim];
    voxelPosY = new double[yDim];
    voxelPosZ = new double[zDim];

    if (success) {
        for (i = 0; i < zDim; i++) {
            memset(objectEst[i], 0, xDim * yDim * sizeof (double));
            memset(data[i], 0, xDim * yDim * sizeof (double));
            memset(objectUpdate[i], 0, xDim * yDim * sizeof (double));
            memset(prevObjectUpdate[i], 0, xDim * yDim * sizeof
(double));
            memset(divQ[i], 0, xDim * yDim * sizeof (double));
            memset(magImage[i], 0, xDim * yDim * sizeof (double));
        }
    }

    init_recondata();
    return success;
}

bool ReconData::initRotation() {
    int i = 0;

    if (!rotInitialized) {
        rotInitialized = true;
        // get memory for the distribution data
        rotData = new double*[zDim];
        brotData = new double*[zDim];
        for (i = 0; i < zDim; i++) {
            rotData[i] = new double[xDim * yDim];
            brotData[i] = new double[xDim * yDim];
        }

        for (i = 0; i < zDim; i++) {
            memset(rotData[i], 0, xDim * yDim * sizeof (double));
            memset(brotData[i], 0, xDim * yDim * sizeof (double));
        }

    }
    return true;
}

void ReconData::init_backScalingFactors() {
    int i = 0;
    int j = 0;
```

```
    bsfData = new double*[zDim];
    tbsfData = new double*[zDim];
    for (i = 0; i < zDim; i++) {
        bsfData[i] = new double[xDim * yDim];
        tbsfData[i] = new double[xDim * yDim];
    }

    for (i = 0; i < zDim; i++) {
        for (j = 0; j < xDim * yDim; j++) {
            bsfData[i][j] = 1.0;
            tbsfData[i][j] = 1.0;
        }
    }
}

bool ReconData::initVectors() {
    int i = 0;
    int j = 0;
    int bufferNum = 2;
    // get memory for the distribution data
    qVector = new double**[bufferNum];
    gradUbarVector = new double**[bufferNum];
    for (i = 0; i < bufferNum; i++) {
        qVector[i] = new double*[zDim];
        gradUbarVector[i] = new double*[zDim];
        for (j = 0; j < zDim; j++) {
            qVector[i][j] = new double[xDim * yDim];
            gradUbarVector[i][j] = new double[xDim * yDim];
        }
    }
    for (i = 0; i < bufferNum; i++)
        for (j = 0; j < zDim; j++) {
            memset(qVector[i][j], 0, xDim * yDim * sizeof (double));
            memset(gradUbarVector[i][j], 0, xDim * yDim * sizeof
(double));
        }

    return true;
}

/* Accessors*/
//Pretty clear
double ReconData::get_LinearInterpolVal(Point3D p, double **d) {
    int xCoor = 0;
    int yCoor = 0;
    int zCoor = 0;
    int xCoorP = 0;
    int yCoorP = 0;

    double prob1 = 0.0;
    double prob2 = 0.0;
    double prob3 = 0.0;
    double prob4 = 0.0;
```

```
    double inten = 0.0;
    double xInter = 0.0;
    double yInter = 0.0;

    xCoor = (int) ((p.x - voxelPosX[0]) / xPixelSize);
    yCoor = (int) ((p.y - voxelPosY[0]) / yPixelSize);
    zCoor = (int) ((p.z - voxelPosZ[0]) / zPixelSize);

    xCoorP = xCoor + 1;
    yCoorP = yCoor + 1;

    if (xCoor > 0 && xCoorP < xDim && yCoor > 0 && yCoorP < yDim) {
        xInter = (double) (p.x - voxelPosX[xCoor]) / xPixelSize;
        yInter = (double) (p.y - voxelPosY[yCoor]) / yPixelSize;

        // compute the probability of each interpolated pixel
        prob1 = (1.0 - xInter) * (1.0 - yInter);
        prob2 = xInter * (1.0 - yInter);
        prob3 = xInter * yInter;
        prob4 = (1.0 - xInter) * yInter;

        inten = (double) d[zCoor][xCoor + (yCoor * xDim)] * prob1
                + (double) d[zCoor][xCoorP + (yCoor * xDim)] * prob2
                + (double) d[zCoor][xCoorP + (yCoorP * xDim)] * prob3
                + (double) d[zCoor][xCoor + (yCoorP * xDim)] * prob4;
    } else {
        inten = 0;
    }
    return (double) inten;
}

// Sets vector objectEst = 1
void ReconData::setFirstObjectEst() {
    int i = 0;
    double iz = 0.0;
    int j = 0;
    double jy = 0.0;
    int k = 0;
    double kx = 0.0;
    long counts = 0;
    double tcounts = 0.0;
    reset_objectEst();

    tcounts = proj->get_totalcounts();
    cout << "tcounts: " << tcounts << endl;
    iz = planeZ[0]; //nee 42
    for (i = 0/*42*/; i < zDim/*87*/; i++) {
        jy = planeY[0];
        for (j = 0; j < yDim; j++) {
            kx = planeX[0];
            for (k = 0; k < xDim; k++) {
                if (((jy - 0.0)*(jy - 0.0)+(kx - 0.0)*(kx - 0.0)) <=
441) {
                    counts++;
```

```
                    objectEst[i][j * xDim + k] = 1;
                } else {
                    objectEst[i][j * xDim + k] = 0;
                }
                kx = kx + xPixelSize;
            }
            jy = jy + yPixelSize;
        }
        iz = iz + zPixelSize;
    };
}

void ReconData::setLength(double l) {
    /** sets the length of the reconstruction space.  */
    int i = 0;

    double xTemp = 0;
    double yTemp = 0;
    double zTemp = 0;

    xLength = yLength = zLength = l;

    //calculate pixel size
    xPixelSize = xLength / xDim;
    yPixelSize = yLength / yDim;
    zPixelSize = zLength / zDim;

    // compute the volume of one voxel
    voxelVolume = xPixelSize * yPixelSize * zPixelSize;

    incrementX.Create(xPixelSize, 0.0, 0.0);
    //incrementXRot.Create(xPixelSize, 0.0, 0.0);

    incrementY.Create(0.0, yPixelSize, 0.0);
    //incrementYRot.Create(0.0, yPixelSize, 0.0);

    incrementZ.Create(0.0, 0.0, zPixelSize);

    xTemp = -xLength / 2 + xPixelSize / 2;
    yTemp = -yLength / 2 + yPixelSize / 2;
    zTemp = -zLength / 2 + zPixelSize / 2;

    startPoint.Create(xTemp, yTemp, zTemp);
    //startPointRot.Create(xTemp, yTemp, zTemp);

    for (i = 0; i < xDim + 2; i++)
        planeX[i] = -(xLength / 2) + i * xPixelSize;
    for (i = 0; i < yDim + 2; i++)
        planeY[i] = -(yLength / 2) + i * yPixelSize;
    for (i = 0; i < zDim + 2; i++)
        planeZ[i] = -(zLength / 2) + i * zPixelSize;

    for (i = 0; i < xDim; i++)
        voxelPosX[i] = startPoint.x + i * xPixelSize;
```

```
    for (i = 0; i < yDim; i++)
        voxelPosY[i] = startPoint.y + i * yPixelSize;
    for (i = 0; i < zDim; i++)
        voxelPosZ[i] = startPoint.z + i * zPixelSize;
}

void ReconData::setProjPointer(ProjData *p) {
    proj = p;
}

void ReconData::reset_data() {
    int i = 0;
    for (i = 0; i < zDim; i++)
        memset(data[i], 0, xDim * yDim * sizeof (double));
}

void ReconData::reset_someData(double** someData) {
    int i = 0;
    for (i = 0; i < zDim; i++)
        memset(someData[i], 0, xDim * yDim * sizeof (double));
}

void ReconData::reset_objectEst() {
    int i = 0;
    for (i = 0; i < zDim; i++) {
        memset(objectEst[i], 0, xDim * yDim * sizeof (double));
    }
}

void ReconData::reset_objectUpdate() {
    int i = 0;
    for (i = 0; i < zDim; i++) {
        memset(objectUpdate[i], 0, xDim * yDim * sizeof (double));
    }
}

void ReconData::reset_brotData() {
    int i = 0;
    for (i = 0; i < zDim; i++)
        memset(brotData[i], 0, xDim * yDim * sizeof (double));
}

void ReconData::reset_rotData() {
    int i = 0;
    for (i = 0; i < zDim; i++)
        memset(rotData[i], 0, xDim * yDim * sizeof (double));
}

void ReconData::reset_rotAttnData() {
    int i = 0;
    for (i = 0; i < zDim; i++)
        memset(rotAttnData[i], 0, xDim * yDim * sizeof (double));
}
```

```
void ReconData::reset_vectors() {
    for (int buf = 0; buf < 2; buf++)
        for (int i = 0; i < zDim; i++) {
            memset(qVector[buf][i], 0, xDim * yDim * sizeof (double));
            memset(gradUbarVector[buf][i], 0, xDim * yDim * sizeof
(double));
        }
}

/* Operations */

// if less than 0 just set to 0
void ReconData::enforcePositivity() {
    int i = 0;
    int j = 0;
    for (i = 0; i < zDim; i++)
        for (j = 0; j < xDim * yDim; j++)
            if (objectEst[i][j] < 0)
                objectEst[i][j] = 0;
}

void ReconData::enforcePositivity(double** data) {
    int i = 0;
    int j = 0;
    for (i = 0; i < zDim; i++)
        for (j = 0; j < xDim * yDim; j++)
            if (data[i][j] < 0)
                data[i][j] = 0;
}

// multiplies data with cylinder that is the size of FOV (se
initrecondata())
void ReconData::enforceFOV() {

    for (int z = 0; z < zDim; z++) {
        for (int y = 0; y < yDim; y++) {
            for (int x = 0; x < xDim; x++) {
                objectEst[z][y * xDim + x] = objectEst[z][y * xDim + x]
* data[z][y * xDim + x];
            }
        }
    }
}

void ReconData::enforceFOV(double** theData) {

    for (int z = 0; z < zDim; z++) {
        for (int y = 0; y < yDim; y++) {
            for (int x = 0; x < xDim; x++) {
                theData[z][y * xDim + x] = theData[z][y * xDim + x] *
data[z][y * xDim + x];
            }
        }
    }
```

```
}

void ReconData::timesview(int ang) {
    int i = 0;
    int j = 0;
    for (i = 0; i < zDim; i++)
        for (j = 0; j < xDim * yDim; j++)
            data[i][j] = data[i][j] * ang;
}

void ReconData::calculateGradient(double** of, double*** destination) {
    // Do x direction first
    for (int z = 0; z < zDim; z++) {
        for (int y = 0; y < yDim; y++) {
            for (int x = 1; x < xDim; x++) {
                destination[0][z][y * xDim + (x - 1)] = of[z][y * xDim
+ x] - of[z][y * xDim + (x - 1)];
            }
            destination[0][z][y * xDim + (xDim - 1)] = -of[z][y * xDim
+ (xDim - 1)];
        }
    }
    enforceFOV(destination[0]);
    // Now y
    for (int z = 0; z < zDim; z++) {
        for (int x = 0; x < xDim; x++) {
            for (int y = 1; y < yDim; y++) {
                destination[1][z][(y - 1) * xDim + x] = of[z][(y *
xDim) + x] - of[z][((y - 1) * xDim) + x];
            }
            destination[1][z][(yDim - 1) * xDim + x] = -of[z][(yDim -
1) * xDim + x];
        }
    }
    enforceFOV(destination[1]);
    //    // Now z //hypothetically
    //    for (int z = 1; z < zDim; z++) {
    //        for (int y = 0; y < yDim; y++) {
    //            for (int x = 0; x < xDim; x++) {
    //                destination[2][z][y * xDim + x] = of[z][(y *
xDim) + x] - of[z - 1][(y * xDim) + x];
    //            }
    //        }
    //    }
}

void ReconData::calculateMagnitudeImage(double*** from, double** to) {
    for (int z = 0; z < zDim; z++) {
        for (int y = 0; y < yDim; y++) {
            for (int x = 0; x < xDim; x++) {
                to[z][y * xDim + x] = (from[0][z][y * xDim + x] *
from[0][z][y * xDim + x])
                        + (from[1][z][y * xDim + x] * from[1][z][y *
xDim + x]); // +
```

```
                //from[2][z][y * xDim + x] * from[2][z][y * xDim + x]);
                to[z][y * xDim + x] = sqrt(to[z][y * xDim + x]);
            }
        }
    }
}

// LB is lower bound of the magnitude image
void ReconData::calculateMagnitudeImageLB(double LB, double*** from,
double** to) {
    reset_someData(to);

    calculateMagnitudeImage(from, to);

    saveSomeData(to, "magImagePre.recon");
    for (int z = 0; z < zDim; z++)
        for (int y = 0; y < yDim; y++)
            for (int x = 0; x < xDim; x++)
                if (to[z][y * xDim + x] < LB) {
                    if (to[z][y * xDim + x] < 0)
                        cout << "here" << endl;
                    to[z][y * xDim + x] = LB;
                } else
                    to[z][y * xDim + x] = to[z][y * xDim + x];

    saveSomeData(to, "magImagePost.recon");
}

// calculates -divq from qvector
void ReconData::calculateNegDivergence(double*** from, double** to) {
    reset_someData(to);

    /**qvector is from, divQ is to*/
    for (int z = 0; z < zDim; z++) {
        for (int y = 1; y < yDim; y++) {
            for (int x = 1; x < xDim; x++) {
                divQ[z][y * xDim + x] = qVector[0][z][y * xDim + (x -
1)] + qVector[1][z][(y - 1) * xDim + x]
                        - qVector[0][z][y * xDim + x] - qVector[1][z][y
* xDim + x];
                divQ[z][0 * xDim + x] = qVector[0][z][0 * xDim + (x -
1)] + 0
                        - qVector[0][z][0 * xDim + x] - qVector[1][z][0
* xDim + x];
            }
            divQ[z][y * xDim + 0] = 0 + qVector[1][z][(y - 1) * xDim +
0]
                    - qVector[0][z][y * xDim + 0] - qVector[1][z][y *
xDim + 0];
        }
        divQ[z][0] = -qVector[0][z][0 * xDim + 0] - qVector[1][z][0 *
xDim + 0];
    }
```

```
    enforceFOV(divQ);

}

/*Rotation Ops*/
bool ReconData::rotateZ(double **from, double **to, double theta) {
    // loop variables
    int k = 0;
    int j = 0;
    int i = 0;

    // some temporary points
    Point3D doPointZ(0.0, 0.0, 0.0);
    Point3D doPointZY(0.0, 0.0, 0.0);
    Point3D doPointZYX(0.0, 0.0, 0.0);

    Point3D startPointRot = startPoint;
    /** The rotated increment to the next point in that row (X)  */
    Point3D incrementXRot = incrementX;
    /** The rotated increment to the next point in that column (Y)  */
    Point3D incrementYRot = incrementY;

    // calculate the rotated points
    incrementXRot.RotateZ(theta);
    incrementYRot.RotateZ(theta);
    startPointRot.RotateZ(theta);

    doPointZ = startPointRot;
    // loop through all z object elements
    for (i = 0; i < zDim; i++) {
        doPointZY = doPointZ;
        // loop through all y object elements
        for (j = 0; j < yDim; j++) {
            doPointZYX = doPointZY;
            // loop through all x object elements
            for (k = 0; k < xDim; k++) {
                to[i][k + j * xDim] +=
get_LinearInterpolVal(doPointZYX, from);
                doPointZYX += incrementXRot;
            } // end xDim
            doPointZY += incrementYRot;
        } // end yDim
        doPointZ += incrementZ;
    } // end zDim
    return true;
}

bool ReconData::rotateZfp(double theta) {

    reset_rotData();
    rotateZ(objectEst, rotData, theta);
    saveRotData("forwardrot.recon");

    return true;
```

```
}

bool ReconData::rotateZfp(double theta, double **from) {

    reset_rotData();
    rotateZ(from, rotData, theta);
    saveRotData("forwardrot.recon");

    return true;
}

bool ReconData::rotateZattn(double theta) {
    if (withAttn) {
        reset_rotAttnData();
        rotateZ(attnData, rotAttnData, theta);
    }
    return true;
}

bool ReconData::rotateZbsf(double theta) {
    reset_rotData();
    rotateZ(bsfData, rotData, theta);
    return true;
}

bool ReconData::rotateZbp(double theta) {
    reset_brotData();
    //        rotateZ(rotData, objectEst, theta);
    rotateZ(rotData, brotData, theta);
    scallb();
    return true;
}

bool ReconData::rotateZbp(double theta, double** to) {
    reset_brotData();
    //        rotateZ(rotData, to, theta);
    rotateZ(rotData, brotData, theta);
    scallb(to);
    return true;
}

void ReconData::scallb() {
    int i = 0;
    int j = 0;
    int k = 0;
    double tbsfvalue = 0.0;

    for (i = 0; i < zDim; i++) {
        for (j = 0; j < yDim; j++) {
            for (k = 0; k < xDim; k++) {
                tbsfvalue = get_tbsf(k, j, i);
                if (tbsfvalue == 0) {
                    objectEst [i][k + j * xDim] += 0.0;
                } else {
```

```
                    objectEst[i][k + j * xDim] += (brotData[i][k + j *
xDim] / tbsfvalue);
                }
            }
        }
    }

}

void ReconData::scallb(double** to) {
    int i = 0;
    int j = 0;
    int k = 0;
    double tbsfvalue = 0.0;

    for (i = 0; i < zDim; i++) {
        for (j = 0; j < yDim; j++) {
            for (k = 0; k < xDim; k++) {
                tbsfvalue = get_tbsf(k, j, i);
                if (tbsfvalue == 0) {
                    to [i][k + j * xDim] += 0.0;
                } else {
                    to[i][k + j * xDim] += (brotData[i][k + j * xDim] /
tbsfvalue);
                }
            }
        }
    }

}

void ReconData::scallb(double** to, double** from) {
    int i = 0;
    int j = 0;
    int k = 0;
    double tbsfvalue = 0.0;

    for (i = 0; i < zDim; i++) {
        for (j = 0; j < yDim; j++) {
            for (k = 0; k < xDim; k++) {
                tbsfvalue = get_tbsf(k, j, i);
                to[i][k + j * xDim] += (from[i][k + j * xDim]); // /
tbsfvalue);
            }
        }
    }

}

/* Copy data*/
void ReconData::copy_rotDataTo(double ** to) {
    int i = 0;
    int j = 0;
    int k = 0;
```

```
    for (i = 0; i < zDim; i++)
        for (j = 0; j < xDim; j++)
            for (k = 0; k < yDim; k++)
                to[i][j + k * xDim] = rotData[i][j + k * xDim];
}

void ReconData::copy_backScalingFactors() {
    int i = 0;
    int j = 0;
    int k = 0;

    for (i = 0; i < zDim; i++)
        for (j = 0; j < xDim; j++)
            for (k = 0; k < yDim; k++)
                bsfData[i][j + k * xDim] = objectEst[i][j + k * xDim];
}

void ReconData::copy_tbackScalingFactors() {
    int i = 0;
    int j = 0;
    int k = 0;

    for (i = 0; i < zDim; i++)
        for (j = 0; j < xDim; j++)
            for (k = 0; k < yDim; k++)
                tbsfData[i][j + k * xDim] = objectEst[i][j + k * xDim];
}

/* File Ops*/

void ReconData::load_objectEst(const string& filename) {
    int i = 0;
    long j = 0;
    float a = 0;


    fstream file;
    file.open(filename.c_str(), fstream::in | fstream::binary);
    if (file.fail()) {
        cerr << "Could not open file " << filename << endl;
    }

    for (i = 0; i < zDim; i++) {
        for (j = 0; j < xDim * yDim; j++) {
            file.read((char*) & a, 4);
            if (a < 0)
                objectEst [i][j] = 0;
            else
                objectEst[i][j] = (double) a;
        }
    }

    file.close();
```

```
}

bool ReconData::saveRotData(const string &filename) {
    int i = 0;
    int j = 0;
    bool success = true;

    fstream file;
    file.open(filename.c_str(), fstream::out | fstream::binary);

    for (i = 0; i < zDim; i++) {
        for (j = 0; j < xDim * yDim; j++) {
            file.write((const char*) & rotData[i][j], sizeof (double));
        }
    }
    file.close();
    return success;
}

void ReconData::save_backScalingFactors(string filename) {
    int i = 0;
    int j = 0;

    fstream file;
    file.open(filename.c_str(), fstream::out | fstream::binary);
    for (i = 0; i < zDim; i++)
        for (j = 0; j < xDim * yDim; j++)
            file.write((const char*) & bsfData[i][j], sizeof (double));
    file.close();

}

void ReconData::saveSomeData(double** theData, const string& filename)
{

    int i = 0;
    int j = 0;

    fstream file;
    file.open(filename.c_str(), fstream::out | fstream::binary);

    for (i = 0; i < zDim; i++)
        for (j = 0; j < xDim * yDim; j++)
            file.write((const char*) & theData[i][j], sizeof (double));

    file.close();
}

void ReconData::saveVector(double*** theVector, const string& filename)
{
    int i = 0;
    int j = 0;

    fstream file;
```

```
    file.open(filename.c_str(), fstream::out | fstream::binary);

    for (int b = 0; b < 2; b++)
        for (i = 0; i < zDim; i++)
            for (j = 0; j < xDim * yDim; j++)
                file.write((const char*) & theVector[b][i][j], sizeof
(double));

    file.close();
}

void ReconData::save_tbackScalingFactors(string filename) {
    int i = 0;
    int j = 0;

    fstream file;
    file.open(filename.c_str(), fstream::out | fstream::binary);

    for (i = 0; i < zDim; i++)
        for (j = 0; j < xDim * yDim; j++)
            file.write((const char*) & tbsfData[i][j], sizeof
(double));
    file.close();
}

// Sets up fftw and makes fft of 2d gaussian
void ReconData::initGaussianBlur(double radius) {
    realData = (long double*) fftwl_malloc(sizeof (long double) * 2 *
xDim * 2 * yDim);
    inData = (fftwl_complex*) fftwl_malloc(sizeof (fftwl_complex) * ((2
* xDim / 2) + 1) *2 * yDim);
    outData = (fftwl_complex*) fftwl_malloc(sizeof (fftwl_complex) *
((2 * xDim / 2) + 1) *2 * yDim);
    filterKernel = (fftwl_complex*) fftwl_malloc(sizeof (fftwl_complex)
* ((2 * xDim / 2) + 1) *2 * yDim);

    memset(realData, 0, 4 * xDim * yDim * sizeof (long double));
    memset(inData, 0, ((2 * xDim / 2) + 1) * 2 * yDim * sizeof
(fftwl_complex));
    memset(outData, 0, ((2 * xDim / 2) + 1) *2 * yDim * sizeof
(fftwl_complex));
    memset(filterKernel, 0, ((2 * xDim / 2) + 1) * 2 * yDim * sizeof
(fftwl_complex));

    pAA = fftwl_plan_dft_r2c_2d(2 * xDim, 2 * yDim, realData, outData,
FFTW_MEASURE);
    pBB = fftwl_plan_dft_c2r_2d(2 * xDim, 2 * yDim, inData, realData,
FFTW_MEASURE);

    filterScaling = 1.0/*21250545*/ / (double) (4 * xDim * yDim);


    createGaussianFilterKernel(radius);
}
```

```
void ReconData::createGaussianFilterKernel(double radius) {
    int i = 0;
    int j = 0;
    int ij = 0;

    double detectBlurGaussValue = 0.0;
    double probGauss = 0.0;
    double probSum = 0.0;
    double ySq = 0.0;
    double xSq = 0.0;

    detectBlurGaussValue = -(1 / (2 * radius * radius));


    for (i = 0; i < 2 * yDim; i++)
        for (j = 0; j < 2 * xDim; j++) {
            ij = j + i * 2 * xDim;

            ySq = ((i - (yDim / 2)) - (yDim / 2))*((i - (yDim / 2)) -
(yDim / 2));
            xSq = ((j - (xDim / 2)) - (xDim / 2))*((j - (xDim / 2)) -
(xDim / 2));

            probGauss = exp(detectBlurGaussValue * (ySq + xSq));
            if (i > yDim / 2 && i < 1.5 * yDim && j > xDim / 2 && j <
1.5 * xDim) {
                realData[ij] = probGauss;
                probSum += probGauss;
            } else
                realData[ij] = 0;
        }

    for (i = 0; i < 2 * yDim; i++)
        for (j = 0; j < 2 * xDim; j++) {
            ij = j + i * 2 * xDim;
            realData[ij] = realData[ij] / probSum;
        }

    cout << probSum << endl;

    fftwl_execute(pAA);

    for (i = 0; i < 2 * yDim; i++)
        for (j = 0; j < ((2 * xDim / 2) + 1); j++) {
            ij = j + i * ((2 * xDim / 2) + 1);
            c_re(filterKernel[ij]) = c_re(outData[ij]);
            c_im(filterKernel[ij]) = c_im(outData[ij]);

            c_re(inData[ij]) = (c_re(outData[ij]));
            c_im(inData[ij]) = c_im(outData[ij]);
        }
```

```
    fftwl_execute(pBB);
}

//fft to and multiply by fft of 2d gaussian, fft back and be happy
void ReconData::applyGaussianBlur(int z, double** to) {
    int i = 0;
    int j = 0;
    int ij = 0;
    int ii = 0;
    int jj = 0;
    int iijj = 0;

    double** tester;
    tester = new double*[zDim];
    for (i = 0; i < zDim; i++)
        tester[i] = new double[2 * xDim * 2 * yDim];
    for (i = 0; i < zDim; i++)
        memset(tester[i], 0, 2 * xDim * 2 * yDim * sizeof (double));

    //set realData to data and zero pad
    for (i = 0; i < 2 * yDim; i++)
        for (j = 0; j < 2 * xDim; j++) {
            ij = j + i * 2 * xDim;
            if (i > yDim / 2 && i < 3 * yDim / 2 && j > xDim / 2 && j <
3 * xDim / 2) {
                realData[ij] = to[z][(j - xDim / 2) + (i - yDim / 2) *
xDim];

            } else
                realData[ij] = 0;
        }

    //forward transform
    fftwl_execute(pAA);

    //multiply filter kernel and complex output data
    for (i = 0; i < 2 * yDim; i++)
        for (j = 0; j < ((2 * xDim / 2) + 1); j++) {
            ij = j + i * ((2 * xDim / 2) + 1);
            c_re(inData[ij]) = (c_re(outData[ij]) *
c_re(filterKernel[ij])
                    - c_im(outData[ij]) * c_im(filterKernel[ij]));
            c_im(inData[ij]) = (c_re(outData[ij]) *
c_im(filterKernel[ij])
                    + c_im(outData[ij]) * c_re(filterKernel[ij]));
        }
    //inverse transform
    fftwl_execute(pBB);

    //set data to interim array before paring off zeros
    for (i = 2 * yDim / 2, ii = 0; i < 2 * yDim; i++, ii++) {
        for (j = 2 * xDim / 2, jj = 0; j < 2 * xDim; j++, jj++) {
            ij = j + i * 2 * xDim;
            iijj = jj + ii * 2 * xDim;
```

```
            tester[0][iijj] = realData[ij] * filterScaling;
            tester[0][ij] = realData[iijj] * filterScaling;

            ij = jj + i * 2 * xDim;
            iijj = j + ii * 2 * xDim;
            tester[0][iijj] = realData[ij] * filterScaling;
            tester[0][ij] = realData[iijj] * filterScaling;
        }
    }

    //get rid of zero pad
    for (i = yDim / 2; i < 1.5 * yDim; i++)
        for (j = xDim / 2; j < 1.5 * xDim; j++)
            to[z][(j - (yDim / 2)) + (i - (xDim / 2)) * xDim] =
tester[0][j + i * 2 * xDim];

    for (i = 0; i < zDim; i++)
        delete[] tester[i];
    delete[] tester;
}
```

recondata.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************************
****
                          recondata.h  -  description
                             -------------------
    begin                : Fri Jan 5 2005
    copyright            : (C) 2005 by Christian Wietholt
    email                : christian.wietholt@marquette.edu

***************************************************************************
****/

/
***************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

***************************************************************************
****/
#ifndef RECONDATA_H
#define RECONDATA_H
```

```
#include "fftw3.h"
#include "point3d.h"

//#include <QObject>

#include <string>
#include <iostream>
using namespace std;

class ProjData;

class ReconData {

public:

    ReconData() {
        rotInitialized = false;
        get_attnDataPtr = &ReconData::get_attnDataNull;
    }

    ~ReconData() {
        delete []data;
        delete []objectEst;
        delete []objectUpdate;
        delete []prevObjectUpdate;
        delete []divQ;
        delete []magImage;
        delete []planeX;
        delete []planeY;
        delete []planeZ;
        delete []voxelPosX;
        delete []voxelPosY;
        delete []voxelPosZ;

        delete []qVector;
        delete []gradUbarVector;

        if (withAttn) {
            delete []attnData;
            delete []rotAttnData;
        }

        if (rotInitialized)
            delete []rotData;
    };

    /**
     * Accessor Methods
     */
    double get_data(int z, int y, int x, double** data) {
        return data[z][x + (y * xDim)];
    };
```

```
double get_objectEst(int z, int y, int x) {
    return objectEst[z][x + (y * xDim)];
};

double** get_objectEst() {
    return objectEst;
};

double get_objectUpdate(int z, int y, int x) {
    return objectUpdate[z][x + (y * xDim)];
};

double** get_objectUpdate() {
    return objectUpdate;
};

double get_prevObjectUpdate(int z, int y, int x) {
    return prevObjectUpdate[z][x + (y * xDim)];
};

double*** get_qVector() {
    return qVector;
};

double get_qVector(int buf, int z, int y, int x) {
    return qVector[buf][z][x + (y * xDim)];
};

double*** get_gradUbarVector() {
    return gradUbarVector;
};

double get_gradUbarVector(int buf, int z, int y, int x) {
    return gradUbarVector[buf][z][x + (y * xDim)];
};

double** get_divQ() {
    return divQ;
}

double get_divQ(int z, int y, int x) {
    return divQ[z][x + (y * xDim)];
};

double** get_magImage() {
    return magImage;
};

double get_magImage(int z, int y, int x) {
    return magImage[z][x + (y * xDim)];
};

double** get_prevObjectUpdate() {
    return prevObjectUpdate;
```

```
};

unsigned short get_xDim() {
    return xDim;
};

unsigned short get_yDim() {
    return yDim;
};

unsigned short get_zDim() {
    return zDim;
};

double get_xPixelSize() {
    return xPixelSize;
};

double get_yPixelSize() {
    return yPixelSize;
};

double get_zPixelSize() {
    return zPixelSize;
};

double get_xLength() {
    return xLength;
};

double get_yLength() {
    return yLength;
};

double get_zLength() {
    return zLength;
};

double get_planeX(int i) {
    return planeX[i];
};

double get_planeY(int i) {
    return planeY[i];
};

double get_planeZ(int i) {
    return planeZ[i];
};

double get_voxelPosX(int i) {
    return voxelPosX[i];
};
```

```
    double get_voxelPosY(int i) {
        return voxelPosY[i];
    };

    double get_voxelPosZ(int i) {
        return voxelPosZ[i];
    };

    Point3D get_startPoint() {
        return startPoint;
    };

    Point3D get_incrementX() {
        return incrementX;
    };

    Point3D get_incrementY() {
        return incrementY;
    };

    Point3D get_incrementZ() {
        return incrementZ;
    };

    bool get_withAttn() {
        return withAttn;
    };

    double get_rotData(int i, int j, int k) {
        return rotData[i][k + (j * xDim)];
    };

    double get_bsfData(int i, int j, int k) {
        return bsfData[i][k + (j * xDim)];
    };

    double get_fsfData(int i, int j, int k) {
        return fsfData[i][k + (j * xDim)];
    };

    /**
     * Operations
     */
    /** This function allocates memory to store the generated density
distribution and
            initializes it with the background intensity */
    bool init(int x, int y, int z);
    bool initRotation();
    bool initVectors();
    bool rotateZbp(double theta);
    bool rotateZbp(double theta, double** to);
    bool rotateZfp(double theta);
    bool rotateZfp(double theta, double** from);
    bool rotateZattn(double theta);
```

```
bool rotateZbsf(double theta);
double get_LinearInterpolVal(Point3D p, double **d);
void init_recondata();
void setFirstObjectEst();
void timesview(int ang);
bool saveRotData(const string &filename);

void setProjPointer(ProjData *p);

/** sets the length of the reconstruction space.  */
void setLength(double l);

void reset_objectEst();
void reset_objectUpdate();
void reset_vectors();

void reset_brotData();
void reset_rotData();
void reset_rotAttnData();

void reset_data();
void reset_someData(double** someData);

void scallb();
void scallb(double**to);
void scallb(double**to, double**from);

void copy_rotDataTo(double ** to);

void enforceFOV();
void enforceFOV(double** data);

void set_objectEst(double val, int z, int y, int x) {
    objectEst[z][x + (y * xDim)] = val;
};

void set_objectEst(double val, int z, int i) {
    objectEst[z][i] = val;
};

void set_qVector(double val, int buf, int z, int y, int x) {
    qVector[buf][z][x + (y * xDim)] = val;
};

void set_gradUbarVector(double val, int buf, int z, int y, int x) {
    gradUbarVector[buf][z][x + (y * xDim)] = val;
};

void set_objectUpdate(double val, int z, int y, int x) {
    objectUpdate[z][x + (y * xDim)] = val;
};

void set_prevObjectUpdate() {
    for (int z = 0; z < zDim; z++)
```

```
            for (int xy = 0; xy < xDim * yDim; xy++)
                prevObjectUpdate[z][xy] = objectUpdate[z][xy];
    }

    void load_objectEst(const string & filename);

    void saveSomeData(double ** theData, const string &filename);
    void saveVector(double *** theVector, const string &filename);

    void set_rotData(int i, int j, int k, double inten) {
        rotData[i][k + (j * xDim)] += inten;
    };

    double (ReconData::*get_attnDataPtr)(int, int, int);

    double get_attnDataLoaded(int i, int j, int k) {
        return rotAttnData[i][k + (j * xDim)];
    }

    double get_attnDataNull(int i, int j, int k) {
        i = j = k = 0;
        return 0;
    };

    double get_tbsf(int x, int y, int z) {
        return tbsfData[z][x + y * xDim];
    }
    void updateMultiply();
    void updateAdd();

    void add_data(int z, int y, int x, double inten, double **data) {
        data[z][x + (y * xDim)] += inten;
    };

    void enforcePositivity();
    void enforcePositivity(double** data);

    void calculateGradient(double** of, double*** destination);
    void calculateMagnitudeImage(double*** from, double** to);
    void calculateMagnitudeImageLB(double LB, double*** from, double**
to);
    void calculateNegDivergence(double*** from, double** to);

    double get_VoxelVolume() {
        return voxelVolume;
    };

    void init_backScalingFactors();
    void copy_backScalingFactors();
    void copy_tbackScalingFactors();
    void save_backScalingFactors(string filename);
    void save_tbackScalingFactors(string filename);
```

```
    void initGaussianBlur(double radius);
    void applyGaussianBlur(int z, double ** to);
    void createGaussianFilterKernel(double radius);
    /**
     * Protected stuff
     */
protected:

private:
    /**
     * Fields
     */

    bool withAttn;
    //indicates wether rotation framework was initialized
    bool rotInitialized;
    /** Dimension in x direction */
    unsigned short xDim;
    /** Dimension in y direction */
    unsigned short yDim;
    /** Dimension in z direction */
    unsigned short zDim;

    /** This is a pointer to the generated density distribution. */
    double **data;
    double **objectUpdate;
    double **prevObjectUpdate;
    double **objectEst;
    double **divQ;
    double **magImage;
    double ***qVector;
    double ***gradUbarVector;

    /** This is a pointer to the rotated density distribution */
    double **rotData;
    double **brotData;
    /** This is a pointer to one eighth of the scaling Factors for
     * used for the ray-driven back-projection
     */
    double **bsfData;
    double **tbsfData;
    double **fsfData;

    double **attnData;
    double **rotAttnData;
    /** the physical length of the distribution */
    double xLength;
    double yLength;
    double zLength;
    /** The size of one pixel */
    double xPixelSize;
    double yPixelSize;
```

```
double zPixelSize;

double voxelVolume;

double *planeX;
double *planeY;
double *planeZ;

double *voxelPosX;
double *voxelPosY;
double *voxelPosZ;

/** first slice lower left corner */
Point3D startPoint;
/** initial increment, to the next point in that row (X) */
Point3D incrementX;
/** initial increment, to the next point in that column (Y) */
Point3D incrementY;
/** increment, to the next slice (Z) */
Point3D incrementZ;

ProjData *proj;

double filterScaling;
long double *realData;

fftwl_complex *inData;
fftwl_complex *outData;
fftwl_complex *filterKernel;

fftwl_plan pAA;
fftwl_plan pBB;

template<typename T> T MIN(const T &x, const T &y) {
    return ( (x) < (y) ? (x) : (y));
}

template<typename T> T MAX(const T &x, const T &y) {
    return ( (x) > (y) ? (x) : (y));
}

/**
 *
 */
/**
 * Constructors
 */
/**
 * Accessor Methods
 */
/**
 * Operations
 */
bool rotateZ(double **from, double **to, double theta);
```

```
};
#endif //RECONDATA_H
```

smpinholerd.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
*************************************************************************
****
                          smpinholerd.cpp  -  description
                             -------------------
    begin                : Wed Apr 05 08:49:48 CST 2006
    copyright            : (C) 2006 by Christian Wietholt
    email                : cwietholt@nhri.org.tw

*************************************************************************
****/

/
*************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

*************************************************************************
****/
#include "smpinholerd.h"

#include "projdata.h"
#include "recondata.h"
#include "point3d.h"
#include "raytbl3d.h"
#include "raytbl3dinc.h"
#include <iostream>
#include <math.h>
#include <time.h>
#include <map>
using namespace std;

/**
 * Constructors/Destructors
 */
SMPinholeRD::SMPinholeRD() {
}
```

```
SMPinholeRD::SMPinholeRD(ProjData *p, ReconData *r, int symR) :
SystemMatrix(p, r, symR) {
    int i = 0;
    int pnum = proj->get_pinholenum();
    alphaXSize = xDim * 10;
    alphaYSize = yDim * 10;
    alphaZSize = zDim * 10;
    alphaAllSize = alphaXSize + alphaYSize + alphaZSize;

    alphaX = new double[alphaXSize];
    alphaY = new double[alphaYSize];
    alphaZ = new double[alphaZSize];
    alphaAll = new double[alphaAllSize];
    testlength = new double[alphaAllSize];
    xposs = new int[alphaXSize];
    zposs = new int[alphaZSize];
    yposs = new int[alphaYSize];


    resetalpha();
    tmpRay = new RayTBL3D(xDim, 0);
    rayInc = new RayTBL3DInc *[pnum];
    for (i = 0; i < pnum; i++) {
        rayInc[i] = new RayTBL3DInc[ proj->get_zDetectDim() /
symetryRatio
                * proj->get_yDetectDim() / symetryRatio
                * proj->get_pinHoleMatrix()
                * proj->get_pinHoleMatrix() ];
        rayInc[i]->clear_all();
    }
}

SMPinholeRD::~SMPinholeRD() {
    delete []alphaX;
    delete []alphaY;
    delete []alphaZ;
    delete []alphaAll;
}

/**
 * Methods
 */



void SMPinholeRD::buildMatrix() {
    time_t start, end;
    time(&start);
    raytrace();
    time(&end);
    cerr << "Time elapsed: " << difftime(end, start) << "s" << endl;

}
```

```
void SMPinholeRD::resetalpha() {
    int i = 0;

    for (i = 0; i < alphaXSize; i++) {
        alphaX[i] = 0.0;
        xposs[i] = 0;
    };
    for (i = 0; i < alphaYSize; i++) {
        alphaY[i] = 0.0;
        yposs[i] = 0;
    };
    for (i = 0; i < alphaZSize; i++) {
        alphaZ[i] = 0.0;
        zposs[i] = 0;
    };
    for (i = 0; i < alphaAllSize; i++) {
        alphaAll[i] = 0.0;
        testlength[i] = 0.0;
    };

}

void SMPinholeRD::buildMatrix(double theta) {
    time_t start, end;
    time(&start);
    raytrace(theta);
    time(&end);
}

/** This function finds all the voxels that are intersected
    by rays from the detector and computes their contribution
    to the detector pixel. */
//Siddon's raytrace
void SMPinholeRD::raytrace(double theta) {
    // some loop variables
    int g = 0;
    int h = 0;
    int i = 0;
    int j = 0;
    int k = 0;
    int l = 0;
    int m = 0;
    int o = 0;
    int xo = 0;
    int yo = 0;
    int zo = 0;
    int q = 0;
    int r = 0;
    int p = 0;
    long s = 0;

    // the minimum and maximum planes in x, y, and z direction
    int kMin = 0;
    int kMax = 0;
```

```
int lMin = 0;
int lMax = 0;
int mMin = 0;
int mMax = 0;
// some voxel positions
int xPos = 0;
int yPos = 0;
int zPos = 0;
// the greatest alpha position possible
int maxAlpha = 0;

int xi = 0;
int yi = 0;
int zi = 0;
// exact position in the object space
double x = 0.0;
double y = 0.0;
double z = 0.0;

// the maximum x, y, z dimension of the object
double maxObject = 0.0;
// distance between the pinhole and the voxel
double xDiff = 0.0;
double yDiff = 0.0;
double zDiff = 0.0;
// length of ray inside a voxel
double length = 0.0;
// total raylength from pinhole to voxel
double rayLength = 0.0;
// ray intersection of a voxel in x, y, z direction
double alphaXPixelSize = 0.0;
double alphaYPixelSize = 0.0;
double alphaZPixelSize = 0.0;
// first/last x, y, z ray-object intersection as an alpha value
double alphaXStart = 0.0;
double alphaYStart = 0.0;
double alphaZStart = 0.0;
double alphaXEnd = 0.0;
double alphaYEnd = 0.0;
double alphaZEnd = 0.0;
// first/last total ray-object intersection as an alpha value
double alphaXMin = 0.0;
double alphaYMin = 0.0;
double alphaZMin = 0.0;

double alphaXMax = 0.0;
double alphaYMax = 0.0;
double alphaZMax = 0.0;
// midpoint between two plane intersects
double alphaMid = 0.0;
double alphaMin = 0.0;
double alphaMax = 0.0;

double alphaMinFudge = 0.0;
```

```
    double alphaMaxFudge = 0.0;

    double distance = 0.0;
    double xLengthHalf = 0.0;

    double inten1 = 0;

    // some temporary points
    Point3D doPointZ(0.0, 0.0, 0.0);
    Point3D doPointZY(0.0, 0.0, 0.0);
    Point3D doPinHolePointZ(0.0, 0.0, 0.0);
    Point3D doPinHolePointZY(0.0, 0.0, 0.0);

    Point3D startPointRot = proj->get_startDetectPoint();
    Point3D startPinholePointRot;
    /** The rotated increment to the next point in that row (y)  */
    Point3D incrementYRot = proj-> get_incrementDetectY();
    /** The rotated increment to the next point in that column (z)  */
    Point3D incrementZRot = proj->get_incrementDetectZ();

    Point3D incrementPinholeZRot = proj->get_incrementPinHoleZ();
    Point3D incrementPinholeYRot = proj->get_incrementPinHoleY();

        // calculate the rotated points
    incrementZRot.RotateZ(-theta);
    incrementYRot.RotateZ(-theta);
    incrementPinholeYRot.RotateZ(-theta);
    incrementPinholeZRot.RotateZ(-theta);
    startPointRot.RotateZ(-theta);


    xLengthHalf = recon->get_xLength() / 2.0;
    // compute maximum object length
    maxObject = MAX(maxObject, recon->get_xLength());
    maxObject = MAX(maxObject, recon->get_yLength());
    maxObject = MAX(maxObject, recon->get_zLength());
    // using that compute the maximum alpha value, (more of a
gestimate)
    maxAlpha = (int) ((double) 2.0
            + ((proj->get_p2oLength() + maxObject / 2.0)
            / proj->get_p2dLength()));

    for (p = 0; p < proj->get_pinholenum(); p++) {
        startPinholePointRot = proj->get_startPinHolePoint(p);
        startPinholePointRot.RotateZ(-theta);

        doPointZ = startPointRot;
        // loop through all z detector elements
        for (i = 0; i < proj->get_zDetectDim() / symetryRatio; i++) {
            doPointZY = doPointZ;
            // loop through all y detector elements
            for (j = 0; j < proj->get_yDetectDim() / symetryRatio; j++)
{
                doPinHolePointZ = startPinholePointRot;
```

```
                // loop through all y pinhole matrix points
                for (g = 0; g < proj->get_pinHoleMatrix(); g++) {
                    doPinHolePointZY = doPinHolePointZ;
                    // loop through all x pinhole matrix points
                    for (h = 0; h < proj->get_pinHoleMatrix(); h++) {
                        // if the pinhole is not on the same plane as
the detector element
                        if ((xDiff = (doPinHolePointZY.x -
doPointZY.x)) != 0) //(1)63.5
                        {
                            // compute the first and last ray intersect
with an x-plane
                            alphaXStart = (recon->get_planeX(0) -
doPointZY.x) / xDiff; //(1)0.9956
                            alphaXEnd = (recon->get_planeX(xDim) -
doPointZY.x) / xDiff; //(1)2.106
                            alphaXMin = MIN(alphaXStart, alphaXEnd);
                            alphaXMax = MAX(alphaXStart, alphaXEnd);
                        } else {
                            // otherwise set standard values
                            alphaXMin = 1.0;
                            alphaXMax = maxAlpha;
                        }

                        // if the pinhole is not on the same plane as
the detector element
                        if ((yDiff = doPinHolePointZY.y -
doPointZY.y) != 0) //(1)73.5
                        {
                            // compute the first and last ray intersect
with an y-plane
                            alphaYStart = (recon->get_planeY(0) -
doPointZY.y) / yDiff; //(1)0.38
                            alphaYEnd = (recon->get_planeY(yDim) -
doPointZY.y) / yDiff; //(1)1.34
                            alphaYMin = MIN(alphaYStart, alphaYEnd);
                            alphaYMax = MAX(alphaYStart, alphaYEnd);
                        } else {
                            // otherwise set standard values
                            alphaYMin = 1.0;
                            alphaYMax = maxAlpha;
                        }
                        // if the pinhole is not on the same plane as
the detector element
                        if ((zDiff = doPinHolePointZY.z -
doPointZY.z) != 0) //(1)63.5
                        {
                            // compute the first and last ray intersect
with an z-plane
                            alphaZStart = (recon->get_planeZ(0) -
doPointZY.z) / zDiff; //(1)0.44
                            alphaZEnd = (recon->get_planeZ(zDim) -
doPointZY.z) / zDiff; //(1)1.55
                            alphaZMin = MIN(alphaZStart, alphaZEnd);
```

```
                            alphaZMax = MAX(alphaZStart, alphaZEnd);
                        } else {
                            // otherwise set standard values
                            alphaZMin = 1.0;
                            alphaZMax = maxAlpha;
                        }
                        // compute the very first intersection with an
object plane
                        alphaMin = 1.0;
                        alphaMin = MAX(alphaMin, alphaXMin);
                        alphaMin = MAX(alphaMin, alphaYMin);
                        alphaMin = MAX(alphaMin, alphaZMin); //1

                        // compute the very last intersection with an
object plane
                        alphaMax = maxAlpha;
                        alphaMax = MIN(alphaMax, alphaXMax);
                        alphaMax = MIN(alphaMax, alphaYMax);
                        alphaMax = MIN(alphaMax, alphaZMax); //1.34

                        if (alphaMin < alphaMax) {
                            // compute the total ray length
                            rayLength = sqrt(xDiff * xDiff + yDiff *
yDiff + zDiff * zDiff); //116.04

                            // compute the intersect distance with a
voxel in x, y, z direction
                            if (xDiff == 0)
                                alphaXPixelSize = 0.0;
                            else {
                                alphaXPixelSize = recon-
>get_xPixelSize() / xDiff;
                            }
                            if (yDiff == 0)
                                alphaYPixelSize = 0.0;
                            else {
                                alphaYPixelSize = recon-
>get_yPixelSize() / yDiff;
                            }
                            if (zDiff == 0)
                                alphaZPixelSize = 0.0;
                            else {
                                alphaZPixelSize = recon-
>get_zPixelSize() / zDiff;
                            }

                            // make sure to get the correct plane after
conversion to x, y, z positions
                            alphaXMin = alphaMin + alphaXPixelSize /
1000000;
                            alphaXMax = alphaMax + alphaXPixelSize /
1000000;
                            alphaYMin = alphaMin + alphaYPixelSize /
1000000;
```

```
                            alphaYMax = alphaMax + alphaYPixelSize /
1000000;
                            alphaZMin = alphaMin + alphaZPixelSize /
1000000;
                            alphaZMax = alphaMax + alphaZPixelSize /
1000000;

                            if (xDiff > 0) {
                                // compute minimum and maximum array
positions

                                kMin = (int) ((double) xDim
                                        - (recon->get_planeX(xDim) -
alphaXMin * xDiff - doPointZY.x)
                                        / recon->get_xPixelSize());


                                kMax = (int) ((doPointZY.x + alphaXMax
* xDiff - recon->get_planeX(0))
                                        / recon->get_xPixelSize());

                                // get all other array positions in
between
                                xo = 0;
                                alphaX[xo] = (recon->get_planeX(kMin) -
doPointZY.x) / xDiff;
                                while (alphaX[xo] < alphaMin) {
                                    kMin = kMin + 1;
                                    alphaX[xo] = alphaX[xo] +
alphaXPixelSize;
                                }
                                xo++;
                                for (k = kMin + 1; k <= kMax; k++, xo+
+)
                                    alphaX[xo] = alphaX[xo - 1] +
alphaXPixelSize;
                                alphaX[xo] = alphaMax + 10;
                            } else if (xDiff < 0) {
                                // compute minimum and maximum array
positions
                                kMin = (int) ((double) xDim
                                        - (recon->get_planeX(xDim) -
alphaXMax * xDiff - doPointZY.x)
                                        / recon->get_xPixelSize());

                                kMax = (int) ((doPointZY.x + alphaXMin
* xDiff - recon->get_planeX(0))
                                        / recon->get_xPixelSize());

                                // get all other array positions in
between
                                xo = 0;
                                alphaX[xo] = (recon->get_planeX(kMax) -
doPointZY.x) / xDiff;
```

```
                                while (alphaX[xo] < alphaMin) {
                                    kMax = kMax - 1;
                                    alphaX[xo] = alphaX[xo] -
alphaXPixelSize;
                                }
                                xo++;
                                for (k = kMin; k < kMax; k++, xo++)
                                    alphaX[xo] = alphaX[xo - 1] -
alphaXPixelSize;
                                alphaX[xo] = alphaMax + 10;
                            } else if (xDiff == 0) {
                                xo = 1;
                                alphaX[0] = alphaMin;
                                alphaX[xo] = alphaMax + 10;
                            }

                            if (yDiff > 0) {
                                // compute minimum and maximum array
positions
                                    lMin = (int) ((double) yDim
                                        - (recon->get_planeY(yDim) -
alphaYMin * yDiff - doPointZY.y)
                                        / recon->get_yPixelSize());

                                lMax = (int) ((doPointZY.y + alphaYMax
* yDiff - recon->get_planeY(0))
                                        / recon->get_yPixelSize());

                                // get all other array positions in
between
                                yo = 0;
                                alphaY[yo] = (recon->get_planeY(lMin) -
doPointZY.y) / yDiff;
                                while (alphaY[yo] < alphaMin) {
                                    lMin = lMin + 1;
                                    alphaY[yo] = alphaY[yo] +
alphaYPixelSize;
                                }
                                yo++;
                                for (l = lMin + 1; l <= lMax; l++, yo+
+)
                                    alphaY[yo] = alphaY[yo - 1] +
alphaYPixelSize;
                                alphaY[yo] = alphaMax + 10;
                            } else if (yDiff < 0) {
                                // compute minimum and maximum array
positions
                                lMin = (int) ((double) yDim
                                        - (recon->get_planeY(yDim) -
alphaYMax * yDiff - doPointZY.y)
                                        / recon->get_yPixelSize());
                                    lMax = (int) ((doPointZY.y +
alphaYMin * yDiff - recon->get_planeY(0))
                                        / recon->get_yPixelSize());
```

```
                                // get all other array positions in
between
                                yo = 0;
                                alphaY[yo] = (recon->get_planeY(lMax) -
doPointZY.y) / yDiff;
                                while (alphaY[yo] < alphaMin) {
                                    lMax = lMax - 1;
                                    alphaY[yo] = alphaY[yo] -
alphaYPixelSize;
                                }
                                yo++;
                                for (l = lMin; l < lMax; l++, yo++)
                                    alphaY[yo] = alphaY[yo - 1] -
alphaYPixelSize;
                                alphaY[yo] = alphaMax + 10;
                            } else if (yDiff == 0) {
                                yo = 1;
                                alphaY[0] = alphaMin;
                                alphaY[yo] = alphaMax + 10;
                            }


                            if (zDiff > 0) {
                                // compute minimum and maximum array
positions
                                mMin = (int) ((double) zDim
                                        - (recon->get_planeZ(zDim) -
alphaZMin * zDiff - doPointZY.z)
                                        / recon->get_zPixelSize());
                                mMax = (int) ((doPointZY.z + alphaZMax
* zDiff - recon->get_planeZ(0))
                                        / recon->get_zPixelSize());

                                // get all other array positions in
between
                                zo = 0;
                                alphaZ[zo] = (recon->get_planeZ(mMin) -
doPointZY.z) / zDiff;
                                while (alphaZ[zo] < alphaMin) {
                                    mMin = mMin + 1;
                                    alphaZ[zo] = alphaZ[zo] +
alphaZPixelSize;
                                }
                                zo++;
                                for (m = mMin + 1; m <= mMax; m++, zo+
+)
                                    alphaZ[zo] = alphaZ[zo - 1] +
alphaZPixelSize;
                                alphaZ[zo] = alphaMax + 10;
                            } else if (zDiff < 0) {
                                // compute minimum and maximum array
positions
                                mMin = (int) ((double) zDim
```

```
                                        - (recon->get_planeZ(zDim) -
alphaZMax * zDiff - doPointZY.z)
                                        / recon->get_zPixelSize());
                                mMax = (int) ((doPointZY.z + alphaZMin
* zDiff - recon->get_planeZ(0))
                                        / recon->get_zPixelSize());

                                // get all other array positions in
between
                                zo = 0;
                                alphaZ[zo] = (recon->get_planeZ(mMax) -
doPointZY.z) / zDiff;
                                while (alphaZ[zo] < alphaMin) {
                                    mMax = mMax - 1;
                                    alphaZ[zo] = alphaZ[zo] -
alphaZPixelSize;
                                }
                                zo++;
                                for (m = mMin; m < mMax; m++, zo++)
                                    alphaZ[zo] = alphaZ[zo - 1] -
alphaZPixelSize;
                                alphaZ[zo] = alphaMax + 10;
                            } else if (zDiff == 0) {
                                zo = 1;
                                alphaZ[0] = alphaMin;
                                alphaZ[zo] = alphaMax + 10;
                            }

                            // there might be some round-off errors, so
be not so rigid
                            alphaMinFudge = alphaMin + 0.000001;
                            alphaMaxFudge = alphaMax - 0.000001;
                            k = kMin;
                            l = lMin;
                            m = mMin;
                            q = 0;

                            // merge the three arrays of alpha values
into one sorted array
                            xi = 0;
                            yi = 0;
                            zi = 0;

                            alphaAll[q] = alphaMin;

                            while (alphaX[xi] <= alphaMin)
                                xi++; //alphaX[xi]>1
                            while (alphaY[yi] <= alphaMin)
                                yi++; //alphaY[yi]>1
                            while (alphaZ[zi] <= alphaMin)
                                zi++; //alphaZ[zi]>1

                            while (xi <= xo && yi <= yo && zi <= zo) {
```

```
                                if (alphaX[xi] <= alphaY[yi] /*&&
alphaX[xi] <= alphaZ[zi]*/
                                        && alphaX[xi] < alphaMaxFudge
&& alphaX[xi] > alphaMinFudge) {
                                    q++;
                                    alphaAll[q] = alphaX[xi];
                                    xi++;
                                } else if (alphaY[yi] <= alphaX[xi]
/*&& alphaY[yi] <= alphaZ[zi]*/
                                        && alphaY[yi] < alphaMaxFudge
&& alphaY[yi] > alphaMinFudge) {
                                    q++;
                                    alphaAll[q] = alphaY[yi];
                                    yi++;
                                } else if (alphaZ[zi] <= alphaX[xi] &&
alphaZ[zi] <= alphaY[yi]
                                        && alphaZ[zi] < alphaMaxFudge
&& alphaZ[zi] > alphaMinFudge) {
                                    q++;
                                    alphaAll[q] = alphaZ[zi];
                                    zi++;
                                } else {
                                    xi = xo + 1;
                                    yi = yo + 1;
                                    zi = zo + 1;
                                }
                            }
                            // finish array with the last intersect
                            //alphaAll[q++] = alphaMax;

                            // calculate the points
                            for (o = 1, r = 0; o <= q; o++) {
                                // compute the intersect length with a
vox
                                if ((alphaAll[o] - alphaAll[o - 1]) <
0) {

                                    cout << o << "in" << q << endl;
                                    length = 0;
                                }//-(alphaAll[o] - alphaAll[o-1]) *
rayLength;}
                                else
                                    length = (alphaAll[o] - alphaAll[o
- 1]) * rayLength;
                                // find the midpoint in that voxel
                                alphaMid = (alphaAll[o] + alphaAll[o -
1]) / 2.0;


                                // find the exact x, y, z positions of
the mid point
                                x = doPointZY.x + alphaMid * xDiff;
                                y = doPointZY.y + alphaMid * yDiff;
                                z = doPointZY.z + alphaMid * zDiff;
                                // compute the array index of that
midpoint
```

```
                                xPos = (int) ((x + (recon-
>get_xLength() / 2.0)) / recon->get_xPixelSize());
                                yPos = (int) ((y + (recon-
>get_yLength() / 2.0)) / recon->get_yPixelSize());
                                zPos = (int) ((z + (recon-
>get_zLength() / 2.0)) / recon->get_zPixelSize());
                                xposs[o - 1] = xPos;
                                yposs[o - 1] = yPos;
                                zposs[o - 1] = zPos;

                                distance = sqrt(pow(x, 2) + pow(y, 2));
                                if (zPos <= zDim - 1 && zPos >= 0 &&
distance < xLengthHalf - recon->get_xPixelSize()) {
                                    tmpRay->xyz[r] = (xPos + yPos *
xDim + zPos * xDim * yDim);
                                    tmpRay->length[r] = length;
                                    testlength[r] = length;
                                    r++;
                                }
                            }
                            tmpRay->xyz[r] = 0;
                            tmpRay->length[r] = 0;

                            s = (h
                                    + g * proj->get_pinHoleMatrix()
                                    + j * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
                                    + i * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
                                    * proj->get_yDetectDim() /
symetryRatio);

                            copyRayInc(p, s, r);

//                                  inten1 += recon->get_rotData(zPos,
yPos, xPos) * length / recon->get_tbsf(xPos, yPos, zPos);
                        } else {
                            s = (h
                                    + g * proj->get_pinHoleMatrix()
                                    + j * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
                                    + i * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
                                    * proj->get_yDetectDim() /
symetryRatio);

                            copyRayInc(p, s, 0);

//                            if ((tbsfvalue = recon->get_tbsf(x, y,
z)) == 0) {
//                                inten1 += 0.0;
//                            } else {
//                                inten1 += recon->get_rotData(z, y, x)
* length / recon->get_tbsf(x, y, z);
```

```
//                              }
                        }

                        tmpRay->reset();
                        resetalpha();

                        doPinHolePointZY += incrementPinholeYRot;
                    }
                    doPinHolePointZ += incrementPinholeZRot;
                }
//                                  proj->add_data(a, j, i, inten1, to);
//                                  inten1 = 0;
                doPointZY += incrementYRot;
            }
            doPointZ += incrementZRot;
        }

    }
    tmpRay->reset();
    resetalpha();
}


void SMPinholeRD::buildMatrix(double theta, int a, double** to,double**
from, int forward) {
    time_t start, end;
    time(&start);
    if(forward)
        raytrace(theta,a,to,from,1);
    else
        raytrace(theta,a,to,from,0);
    time(&end);

}

void SMPinholeRD::raytrace(double theta, int a,double** to,double **
from, int forward) {
    // some loop variables
    int g = 0;
    int h = 0;
    int i = 0;
    int j = 0;
    int k = 0;
    int l = 0;
    int m = 0;
    int o = 0;
    int xo = 0;
    int yo = 0;
    int zo = 0;
    int q = 0;
    int r = 0;
    int p = 0;
    long s = 0;
```

```
// the minimum and maximum planes in x, y, and z direction
int kMin = 0;
int kMax = 0;
int lMin = 0;
int lMax = 0;
int mMin = 0;
int mMax = 0;
// some voxel positions
int xPos = 0;
int yPos = 0;
int zPos = 0;
// the greatest alpha position possible
int maxAlpha = 0;

int xi = 0;
int yi = 0;
int zi = 0;
// exact position in the object space
double x = 0.0;
double y = 0.0;
double z = 0.0;

// the maximum x, y, z dimension of the object
double maxObject = 0.0;
// distance between the pinhole and the voxel
double xDiff = 0.0;
double yDiff = 0.0;
double zDiff = 0.0;
// length of ray inside a voxel
double length = 0.0;
// total raylength from pinhole to voxel
double rayLength = 0.0;
// ray intersection of a voxel in x, y, z direction
double alphaXPixelSize = 0.0;
double alphaYPixelSize = 0.0;
double alphaZPixelSize = 0.0;
// first/last x, y, z ray-object intersection as an alpha value
double alphaXStart = 0.0;
double alphaYStart = 0.0;
double alphaZStart = 0.0;
double alphaXEnd = 0.0;
double alphaYEnd = 0.0;
double alphaZEnd = 0.0;
// first/last total ray-object intersection as an alpha value
double alphaXMin = 0.0;
double alphaYMin = 0.0;
double alphaZMin = 0.0;

double alphaXMax = 0.0;
double alphaYMax = 0.0;
double alphaZMax = 0.0;
// midpoint between two plane intersects
double alphaMid = 0.0;
double alphaMin = 0.0;
```

```
    double alphaMax = 0.0;

    double alphaMinFudge = 0.0;
    double alphaMaxFudge = 0.0;

    double distance = 0.0;
    double xLengthHalf = 0.0;

    double inten1 = 0;

    double tbsfvalue = 0.0;

    // some temporary points
    Point3D doPointZ(0.0, 0.0, 0.0);
    Point3D doPointZY(0.0, 0.0, 0.0);
    Point3D doPinHolePointZ(0.0, 0.0, 0.0);
    Point3D doPinHolePointZY(0.0, 0.0, 0.0);

    Point3D startPointRot = proj->get_startDetectPoint();
    Point3D startPinholePointRot;
    /** The rotated increment to the next point in that row (y)  */
    Point3D incrementYRot = proj-> get_incrementDetectY();
    /** The rotated increment to the next point in that column (z)  */
    Point3D incrementZRot = proj->get_incrementDetectZ();

    Point3D incrementPinholeZRot = proj->get_incrementPinHoleZ();
    Point3D incrementPinholeYRot = proj->get_incrementPinHoleY();

        // calculate the rotated points
    incrementZRot.RotateZ(-theta);
    incrementYRot.RotateZ(-theta);
    incrementPinholeYRot.RotateZ(-theta);
    incrementPinholeZRot.RotateZ(-theta);
    startPointRot.RotateZ(-theta);


    xLengthHalf = recon->get_xLength() / 2.0;
    // compute maximum object length
    maxObject = MAX(maxObject, recon->get_xLength());
    maxObject = MAX(maxObject, recon->get_yLength());
    maxObject = MAX(maxObject, recon->get_zLength());
    // using that compute the maximum alpha value, (more of a
gestimate)
    maxAlpha = (int) ((double) 2.0
            + ((proj->get_p2oLength() + maxObject / 2.0)
            / proj->get_p2dLength()));

    for (p = 0; p < proj->get_pinholenum(); p++) {
        startPinholePointRot = proj->get_startPinHolePoint(p);
        startPinholePointRot.RotateZ(-theta);

        doPointZ = startPointRot;
        // loop through all z detector elements
        for (i = 0; i < proj->get_zDetectDim() / symetryRatio; i++) {
```

```
            doPointZY = doPointZ;
            // loop through all y detector elements
            for (j = 0; j < proj->get_yDetectDim() / symetryRatio; j++)
{
                doPinHolePointZ = startPinholePointRot;
                if(!forward) {
                    inten1 = proj->get_data(a, j, i, from);

                    //uncomment for sensitivity correction
//                    if (proj->get_pinholenum() == 1 && proj-
>get_yDetectDim() == 128 && proj->get_zDetectDim() == 1) {
//                        inten1 = inten1 * proj->get_sensitiveMap(p,
i, j);
//                    }
                }
                // loop through all y pinhole matrix points
                for (g = 0; g < proj->get_pinHoleMatrix(); g++) {
                    doPinHolePointZY = doPinHolePointZ;
                    // loop through all x pinhole matrix points
                    for (h = 0; h < proj->get_pinHoleMatrix(); h++) {
                        // if the pinhole is not on the same plane as
the detector element
                        if ((xDiff = (doPinHolePointZY.x -
doPointZY.x)) != 0) //(1)63.5
                        {
                            // compute the first and last ray intersect
with an x-plane
                            alphaXStart = (recon->get_planeX(0) -
doPointZY.x) / xDiff; //(1)0.9956
                            alphaXEnd = (recon->get_planeX(xDim) -
doPointZY.x) / xDiff; //(1)2.106
                            alphaXMin = MIN(alphaXStart, alphaXEnd);
                            alphaXMax = MAX(alphaXStart, alphaXEnd);
                        } else {
                            // otherwise set standard values
                            alphaXMin = 1.0;
                            alphaXMax = maxAlpha;
                        }

                        // if the pinhole is not on the same plane as
the detector element
                        if ((yDiff = doPinHolePointZY.y -
doPointZY.y) != 0) //(1)73.5
                        {
                            // compute the first and last ray intersect
with an y-plane
                            alphaYStart = (recon->get_planeY(0) -
doPointZY.y) / yDiff; //(1)0.38
                            alphaYEnd = (recon->get_planeY(yDim) -
doPointZY.y) / yDiff; //(1)1.34
                            alphaYMin = MIN(alphaYStart, alphaYEnd);
                            alphaYMax = MAX(alphaYStart, alphaYEnd);
                        } else {
                            // otherwise set standard values
```

```
                            alphaYMin = 1.0;
                            alphaYMax = maxAlpha;
                        }
                        // if the pinhole is not on the same plane as
the detector element
                        if ((zDiff = doPinHolePointZY.z -
doPointZY.z) != 0) //(1)63.5
                        {
                            // compute the first and last ray intersect
with an z-plane
                            alphaZStart = (recon->get_planeZ(0) -
doPointZY.z) / zDiff; //(1)0.44
                            alphaZEnd = (recon->get_planeZ(zDim) -
doPointZY.z) / zDiff; //(1)1.55
                            alphaZMin = MIN(alphaZStart, alphaZEnd);
                            alphaZMax = MAX(alphaZStart, alphaZEnd);
                        } else {
                            // otherwise set standard values
                            alphaZMin = 1.0;
                            alphaZMax = maxAlpha;
                        }
                        // compute the very first intersection with an
object plane
                        alphaMin = 1.0;
                        alphaMin = MAX(alphaMin, alphaXMin);
                        alphaMin = MAX(alphaMin, alphaYMin);
                        alphaMin = MAX(alphaMin, alphaZMin); //1

                        // compute the very last intersection with an
object plane
                        alphaMax = maxAlpha;
                        alphaMax = MIN(alphaMax, alphaXMax);
                        alphaMax = MIN(alphaMax, alphaYMax);
                        alphaMax = MIN(alphaMax, alphaZMax); //1.34

                        if (alphaMin < alphaMax) {
                            // compute the total ray length
                            rayLength = sqrt(xDiff * xDiff + yDiff *
yDiff + zDiff * zDiff); //116.04

                            // compute the intersect distance with a
voxel in x, y, z direction
                            if (xDiff == 0)
                                alphaXPixelSize = 0.0;
                            else {
                                alphaXPixelSize = recon-
>get_xPixelSize() / xDiff;
                            }
                            if (yDiff == 0)
                                alphaYPixelSize = 0.0;
                            else {
                                alphaYPixelSize = recon-
>get_yPixelSize() / yDiff;
                            }
```

```
                            if (zDiff == 0)
                                alphaZPixelSize = 0.0;
                            else {
                                alphaZPixelSize = recon-
>get_zPixelSize() / zDiff;
                            }

                            // make sure to get the correct plane after
conversion to x, y, z positions
                            alphaXMin = alphaMin + alphaXPixelSize /
1000000;
                            alphaXMax = alphaMax + alphaXPixelSize /
1000000;
                            alphaYMin = alphaMin + alphaYPixelSize /
1000000;
                            alphaYMax = alphaMax + alphaYPixelSize /
1000000;
                            alphaZMin = alphaMin + alphaZPixelSize /
1000000;
                            alphaZMax = alphaMax + alphaZPixelSize /
1000000;

                            if (xDiff > 0) {
                                // compute minimum and maximum array
positions

                                kMin = (int) ((double) xDim
                                        - (recon->get_planeX(xDim) -
alphaXMin * xDiff - doPointZY.x)
                                        / recon->get_xPixelSize());


                                kMax = (int) ((doPointZY.x + alphaXMax
* xDiff - recon->get_planeX(0))
                                        / recon->get_xPixelSize());

                                // get all other array positions in
between
                                xo = 0;
                                alphaX[xo] = (recon->get_planeX(kMin) -
doPointZY.x) / xDiff;
                                while (alphaX[xo] < alphaMin) {
                                    kMin = kMin + 1;
                                    alphaX[xo] = alphaX[xo] +
alphaXPixelSize;
                                }
                                xo++;
                                for (k = kMin + 1; k <= kMax; k++, xo+
+)
                                    alphaX[xo] = alphaX[xo - 1] +
alphaXPixelSize;
                                alphaX[xo] = alphaMax + 10;
                            } else if (xDiff < 0) {
```

```
                                // compute minimum and maximum array
positions
                                kMin = (int) ((double) xDim
                                        - (recon->get_planeX(xDim) -
alphaXMax * xDiff - doPointZY.x)
                                        / recon->get_xPixelSize());

                                kMax = (int) ((doPointZY.x + alphaXMin
* xDiff - recon->get_planeX(0))
                                        / recon->get_xPixelSize());

                                // get all other array positions in
between
                                xo = 0;
                                alphaX[xo] = (recon->get_planeX(kMax) -
doPointZY.x) / xDiff;
                                while (alphaX[xo] < alphaMin) {
                                    kMax = kMax - 1;
                                    alphaX[xo] = alphaX[xo] -
alphaXPixelSize;
                                }
                                xo++;
                                for (k = kMin; k < kMax; k++, xo++)
                                    alphaX[xo] = alphaX[xo - 1] -
alphaXPixelSize;
                                alphaX[xo] = alphaMax + 10;
                            } else if (xDiff == 0) {
                                xo = 1;
                                alphaX[0] = alphaMin;
                                alphaX[xo] = alphaMax + 10;
                            }

                            if (yDiff > 0) {
                                // compute minimum and maximum array
positions
                                    lMin = (int) ((double) yDim
                                        - (recon->get_planeY(yDim) -
alphaYMin * yDiff - doPointZY.y)
                                        / recon->get_yPixelSize());

                                lMax = (int) ((doPointZY.y + alphaYMax
* yDiff - recon->get_planeY(0))
                                        / recon->get_yPixelSize());

                                // get all other array positions in
between
                                yo = 0;
                                alphaY[yo] = (recon->get_planeY(lMin) -
doPointZY.y) / yDiff;
                                while (alphaY[yo] < alphaMin) {
                                    lMin = lMin + 1;
                                    alphaY[yo] = alphaY[yo] +
alphaYPixelSize;
                                }
```

```
                                yo++;
                                for (l = lMin + 1; l <= lMax; l++, yo+
+)
                                    alphaY[yo] = alphaY[yo - 1] +
alphaYPixelSize;
                                alphaY[yo] = alphaMax + 10;
                            } else if (yDiff < 0) {
                                // compute minimum and maximum array
positions
                                lMin = (int) ((double) yDim
                                        - (recon->get_planeY(yDim) -
alphaYMax * yDiff - doPointZY.y)
                                        / recon->get_yPixelSize());
                                lMax = (int) ((doPointZY.y +
alphaYMin * yDiff - recon->get_planeY(0))
                                        / recon->get_yPixelSize());

                                // get all other array positions in
between
                                yo = 0;
                                alphaY[yo] = (recon->get_planeY(lMax) -
doPointZY.y) / yDiff;
                                while (alphaY[yo] < alphaMin) {
                                    lMax = lMax - 1;
                                    alphaY[yo] = alphaY[yo] -
alphaYPixelSize;
                                }
                                yo++;
                                for (l = lMin; l < lMax; l++, yo++)
                                    alphaY[yo] = alphaY[yo - 1] -
alphaYPixelSize;
                                alphaY[yo] = alphaMax + 10;
                            } else if (yDiff == 0) {
                                yo = 1;
                                alphaY[0] = alphaMin;
                                alphaY[yo] = alphaMax + 10;
                            }


                            if (zDiff > 0) {
                                // compute minimum and maximum array
positions
                                mMin = (int) ((double) zDim
                                        - (recon->get_planeZ(zDim) -
alphaZMin * zDiff - doPointZY.z)
                                        / recon->get_zPixelSize());
                                mMax = (int) ((doPointZY.z + alphaZMax
* zDiff - recon->get_planeZ(0))
                                        / recon->get_zPixelSize());

                                // get all other array positions in
between
                                zo = 0;
```

```
                                alphaZ[zo] = (recon->get_planeZ(mMin) -
doPointZY.z) / zDiff;
                                while (alphaZ[zo] < alphaMin) {
                                    mMin = mMin + 1;
                                    alphaZ[zo] = alphaZ[zo] +
alphaZPixelSize;
                                }
                                zo++;
                                for (m = mMin + 1; m <= mMax; m++, zo+
+)
                                    alphaZ[zo] = alphaZ[zo - 1] +
alphaZPixelSize;
                                alphaZ[zo] = alphaMax + 10;
                            } else if (zDiff < 0) {
                                // compute minimum and maximum array
positions
                                mMin = (int) ((double) zDim
                                        - (recon->get_planeZ(zDim) -
alphaZMax * zDiff - doPointZY.z)
                                        / recon->get_zPixelSize());
                                mMax = (int) ((doPointZY.z + alphaZMin
* zDiff - recon->get_planeZ(0))
                                        / recon->get_zPixelSize());

                                // get all other array positions in
between
                                zo = 0;
                                alphaZ[zo] = (recon->get_planeZ(mMax) -
doPointZY.z) / zDiff;
                                while (alphaZ[zo] < alphaMin) {
                                    mMax = mMax - 1;
                                    alphaZ[zo] = alphaZ[zo] -
alphaZPixelSize;
                                }
                                zo++;
                                for (m = mMin; m < mMax; m++, zo++)
                                    alphaZ[zo] = alphaZ[zo - 1] -
alphaZPixelSize;
                                alphaZ[zo] = alphaMax + 10;
                            } else if (zDiff == 0) {
                                zo = 1;
                                alphaZ[0] = alphaMin;
                                alphaZ[zo] = alphaMax + 10;
                            }

                            // there might be some round-off errors, so
be not so rigid
                            alphaMinFudge = alphaMin + 0.000001;
                            alphaMaxFudge = alphaMax - 0.000001;
                            k = kMin;
                            l = lMin;
                            m = mMin;
                            q = 0;
```

```
                            // merge the three arrays of alpha values
into one sorted array
                            xi = 0;
                            yi = 0;
                            zi = 0;

                            alphaAll[q] = alphaMin;

                            while (alphaX[xi] <= alphaMin)
                                xi++; //alphaX[xi]>1
                            while (alphaY[yi] <= alphaMin)
                                yi++; //alphaY[yi]>1
                            while (alphaZ[zi] <= alphaMin)
                                zi++; //alphaZ[zi]>1

                            while (xi <= xo && yi <= yo && zi <= zo) {
                                if (alphaX[xi] <= alphaY[yi] /*&&
alphaX[xi] <= alphaZ[zi]*/
                                        && alphaX[xi] < alphaMaxFudge
&& alphaX[xi] > alphaMinFudge) {
                                    q++;
                                    alphaAll[q] = alphaX[xi];
                                    xi++;
                                } else if (alphaY[yi] <= alphaX[xi]
/*&& alphaY[yi] <= alphaZ[zi]*/
                                        && alphaY[yi] < alphaMaxFudge
&& alphaY[yi] > alphaMinFudge) {
                                    q++;
                                    alphaAll[q] = alphaY[yi];
                                    yi++;
                                } else if (alphaZ[zi] <= alphaX[xi] &&
alphaZ[zi] <= alphaY[yi]
                                        && alphaZ[zi] < alphaMaxFudge
&& alphaZ[zi] > alphaMinFudge) {
                                    q++;
                                    alphaAll[q] = alphaZ[zi];
                                    zi++;
                                } else {
                                    xi = xo + 1;
                                    yi = yo + 1;
                                    zi = zo + 1;
                                }
                            }
                            // finish array with the last intersect
                            //alphaAll[q++] = alphaMax;

                            // calculate the points
                            for (o = 1, r = 0; o <= q; o++) {
                                // compute the intersect length with a
vox
                                if ((alphaAll[o] - alphaAll[o - 1]) <
0) {
                                    cout << o << "in" << q << endl;
                                    length = 0;
```

```
                                }//-(alphaAll[o] - alphaAll[o-1]) *
rayLength;}
                                else
                                    length = (alphaAll[o] - alphaAll[o
- 1]) * rayLength;
                                // find the midpoint in that voxel
                                alphaMid = (alphaAll[o] + alphaAll[o -
1]) / 2.0;

                                // find the exact x, y, z positions of
the mid point
                                x = doPointZY.x + alphaMid * xDiff;
                                y = doPointZY.y + alphaMid * yDiff;
                                z = doPointZY.z + alphaMid * zDiff;
                                // compute the array index of that
midpoint
                                xPos = (int) ((x + (recon-
>get_xLength() / 2.0)) / recon->get_xPixelSize());
                                yPos = (int) ((y + (recon-
>get_yLength() / 2.0)) / recon->get_yPixelSize());
                                zPos = (int) ((z + (recon-
>get_zLength() / 2.0)) / recon->get_zPixelSize());
//                                xposs[o - 1] = xPos;
//                                yposs[o - 1] = yPos;
//                                zposs[o - 1] = zPos;

                                  if (xPos >= 128)
                                      xPos = 127;
                                  if (xPos < 0)
                                      xPos = 0;
                                  if (yPos >= 128)
                                      yPos = 127;
                                  if (yPos < 0)
                                      yPos = 0;
                                  if (zPos >= 128)
                                      zPos = 127;
                                  if (zPos < 0)
                                      zPos = 0;
                                if(forward){
                                    inten1 += recon->get_data(zPos,
yPos, xPos, from) * length / recon->get_tbsf(xPos, yPos, zPos);
                                } else
                                    recon->set_rotData(zPos, yPos,
xPos, (double) (inten1 * length) / recon->get_tbsf(xPos, yPos, zPos));

                            }

                            s = (h
                                    + g * proj->get_pinHoleMatrix()
                                    + j * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
                                    + i * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
```

```
                                    * proj->get_yDetectDim() /
symetryRatio);

                        } else {
                            s = (h
                                    + g * proj->get_pinHoleMatrix()
                                    + j * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
                                    + i * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
                                    * proj->get_yDetectDim() /
symetryRatio);

                            if ((tbsfvalue = recon->get_tbsf(x, y, z))
== 0) {
                                if(forward){
                                    inten1 += 0;
                                } else
                                    recon->set_rotData(zPos, yPos,
xPos,0);
                            } else {
                                if(forward){
                                    inten1 += recon->get_data(zPos,
yPos, xPos, from) * length / recon->get_tbsf(xPos, yPos, zPos);
                                } else
                                    recon->set_rotData(zPos, yPos,
xPos, (double) (inten1 * length) / recon->get_tbsf(xPos, yPos, zPos));
                            }
                        }

                        doPinHolePointZY += incrementPinholeYRot;
                    }
                    doPinHolePointZ += incrementPinholeZRot;
                }
                if(forward) {
                //uncomment for sensitivity correction
//                    if (proj->get_pinholenum() == 1 && proj-
>get_yDetectDim() == 128 && proj->get_zDetectDim() == 1) {
//                        inten1 = inten1 * proj->get_sensitiveMap(p,
i, j);
//                    }
                    proj->add_data(a, j, i, inten1, to);
                    inten1 = 0.0;
                }
                doPointZY += incrementYRot;
            }
            doPointZ += incrementZRot;
        }

    }

    tmpRay->reset();
    resetalpha();
}
```

```
// This is the old way
void SMPinholeRD::raytrace() {
    cout << "raytrace...1" << endl;
    // some loop variables
    int g = 0;
    int h = 0;
    int i = 0;
    int j = 0;
    int k = 0;
    int l = 0;
    int m = 0;
    int o = 0;
    int xo = 0;
    int yo = 0;
    int zo = 0;
    int q = 0;
    int r = 0;
    int p = 0;
    long s = 0;

    // the minimum and maximum planes in x, y, and z direction
    int kMin = 0;
    int kMax = 0;
    int lMin = 0;
    int lMax = 0;
    int mMin = 0;
    int mMax = 0;
    // some voxel positions
    int xPos = 0;
    int yPos = 0;
    int zPos = 0;
    // the greatest alpha position possible
    int maxAlpha = 0;

    int xi = 0;
    int yi = 0;
    int zi = 0;
    // exact position in the object space
    double x = 0.0;
    double y = 0.0;
    double z = 0.0;

    // the maximum x, y, z dimension of the object
    double maxObject = 0.0;
    // distance between the pinhole and the voxel
    double xDiff = 0.0;
    double yDiff = 0.0;
    double zDiff = 0.0;
    // length of ray inside a voxel
    double length = 0.0;
    // total raylength from pinhole to voxel
    double rayLength = 0.0;
    // ray intersection of a voxel in x, y, z direction
```

```
    double alphaXPixelSize = 0.0;
    double alphaYPixelSize = 0.0;
    double alphaZPixelSize = 0.0;
    // first/last x, y, z ray-object intersection as an alpha value
    double alphaXStart = 0.0;
    double alphaYStart = 0.0;
    double alphaZStart = 0.0;
    double alphaXEnd = 0.0;
    double alphaYEnd = 0.0;
    double alphaZEnd = 0.0;
    // first/last total ray-object intersection as an alpha value
    double alphaXMin = 0.0;
    double alphaYMin = 0.0;
    double alphaZMin = 0.0;

    double alphaXMax = 0.0;
    double alphaYMax = 0.0;
    double alphaZMax = 0.0;
    // midpoint between two plane intersects
    double alphaMid = 0.0;
    double alphaMin = 0.0;
    double alphaMax = 0.0;

    double alphaMinFudge = 0.0;
    double alphaMaxFudge = 0.0;

    double distance = 0.0;
    double xLengthHalf = 0.0;

    // some temporary points
    Point3D doPointZ(0.0, 0.0, 0.0);
    Point3D doPointZY(0.0, 0.0, 0.0);
    Point3D doPinHolePointZ(0.0, 0.0, 0.0);
    Point3D doPinHolePointZY(0.0, 0.0, 0.0);

    xLengthHalf = recon->get_xLength() / 2.0;
    // compute maximum object length
    maxObject = MAX(maxObject, recon->get_xLength());
    maxObject = MAX(maxObject, recon->get_yLength());
    maxObject = MAX(maxObject, recon->get_zLength());
    // using that compute the maximum alpha value, (more of a
gestimate)
    maxAlpha = (int) ((double) 2.0
            + ((proj->get_p2oLength() + maxObject / 2.0)
            / proj->get_p2dLength()));

    for (p = 0; p < proj->get_pinholenum(); p++) {
        cout << "pinnum " << p << endl;

        doPointZ = proj->get_startDetectPoint();
        // loop through all z detector elements
        for (i = 0; i < proj->get_zDetectDim() / symetryRatio; i++) {
            doPointZY = doPointZ;
            // loop through all y detector elements
```

```
            for (j = 0; j < proj->get_yDetectDim() / symetryRatio; j++)
{
                doPinHolePointZ = proj->get_startPinHolePoint(p);
                // loop through all y pinhole matrix points
                for (g = 0; g < proj->get_pinHoleMatrix(); g++) {
                    doPinHolePointZY = doPinHolePointZ;
                    // loop through all x pinhole matrix points
                    for (h = 0; h < proj->get_pinHoleMatrix(); h++) {
                        if ((xDiff = doPinHolePointZY.x -
doPointZY.x) != 0) //(1)63.5
                        {
                            // compute the first and last ray intersect
with an x-plane
                            alphaXStart = (recon->get_planeX(0) -
doPointZY.x) / xDiff; //(1)0.9956
                            alphaXEnd = (recon->get_planeX(xDim) -
doPointZY.x) / xDiff; //(1)2.106
                            alphaXMin = MIN(alphaXStart, alphaXEnd);
                            alphaXMax = MAX(alphaXStart, alphaXEnd);
                        } else {
                            // otherwise set standard values
                            alphaXMin = 1.0;
                            alphaXMax = maxAlpha;
                        }

                        // if the pinhole is not on the same plane as
the detector element
                        if ((yDiff = doPinHolePointZY.y -
doPointZY.y) != 0) //(1)73.5
                        {
                            // compute the first and last ray intersect
with an y-plane
                            alphaYStart = (recon->get_planeY(0) -
doPointZY.y) / yDiff; //(1)0.38
                            alphaYEnd = (recon->get_planeY(yDim) -
doPointZY.y) / yDiff; //(1)1.34
                            alphaYMin = MIN(alphaYStart, alphaYEnd);
                            alphaYMax = MAX(alphaYStart, alphaYEnd);
                        } else {
                            // otherwise set standard values
                            alphaYMin = 1.0;
                            alphaYMax = maxAlpha;
                        }

                        // if the pinhole is not on the same plane as
the detector element
                        if ((zDiff = doPinHolePointZY.z -
doPointZY.z) != 0) //(1)63.5
                        {
                            // compute the first and last ray intersect
with an z-plane
                            alphaZStart = (recon->get_planeZ(0) -
doPointZY.z) / zDiff; //(1)0.44
```

```
                            alphaZEnd = (recon->get_planeZ(zDim) -
doPointZY.z) / zDiff; //(1)1.55
                            alphaZMin = MIN(alphaZStart, alphaZEnd);
                            alphaZMax = MAX(alphaZStart, alphaZEnd);
                        } else {
                            // otherwise set standard values
                            alphaZMin = 1.0;
                            alphaZMax = maxAlpha;
                        }

                        // compute the very first intersection with an
object plane
                        alphaMin = 1.0;
                        alphaMin = MAX(alphaMin, alphaXMin);
                        alphaMin = MAX(alphaMin, alphaYMin);
                        alphaMin = MAX(alphaMin, alphaZMin); //1

                        // compute the very last intersection with an
object plane
                        alphaMax = maxAlpha;
                        alphaMax = MIN(alphaMax, alphaXMax);
                        alphaMax = MIN(alphaMax, alphaYMax);
                        alphaMax = MIN(alphaMax, alphaZMax); //1.34

                        if (alphaMin < alphaMax) {
                            // compute the total ray length
                            rayLength = sqrt(xDiff * xDiff + yDiff *
yDiff + zDiff * zDiff); //116.04

                            // compute the intersect distance with a
voxel in x, y, z direction
                            if (xDiff == 0) alphaXPixelSize = 0.0;
                            else {
                                alphaXPixelSize = recon-
>get_xPixelSize() / xDiff;
                            }
                            if (yDiff == 0) alphaYPixelSize = 0.0;
                            else {
                                alphaYPixelSize = recon-
>get_yPixelSize() / yDiff;
                            }
                            if (zDiff == 0) alphaZPixelSize = 0.0;
                            else {
                                alphaZPixelSize = recon-
>get_zPixelSize() / zDiff;
                            }

                            // make sure to get the correct plane after
conversion to x, y, z positions
                            alphaXMin = alphaMin + alphaXPixelSize /
1000000;
                            alphaXMax = alphaMax + alphaXPixelSize /
1000000;
```

```
                            alphaYMin = alphaMin + alphaYPixelSize /
1000000;
                            alphaYMax = alphaMax + alphaYPixelSize /
1000000;
                            alphaZMin = alphaMin + alphaZPixelSize /
1000000;
                            alphaZMax = alphaMax + alphaZPixelSize /
1000000;

                            if (xDiff > 0) {
                                // compute minimum and maximum array
positions
                                kMin = (int) ((double) xDim
                                        - (recon->get_planeX(xDim) -
alphaXMin * xDiff - doPointZY.x)
                                        / recon->get_xPixelSize());
                                kMax = (int) ((doPointZY.x + alphaXMax
* xDiff - recon->get_planeX(0))
                                        / recon->get_xPixelSize());

                                // get all other array positions in
between
                                xo = 0;
                                alphaX[xo] = (recon->get_planeX(kMin) -
doPointZY.x) / xDiff;
                                while (alphaX[xo] < alphaMin) {
                                    kMin = kMin + 1;
                                    alphaX[xo] = alphaX[xo] +
alphaXPixelSize;
                                }
                                xo++;
                                for (k = kMin + 1; k <= kMax; k++, xo+
+)
                                    alphaX[xo] = alphaX[xo - 1] +
alphaXPixelSize;
                                alphaX[xo] = alphaMax + 10;
                            } else if (xDiff < 0) {
                                // compute minimum and maximum array
positions
                                kMin = (int) ((double) xDim
                                        - (recon->get_planeX(xDim) -
alphaXMax * xDiff - doPointZY.x)
                                        / recon->get_xPixelSize());
                                kMax = (int) ((doPointZY.x + alphaXMin
* xDiff - recon->get_planeX(0))
                                        / recon->get_xPixelSize());

                                // get all other array positions in
between
                                xo = 0;
                                alphaX[xo] = (recon->get_planeX(kMax) -
doPointZY.x) / xDiff;
                                while (alphaX[xo] < alphaMin) {
                                    kMax = kMax - 1;
```

```
                                    alphaX[xo] = alphaX[xo] -
alphaXPixelSize;
                                }
                                xo++;
                                for (k = kMin; k < kMax; k++, xo++)
                                    alphaX[xo] = alphaX[xo - 1] -
alphaXPixelSize;
                                alphaX[xo] = alphaMax + 10;
                            } else if (xDiff == 0) {
                                xo = 1;
                                alphaX[0] = alphaMin;
                                alphaX[xo] = alphaMax + 10;
                            }

                            if (yDiff > 0) {
                                // compute minimum and maximum array
positions
                                lMin = (int) ((double) yDim
                                        - (recon->get_planeY(yDim) -
alphaYMin * yDiff - doPointZY.y)
                                        / recon->get_yPixelSize());
                                lMax = (int) ((doPointZY.y + alphaYMax
* yDiff - recon->get_planeY(0))
                                        / recon->get_yPixelSize());

                                // get all other array positions in
between
                                yo = 0;
                                alphaY[yo] = (recon->get_planeY(lMin) -
doPointZY.y) / yDiff;
                                while (alphaY[yo] < alphaMin) {
                                    lMin = lMin + 1;
                                    alphaY[yo] = alphaY[yo] +
alphaYPixelSize;
                                }
                                yo++;
                                for (l = lMin + 1; l <= lMax; l++, yo+
+)
                                    alphaY[yo] = alphaY[yo - 1] +
alphaYPixelSize;
                                alphaY[yo] = alphaMax + 10;
                            } else if (yDiff < 0) {
                                // compute minimum and maximum array
positions
                                lMin = (int) ((double) yDim
                                        - (recon->get_planeY(yDim) -
alphaYMax * yDiff - doPointZY.y)
                                        / recon->get_yPixelSize());
                                lMax = (int) ((doPointZY.y + alphaYMin
* yDiff - recon->get_planeY(0))
                                        / recon->get_yPixelSize());

                                // get all other array positions in
between
```

```
                                yo = 0;
                                alphaY[yo] = (recon->get_planeY(lMax) -
doPointZY.y) / yDiff;
                                while (alphaY[yo] < alphaMin) {
                                    lMax = lMax - 1;
                                    alphaY[yo] = alphaY[yo] -
alphaYPixelSize;
                                }
                                yo++;
                                for (l = lMin; l < lMax; l++, yo++)
                                    alphaY[yo] = alphaY[yo - 1] -
alphaYPixelSize;
                                alphaY[yo] = alphaMax + 10;
                            } else if (yDiff == 0) {
                                yo = 1;
                                alphaY[0] = alphaMin;
                                alphaY[yo] = alphaMax + 10;
                            }


                            if (zDiff > 0) {
                                // compute minimum and maximum array
positions
                                mMin = (int) ((double) zDim
                                        - (recon->get_planeZ(zDim) -
alphaZMin * zDiff - doPointZY.z)
                                        / recon->get_zPixelSize());
                                mMax = (int) ((doPointZY.z + alphaZMax
* zDiff - recon->get_planeZ(0))
                                        / recon->get_zPixelSize());

                                // get all other array positions in
between
                                zo = 0;
                                alphaZ[zo] = (recon->get_planeZ(mMin) -
doPointZY.z) / zDiff;
                                while (alphaZ[zo] < alphaMin) {
                                    mMin = mMin + 1;
                                    alphaZ[zo] = alphaZ[zo] +
alphaZPixelSize;
                                }
                                zo++;
                                for (m = mMin + 1; m <= mMax; m++, zo+
+)
                                    alphaZ[zo] = alphaZ[zo - 1] +
alphaZPixelSize;
                                alphaZ[zo] = alphaMax + 10;
                            } else if (zDiff < 0) {
                                // compute minimum and maximum array
positions
                                mMin = (int) ((double) zDim
                                        - (recon->get_planeZ(zDim) -
alphaZMax * zDiff - doPointZY.z)
                                        / recon->get_zPixelSize());
```

```
                                mMax = (int) ((doPointZY.z + alphaZMin
* zDiff - recon->get_planeZ(0))
                                        / recon->get_zPixelSize());

                                // get all other array positions in
between
                                zo = 0;
                                alphaZ[zo] = (recon->get_planeZ(mMax) -
doPointZY.z) / zDiff;
                                while (alphaZ[zo] < alphaMin) {
                                    mMax = mMax - 1;
                                    alphaZ[zo] = alphaZ[zo] -
alphaZPixelSize;
                                }
                                zo++;
                                for (m = mMin; m < mMax; m++, zo++)
                                    alphaZ[zo] = alphaZ[zo - 1] -
alphaZPixelSize;
                                alphaZ[zo] = alphaMax + 10;
                            } else if (zDiff == 0) {
                                zo = 1;
                                alphaZ[0] = alphaMin;
                                alphaZ[zo] = alphaMax + 10;
                            }

                            // there might be some round-off errors, so
be not so rigid
                            alphaMinFudge = alphaMin + 0.000001;
                            alphaMaxFudge = alphaMax - 0.000001;
                            k = kMin;
                            l = lMin;
                            m = mMin;
                            q = 0;

                            // merge the three arrays of alpha values
into one sorted array
                            xi = 0;
                            yi = 0;
                            zi = 0;

                            alphaAll[q] = alphaMin;

                            while (alphaX[xi] <= alphaMin)
                                xi++; //alphaX[xi]>1
                            while (alphaY[yi] <= alphaMin)
                                yi++; //alphaY[yi]>1
                            while (alphaZ[zi] <= alphaMin)
                                zi++; //alphaZ[zi]>1

                            while (xi <= xo && yi <= yo && zi <= zo) {
                                if (alphaX[xi] <= alphaY[yi] &&
alphaX[xi] <= alphaZ[zi]
                                        && alphaX[xi] < alphaMaxFudge
&& alphaX[xi] > alphaMinFudge) {
```

```
                                    q++;
                                    alphaAll[q] = alphaX[xi];
                                    xi++;
                                } else if (alphaY[yi] <= alphaX[xi] &&
alphaY[yi] <= alphaZ[zi]
                                        && alphaY[yi] < alphaMaxFudge
&& alphaY[yi] > alphaMinFudge) {
                                    q++;
                                    alphaAll[q] = alphaY[yi];
                                    yi++;
                                } else if (alphaZ[zi] <= alphaX[xi] &&
alphaZ[zi] <= alphaY[yi]
                                        && alphaZ[zi] < alphaMaxFudge
&& alphaZ[zi] > alphaMinFudge) {
                                    q++;
                                    alphaAll[q] = alphaZ[zi];
                                    zi++;
                                } else {
                                    xi = xo + 1;
                                    yi = yo + 1;
                                    zi = zo + 1;
                                }
                            }
                            // finish array with the last intersect
                            //alphaAll[q++] = alphaMax;

                            // calculate the points
                            for (o = 1, r = 0; o <= q; o++) {
                                // compute the intersect length with a
vox
                                if ((alphaAll[o] - alphaAll[o - 1]) <
0) {
                                    cout << o << "in" << q << endl;
                                    length = 0;
                                }//-(alphaAll[o] - alphaAll[o-1]) *
rayLength;}
                                else
                                    length = (alphaAll[o] - alphaAll[o
- 1]) * rayLength;
                                // find the midpoint in that voxel
                                alphaMid = (alphaAll[o] + alphaAll[o -
1]) / 2.0;

                                // find the exact x, y, z positions of
the mid point
                                x = doPointZY.x + alphaMid * xDiff;
                                y = doPointZY.y + alphaMid * yDiff;
                                z = doPointZY.z + alphaMid * zDiff;

                                // compute the array index of that
midpoint
                                xPos = (int) ((x + (recon-
>get_xLength() / 2.0)) / recon->get_xPixelSize());
```

```
                                yPos = (int) ((y + (recon-
>get_yLength() / 2.0)) / recon->get_yPixelSize());
                                zPos = (int) ((z + (recon-
>get_zLength() / 2.0)) / recon->get_zPixelSize());
                                xposs[o - 1] = xPos;
                                yposs[o - 1] = yPos;
                                zposs[o - 1] = zPos;

                                distance = sqrt(pow(x, 2) + pow(y, 2));

                                if (zPos <= zDim - 1 && zPos >= 0 &&
distance < xLengthHalf - recon->get_xPixelSize()) {
                                    tmpRay->xyz[r] = (xPos + yPos *
xDim + zPos * xDim * yDim);
                                    tmpRay->length[r] = length;
                                    testlength[r] = length;
                                    r++;
                                }
                            }
                            tmpRay->xyz[r] = 0;
                            tmpRay->length[r] = 0;

                            s = (h
                                    + g * proj->get_pinHoleMatrix()
                                    + j * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
                                    + i * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
                                    * proj->get_yDetectDim() /
symetryRatio);

                            copyRayInc(p, s, r);
                        } else {
                            s = (h
                                    + g * proj->get_pinHoleMatrix()
                                    + j * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
                                    + i * proj->get_pinHoleMatrix() *
proj->get_pinHoleMatrix()
                                    * proj->get_yDetectDim() /
symetryRatio);

                            copyRayInc(p, s, 0);
                        }

                        tmpRay->reset();
                        resetalpha();

                        doPinHolePointZY += proj-
>get_incrementPinHoleY();
                    }
                    doPinHolePointZ += proj->get_incrementPinHoleZ();
                }
                doPointZY += proj->get_incrementDetectY();
```

```
                //
cout<<rayLength<<endl;
            }
            doPointZ += proj->get_incrementDetectZ();
        }

    }
    tmpRay->reset();
    resetalpha();
    cout << "raytrace end .." << endl;
}

bool SMPinholeRD::copyRayInc(int p, long rayNum, int raySize) {
    // p:= pinhole number; rayNum := s from above; raySize := r
    //    cout << p << "," << rayNum << "," << raySize << endl;

    bool success = true;
    int i = 0;
    int k = 0;
    int xyz = 0;
    int xyDim = 0;
    int zxyDim = 0;

    unsigned short xS = 0;
    unsigned short yS = 0;
    unsigned short zS = 0;

    unsigned short x = 0;
    unsigned short y = 0;
    unsigned short z = 0;

    unsigned short xInc = 0;
    unsigned short yInc = 0;
    unsigned short zInc = 0;

    unsigned short xSgn = 0;
    unsigned short ySgn = 0;
    unsigned short zSgn = 0;

    unsigned char inc = 0;
    unsigned char sgn = 0;

    unsigned char diff = 0;
    unsigned char sign = 0;

    char sdiff = 0;
    rayInc[p][rayNum].length.clear();
    rayInc[p][rayNum].xyzInc.clear();
    rayInc[p][rayNum].xyzSgn.clear();

    if (raySize == 0) {
        rayInc[p][rayNum].xStart = 0;
        rayInc[p][rayNum].yStart = 0;
        rayInc[p][rayNum].zStart = 0;
```

```
    rayInc[p][rayNum].length.push_back(0);
} else {
    xyDim = xDim * yDim;
    xyz = tmpRay->xyz[0];
    zS = xyz / xyDim;
    zxyDim = zS * xyDim;
    yS = (xyz - zxyDim) / xDim;
    xS = xyz - zxyDim - yS * xDim;

    //        cout<<"xs"<<xS<<"ys"<<yS<<"zS"<<zS<<endl;
    rayInc[p][rayNum].xStart = xS;
    rayInc[p][rayNum].yStart = yS;
    rayInc[p][rayNum].zStart = zS;
    rayInc[p][rayNum].length.push_back(tmpRay->length[i]);
}

for (i = 1, k = 0; i < raySize; i++, k++) {
    inc = 0;
    sgn = 0;

    rayInc[p][rayNum].length.push_back(tmpRay->length[i]);
    xyz = tmpRay->xyz[i];
    z = xyz / xyDim;
    zxyDim = z * xyDim;
    y = (xyz - zxyDim) / xDim;
    x = xyz - zxyDim - y * xDim;

    if ((sdiff = x - xS) < 0) {
        xInc = -sdiff;
        xSgn = 1;
    } else {
        xInc = sdiff;
        xSgn = 0;
    }

    if ((sdiff = y - yS) < 0) {
        yInc = -sdiff;
        ySgn = 1;
    } else {
        yInc = sdiff;
        ySgn = 0;
    }

    if ((sdiff = z - zS) < 0) {
        zInc = -sdiff;
        zSgn = 1;
    } else {
        zInc = sdiff;
        zSgn = 0;
    }

    xS = x;
    yS = y;
    zS = z;
```

```
i++;

rayInc[p][rayNum].length.push_back(tmpRay->length[i]);
xyz = tmpRay->xyz[i];
z = xyz / xyDim;
zxyDim = z * xyDim;
y = (xyz - zxyDim) / xDim;
x = xyz - zxyDim - y * xDim;

if (tmpRay->length[i] == 0 && tmpRay->xyz[i] == 0) {
    diff = 0;
    inc |= diff;
    inc <<= 1;
    inc |= diff;
    inc <<= 1;
    inc |= diff;
    inc <<= 1;

    sign = 0;
    sgn |= sign;
    sgn <<= 1;
    sgn |= sign;
    sgn <<= 1;
    sgn |= sign;
    sgn <<= 1;

} else {
    if ((sdiff = z - zS) < 0) {
        diff = -sdiff;
        inc |= diff;
        inc <<= 1;

        sign = 1;
        sgn |= sign;
        sgn <<= 1;
    } else {
        diff = sdiff;
        inc |= diff;
        inc <<= 1;

        sign = 0;
        sgn |= sign;
        sgn <<= 1;
    }

    if ((sdiff = y - yS) < 0) {
        diff = -sdiff;
        inc |= diff;
        inc <<= 1;

        sign = 1;
        sgn |= sign;
        sgn <<= 1;
```

```
            } else {
                diff = sdiff;
                inc |= diff;
                inc <<= 1;

                sign = 0;
                sgn |= sign;
                sgn <<= 1;
            }

            if ((sdiff = x - xS) < 0) {
                diff = -sdiff;
                inc |= diff;
                inc <<= 1;

                sign = 1;
                sgn |= sign;
                sgn <<= 1;
            } else {
                diff = sdiff;
                inc |= diff;
                inc <<= 1;

                sign = 0;
                sgn |= sign;
                sgn <<= 1;
            }
        }

        inc |= zInc;
        inc <<= 1;
        sgn |= zSgn;
        sgn <<= 1;

        inc |= yInc;
        inc <<= 1;
        sgn |= ySgn;
        sgn <<= 1;

        inc |= xInc;
        inc <<= 1;
        sgn |= xSgn;
        sgn <<= 1;

        xS = x;
        yS = y;
        zS = z;

        rayInc[p][rayNum].xyzInc.push_back(inc);
        rayInc[p][rayNum].xyzSgn.push_back(sgn);
    }
    return success;
}
```

smpinholerd.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
*************************************************************************
****
                          smpinholerd.h  -  description
                             -------------------
    begin                : Wed Apr 21 08:48:48 CST 2006
    copyright            : (C) 2006 by Christian Wietholt
    email                : cwietholt@nhri.org.tw

*************************************************************************
****/

/
*************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

*************************************************************************
****/
#ifndef SMPINHOLERD_H
#define SMPINHOLERD_H

#include <vector>

#include "systemmatrix.h"

class ProjData;
class ReconData;
class RayTBL3D;

/**
 * Class SMPinholeError
 * Implementation of a system matrix with a pinhole geometry using a
 * error function to model the aperture.
 */
class SMPinholeRD : public SystemMatrix {
    /**
     * Public stuff
     */
public:
    /**
     * Fields
```

```
     */
    /**
     *
     */
    /**
     * Constructors
     */
    SMPinholeRD();

    SMPinholeRD(ProjData *p, ReconData *r, int symetryRatio);

    ~SMPinholeRD();
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
    virtual void buildMatrix(double theta);
    virtual void buildMatrix(double theta,int a, double** to,double**
from,int forward);
    virtual void buildMatrix();

    /**
     * Protected stuff
     */
protected:
    /**
     * Fields
     */
    /**
     *
     */
    /**
     * Constructors
     */
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
    /**
     * Private stuff
     */
private:
    /**
     * Fields
     */
    double *alphaX;
    double *alphaY;
    double *alphaZ;
    double *alphaAll;
```

```
    int *xposs;
    int *yposs;
    int *zposs;
    double *testlength;

    int alphaXSize;
    int alphaYSize;
    int alphaZSize;
    int alphaAllSize;

    RayTBL3D *tmpRay;
    /**
     *
     */
    /**
     * Constructors
     */
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
    void raytrace(double theta);
    void raytrace(double theta,int a, double** to,double** from,int
forward);
    void raytrace();
    bool copyRay(int rayNum, int raySize);
    bool copyRayInc(int p, long rayNum, int raySize);
    void resetalpha();

    template<typename T> T MIN(const T &x, const T &y) {
        return ( (x) < (y) ? (x) : (y));
    }

    template<typename T> T MAX(const T &x, const T &y) {
        return ( (x) > (y) ? (x) : (y));
    }

    /** This function generates the values of the
     * error function with a certain step size.
     */
};
#endif //SMPINHOLERD_H
```

systematrix.cpp

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************************
****
                          systemmatrix.cpp  -  description
                             -------------------
    begin                : Wed Apr 05 07:51:48 CST 2006
    copyright            : (C) 2006 by Christian Wietholt
    email                : cwietholt@nhri.org.tw
```

```
**************************************************************************
****/

/
**************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

**************************************************************************
****/

#include "systemmatrix.h"

#include "recondata.h"
#include "projdata.h"
#include <cstring>
#include <string>
#include <iostream>
#include <fstream>
using namespace std;

/**
 * Constructors/Destructors
 */

SystemMatrix::SystemMatrix() {
}

SystemMatrix::SystemMatrix(ProjData *p, ReconData *r, int symR) {
    symetryRatio = symR;
    proj = p;
    recon = r;
    xDim = recon->get_xDim();
    yDim = recon->get_yDim();
    zDim = recon->get_zDim();
    matrixSize = 0;
}

SystemMatrix::~SystemMatrix() {
}

/**
```

```
 * Methods
 */


void SystemMatrix::saveFunctionData(const string &filename) {
    int i = 0;
    int j = 0;

    fstream file;
    file.open( filename.c_str(), fstream::out);

    for (i = 0; i < numFunctions; i++) {
        for (j = 0; j < function[i].numValues; j++) {
            file << function[i].functionValues[j] << ",";
        }
        file << endl;
    }
    file.close();
}

void SystemMatrix::save(const string &filename) {
    int i = 0;
    int j = 0;
    int k = 0;

    fstream file;
    file.open(filename.c_str(), fstream::out | fstream::binary);

    for (i = 0; i < zDim; i++) {
        for (j = 0; j < yDim; j++) {
            for (k = 0; k < xDim; k++) {
                file << (double) detect[k + j * yDim + i * yDim *
xDim].scale;
            }
        }
    }
    file.close();
}
```

systematrix.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************************
****
                          systemmatrix.h  -  description
                             -------------------
    begin                : Wed Apr 5 07:50:48 CST 2006
    copyright            : (C) 2006 by Christian Wietholt
    email                : cwietholt@nhri.org.tw

***************************************************************************
****/
```

```
/
*************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

*************************************************************************
****/

#ifndef SYSTEMMATRIX_H
#define SYSTEMMATRIX_H

#include "defs.h"
#include "point3d.h"
#include <string>
#include <iostream>
using namespace std;


class ProjData;
class ReconData;
class RayTBL3D;
class RayTBL3DInc;

/**
 * Class ForwardProjector
 *
 */
class SystemMatrix {
    /**
     * Public stuff
     */
public:
    /**
     * Fields
     */
    Funct *function;
    IntersectScale *detect;
    ProjRadii *radii;

    RayTBL3DInc **rayInc;

    double functionValueStepSize;
```

```
    /**
     *
     */
    /**
     * Constructors
     */
    SystemMatrix();
    SystemMatrix(ProjData *p, ReconData *r, int symR);

    virtual ~SystemMatrix();
    /**
     * Accessor Methods
     */

    /**
     * Operations
     */
    virtual void buildMatrix() {
    };

    virtual void buildMatrix(double theta) {
    };

    virtual void buildMatrix(double theta, int a, double** to,double**
from, int forward) {
    };
    void saveFunctionData(const string &filename);
    void save(const string &filename);

    /**
     * Protected stuff
     */
protected:
    /**
     * Fields
     */
    ProjData *proj;
    ReconData *recon;

    /** The rotated starting point. */
    Point3D startPoint;
    /** The increment to the next point in that row (X)  */
    Point3D incrementX;
    /** The increment to the next point in that column (Y)  */
    Point3D incrementY;
    /** The increment to the next point in that slice (Z)  */
    Point3D incrementZ;

    int xDim;
    int yDim;
    int zDim;

    int matrixSize;
    int symetryRatio;
```

```
    int numFunctions;
    /**
     *
     */
    /**
     * Constructors
     */
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
    /**
     * Private stuff
     */
private:
    /**
     * Fields
     */
    /**
     *
     */
    /**
     * Constructors
     */
    /**
     * Accessor Methods
     */
    /**
     * Operations
     */
};
#endif //FORWARDPROJECTOR_H
```

defs.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
**********************************************************************
****
 *   Copyright (C) 2005 by Christian Wietholt
*
 *   chris@gamma.mipl.cgu.edu.tw
*
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
```

```
 *
*
 *   This program is distributed in the hope that it will be useful,
*
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
*
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
*
 *   GNU General Public License for more details.
*
 *
*
 *   You should have received a copy of the GNU General Public License
*
 *   along with this program; if not, write to the
*
 *   Free Software Foundation, Inc.,
*
 *   59 Temple Place - Suite 330, Boston, MA  02111-1307, USA.
*

**********************************************************************
****/
#ifndef DEFS_H
#define DEFS_H

typedef struct {
    double rad;
    double sin;
    double cos;
    double tan;
} SETANG;

typedef struct {
    double i;
    double j;
    double x;
    double y;
    double intercept;
    double ratio;
} COR;

typedef struct {
    /** X_focal_point=cor.x-COR2FocPt*cos(theta)
     */
    double xf;
    /** Y_focal_point=cor.y-COR2FocPt*sin(theta)
     */
    double yf;
    /** =theta-(angle between focal line and ray thru bin t)
     */
    double tao;
    /** Focal length of the Fan beam
     */
```

```
    double FocLen;
    /** the distance between COR and Focal Point
     */
    double COR2FocPt;
} FAN;

typedef struct {
    /** The values for an error function
     */
    double *functionValues;
    /** The number of values stored until error function goes to zero
     */
    int numValues;
} Funct;

typedef struct {
    /** The y detector coordinate of the ray intersection
     */
    double y;
    /** The z detector coordinate of the ray intersection
     */
    double z;
    /** A scaling factor for this ray to make the probability
     * of the corresponding interpolation function zero
     */
    double scale;
} IntersectScale;

typedef struct {
    /** The projected pinhole radius */
    double radius;
    /** The radius of the pinhole projection in measures of pixel
     */
    int radiusGaussPixel;
    /** The square of the above computed radius in measures of pixel
     */
    int radiusGaussSquare;
    /** Magnification factor squared */
    double magFactor;
} ProjRadii;


#endif
```

projectorenum.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************
****
                          projectorenum.h  -  description
                             -------------------
    begin                : Fri Mar 5 2004
    copyright            : (C) 2004 by Christian Wietholt
    email                : christian.wietholt@marquette.edu
```

```
*********************************************************************
****/

/
*********************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

*********************************************************************
****/


#ifndef PROJECTORENUM_H
#define PROJECTORENUM_H


      /** The type of region tool.  */
      enum ProjectorType
      {
            NOPROJ                              = -1,
            POINTPINHOLE_VD               =  0,
            ROUNDPINHOLE_VD               =  1,
            ERRORPINHOLE_VD               =  2,
            GAUSSPINHOLE_VD               =  3,
            ERRORPINHOLEROT_VD      =  4,
    PINHOLE_RD          =  5,
    PINHOLE_ATTN_RD     =  6,
            FANBEAM_RD                          =  7,
            CONEBEAMCT                          =  8
      };  // Define one

#endif

#ifndef PINHOLEENUM_H
#define PINHOLEENUM_H

  enum PinholeTyp
  {
    INFILE              = 0,
    ROUND               = 1,
    GAUSSIAN            = 2
  };
```

```
#endif

#ifndef LOCATIONENUM_H
#define LOCATIONENUM_H


  /** The type of region tool.  */
  enum LOCATION
  {
    CENTER       = 0,
    CORNER       = 1
  };  // Define one

#endif
```

purge.h

```
/** Adapted by Paul Wolf for CPSPECT, 2012 */
/
***************************************************************************
****
                          purge.h  -  description
                             -------------------
    begin                : Thu Jun 20 2002
    copyright            : (C) 2002 by Christian Wietholt
    email                : christian.wietholt@marquette.edu

***************************************************************************
****/

/
***************************************************************************
****
 *
*
 *   This program is free software; you can redistribute it and/or
modify  *
 *   it under the terms of the GNU General Public License as published
by  *
 *   the Free Software Foundation; either version 2 of the License, or
*
 *   (at your option) any later version.
*
 *
*

***************************************************************************
****/

//: :purge.h
// Delete pointers in an STL sequence container
#ifndef PURGE_H
#define PURGE_H
#include <algorithm>
```

```
template<class Seq> void purge(Seq& c)
{
      typename Seq::iterator i;
      for(i = c.begin(); i != c.end(); i++)
      {
            delete *i;
            *i = 0;
      }
}

// Iterator version:
template<class InpIt>
void purge(InpIt begin, InpIt end)
{
      while(begin != end)
      {
            delete *begin;
            *begin = 0;
            begin++;
      }
}
#endif // PURGE_H ///:~
```