Politecnico di Milano
Facoltà di Ingegneria dei Sistemi
Corso di Studi in Ingegneria Fisica

DESIGN AND DEVELOPMENT OF A TOMOGRAPHIC LIBRARY WITH PHYSICAL CORRECTIONS FOR QUANTITATIVE ANALYSIS

Relatore interno: Prof. Giacomo Claudio Ghiringhelli

Correlatore: Dr. Vicente Armando Solé

Tesi di Laurea Specialistica di:

Nicola Viganò

Matr. Nr. 725109

Anno Accademico 2010-2011

"Talk is cheap. Show me the code."

Linus Torvalds

Contents

List of Figures

List of Listings

Abstract (Italiano)

Questo lavoro di tesi tratta il design e l'implementazione di una libreria per la ricostruzione tomografica, basata sulle Tecniche di Ricostruzione Algebrica (ART, acronimo della dicitura inglese), con l'aggiunta di un set peculiare di correzioni.
Queste correzioni sono incentrate sull'introduzione di alcuni processi fisici nel contesto delle tecniche ART, per migliorare la qualità di ricostruzione, e possibilmente portare anche a risultati quantitativi.
Le nozioni matematiche, le scelte implementative, e gli esempi di codice riportati in questo testo, sono la base per la libreria FreeART, la quale è una libraria tomografica sviluppata da zero dall'autore della tesi. Lo sviluppo è avvenuto dall'inizio di ottobre 2010 alla fine di marzo 2011 presso ESRF, Grenoble, Francia.
Questo testo può esser sia considerato un manuale per la comprensione di FreeART, che un tutorial su come scrivere un codice ART con correzioni fisiche.
FreeART è una libreria open source ed è distribuita con una licenza LGPLv2+, il che la rende estremamente utile per le persone interessate ad imparare come scrivere uno strumento di ricostruzione tomografica ART, o ad usarla liberamente.
I sorgenti sono liberamente scaricabili dal sito del progetto:

`https://forge.epn-campus.eu/projects/freeart`

Dove possono anche essere reperite informazioni sul prodotto ed istruzioni per il suo utilizzo.

Nicola Viganò

Abstract (English)

This thesis work is about the design and the implementation of a tomographic reconstruction library based on the Algebraic Reconstruction Techniques (ART), with the addition of a peculiar set of corrections.
These corrections try to introduce some physical processes into the ART framework, in order to improve the reconstruction quality, and possibly deal quantitative results.
The mathematical background, the implementation choices, and the code samples reported in this text, are the basis for the FreeART library, which is a tomographic library started from scratch and developed by the author, from the beginning of October 2010 until the end of March 2011 at the ESRF, Grenoble, France.
This text can be considered both a handbook for the library, and a tutorial on how to write an ART code, with physical corrections.
FreeART is an open source library and is distributed under a LGPLv2+ license, which makes it extremely useful for people interested in learning how to write a Simultaneous ART tool, or in using it freely.
The sources can be downloaded from the website of the project:

`https://forge.epn-campus.eu/projects/freeart`

Where it is also possible to find information about the product and user guides.

Nicola Viganò

Estratto della tesi (Italiano)

Il testo è strutturato in sei capitoli, i quali danno sia un'immagine di quelle che erano le soluzioni preesistenti, sia spiegano i risultati e nuovi concetti che emergono da questo lavoro.

Il capitolo 1, che comunemente introduce il lettore all'argomento da un punto di vista più fenomenologico, comincia con un'introduzione alla tomografia oltre che alle ragioni che portano all'utilizzo dei raggi X come strumento privilegiato. Prosegue poi introducendo le particolarità più interessanti della fisica della radiazione X, insieme alle sue fonti più comuni.

Nel capitolo 2, vengono presentati al lettore i metodi tomografici. Vi si trova infatti sia una breve comparativa delle loro differenze, che i principi matematici del semplice algoritmo ART.

Le cose cambiano poi nel capitolo 3, dove le correzioni fisiche sono introdotte nello schema delle tecniche ART. Tre differenti set-up sperimentali sono considerati, e vengono poi discusse le approssimazioni necessarie per inserirli nell'implementazione algoritmica.

Alcuni selezionati estratti del codice di FreeART sono mostrati e commentati nel corso del capitolo 4. Questo capitolo è indubbiamente il più lungo e spiega i principali algoritmi implementati nella libreria, i quali, tutti insieme, rendono possibile la ricostruzione dei dati sperimentali. Dal momento che il codice C/C++ riportato è estremamente orientato alle prestazioni (per un'esecuzione sequenziale su CPU), potrebbe risultare molto difficile da seguire, di conseguenza sono anche riportate le tecniche di ottimizzazione utilizzate nella scrittura del codice.

Nel capitolo 5, sono mostrate le altre importanti caratteristiche di FreeART. Viene infatti spesa molta attenzione all'interfaccia python, la quale rende possibile il suo

utilizzo sia in semplici e veloci scipt, o in ricostruzioni automatizzate di grandi dataset.

In fine, nel capitolo 6, vengono riportati alcuni risultati d'esempio, ottenuti con la libreria, e vengono presentate le caratteristiche, i punti di forza e gli svantaggi di questa libraria. I risultati sono sia valutati dal punto di vista della qualità dell'immagine ricostruita, che confrontati con gli stessi risultati ottenuti dai codici di tomografia preesistenti.

Chapter 1

Introduction

X-rays were discovered in late nineteenth century, and since then they have been successfully used in visualizing the internal structure of thing in a non-invasive way. In the 1970s, X-ray computed tomography (CT) was developed. The advantage of CT over regular radiography is that it provides a complete 3-dimensional representation of the object, instead of 2-dimensional projections.
Medical world is the main field that is know to use intensively CT, but it is also becoming more and more used in industrial and scientific applications, since it proved to be a valuable tool for various research fields.
It is now possible to image objects with a resolution of less than one micrometer, thanks to the current high resolution systems.

1.1 Tomography

The idea behind this technique, as opposed to radiography, is to acquire a series of projection images at different angles of the scanned object. This can be achieved in the medical world rotating the source and detector around the patient or in facilities like synchrotrons rotating the sample itself.
The first mathematical formalization of Tomography came in 1917, thanks to the work of Johann Karl August Radon, who introduced the Radon transform. This transform could be used to obtain the reconstruction of an object based on its projection data.

Tomography, however, became of any use just later, with the advent of computers, and the first CT scanner was developed in 1972 by Sir Godfrey Hounsfield, an English electrical engineer, who based his work on the papers published in 1963 and

1964 by Allan MacLeod Cormack, an American physicist.
Apart from the medical world, X-Ray CT was also quickly introduced in industrial applications for quality control and non-destructive testing.
The development of CT with a resolution within the micrometer scale quickly caught the attention of researchers from a wide variety of fields such as biology, pharmacy, engineering, palaeontology, geology and many more.
For what concerns my thesis work, the intended application was exactly the tomographic reconstruction in the high resolution domain, but still it can be applied to macroscopic samples without loss of functionality.

Nowadays, other kinds of tomography do exist. Different probing techniques have been developed, based on ultra-sounds or regions of the electromagnetic spectrum away from the X-rays. What makes X-rays so special about tomography is clearly their long penetration length in materials. An example can be seen in figure 1.1, which shows the penetration length dependence on wavelength in silicon. In the region of X-rays, around $25keV$, there is a peak in penetration length that is from 10 to 100 times longer than the one for visible light.
However, tomography in biological tissues, by means of visible light, is possible but just on very thin layers: not only because of the reduced penetration length, but also because light is subject to a much greater diffusion.
So it is now important to introduce some important concepts about X-rays that will be needed to better understand dynamics and shortcomings of computed tomography.

1.2 Introduction to X-Ray Physics

X-rays are a form of highly energetic electromagnetic waves or photons (wave-particle duality). The energy E of such photons is related to their wavelength λ or frequency ν by the well known equation:

$$E = \frac{hc}{\lambda} = h\nu \tag{1.1}$$

where h is Planck's constant, equal to $4.136 \cdot 10^{-15} eVs$ and c the speed of light

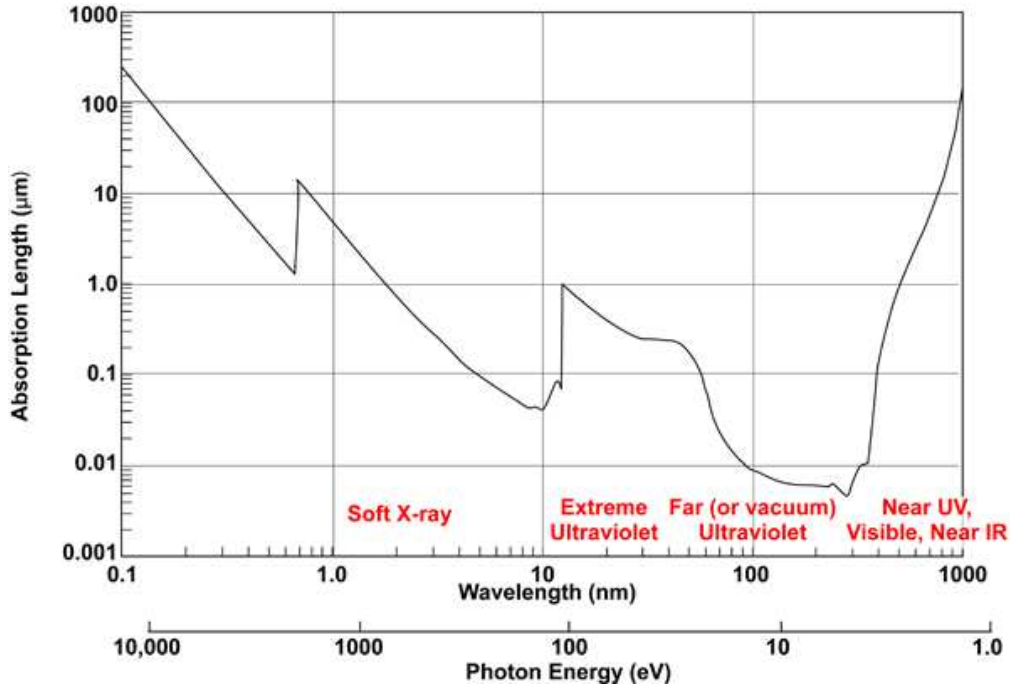


Figure 1.1: Light absorption length in Silicon (Source: MIT High-Sensitivity Sensors[1])

in vacuum, equal to $3 \cdot 10^8 m/s$. Nowadays, the difference between X-rays and γ-rays is just related to the way they are produced: X-rays are defined as electromagnetic radiation emitted by charged particles, while γ-rays are emitted by the nucleus in processes of radioactive decay or are created in annihilation processes.
In this thesis work, the distinction between the two was never took into account, because the algorithm is independent of such phenomenological differences.

There is now the need to briefly introduce some aspects of the interaction of X-rays with a medium, first using a particle-like model and then a wave-like model.

1.2.1 Scattering processes

There are different scattering processes that X-ray photons can undergo in the interaction with matter:

Photo-electric absorption: A photon can transfer all of its energy to a shell electron, ejecting it from the atom. This interaction is only possible when the energy of the photon is higher than the binding energy of the electron. The remaining energy

is converted into kinetic energy of the ejected electron. The interaction probability for photo-electric absorption τ can, for the typical energies encountered in X-ray CT (5 to 150 keV), be approximated by:

$$\frac{\tau}{\rho} \propto \left(\frac{Z}{E}\right)^3 \tag{1.2}$$

where ρ is the mass density, Z the atomic number of the element and E the energy of the photon.

Compton scattering: A photon can also interact with an atomic electron, transferring to it some of its energy. This event can be considered similar to an elastic collision between two particles, because X-ray photons begin to have a significant momentum, and therefore can undergo this kind of interactions.
Since the photon leaves the interaction site in a different direction and energy, this interaction is classified as an inelastic scattering process, despite the fact that the collision is considered elastic.
Compton scattering in the object can be an undesirable effect in transmission tomography, as some of the deviated photons reach the detector, making the object a radiation scatterer, which may distort the image quality. In other cases this process exploited to compute the density of the sample.
The interaction probability for Compton scattering σ can be described by:

$$\frac{\sigma}{\rho} \propto \left(\frac{Z}{A}\right) f(E) \tag{1.3}$$

where A is the mass number and $f(E)$ is an energy dependent factor that can be calculated using the Klein-Nishina formula.

Rayleigh scattering: In this process, the photon is scattered by the whole electron cloud instead of a single electron. This is an elastic scattering process, in which no energy is transferred. Although this interaction occurs at low energies and results in relative large scattering angles, it generally poses no significant contribution.
This is the basic scattering event, that gives rise to diffraction patterns, through which it's possible to determine the crystallographic orientation of the sample.

Higher energies scattering events: There are then other processes that just have a reasonable cross-section at higher energies, that usually are not reached in CT. These processes are *pair production* (above $1.022MeV$), and *nuclear reactions* (at even higher energies than *pair production* event).

1.2.2 Propagation through the media

Due to the wave-particle duality, all photons, including X-rays, are subject to wave-related effects. Every wave can be described by its amplitude, wavelength (or frequency) and phase. The interaction of a wave with a medium is determined by the complex refraction index $n = 1 - \delta + i\beta$, where δ is responsible for the *attenuation* of the wave and β for the *phase shift*, which is due to a difference in propagation speed between the medium and vacuum.

Phase shift: It causes a deformation of the wave-front as the parts of it traversing the medium move at a different speed. When the studied features are much larger than the wavelength, the ray optical approach can be used to represent the waves. Using this approach, each part of the wave-front can be represented by a ray perpendicular to the wave-front.
The introduction of a phase shift can then be seen as a change in the direction of the incoming ray when it goes from one medium to another. The effect is called *refraction*. In addition, at distances further away from the medium, interference between the original and the deformed wave-front results in a complex pattern of intensities, called *diffraction*, for which the ray optical approach is no longer valid. Both refraction and diffraction effects are inevitably encountered in high resolution X-ray imaging systems and should be appropriately accounted for.
Thankfully these variations in phase are usually just small corrections and can be easily neglected, without the introduction of sensible errors.

Attenuation: For a ray passing through the media, it causes a decrement in the ray intensity.
Let's consider a monochromatic X-ray beam of intensity I, which is proportional to

the number of photons per unit time and unit area, and an infinitesimally thin slab of thickness ds. The slab consists of a material with linear attenuation coefficient $\mu = \tau + \sigma$, which combines both contributions of photo-electric absorption and Compton scattering. The change in intensity of the beam after passing through the slab is then given by:

$$\frac{dI}{ds} = -\mu I \tag{1.4}$$

Integrating this along the path L from the source to the detector position yields the law of Lambert-Beer:

$$I = I_0 \cdot \exp\left[-\int_L \mu\,(s)\,ds\right] \tag{1.5}$$

where I_0 is the unattenuated beam intensity and where the linear attenuation coefficient $\mu(s)$ depends on the material composition at position s along the path L. Even though the individual interactions of photons with matter are of statistical nature, the macroscopic intensity of the beam can thus be described using a deterministic exponential law.
Note that this formula, which is the basic equation in CT, is only valid for attenuation processes and for a monochromatic beam. Some of the most important artefacts that arise in high resolution CT are due to a violation of these conditions.

1.3 X-Ray Sources

Usually there are two different kinds of X-ray sources: X-ray Tubes, and Synchrotrons. In this section there will be a brief explanation of how these sources work, and what are their key features.

1.3.1 X-ray tubes

The principle behind X-ray tubes is quite simple. As we can see in the image 1.2, an electric current is used to heat a filament, the cathode (C), which emits electrons due to the thermionic effect. These electrons are accelerated in a vacuum tube towards

a target plate, the anode (A), by applying a high voltage U_a between cathode and anode, resulting in a current. These fast electrons collide with the target and deposit their energy in it. A small amount of the deposited energy is used to generate X-rays, which emit from the target and escape the tube through an exit window. The rest of the energy is released as heat.

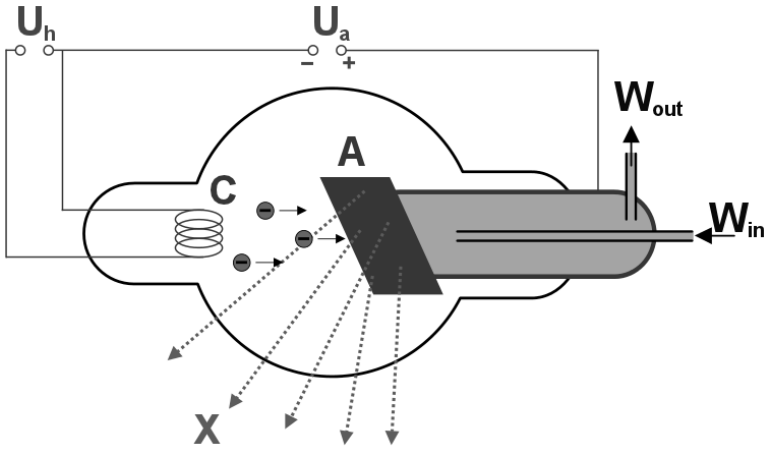


Figure 1.2: Coolidge tube, also called hot cathode tube (Source: Wikipedia[2])

By using electromagnetic lenses, the electrons can be focused onto the target such that the X-rays are generated within a small area of the target, which is called the focal spot of the tube. Since the spot heats up due to the energy deposit of the electrons, the electron current needs to be limited to prevent the target from melting. A smaller spot size thus requires a lower current, which implies a decrease in the number of photons that are generated in the target, thus a lower X-ray flux. Depending on the design, the target plate can be cooled which allows for a higher electron current and X-ray flux.

When an electron beam strikes the target material, X-rays can be created by two processes:

Characteristic radiation: An incoming electron can collide with a shell electron, transferring a part of its energy to the stuck electron, which is dissipated into heat. The majority of the incoming electrons interacts by this process, which accounts for the heating of the target. A fraction of these collisions results in the removal of the

shell electron, leaving a hole in the shell. This gap is immediately filled by a higher shell electron dropping into the hole while emitting a photon of a specific energy. The energy of the photon is well-defined and equal to the difference in energy between the two electron states, yielding a characteristic peak in the emitted spectrum.
This kind of radiation is also called *Fluorescence*, and every chemical element has its own characteristic fluorescence spectrum.
The same scattering process can happen also in the sample, so it can be used to obtain a map of the sample's chemical composition.

Bremsstrahlung: An electron can also interact with the nuclei of the target material by Coulomb interaction, losing a significant amount of energy by emitting a photon. Bremsstrahlung yields a continuous X-ray spectrum, where the energy of the emitted photons lies between 0 and $E_{max} = qU$, with q the electric charge of an electron and U the high voltage of the tube.

1.3.2 Synchrotrons

In a synchrotron, charged particles (usually electrons) are accelerated to very high energies and injected in a quasi-circular storage ring, consisting of straight sections and bending magnets. When a relativistic particle is deflected from its path by a magnetic field, it loses some of its energy by emitting high energy photons. When it was first observed, synchrotron radiation was seen as a nuisance, causing unwanted energy loss. Soon afterwards however, it was found that this radiation can be very useful in numerous experiments, and hence now it is deliberately produced.
In the first generations of synchrotrons, the electromagnetic radiation is only created by the bending magnets, creating radiation with a wide spectral distribution. To obtain monochromatic photons, crystalline monochromators are added to the experimental set-up to extract a monochromatic beam, resulting in a decrease in flux.
In the current third generation synchrotrons, devices are installed in the straight sections which are specially designed to produce higher photon fluxes and/or monochromatic radiation. These insertion devices consist of a number of periodically po-

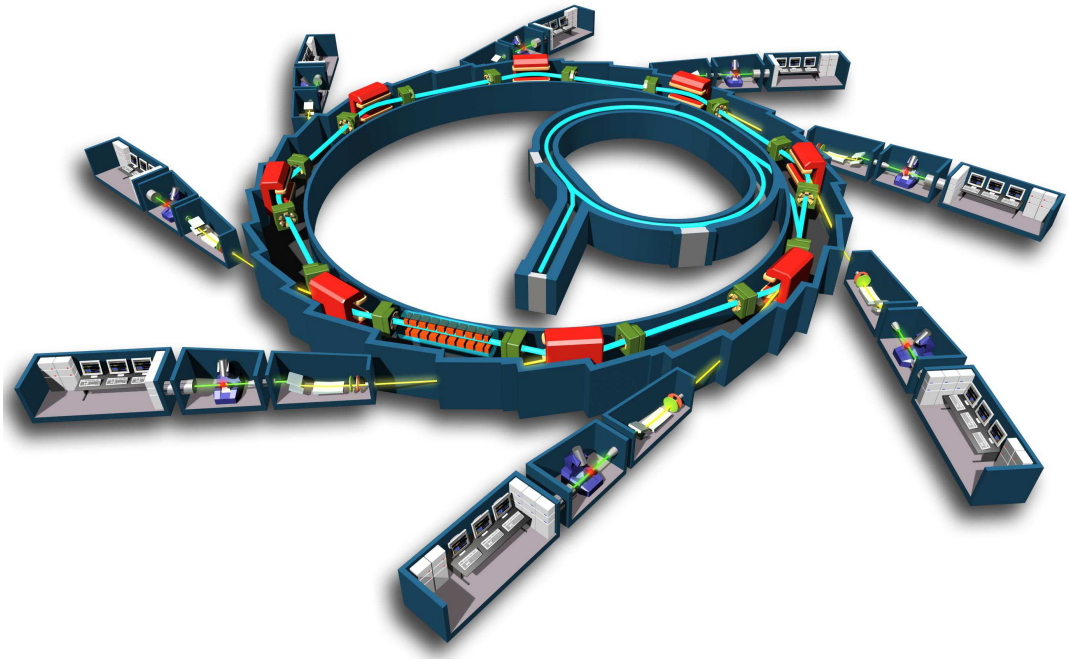


Figure 1.3: Scheme of a synchrotron (Source: Wikipedia[3])

sitioned magnets, forcing the electrons to following a sinusoidal trajectory. Two types of magnetic devices are commonly used: undulators, which produce an almost monochromatic photon beam, and wigglers, which produce a polychromatic beam of high intensity.

Synchrotron radiation offers some unique advantages, making it highly suitable for X-ray tomography. It consists of an almost parallel beam of high intensity, providing sufficient statistical information at relative small scanning times even at a position far away from the source.

The radiation is also spatially coherent, as it originates from a very small area. This allows imaging using wave-related effects, which can be used to increase resolution and contrast, especially in low attenuating materials. Furthermore, synchrotron radiation can be used to produce an almost monochromatic beam with an energy that is tunable to some extent.

The downside is that synchrotron installations are quite expensive and accessibility is limited. Furthermore, magnification of the imaged object can only be achieved using complex X-ray optics.

Another important property of the synchrotron radiation is related to its *Brilliance*. This property gives an estimation of the quality of the beam. It is defined as:

$$\text{Brilliance} = \frac{\text{Photons/second}}{(\text{mrad})^2\,(\text{mm}^2\text{ source area})\,(0.1\%\text{ bandwidth})} \tag{1.6}$$

and as can be seen from 1.6, it is a figure-of-merit that includes the photon flux, the divergence of the beam, the dimension of the source area, and the monochromaticity of the light itself.

Brilliance of synchrotron light is usually way much superior to the one of other X-ray sources.

Chapter 2

Computed Tomography

In this chapter, there will be first a brief introduction to the techniques available nowadays, and then the Algebraic Reconstruction Techniques will be explored in more details, which are at the basis for the reconstruction algorithms used in FreeART.

The treatment of these techniques is, for simplicity, in the case of transmission tomography for a 2-dimensional object. Let's also do the assumption that data has been previously normalized in such a way that every value in the input data can be considered a line integral.
In practice, assigning to $f(x, y)$ the meaning of the real quantity for the point (x, y), which in this case is the linear attenuation coefficient μ, so every value $P_\theta(t)$ from the input will be given by:

$$P_\theta(t) = \int_L \mu(x, y)\, ds \tag{2.1}$$

where θ is the angle of the projection, t is the offset of the given ray related to the point in the input data, L is the path of the given ray through the object which is given by the relation:

$$x \cos\theta + y \sin\theta = t \tag{2.2}$$

Making now the supposition that the incoming beam is monochromatic, and calling the incoming beam intensity $I_\theta^0(t)$, and $I_\theta(t)$ the intensity on the detector, the transmission is given by the equation 1.5, which becomes:

$$I_\theta(t) = I_\theta^0(t) \cdot \exp\left[-\int_L \mu(x, y)\, ds\right] \tag{2.3}$$

where t stands again for the considered ray in the projection θ.

From 2.3 we can obtain the input in the desired format for the algorithms:

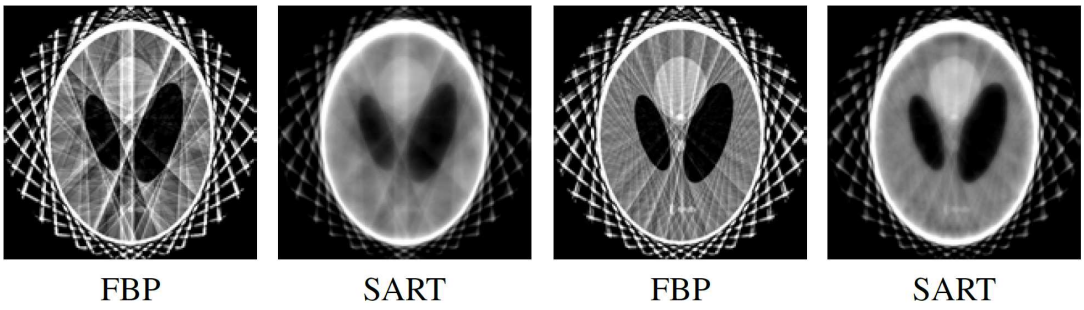
$$\int_L \mu(x,y)\,ds = -\ln\frac{I_\theta(t)}{I_\theta^0(t)} \tag{2.4}$$

2.1 Techniques and their Differences

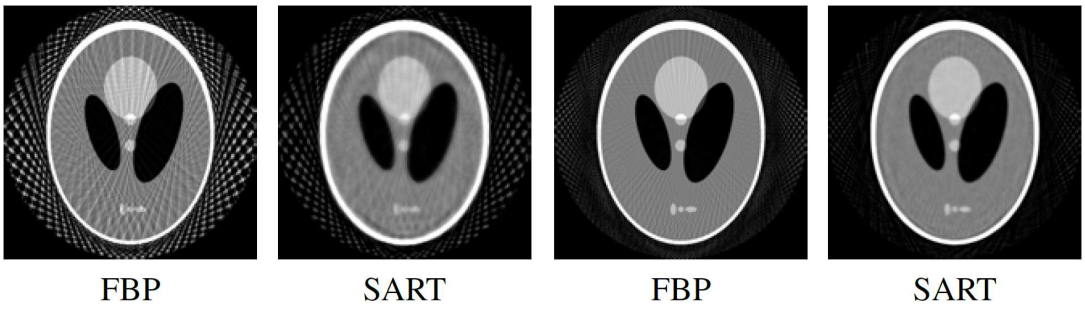
In CT there are mainly two different classes of reconstruction techniques: *Filtered Back Projection* and *Algebraic Reconstruction Techniques*.

The most used is the first one: it is the fastest and gives very nice results when many projections are available.

The second, instead, is usually slower but tends to perform better in many ways: noise is lower when few projections are available or when some angles are missing, and gives the possibility to introduce many kinds of different corrections in an easy and straight-forward way.



(a) 16 projections (b) 32 projections



(c) 64 projections (d) 128 projections

Figure 2.1: Comparison of reconstruction quality between ART and FBP, with an increasing number of projections (Source: *De Witte*[4])

It is possible to see from figure 2.1, how ART and FBP algorithms perform with few projections. The fact that the ART family performs better than the FBP technique, can be appreciated quite easily.
However the difference is more evident when the number of projections is very low. With the increase of projections number, the difference becomes less visible.
A similar comparison, but with missing angle ranges instead, can be found in literature.

2.1.1 Filtered Back Projection

This method is based on the analytical reconstruction techniques, and many algorithms have been developed to cope with different acquisition set-ups. As a result, as an example, it is possible to find methods for parallel-beam, cone-beam, fan-beam and helical-scan set-ups. Some basic concepts about FBP will now be discussed, but without too many the details, because this class of reconstruction techniques is not implemented in FreeART. There will also be the assumption of parallel-beam, without helical scan.
The basic concept behind analytical techniques is the Fourier Slice Theorem [6]:

> The 1D Fourier transform of a parallel projection of a 2D object function $f(x, y)$ at an angle θ with respect to the X-axis, gives a slice of the 2D Fourier transform $F(u, v)$ of the function $f(x, y)$ at an angle θ with respect to the u-axis.

This means that from a projection, thanks to their Fourier transforms, we obtain objects in a 2D space, that when anti-transformed back, will give us the original object. This works well in theory, but in practice it has some flaws. An example is the sparsity of the grid far from the centre can give bad results on the higher frequencies, because the grid is interpolated in the anti-transform.
Filtered back projection overcomes these limits introducing a filter in the 2D Fourier transform [4].

2.1.2 Algebraic Reconstruction Techniques

The main idea behind this class of methods is to divide the object to reconstruct into voxels (or pixels if in 2D) that are represented by the quantity $f(x, y)$ as explained just at the beginning of this chapter.
So there will be a set of values f_i with $i = 1 \dots N$ where N is the number of voxels, and another set of values p_j called *ray sums*, which are the signals measured by the detector for each ray passing through the sample. In the latter case, $j = 1 \dots M$ where M is the total number of rays in all the projections.
The relationship between these two quantities can be expressed as a system of linear equations:

$$\sum_{j=1}^{N} w_{ij} f_j = p_i, i = 1 \dots M \tag{2.5}$$

where w_{ij} are weights that relate every ray to every voxel. In practice the matrix W of the weights will be an highly sparse matrix. The ray sums are instead the line integrals seen before 2.4:

$$p_i = -\ln \frac{I_i}{I_i^0} \tag{2.6}$$

So the equations 2.5 can be seen explicitly as a system of linear equation that reads out:

$$\begin{aligned} w_{11} f_1 + w_{12} f_2 + \cdots + w_{1N} f_N &= p_1 \\ w_{21} f_1 + w_{22} f_2 + \cdots + w_{2N} f_N &= p_2 \\ \vdots &= \vdots \\ w_{M1} f_1 + w_{M2} f_2 + \cdots + w_{MN} f_N &= p_M \end{aligned} \tag{2.7}$$

That can be written more compactly:

$$\mathbf{W} \cdot \mathbf{F} = \mathbf{P} \tag{2.8}$$

This is very elegant and nice, but really difficult to manage. This system has usually infinitely many solutions, and would result impossible because of the measurement error. So the usually preferred technique is an *iterative* approach.

2.2 ART in details

Since the core algorithm of FreeART, like the name suggests, is of the family of ART methods, discussion about how Algebraic Techniques work, will proceed in a deeper detail.

2.2.1 Mathematical model: Kaczmarz method

The algorithm that dives the basis for the iterative procedures is the *Kaczmarz method*. The 2-dimensional object to be reconstructed is represented by a vector $\vec{f} = (f_1, f_2, \dots f_N)$ in a N-dimensional vector space. For every equation in 2.7 there is a different hyperplane in such vector space. If a unique solution exists, they do intersect in a unique point, which is the solution to the system of linear equations 2.8.

To understand easily the way this algorithm operates, let's reduce the system to a smaller dimension, with number of dimensions and equations $M = N = 2$:

$$\begin{aligned} w_{11}f_1 + w_{12}f_2 &= p_1 \\ w_{21}f_1 + w_{22}f_2 &= p_2 \end{aligned} \tag{2.9}$$

These two equations are two different lines in a 2-dimensional space. As can be seen in figure 2.2, their intersection is the solution and we reach it iteratively. Starting from an initial guess $\vec{f}^{(0)}$, we obtain the next iteration step $\vec{f}^{(1)}$, projecting it on the line represented by one of the two equations. Then again, the next step $\vec{f}^{(2)}$ is retrieved projecting the previous solution to the other line, and so on. If the solution $\vec{f}$ exists and it is unique, the procedure converges to it.

It is now possible to show the equation in the N-dimensional space:

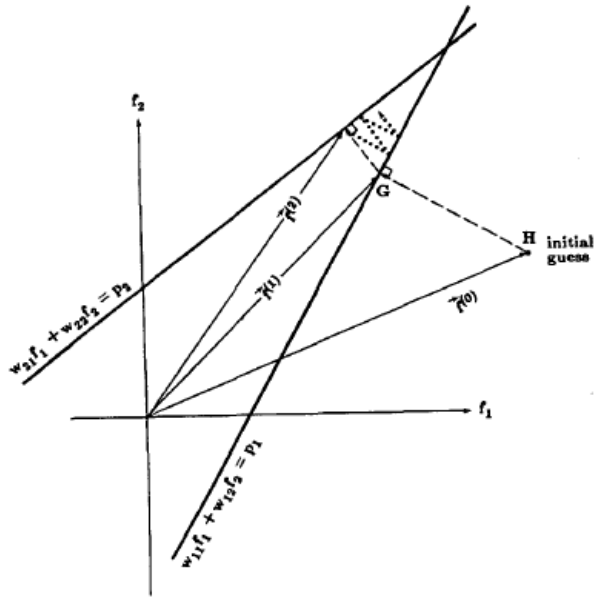


Figure 2.2: 2-dimensional case of Kaczmarz algorithm (Source: *Kak et al.*[6])

$$\vec{f}^k = \vec{f}^{k-1} + \frac{p_k - \vec{w}_k \cdot \vec{f}^{k-1}}{|\vec{w}_k|^2}\vec{w}_k \tag{2.10}$$

where $\vec{w}_k = (w_{k1}, w_{k2}, \ldots, w_{kN})$.

Sadly enough, in real world data, affected by noise, hyperplanes do not intersect in a point, but they form a polyhedron. Once reached an approximate solution in this polyhedron, the algorithm will cycle from one approximate solution to the other, without converging to the real solution.

2.2.2 The ART Method

It is now easy to derive the base iterative formula for Algebraic Techniques, modifying 2.10 in this way:

$$\Delta f_j^{(m)} = f_j^{(m)} - f_j^{(m-1)} = \lambda \frac{p_i - q_i^{(m-1)}}{\sum_{n=1}^{N} w_{in}^2} w_{ij} \; ; \qquad q_i = \sum_{n=1}^{N} w_{in} f_n \tag{2.11}$$

where λ is a *damping factor*, a scalar in the range $[0, 1]$, which helps to reduce the introduction of noise. Everything else stays the same, made exception for the fact that it was translated from vector notation to explicit vector components notation.

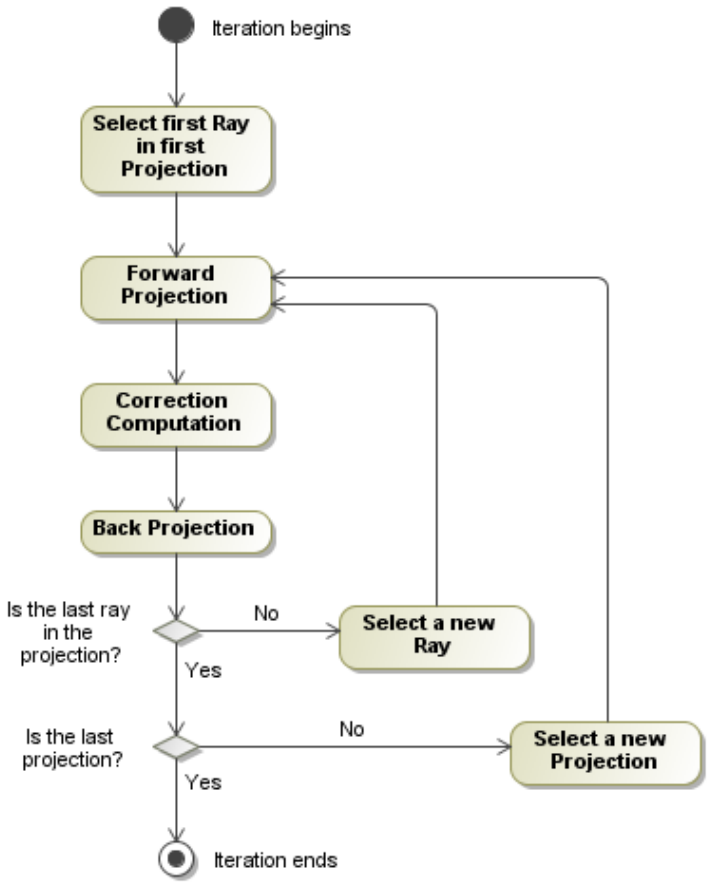


Figure 2.3: Flow diagram of one iteration of the ART algorithms

Let's now describe the three main steps in applying this formula. The first step is to compute the *ray sum* for the given ray:

$$q_i = \sum_{n=1}^{N} w_{in} f_n \tag{2.12}$$

that is sometimes called *forward projection* or *re-projection* of the current solution.

Then it comes the *correction computation*:

$$c_i = \lambda \frac{p_i - q_i^{(m-1)}}{\sum_{n=1}^{N} w_{in}^2} \tag{2.13}$$

which, in this phase of the algorithm, is the same for every voxel reached by the ray i.

Finally it comes the *back-projection* step, in which the corrections are applied to the related voxels, weighted by the factors w_{ij}, and the solution is updated:

$$f_j^{(m)} = f_j^{(m-1)} + w_{ij}c_i \tag{2.14}$$

This procedure is so applied for every voxel reached by the ray i, for every ray in the projection P_θ, and for all the projections.
Once operated the procedure over all the projections, an iteration of the algorithm will be done. This can be summarized with a flow diagram, like the one reported in picture 2.3.
It is then possible to do as many iterations as necessary to converge. This may be a direct requirement of the possibility that for different damping parameters λ, convergence can be reached with varying number of iterations.

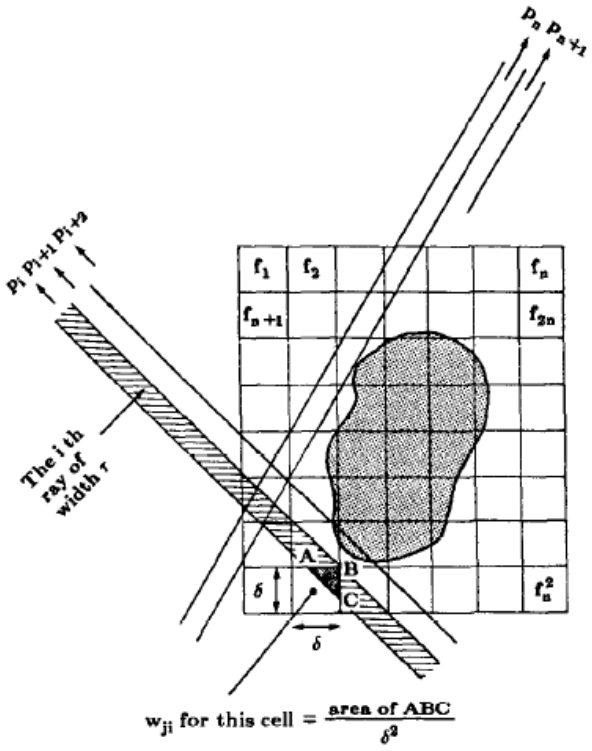


Figure 2.4: ART: Strip-based ray implementation (Source: *Kak et al.*[6])

2.2.3 Implementation details

A not trivial part in the implementation of an ART code is the way the weights w_{ij} are computed.

Strip-based: One of the most accurate ways to compute them can be seen in image 2.4, and it is to think about the rays as strips of given width r, and the weights as the intersection areas between the rays and the voxels, with edges of given length δ.
This implementation, however, apart from transmission tomography, is very difficult to work with. It will be clear in the next chapter, where the physical corrections will be introduced, that to apply some of them it is mandatory to order the voxels on the direction of a given ray. This is quite difficult with such an approach, while it is totally straight-forward in a sampling based implementation.

Sampling-based: This quite simple implementation keeps the ray as an infinitesimal width line, and samples the voxels grid every fixed-length interval. The length of the interval, and the alignment of the sampling points in respect to the voxels' centres are the most important things to care about in this implementation.

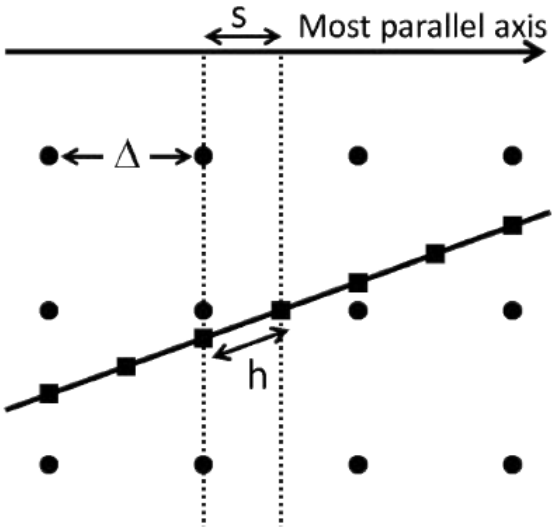


Figure 2.5: ART: Joseph's sampling-based ray implementation (Source: *De Witte*[4])

An interesting implementation, which can be seen in image 2.5, is based on

Joseph's algorithm [12]. The length interval, on the most parallel axis to the ray, is given by the fraction of the distance δ between two adjacent voxels, and an integer number S called *oversampling factor*.
The alignment is to the centre of the voxels on the most parallel axis to the ray, so that every S sampling points, the "$S+1$ point" is again aligned to the centre of a voxel.

Sampling is then performed through linear interpolation of the closest voxels. In 2D, four voxels will be sampled in a bilinear interpolation, while in 3D eight voxels will be sampled in a trilinear interpolation.
In this implementation a slightly meaningful modification to the basic ART formula 2.11 needs to be done, in order to reflect the introduction of both the interpolation and the summation done on the sampling points:

$$\Delta f_j^{(m)} = f_j^{(m)} - f_j^{(m-1)} = \lambda \frac{p_i - q_i^{(m-1)}}{\sum_{k=1}^{K} \sum_{n=1}^{N} (w_{ikn})^2} w_{ij} \; ; \qquad q_i = \sum_{k=1}^{K} \sum_{n=1}^{N} w_{ikn} f_{kn} \quad (2.15)$$

where the summation $\sum_{k=1}^{K}$ is over the sampling points, and the weights w_{ikn} now relate the voxel n to the sampling point k in the ray i.

The introduction of some corrections to the reconstruction algorithm, which are based on physical processes, is quite simple and straight-forward.
An easy but effective modification to the formula 2.11 can be done introducing a correction term K_{in} that depends both on the ray i and the voxel n [7]:

$$\Delta f_j^{(m)} = f_j^{(m)} - f_j^{(m-1)} = \lambda \frac{p_i - q_i^{(m-1)}}{\sum_{n=1}^{N} K_{in} \left(w_{in}\right)^2} w_{ij} \; ; \qquad q_i = \sum_{n=1}^{N} w_{in} K_{in} f_n \tag{3.1}$$

A similar expression can also be deduced for the sampling-based formula. Starting from this derived expression, the author introduced physical corrections to ART algorithms in a systematic and rigorous way. This approach was never applied before to such a wide number of problem at the same time, so an important feature of this thesis work is that it gives a common pattern to introduce physical corrections, and documents all the needed steps to do it.
Moreover, in this text, the author introduces and explains his own specific approximations for the different physical cases, which are needed in order to make the elegant mathematical formulation easily translatable to the algorithmic form that computers can elaborate, while preserving physical consistency.
Let's now discuss in detail the term K_{in} for the different physical set-ups.

3.1 Physical Models for the Experimental Data

To show how the correction term K can be computed, first of all, it is needed a change from the implicit notation based on the summation over the voxels, to the physical system of reference given by the ray and the centre of rotation of the sample.

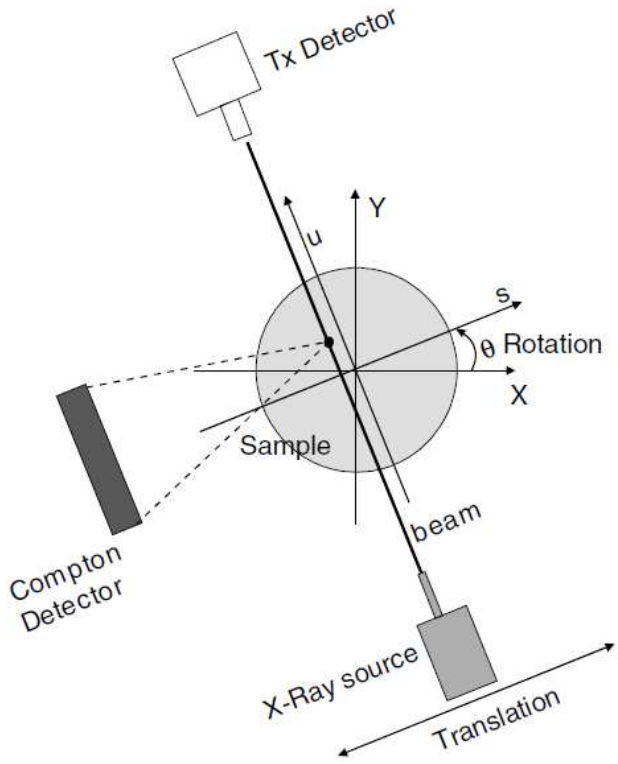


Figure 3.1: Scheme of a Compton experiment (Source: *Golosio et al.*[7])

As can be seen in picture 3.1, let u be the axis parallel to the rays for a given projection P_θ, and s the axis perpendicular to u and that measures the offset of a given ray i.
So in the physical cases considered in my code, K can be expressed as product of correction functions:

$$K_{in} = h_i(s,u)\ \ g_i(s,u) \tag{3.2}$$

In this case $h_i(s,u)$ is the function that takes into account the attenuation of the incoming ray, while $g_i(s,u)$ represents the attenuation of the scattered radiation.
So h_i is the well known linear attenuation formula:

$$h_i(s,u) = \exp\left(-\int^{u} \mu(s,u',E_0)\,du'\right) \tag{3.3}$$

which is a line integral from the origin of the ray, to the scattering point (s,u), and where μ is the linear coefficient of every point in the sample for the given energy E_0 of the incoming ray.

While the function h_i is quite simple to implement, the function g_i, which accounts for the self-absorption, can be really difficult to deal with. It also varies from physical set-up to physical set-up.

3.2 Approximations for Self-Absorption

Different experimental set-ups can be expressed by different self-absorption formulas. Those different terms depend on the physical origin of such signals and for a complete and instructive treatment, the reader is suggested to look for further explanations and details on the specific literature.

3.2.1 Compton

The self-absorption term for the set-up in figure 3.1 is:

$$g_i(s,u) = \int_{\Omega_D} d\Omega \exp\left(-\int_{(s,u)}^{Det} \mu(l,E')\,dl\right) \tag{3.4}$$

which can be nearly impossible to compute, unless we perform some approximations. A first important approximation is based on an experimental fact. Compton detector is nothing else than the Fluorescence detector. Its area is much smaller than the distance from the sample and the detector itself. A rough estimation tells us that the divergence angle seen from a point source located wherever in the sample to the detector surface is $\sim 0.5°$.

This makes it possible to safely use the parallel beam approximation, and so to decouple the solid angle integral from the attenuation exponential. Equation 3.4 then becomes:

$$g_i(s,u) = \Omega \exp\left(-\int_{(s,u)}^{(s_d,u_d)} \mu(l,E')\,dl\right) \tag{3.5}$$

where (s_d, u_d) is now the centre of the detector. The formidable integral on a conic volume, from a point source to the area of the detector, is now a simple line integral from a point source to the centre of the detector. The integral on the solid angle becomes simply the solid angle covered by the detector from the source point.

Even if this approximation makes the implementation a lot easier, another major problem is still present. The linear attenuation coefficient is referred to the scattered energy E', and we don't know such attenuation coefficient *a priori*.
To solve this problem, is again useful to refer to the typical experimental values. A complete treatment of Compton scattering can easily be found in literature, but it convenient to report the equation that relates the wavelength-shift $\Delta\lambda$ to the angle θ [8]:

$$\Delta\lambda = \lambda_{emitted} - \lambda_{incoming} = \lambda_C \left(1 - \cos\theta\right) \tag{3.6}$$

where:

$$\lambda_C = \frac{h}{m_0 c} = 2.43 \cdot 10^{-12} m \tag{3.7}$$

In a Compton scattering experiment, the detector is usually at 90 degrees from the incoming ray, so we have a λ-shift of exactly λ_C. Since the usual X-ray energies are around $25KeV$, the wavelength associated, inverting equation 1.1, then reads $5.1648 \cdot 10^{-11}$: which is more than twenty times the shift. This leads me to say that we can take $E' \sim E_0$ and $\mu\left(l, E'\right) \sim \mu\left(l, E_0\right)$, so the equation 3.4 finally becomes:

$$g_i\left(s, u\right) = \Omega \exp\left(-\int_{(s,u)}^{(s_d,u_d)} \mu\left(l, E_0\right) dl\right) \tag{3.8}$$

The last equation can now be easily computed once we know the coefficients $\mu\left(l, E_0\right)$ along the line l. This can be achieved thanks to transmission tomography.

3.2.2 Fluorescence

When it comes to fluorescence, things change a bit. The two aspects that remain valid are the basic function for self-absorption 3.4, and the parallel beam approximation, because geometry is unchanged. So the use of 3.5 is legitimate.
What is not legitimate is the approximation on the linear absorption coefficient, because in this case the energies E', of the emitted radiation, are much more shifted from the energy E_0 of the probing beam, than it was in the Compton case.
This time a different approach can be used. All the absorption coefficients for every

incoming radiation of energy E' on every element are known *a priori*. What is not known *a priori*, is the real chemical composition of the sample, and a fluorescence experiment is the right experiment to perform, in order to obtain it. The problem is now just the recursive, or cyclic, dependence between chemical composition determination and self-absorption correction. To solve this problem it is possible to follow a simple iterative procedure, enveloping the iterative reconstruction procedure.

First of all, let's suppose that a previous analysis on the fluorescence data [1], has produced sinograms for the detected emission lines of the elements that compose the sample.

Using those sinograms, a qualitative and approximate reconstruction on the chemical composition of the sample can be performed, and so can be determined an approximate ratio of the chemical composition for every voxel. Information on the density for every voxel can be obtained by the Compton reconstruction. Interesting to notice is that along with fluorescence data, Compton data is always available since they can be detected by the same detector.

It is now possible to write a formula that gives an approximate coefficient for the temporary reconstruction for a given line α in the voxel n:

$$\mu_{\alpha n} = \frac{\sum_\beta \sigma_{\alpha\beta} x_{\beta n}}{\sum_\beta x_{\beta n}} \rho_n \tag{3.9}$$

where $\sigma_{\alpha\beta}$ is the absorption cross-section for the given element β at the energy α, ρ_n is the density for the given voxel n and $x_{\beta n}$ is the temporary value of the chemical concentration of the element β in the voxel n.

Using these approximate absorption coefficients for every line α, it is possible to perform a better fluorescence reconstruction. From that reconstruction is then possible to extract a better estimation to those coefficients by equation 3.9, which would lead to an even better chemical composition determination and so on in a self-consistent manner.

[1] An example of analysis tool is the well known PyMca, dedicated in particular to fluorescence analysis, and developed by V. Armando Solé at the ESRF, Grenoble

3.2.3 Diffraction

The case of diffraction is different from the previous cases: while the linear absorption coefficient is now known, because diffraction is based on an event of elastic scattering, the geometrical part gets more complicated.
The self-absorption term is:

$$g_i(s,u) = \exp\left(-\int_{(s,u)}^{Det} \mu(l, E_0)\, dl\right) \tag{3.10}$$

where now the integral is on the surface of a cone from the source point to a circle on the diffraction CCD. What characterizes the cone is the aperture angle that depends on the Brag condition for both the energy E_0 of the incoming radiation, and the crystal orientation of the voxels in the sample [9].
Since these angles are surely larger than in case of Compton or fluorescence experiments, the parallel beam approximation cannot be done. Moreover the surface of the cone does not remain in-plane, hence a 3-dimensional geometry is required. In such case is possible to sample on the conical surface every fixed angle, and the surface integral becomes simply a sum of line integrals:

$$g_i(s,u) = \frac{1}{D}\sum_{d=1}^{D} \exp\left(-\int_{(s,u)}^{(s_d,u_d)} \mu(l, E_0)\, dl\right) \tag{3.11}$$

where D is the number of line integrals, and (s_d, u_d) is the point on the detector reached by the line d.

So far, just a qualitative discussion about the algorithms and mathematical ideas was done. In this chapter the discussion will now move to the introduction of a more detailed description of those algorithms, analysing the implementation of FreeART and showing some of its code snippets as examples.
Since these code examples will be taken directly from the library, which is released with a LGPL2+ license, the reader is free to act under the terms of the license.
The core of the library is written in C++, and the majority of the examples will be in C++, so a basic understanding of such programming language is required.
Instead of introducing directly the most important algorithms in the library, there will now be the introduction of some classes that serve as a sort of minimal framework in the code, implementing 2-dimensional arrays, or geometric properties, like euclidean 2D vectors.

4.1 Utility classes

The need for a mini-framework in the library is based on the strong requirement for portability. Many extensive and rich frameworks for mathematical programming exist, but are usually big dependencies and put constrains on portability.
The decision to base the data-structures on STL[1], was taken because the STL provides a standard interface and is well tested, and has performance oriented classes.
Before getting to the classes, let's first introduce the reason why two different sizes for floating point numbers are used in the code. Then an example of template geo-

[1]Standard Template Library of C++, nowadays implemented in every stdc++ library that comes along with any recent compiler.

Listing 4.1: Definition of floats in macros.h

```
/* storage float type */
typedef float    float_S;

/* calculus float type */
typedef double   float_C;

/* utility macros for switching between formats */
#define _FT_C( x )     ((float_C)(x))
#define _FT_S( x )     ((float_S)(x))
```

metric class will be treated, and finally a powerful implementation of array objects will be shown in its full glory.

4.1.1 Floating point sizes

From listing 4.1, it appears clear that the code makes mixed use of 32bits and 64bits floats. As can be guessed from the comments that come along with the *typedef*s, the type *float_S* is well suited for storage purposes, while *float_C* should be used in performing floating point operations.

The principle behind this decision is that the larger precision granted by *double*s is only needed when numeric deletion[2], or other rounding errors can happen. This is the case in arithmetic operations, but usually it is safe to store final results in single precision floats, that still guarantee more precision than the experimental techniques do, but consume much less memory than 64bits floats.

4.1.2 Geometric Classes

One of the most interesting and powerful uses of templates can be easily learned from the *Position* class in listing 4.2. It represents the Euclidean vector in two dimensions, and can be used as an index in a 2-dimensional matrix or as a geometric position in a real space representation.

Listing 4.2: Position template in 2D

```
template<typename Type>
struct Position {
```

[2] A basic and subtle source of numeric errors

```
    Type x, y;

    Position() throw() : x(0), y(0) { }
    template<typename Type2>
    Position(const Type2 & _x,const Type2 & _y) throw()
      : x( (Type)_x ), y( (Type)_y ) { }
    template<typename Type2>
    Position(const Position<Type2> & old) throw()
      : x( (Type)old.x ), y( (Type)old.y ) { }

    Type operator*(const Position<Type> & other) const throw() {
      return (this->x*other.x + this->y*other.y);
    }
    template<typename Type2>
    Position<Type> operator*(const Type2 & scalar) const throw() {
      return Position<Type>(this->x*scalar, this->y*scalar);
    }
    template<typename Type2>
    Position<Type> operator/(const Type2 & scalar) const throw() {
      return Position<Type>(this->x/scalar, this->y/scalar);
    }
    template<typename Type2>
    Position<Type> operator+(const Position<Type2> & other) const throw() {
      return Position<Type>(this->x+other.x, this->y+other.y);
    }
    template<typename Type2>
    Position<Type> & operator+=(const Position<Type2> & other) throw() {
      this->x += other.x, this->y += other.y;
      return *this;
    }
    template<typename Type2>
    Position<Type> operator-(const Position<Type2> & other) const throw() {
      return Position<Type>(this->x-other.x, this->y-other.y);
    }
    template<typename Type2>
    Position<Type> & operator-=(const Position<Type2> & other) throw() {
      this->x -= other.x, this->y -= other.y;
      return *this;
    }
    Type quadNorm() const throw() {
      return (SQUARE(x) + SQUARE(y));
    }
    Type norm() const throw() {
      return sqrt(this->quadNorm());
    }
  };
```

```
typedef Position<int32_t>  Position_I32;
typedef Position<uint32_t> Position_UI32;
typedef Position<float_C>  Position_FC;
typedef Position<float_S>  Position_FS;
```

Using the power of templates, many of the algebraic operations that can be performed on geometric vector, are implemented in an abstract way. The compiler will then take the burden to create different overloaded versions of the same member method, in order to allow all the different usages with different data types.

This makes such a template class a powerful tool that can be re-used and deployed easily into the code, with vast and different purposes. The use of such objects is then simple and straightforward.

Another interesting note, is about the last four lines of code block 4.2, where four different kinds of position classes are *typedef*fed, and their names describe the precise base data type used in the corresponding implementation. Thus *UI32* means *Unsigned Integer of 32bits*, while *FC* means *Floating point number of 64bits* as seen before.

4.1.3 Arrays

Some of the most important classes in the whole library come from the header file *BinaryVectors.h*, where different types of generic arrays are implemented. The name *Binary* comes from the fact that usually the base type of these templates are integer or float data types, instead of chars like it happens in *string* objects.

Listing 4.3: 2-dimensional binary arrays definition

```
template<typename Type>
class BinVec2D : public BinVec<Type> {
  /**
   * Width of the rows
   */
  size_t width;
  /**
   * Number of rows
   */
  size_t height;

  [..]
```

```
public:
  [..]

  /**
17 * Constructor
   *
   * @param _width (Optional) width of the rows
   * @param _height (Optional) number of the rows
   * @param init (Optional) default value for the elements in the slices
22 */
  BinVec2D( const size_t & _width = 0, const size_t & _height = 0,
            const Type & init = Type())
    : BinVec<Type>(_width*_height, init), width(_width), height(_height)
  {
27  this->clean();
  }

  [..]

32 /**
   * Resetting method: it changes the shape of the vector and cleans the values
   *
   * @param _width new width of the rows
   * @param _height new number of rows
37 */
  void reset(const size_t & _width, const size_t & _height)
  {
    width = _width;
    height = _height;
42  this->resize(height*width);
    this->clean();
  }

  [..]
47

  /**
   * Sums the values from the given matrix to the corresponding ones in this
   * matris, eventually checking for constrains on the negativity.
   *
52 * @param matr The matrix with the new values
   * @param nonNegativity A boolean telling whether checking for it or not
   */
  template<typename Type2>
  void setCorrections(const BinVec2D<Type2> & matr,
57                    const bool & nonNegativity = false)
  {
    typedef typename std::vector<Type2>::const_iterator  type2_const_iterator;
```

```
    CHECK_THROW(width == matr.getWidth(),
62          WrongArgException("Matrix has not the same width"));
    CHECK_THROW(height == matr.getHeight(),
          WrongArgException("Matrix has not the same height"));

    /* We use iterators in order to be faster: there are no order needs */
67  type2_const_iterator itNewVals = matr.begin();
    const iterator & endMatrix = this->end();
    for(iterator itMatrix = this->begin(); itMatrix < endMatrix;
          itMatrix++, itNewVals++)
    {
72    *itMatrix += (Type) *itNewVals;
    }
    if (nonNegativity) {
      for(iterator itMatrix = this->begin(); itMatrix < endMatrix; itMatrix++)
      {
77      if (*itMatrix < 0) { *itMatrix = 0; }
      }
    }
  }

82 const size_t &getWidth() const throw() { return width; }
  const size_t &getHeight() const throw() { return height; }

  [..]

87 /**
   * Const double indexed getter
   * (with boundary checks when compiled in debug mode)
   *
   * @param iy the row to select
92 * @param ix the index of the element in the row
   */
  const_reference get(const size_t &iy, const size_t &ix) const
  {
    DEBUG_CHECK( checkBoundaries(iy, ix) );
97  return (*this)[iy*width + ix];
  }
  /**
   * Double indexed getter (with boundary checks when compiled in debug mode)
   *
102 * @param iy the row to select
   * @param ix the index of the element in the row
   */
  reference get(const size_t &iy, const size_t &ix)
  {
```

```
107     DEBUG_CHECK( checkBoundaries(iy, ix) );
        return (*this)[iy*width + ix];
      }
    };
    typedef BinVec2D<float_C>  BinVec2D_D;
112 typedef BinVec2D<float_S>  BinVec2D_FS;
    typedef BinVec2D<bool>     BinVec2D_B;
    typedef BinVec2D<uint32_t> BinVec2D_UI32;
```

In listing 4.3 many details of member functions were omitted for the sake of compactness, since it is still very large and full of details. Interesting features of this class can be seen directly looking at the code and the member methods.

Getter methods for accessing the arrays as a real 2-dimensional matrix were added, along with the inherited methods for accessing the object as a 1-dimensional array. Interesting is also to take attention to the conditional check of the array boundaries in debug build, that automatically shuts down in non-debug builds. The automated boundary check can be extremely precious in development, since can detect subtle problems that otherwise can hide for a long time after their introduction.

Re-shaping ability, along with default value reset can be very useful in everyday programming, too. So the *void reset(..)* method and the *init* field in the constructor can be used for these purposes.

Finally let's spend some words about the method *void setCorrections(..)*, which performs a matrix addition from the matrix in the arguments. It assumes that, in the given 2-dimensional array of the same shape, correction values for the callee matrix are stored. It is used in the main algorithm of this library, that will be discussed at the end of this chapter.

4.2 Sampling

The first algorithm implementation that will be discussed is the sampling of the voxels along the rays. In paragraph 2.2.3 a compact and effective formula (2.15) for sampling along the rays was given, but this mathematically elegant expression can hardly be efficiently implemented in a straight forward manner. Some rework needs to be performed in order to exploit the big sparseness of those matrices.

There can be a great gain in time and memory consumption, if the restriction of the interpolation to the nearest voxel centres of every sampling point is performed. In 2D this results in maximum four voxels sampled per time, and in 3D this enlarges to eight. So the formula 2.15 can be recast in 2D:

$$\Delta f_j^{(m)} = f_j^{(m)} - f_j^{(m-1)} = \lambda \frac{p_i - q_i^{(m-1)}}{\sum_{k=1}^{K} \sum_{n=1}^{N_k} (w_{ikn})^2} w_{ij} \ ; \qquad q_i = \sum_{k=1}^{K} \sum_{n=1}^{N_k} w_{ikn} f_{kn} \quad (4.1)$$

Another consideration is that also the index k does not need to run over all the points, but just on the sampling points related to a ray. A resulting formula can be derived like in the case of 4.1. Moreover, thanks to the Joseph's principle of aligning to the voxel centres, given the S oversampling factor, every S sampling points the number of interpolated voxels will be divided by two, saving computation time and possibly occupation in memory.

However, to be as efficient as possible during the reconstruction, the tuples composed by the index of the voxel and its related interpolation coefficient need to be stored contiguously in memory. The best approach to exploit pre-fetching abilities of modern CPUs is to line-up all the indexes and coefficients separately, making two different vectors of equal length.

The choice in FreeART was to allocate two of such vectors for every ray. As can be see in code 4.4, *BinVec_ UI32 indexes;* is the vector of unsigned 32bits integers which correspond to the voxels, while *BinVec_ FS weights;* is the vector of single precision floats that contain the associated interpolation coefficients for every entry in the *indexes* vector. A third important vector is *BinVec_ UI8 sizes;*, made of unsigned 8bits integers, which holds the number of sampled voxels per sampling point. It is extremely important to determine which indexes correspond to the different sampling points.

Listing 4.4: Subray Class from Ray.h

```
1 class SubRay {
  public:
    /**
     * Default constructor
     */
```

```
6   SubRay() : lossFractionOutput(1) { }

    /**
     * Max size of the samplable voxels in a 2D geometry
     */
11  static const uint8_t max_size = 4;

    /**
     * This is the output of the ray in the transmission setup. In principle this
     * is the integral over the absorption coefficient
16   */
    float_S lossFractionOutput;

    /**
     * Position of the initial point in the ray
21   */
    Position_FS initPosition;

    /**
     * Sizes of the single sampled points (how many voxels they sample)
26   */
    BinVec_UI8  sizes;

    /**
     * Indexes of the sampled points
31   */
    BinVec_UI32 indexes;
    /**
     * Weights associated to the indexes of the sampled points
     */
36  BinVec_FS   weights;

    size_t size() const throw() { return sizes.size(); }
};
```

This structure is very powerful and both reduces memory consumption and allows optimizations thanks vectorization of the computation. However, it can be really difficult to work with. Function *void ScannerPhantom2D::sampleLine(SubRay& subRay, IterationData& data)*, first does an estimate of the number of sampling points in the given ray, based on the fixed interval between one point and the other. Then, allocating the memory in the *sizes* vector, it counts how many voxels it will be sampling at worse.

Finally, after allocating enough space in the indexes and weights arrays, it does the

real sampling of the voxels.

Listing 4.5: void sampleLine(SubRay& subRay, IterationData& data) from ScannerPhantom2D.cpp

```
1 INLINE void
  ScannerPhantom2D::sampleLine( SubRay& subRay, IterationData& data)
  {
    [..]

6   const uint32_t numPoints =
        _FT_UI32(floor(limitsIndep.getLength()/data.increment));
    subRay.sizes.reserve(numPoints);

    BinVec<Position_FS> & posVec = data.rot.posBuffer;
11  posVec.clear(); posVec.reserve(numPoints);

    /* Counting the points that can be sampled */
    for(; data.limits.contains(data.pos);)
    {
16    /* Push the new point to the list of sampled points: using "push_back"
       * method to be safe with memory bounds and previous estimate calculation */
      subRay.sizes.push_back(0);
      posVec.push_back(data.pos);

21    /* if the point is aligned to the integral part in the main axis, we will
       * sample just 2 points instead of 4 */
      if ( abs(indepCoord - floor(indepCoord)) < TOLL_COMP) {
        data.sampPartial++;
      } else {
26      data.sampComplete++;
      }

      /* Next sample point */
      data.pos += data.rot.pointIncrement;
31  }

    /* Let's allocate enough place for sampling the voxels */
    const uint32_t totSamplable = 4*data.sampComplete + 2*data.sampPartial;
    subRay.indexes.resize(totSamplable, 0);
36  subRay.weights.resize(totSamplable, 0.0);

    /* Sampling */
    BinVec<Position_FS>::const_iterator posIt = posVec.begin();
    for(PartialSubRayIterator point(subRay); !point.isEnd(); point++)
41  {
      /* Get the next position from the buffer */
```

```
      const Position_FS & pos = *posIt++;

      /* Sample it! */
      selectVoxels(point, pos);

      /* Sanity check test! (I should throw an exception) */
      if (!point.size())  {
        [..]
      }
    }

    [..]
}
```

*SubRayIterator*s (from file *RayHelpers.h*) are nice and interesting tools, which iterate over the *Subray* class, keeping consistency. They are complex templates, which hold pointers to indexes and weights, and according to function *void shiftPartialBase() throw()* in code block 4.6, they shift those internal pointers, by the size of current sampling points.

Listing 4.6: Private members of the base for Subray Iterators

```
/**
 * Base Ray iterator class
 */
template< typename lnuiIter, typename flsIter, typename sizeIter>
class PartialRayBaseIterator {
protected:
  /**
   * iterator pointing to the initial value of the indexes for the pointed point
   */
  lnuiIter indexes;
  /**
   * iterator pointing to the initial value of the weights for the pointed point
   */
  flsIter weights;

  /**
   * iterator pointing to the number of voxels sampled in the pointed point
   */
  sizeIter sizes;
  /**
   * End of the sizes vector
   */
  sizeIter endSizes;
```

```
25    /**
       * Moves to the next point all the iterators
       */
      void shiftPartialBase() throw() {
        const uint8_t & shift = *(this->sizes)++;
30      indexes += shift, weights += shift;
      }
    public:
      [..]

35    /**
       * Public advancing operator that moves to next point
       */
      PartialRayBaseIterator & operator++(int) throw() {
        shiftPartialBase();
40      return *this;
      }
      [..]
    };
```

So to do a full recap, the main idea is to allocate all the indexes and weights for a given ray contiguously, and accessing them sequentially. A very efficient way is doing pointer arithmetic, but even nicer is hiding it behind a simple "unit addition" operator (++). Contiguous storage can also allow to introduce *SIMD operations*, thanks to the specific instructions in the modern CPUs.

Pointer arithmetic is widely used in my code, because for a sequential way of programming is faster than every other implementation. However, for parallel programming, pointer arithmetic is bad, but parallel programming was not a goal of my project. An "external" parallelism was instead suggested to the users of the library, since in the current C++ standard there is nothing about parallel programming. So every compiler and every platform can offer different implementation, which would require huge efforts to guarantee portability.

A way to apply the so called external parallelism is based on the fact that reconstructions are usually performed "slice-by-slice". It means that a full volume reconstruction is made of many 2-dimensional flat reconstructions. So the parallelization could be on the set of slices, thanks to the use of different threads or processes in the code written by the user of the library.

Listing 4.7: Method to interpolate on the fly the values of the sampled voxels

```
template< typename lnuiIter , typename flsIter , typename sizeIter>
class PartialRayBaseIterator {
protected:
  [..]
public:
  [..]
  float_C getMeanField(const BinVec2D_FS &mtr) const throw() {
    switch (*sizes) {
      case 4: {
        return _FT_C(mtr.get(indexes[0])) * _FT_C(weights[0])
            + _FT_C(mtr.get(indexes[1])) * _FT_C(weights[1])
            + _FT_C(mtr.get(indexes[2])) * _FT_C(weights[2])
            + _FT_C(mtr.get(indexes[3])) * _FT_C(weights[3]);
      }
      case 2: {
        return _FT_C(mtr.get(indexes[0])) * _FT_C(weights[0])
            + _FT_C(mtr.get(indexes[1])) * _FT_C(weights[1]);
      }
      case 3: {
        return _FT_C(mtr.get(indexes[0])) * _FT_C(weights[0])
            + _FT_C(mtr.get(indexes[1])) * _FT_C(weights[1])
            + _FT_C(mtr.get(indexes[2])) * _FT_C(weights[2]);
      }
      case 1: {
        return _FT_C(mtr.get(indexes[0])) * _FT_C(weights[0]);
      }
      default: {
        return 0;
      }
    }
  }
  [..]
};
```

Apart from storing those values, there exist also functions that load them, interpolating on the fly the values of the voxels pointed by their indexes. From the code snippet 4.7, it can be seen the interpolation on the current sampling point in the given SubrayIterator. It is just a geometrical (or linear) interpolation and it does not deserve further discussion.

A nice example on how to use these Iterators to extract information for a given ray, is shown in code snippet 4.8, which quickly iterates over all the ray, with a simple API and in a manner that couples very well with pointer arithmetic.

Listing 4.8: Method to load interpolated values over a sampled line

```
INLINE void
GeometryFactory::loadMeanCoeffs(const SubRay & subray,
    const BinVec2D_FS & matr, float_C * coeff)
{
  *coeff++ = 0;
  for(PartialConstSubRayIterator point(subray); !point.isEnd(); point++)
  {
    *coeff++ = point.getMeanField(matr);
  }
}
```

The suffixes *Partial* and *Const* refer to the properties of the iterator, that first does just partial shifting of the internal pointers, ignoring the "geometric" position of the point in the ray (it is used only for debug purposes), and does not allow modifications on the ray structure.

4.3 Attenuation computation

Another topic that was already discussed in a detailed way in paragraph 1.2.2, and chapter 3, is the attenuation of light through the media.

In particular, in chapter 3, every surface or volume integral was re-cast to line integrals, which are a lot easier to work with. The attenuation computation code can be seen in block 4.9.

Listing 4.9: Attenuation computation over a line

```
INLINE void
GeometryFactory::updateIncomingLossFraction(Rotation & rot,
                                            const BinVec2D_FS & absMatr,
                                            float_S * lossFractionIncident)
{
  const uint32_t & oversamp = _FT_UI8(prefs.unsIntPrefs.get(OVERSAMP_NAME));
  const float_C incr = 1/_FT_C(oversamp);

  /* The physical size of the voxel is important for quantitative results */
  const float_C & physicalSize = (rot.increment.x >= rot.increment.y)
                                 ? prefs.doublePrefs.get(VOXEL_WIDTH_NAME)
                                 : prefs.doublePrefs.get(VOXEL_HEIGHT_NAME);

  const float_C interactLen = incr * physicalSize * rot.integralNormalization;
```

```
15
    /* Let's allocate a buffer for loading the coefficients */
    const uint32_t maxPoints = 1 + oversamp
                * (1 + _FT_UI32( max(absMatr.getWidth(), absMatr.getHeight()) ) );
    BinVec_FC coeffsBufferObj(maxPoints);
20  float_C * const coeffsBuffer = &*coeffsBufferObj.begin();

    for(uint32_t numRay = 0; numRay < rot.size(); numRay++)
    {
      Ray & ray = rot.getRay(numRay);
25    float_C fract = 1;

      loadMeanCoeffs(ray, absMatr, coeffsBuffer);

      const float_C * const endCoeff  = coeffsBuffer + ray.size() +1;
30    const float_C * coeff           = coeffsBuffer + 1;
      const float_C * prevCoeff       = coeffsBuffer;

      float_C previousProdStep = *prevCoeff * interactLen;

35    for(; coeff < endCoeff; coeff++, prevCoeff++)
      {
        *lossFractionIncident++ = _FT_S(fract);

        const float_C currentProdStep = *coeff * interactLen;
40      const float_C maxProd = max(abs(currentProdStep), abs(previousProdStep));

        const float_C scaleFactor = 1 + 2*maxProd*(maxProd >= 1.0);
        const float_C newLen = interactLen/scaleFactor;

45      fract *= pow(
            1 - newLen * (*prevCoeff + (*coeff) * (1 - newLen*(*prevCoeff)) )/2,
            scaleFactor);

        previousProdStep = currentProdStep;
50    }
      ray.lossFractionOutput = _FT_S(fract);
    }
  }
```

This routine is based on the Heun method which is a second order Runge-Kutta numerical method for solving ordinary differential equations. The use of such method can sound strange at the beginning, since it was already showed the analytical solution to equation 1.4, and a numerical solution to the problem should not be needed. Since the partial decrement in intensity over all the points sampled along

the ray is strictly needed, the Lamber-Beer equation (1.5) can be expressed for a point along the ray:

$$I_{ik} = I_0 \cdot \exp\left[-\int_{(s_a,u_a)}^{(s_k,u_k)} \mu(s)\, ds\right] \tag{4.2}$$

where (s_a, u_a) is the first point of the ray in the sample, and (s_k, u_k) is the considered point k.

A numerical approach based on 4.2 can be:

$$l_{ik} = \sum_{t=1}^{k} \mu_t d_\theta = d_\theta \sum_{t=1}^{k} \mu_t \ ; \qquad I_{ik} = I_0 \cdot \exp\left[-l_{ik}\right] \tag{4.3}$$

where d_θ is the distance between every sampling point, which is fixed for a given projection, μ_t is the interpolated coefficient in the sampled point k and l_{ik} is the numerical line integral of the linear absorption coefficient μ along the given ray i. Equation 4.3 can also be recast to a recursive formula for the computation of the line integral:

$$l_{i0} = 0 \ ; \qquad l_{ik} = l_{ik-1} + \mu_t d_\theta \ ; \qquad I_{ik} = I_0 \cdot \exp\left[-l_{ik}\right] \tag{4.4}$$

However, even if these equations can look very attractive and easy to work with, are extremely expensive by the means of computational time. The *double exp(double x);* function of the C library becomes very slow when called for every sampled point in every ray of every projection.

The only solution is to use a faster solution to the ray integral, starting from the differential problem and solving the associated Cauchy problem. A nice implementation is the Heun method, since it is a second order Runge-Kutta method, and guarantees a nice convergence to the solution if the stability conditions are met. For a deeper insight in the method the reader is suggested to consult the specific literature [10]. The only considerations on stability needed in this case, where the line integral l_{ik} is non negative and monotonely growing, are on the modulus of the product $\mu_t d_\theta$ at every step, that needs to be $\in [0, 2)$.

All the aspects discussed till now can be seen in code block 4.9, where the partial attenuation for all the sampled points in a given projection are computed. From

line 6 to line 15 the parameter d_θ is computed, then till line 20 enough buffer space is allocated for future calculations. Starting from line 22 begins the loop over the rays in the projection. Indeed, on line 27 is used the function from code snippet 4.8, encountered in the previous section, that will load the interpolated coefficients into the just allocated buffer.

Then starting from line 35 it begins the computation of the line integral for the rays in the current rotation, where it was adopted a trick to keep all the steps in the stability region. If the coefficients μ_t in the product $\mu_t d_\theta$ are too big, the step d_θ is just virtually reduced, dividing it by a scale factor and virtually iterating on the fixed μ_t, and finally elevating the iteration value to the power of the scale factor. This is the equivalent to reducing the step, and iterating more times, according to the step reduction. A mathematical expression for the implemented formula is:

$$d_{\theta ik} = \frac{d_\theta}{s_{ik}} \; ; \qquad I_{ik} = I_{ik-1}\left(1 - \frac{1}{2} d_{\theta it}\left(\mu_{k-1} + \mu_k\left(1 - d_{\theta ik}\mu_{k-1}\right)\right)\right)^{s_{ik}} \tag{4.5}$$

where s_{ik} is the scale factor for the point k in the ray i, and so $d_{\theta ik}$ is the virtually reduced step.

The reader is also suggested to perform the derivation of equation 4.5 from the Heun standard formula and the known Cauchy problem associated to equation 1.4.

4.4 Self-Absorption matrices computation

When it comes to self-absorption, computing and storing all the coefficients associated to the sampled points, becomes a big problem in memory occupation and computational time.

To give a better idea of the size of that problem, let's now do a rough estimation of the relation between the number of voxels and the other quantities.

Let the matrix be a square matrix, so the total number of voxels is $O\left(n^2\right)$, where n is the number of voxels per edge. Given the spacing between two adjacent rays Δ, and the spacing between to adjacent points in the same ray, they can be related to the square voxel edge l by the two relations:

$$\begin{aligned} \Delta &= al \\ \Lambda &= \frac{l}{S|\cos\theta|} \end{aligned} \tag{4.6}$$

where θ is the angle between the ray and the most parallel axis, S is the over-sampling factor, and $a \in \Re$. Taking some common values like $a = 1$ and $S = 2$, thus it is easy to deduce that $\Delta = l$, $\Lambda \in \left[\frac{1}{2}l, \frac{1}{\sqrt{2}}l\right]$ and the mean value $\bar{\lambda} \simeq 0.55$. So the area associated to a mean sampling point is $\delta = \Delta\Lambda = 0.55l^2$. Useful to know is that the reconstruction will be just over a circle centred in the centre of the square matrix, so that will be the area to sample, which is $\Gamma = \frac{\pi}{4}l^2$. Last, given the number of projections p, the relative number of sampling points per reconstruction to the $O(n^2)$ voxels will be:

$$N = \frac{\Gamma}{\delta}p = \frac{\pi}{4 \cdot 0.55}p = \frac{\pi}{2.2}p \tag{4.7}$$

So the order of sampling points is $O(n^2 p)$.

It is now interesting to compute the number of the self-absorption sampling points to consider. If t is the number of self-absorption rays per sampled point, as rule of thumb, each of those rays will be long $b \sim \frac{1}{2}l$. Based on this last fact and the previous derivation, the order of points to sample and compute on the self-absorption rays will be: $O(n)$.

So finally the order of the points to consider and store, in case of self-absorption corrections will be $O(n^3 tp)$.

In principle, with an extremely powerful processor the memory constrain could be removed, calculating every time on the fly all the needed parameters. This could be possible with latest GPUs, that perform very well in huge number crunching applications. However this is in contrast with the principle of "easy portability": as of today, the only open standard for GPGPU[3] is *OpenCL* but it lacks the parity level to closed standards in both performance and features. On the other side, closed

[3]General Purpose GPU

standards do not offer cross portability between different vendors, and can impose really bad policies to customers.

To manage the problem of self-absorption with sequential CPU computation, it is needed to perform some approximations. The main idea is to not consider all the "scattering rays" on their own, but for a given projection θ and a given direction t for the emitted rays, to compute a matrix with a 1-to-1 mapping to the image matrix that assigns to every voxel an approximate value of self-absorption attenuation to the detector. The self-absorption correction coefficient for a given sampled point in the projection θ, can then be extracted interpolating the values of the closest voxels in the generated self-absorption matrix.

This approach reduces the number of evaluations from $O(n^3tp)$ to $O(n^2tp)$. Since usually $t \sim 1-2$ or in extreme cases ~ 10, t can be considered a constant and the complexity reduces to $O(n^2p)$ which is the same of the "not self-absorption corrected" problem.

In chunks of code 4.10 and 4.11 is reported the function that creates the approximate self-absorption matrices.

Listing 4.10: Computation of self-matrices for every point of the image (Part 1)

```
INLINE void
GeometryFactory::updateSelfAbsorptionMatrices(Rotation & rot,
    const BinVec2D_FS & absorbMatr, BinVec2D_FS & selfAbsorbMatr)
{
  BinVec2D_FS coeffsMatr(absorbMatr.getWidth(), absorbMatr.getHeight());

  BinVec_FS lossFractBuffer(rot.totSampledPoints);
  updateIncomingLossFraction(rot, absorbMatr, &*lossFractBuffer.begin());

  const float_C squareNorm = SQUARE(_FT_C(rot.integralNormalization));

  BinVec_FS::iterator buff = lossFractBuffer.begin();
  for(Rotation::const_iterator ray = rot.begin(); ray != rot.end(); ray++) {
    const float_C & outRayFract = ray->lossFractionOutput;
    for(uint32_t numPoint = 0; numPoint < ray->size(); numPoint++, buff++) {
      *buff = _FT_S(1 - (_FT_C(*buff) - outRayFract) * squareNorm
                    / (_FT_C(*buff) + (_FT_C(*buff) < TOLL_COMP)));
    }
  }

  [..]
```

```
22 }
```

The self-absorption matrices are related to the projections in a 1-to-1 relation. The basic idea is to sample with n parallel rays the image matrix in the direction of the detector, and then associate to all of those sampled points value of the self-absorption function that is computed cumulatively along the ray. Those sampled points are then interpolated to create the final correction matrix.

As can be seen in code block 4.10, the first part of the function creates a buffer that can hold all the sampled points for the projection θ, and then starting from the detector it computes the attenuation for all the points in all the rays in the opposite direction of what should be expected. This is in principle an error, but in the loop from line 13, these values are reverted to the right ones, and are normalized on the basis of the relative distance between one point and the next in rotation θ.

Listing 4.11: Computation of self-matrices for every point of the image (Part 2)

```
INLINE void
GeometryFactory::updateSelfAbsorptionMatrices(Rotation & rot,
3     const BinVec2D_FS & absorbMatr, BinVec2D_FS & selfAbsorbMatr)
{
  [..]

  buff = lossFractBuffer.begin();
8 for(Rotation::const_iterator ray = rot.begin(); ray != rot.end(); ray++)
  {
    for(PartialConstSubRayIterator point(*ray); !point.isEnd(); point++, buff++)
    { /* We now prepare the matrices with the absorption seen from the voxels */
      const float_S & lossFract = *buff;
13
      const uint32_t * const voxlist = point.getIndexesList();
      const float_S * const weights = point.getWeightsList();

      switch (point.size()) {
18      case 4: {
          selfAbsorbMatr.get(voxlist[0]) += lossFract * weights[0],
              selfAbsorbMatr.get(voxlist[1]) += lossFract * weights[1],
              selfAbsorbMatr.get(voxlist[2]) += lossFract * weights[2],
              selfAbsorbMatr.get(voxlist[3]) += lossFract * weights[3];
23        coeffsMatr.get(voxlist[0]) += weights[0],
              coeffsMatr.get(voxlist[1]) += weights[1],
              coeffsMatr.get(voxlist[2]) += weights[2],
              coeffsMatr.get(voxlist[3]) += weights[3];
```

```
            break;
28        }
          case 2: {
            selfAbsorbMatr.get(voxlist[0]) += lossFract * weights[0],
                selfAbsorbMatr.get(voxlist[1]) += lossFract * weights[1];
            coeffsMatr.get(voxlist[0]) += weights[0],
33              coeffsMatr.get(voxlist[1]) += weights[1];
            break;
          }
          case 3: {
            selfAbsorbMatr.get(voxlist[0]) += lossFract * weights[0],
38              selfAbsorbMatr.get(voxlist[1]) += lossFract * weights[1],
                selfAbsorbMatr.get(voxlist[2]) += lossFract * weights[2];
            coeffsMatr.get(voxlist[0]) += weights[0],
                coeffsMatr.get(voxlist[1]) += weights[1],
                coeffsMatr.get(voxlist[2]) += weights[2];
43          break;
          }
          case 1: {
            selfAbsorbMatr.get(voxlist[0]) += lossFract * weights[0];
            coeffsMatr.get(voxlist[0]) += weights[0];
48          break;
          }
          default: {
            WarningPrintf(("No Voxel sampled here!\n"));
            break;
53        }
        }
      }
    }

58  BinVec_FS::const_iterator coeff = coeffsMatr.begin();
    for(BinVec2D_FS::iterator fract = selfAbsorbMatr.begin();
        fract != selfAbsorbMatr.end(); fract++, coeff++)
    {
      *fract /= (*coeff + (*coeff < TOLL_COMP));
63  }
}
```

In code block 4.11 these sampled values are then interpolated to create the final self-absorption matrix. The only non-trivial note is about the *coeffsMatr* that is needed to normalize the final matrix, since the coverage done bay the sampled points is not uniform. Indeed, from line 59, the values of the final matrix are divided by the sum of weights of the sampled points that contributed to determine the value of

the voxel. So if there was an over coverage of the voxel (sum of weights > 1), or an under coverage (sum of weights < 1), the values is restored to the expected one.
A final note on a trick used in both the pieces of code 4.10 and 4.11 at lines 17 and 62 respectively: in the denominator, division by zero was avoided simply adding a quantity that is usually 0 and becomes 1 only when the denominator is 0 itself. Dividing by 1 simply leaves things unchanged, while preventing floating point exceptions and avoiding *if {..} else {..}* constructs in loops.

4.5 Simultaneous ART

All the pieces will now be put together and it will be demonstrated how the procedures shown in this chapter can serve to construct a SART algorithm with physical corrections.
As a study case, it was chosen to discuss the code that performs reconstructions on data from diffraction experimental set-ups. The purpose of this kind of reconstruction is to correlate the Bragg peaks on the CCD to the voxels in the image matrix. It also useful to note that this implementation is optimized C++ code, so it may seem obfuscated and difficult to understand.
Simultaneous ART implementation is pretty much similar to ordinary ART, except from the fact that instead of updating the image matrix after every ray back-projection, the corrections are stored and summed in a service matrix, and applied only after the iteration on the current projection θ ends.
To explain SART implementation, it is first needed to introduce some utility functions.

Listing 4.12: Component wise product of two vectors

```
1 template<typename Precision>
  INLINE void
  DiffrGeometryClient::compute_InAndOut_LossFract_product(
      const float_S * inLossFract, const Precision * outLeftLossFract,
      const Precision * outRightLossFract, const uint32_t & totPoints)
6   throw()
  {
    const BinVec_FC::iterator endOfPoints = voxIndepParamBuff.begin() + totPoints;
    for(BinVec_FC::iterator vecBuffer = voxIndepParamBuff.begin();
```

```
        vecBuffer != endOfPoints;
11      vecBuffer++, outLeftLossFract++, outRightLossFract++, inLossFract++)
    {
      *vecBuffer = (_FT_C(*outLeftLossFract) + _FT_C(*outRightLossFract))
                   * _FT_C(*inLossFract) / 2;
    }
16 }
```

The first is shown in the code block 4.12, and is the composition of two different vector operations. Two of the input vectors are the in-plane components of the self-absorption corrections for the points in the ray, another argument is the vector for the incoming beam absorption correction, and finally *totPoints* gives the number of points into the vectors. The operation performed is the component wise average between the self-absorption vectors, and finally the component wise product of the result with the vector of the incoming beam attenuation values. The function in listing 4.12, is used in the function that computes the self absorption correction joined to the other corrections, shown in code block 4.13.

Listing 4.13: Computes self-absorption correction parameters

```
INLINE void
DiffrGeometryClient::computeDiffrSelfAbsCorrectionParams(
    const GeometryTable & gt, const uint32_t & numRot, const SubRay & ray,
4   const float_S * inLossFract, BinVec2D_D & selfAbsBuff)
{
  float_C * const leftOutLossFract  = &*selfAbsBuff.begin();
  float_C * const rightOutLossFract = &selfAbsBuff.get(1, 0);

9 const BinVec2D_FS & leftMatr  = gt.getSelfAbsorpAttenuation(0, numRot);
  computeSelfAbsCorrections(leftMatr, ray, leftOutLossFract);

  const BinVec2D_FS & rightMatr = gt.getSelfAbsorpAttenuation(1, numRot);
  computeSelfAbsCorrections(rightMatr, ray, rightOutLossFract);
14
  compute_InAndOut_LossFract_product( inLossFract, leftOutLossFract,
                                      rightOutLossFract, ray.size() );
}
```

There is nothing special about function *computeDiffrSelfAbsCorrectionParams*, it just interpolates the values in the self-absorption matrices along the ray, and then composes those vectors, using function from code 4.12.

What could puzzle the reader, about this function is at lines 7 and 8, where the

pointer to two different areas of a matrix are taken. The matrix *selfAbsBuff* is a preallocated buffer area, where data can be written and used just after, without any worries about cleaning the memory area. Before the execution of this piece of code, it is enlarged enough to fit all the possible vectors for the given rotation. This buffer is a multi-dimensional buffer, since it is made of a 2-dimensional vector. In one axis it measures the maximum length of a sampling vector, and in the other the number of buffer vectors needed.

In this specific example, the number of buffer vectors needed is two, so while the first just starts from the beginning of the buffer area, the other takes as initial point the beginning of the next row in the 2-dimensional buffer area.

Listing 4.14: Re-projection of corrections from a given ray

```
inline void
GeometryClient::applyCorrections( BinVec2D_D & matr,
                                  const SubRay & subray,
                                  const float_C & correction)
{
  for(PartialConstSubRayIterator point(subray); !point.isEnd(); point++)
  {
    const uint32_t * const voxlist = point.getIndexesList();
    const float_S * const weights = point.getWeightsList();
    /* Simple loop unrolling on the voxels sampled by the point */
    switch (point.size()) {
      case 4: {
        matr.get(voxlist[0]) += correction * _FT_C(weights[0]),
        matr.get(voxlist[1]) += correction * _FT_C(weights[1]),
        matr.get(voxlist[2]) += correction * _FT_C(weights[2]),
        matr.get(voxlist[3]) += correction * _FT_C(weights[3]);
        break;
      }
      case 2: {
        matr.get(voxlist[0]) += correction * _FT_C(weights[0]),
        matr.get(voxlist[1]) += correction * _FT_C(weights[1]);
        break;
      }
      case 3: {
        matr.get(voxlist[0]) += correction * _FT_C(weights[0]),
        matr.get(voxlist[1]) += correction * _FT_C(weights[1]),
        matr.get(voxlist[2]) += correction * _FT_C(weights[2]);
        break;
      }
      case 1: {
```

```
            matr.get(voxlist[0]) += correction * _FT_C(weights[0]);
            break;
        }
        default: {
            WarningPrintf(("No Voxel sampled here!\n"));
            break;
        }
    }
  }
}
```

Let's now just spend a little time in how the function from code block 4.14, works. It behaves in a much similar way to another piece of code in listing 4.11. It is an unrolled loop, nested in a loop that iterates over all the sampled points in the ray. Different is the action taken: it back-projects the correction obtained for the current ray from the main formula to the voxels touched by the ray, in the given storage matrix *matr*.

Next is the function that efficiently performs the ray sum, interpolating the image matrix, and at the same time, computes the denominator of the formula 3.1. Both the interpolation of the matrix, and the square weights associated to the sampled points, are multiplied by the attenuation correction value from the function 3.2.

Listing 4.15: Ray sum, and denominator of the correction formula calculation

```
template<typename Precision>
INLINE void
GeometryClient::computeSignalAndDenom(const BinVec2D_FS & matr,
                                      float_C & signal, float_C & denom,
                                      const Ray & ray, const Precision * params)
{
  for(PartialConstSubRayIterator point(ray); !point.isEnd(); point++, params++)
  {
    signal += _FT_C(*params) * point.getMeanField(matr);
    denom += _FT_C(*params) * point.getSquareWeight();
  }
}
```

Finally it comes the function that encloses all the previous functions and performs a complete iteration of the SART algorithm over all the projections in a sinogram.

Listing 4.16: Main function for doing a SART reconstruction iteration

```
void
DiffrGeometryClient::reconstructionIteration( const GeometryTable &gt,
                                              const Sinogram &sino,
                                              Phantom2D &ph)
{
  const bool & nonNegative = prefs.boolPrefs.get(NON_NEGATIV_NAME);
  const float_C oversamp = _FT_C(prefs.unsIntPrefs.get(OVERSAMP_NAME));
  const bool & selfAbs = prefs.boolPrefs.get(SELF_ABSORP_NAME);

  BinVec2D_FS & matr = ph.getMatrix();

  ReconstructionParameters rp(ph, 1/(oversamp));

  checkAndPrepareIteration(rp, ph, sino, 2);

  /* Let's iterate on all the rotations */
  for(size_t numRot = 0; numRot < gt.size(); numRot++) {
    diffMatr.clean();

    rp.realProjSel = rndmAccessor[numRot];
    const Rotation & rot = gt.getRotation(rp.realProjSel);
    const uint32_t & offset = gt.getOffsets().getRotOffset(rp.realProjSel);

    /* Pointers to the physical quantities, offsetted to the right rotation */
    const float_S * inLossFract = gt.getIncidentLossFractions() + offset;

    /* Normalization factors for rotation (integral normalization) */
    const float_C rotationFactor = _FT_C(rot.integralNormalization);

    for(uint32_t numRay = 0; numRay < rot.size(); numRay++)
    {
      const Ray &ray = rot.getRay(numRay); const uint32_t &raySize = ray.size();

      float_C denom = 0, calculatedSignal = 0;
      if (selfAbs) {
        computeDiffrSelfAbsCorrectionParams(gt, rp.realProjSel, ray,
                                            inLossFract, rp.selfAbsBuff);

        computeSignalAndDenom<float_C>( matr, calculatedSignal, denom, ray,
                                        &*voxIndepParamBuff.begin());
      } else {
        computeSignalAndDenom<float_S>( matr, calculatedSignal, denom, ray,
                                        inLossFract);
      }
      /* Move the pointer to the next ray */
      inLossFract += raySize;
```

```
48          calculatedSignal *= rotationFactor, denom *= rotationFactor;
            const float_C sampNormaliz = _FT_C(raySize)/(rp.diameter*oversamp);

            const float_C correctionFactor = sampNormaliz * rp.damping / denom *
                    (_FT_C(sino.getPoint(rp.realProjSel, numRay)) - calculatedSignal);
53
            applyCorrections(diffMatr, ray, correctionFactor);
          }

          /* let's apply the corrections to the phantom */
58        matr.setCorrections(diffMatr, nonNegative);
        }
      }
```

At the beginning, some useful constants and preference switches are collected. Then at line 12, an object called *rp* is created. This object contains all the needed information for the current iteration of the algorithm. The function call *checkAndPrepareIteration* just fixes some more properties in the reconstruction object, like allocating dirty buffers for fast self-absorption corrections computation, and resizing of the randomizing vector.

Now, at line 17 comes the loop over all the projections in the sinogram. After cleaning the temporary matrix of the SART algorithm, the projection to use is selected in a randomized way. The vector *rndmAccessor* has the same number of components as the total number Θ of projections, and contains integer numbers from 0 to $\Theta - 1$, rearranged randomly in the vector. This ensures that accessing sequentially to the vector, it returns random indexes for accessing the projections in the sinogram.

The reason behind a randomized access scheme in processing the projections is that it reduces significantly the degree of similarity between one projection and the next. From a mathematical point of view, projections should be as much perpendicular to each others as possible, in order to decrease the correlation between the corrections. In noisy reconstructions, which of course is always the case with real data, this prevents some peculiar artefacts from arising into the reconstructed image.

After some useful quantities are fixed for the current projection θ, at line 30, begins the loop over the rays in the rotation. All the function calls in the loop are already known, since they were discussed in this section, so it is now easy to point out that the formula at lines 51 and 52, is exactly the equation 2.13, that computes the cor-

rection value to be back-projected on the voxels touched by the ray.

At the end of function, in line 58, all the summed corrections are applied back to the original image matrix.

What can be deduced in the previous chapter is that, using FreeART library, can be possible to perform 2-dimensional tomography on experimental data coming from different kinds of set-ups. Tomographic reconstructions can be performed using various physical corrections, depending on the case, and some other details can be tweaked in order to suite the needs of the user.
However, along with the pure reconstruction logic, other useful tools have been developed and added to the library. A selection of them is reported and discussed in this chapter.

5.1 Projection of Sinograms

Starting from a reconstructed image, or a loaded theoretical image matrix, it is possible to generate theoretically predicted sinograms. This is a specular feature to the reconstruction algorithm. It is based on the same concepts, so the two code paths share most of the code.
This materializes in being just one the function that is different from the two use cases.

Listing 5.1: Function that generates a new sinogram

```
void
DiffrGeometryClient::makeSinogram(const GeometryTable & gt, Sinogram & sino,
                                  const Phantom2D & ph, const BinVec2D_B & mask)
{
  const bool completeSinogram = !mask.size();
  if (!completeSinogram) { checkMask(mask, ph); }

  const bool & selfAbs = prefs.boolPrefs.get(SELF_ABSORP_NAME);
  const BinVec2D_FS & matr = ph.getMatrix();
```

```
10
    const size_t & totProj = gt.size();
    sino.reset(totProj, gt.getTotRaysPerRot());

    const float_S * inLossFract = gt.getIncidentLossFractions();
15  const uint32_t maxPointNum = this->computeMaxRayLength();
    BinVec2D_D outLossFractBuff(selfAbs ? maxPointNum : 0, 2);

    for(size_t numRot = 0; numRot < totProj; numRot++)
    {
20    const Rotation & rotation = gt.getRotation(numRot);
      SinogramProj & sinoSlice = sino.getSlice(numRot);
      sinoSlice.angle = rotation.angle;

      for(uint32_t numRay = 0; numRay < gt.getTotRaysPerRot(); numRay++)
25    {
        const Ray & ray = rotation[numRay];
        float_C signal = 0;

        if (selfAbs) {
30        computeDiffrSelfAbsCorrectionParams(gt, numRot, ray, inLossFract,
                                              outLossFractBuff);

          signal = computeRaySum<float_C>(matr, ray, completeSinogram, mask,
                                          &*voxIndepParamBuff.begin());
35      } else {
          signal = computeRaySum<float_S>(matr, ray, completeSinogram, mask,
                                          inLossFract);
        }
        inLossFract += ray.size();
40
        sinoSlice.getPoint(numRay) =
                          _FT_S(signal * _FT_C(rotation.integralNormalization));
      }
    }
45 }
```

As an example, a method deputed to generate the sinograms is reported in code block 5.1. Much of the code in this function already appeared before and should be already clear how it works.

There are just two main differences from this function and the one reported in listing 4.16: the generation of the sinogram, storing the computed ray sums directly in the sinogram, and the use of a conditional mask in the computation of the ray sum.

While the first is obvious and is performed in lines 12 and 41-42, the second is not

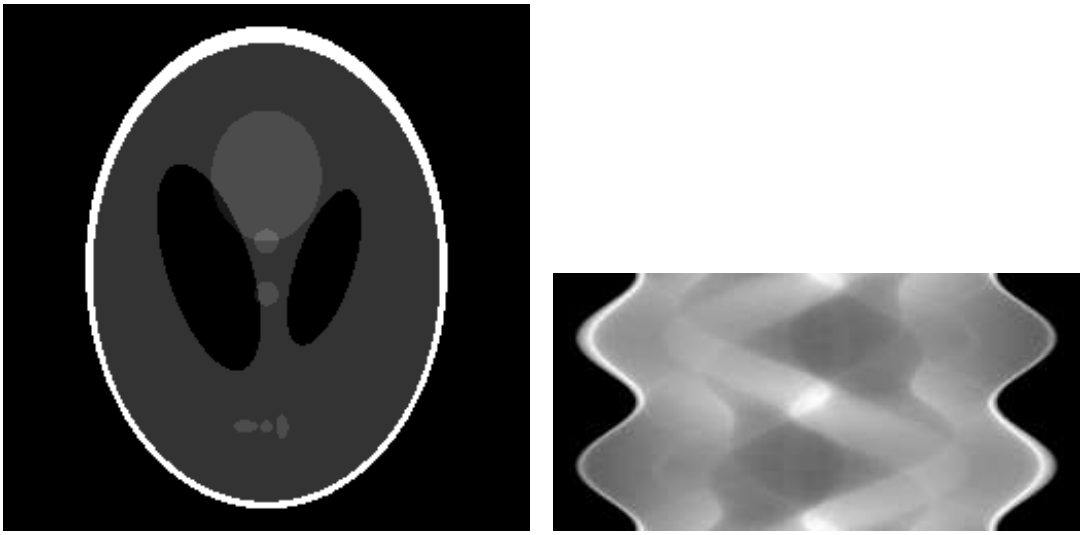
so trivial, so it is delegated to a separate function, which is called at lines 33 and 36.

Listing 5.2: Function to compute ray sum, using a mask for the image

```
template<typename Precision>
INLINE float_C
GeometryClient::computeRaySum(const BinVec2D_FS & matr, const SubRay & ray,
    const bool & completeSinogram, const BinVec2D_B & mask,
    const Precision * voxIndepParams)
{
  float_C signal = 0;
  for(PartialConstSubRayIterator point(ray); !point.isEnd();
      point++, voxIndepParams++)
  {
    /* If it is complete we don't mind to consider voxel by voxel through the
     * mask */
    if (completeSinogram) {
      signal += point.getMeanField(matr) * _FT_C(*voxIndepParams);
    } else {
      /* Otherwise let's check voxel by voxel */
      float_C signalSmplPoint = 0;
      const uint32_t * voxIndexes = point.getIndexesList();
      const float_S * voxWeights = point.getWeightsList();
      const uint32_t numTotVox = point.size();
      for(uint32_t numVox = 0; numVox < numTotVox;
          numVox++, voxIndexes++, voxWeights++)
      {
        /* if this voxel is allowed by the mask, let's sum it up */
        if ( mask.get( *voxIndexes ) ) {
          signalSmplPoint += _FT_C(*voxWeights) *
                             /* Weight of the voxel */
                             _FT_C(matr.get(*voxIndexes));
                             /* Emission probab of the voxel */
        }
      }
      signal += signalSmplPoint * _FT_C(*voxIndepParams);
    }
  }
  return signal;
}
```

In code block 5.2, the function that computes the ray sum is reported. If the boolean *completeSinogram* is **true**, then the ray sum takes the straight forward way, without mask. The full generation of a sinogram can be seen in the image 5.1, where no mask

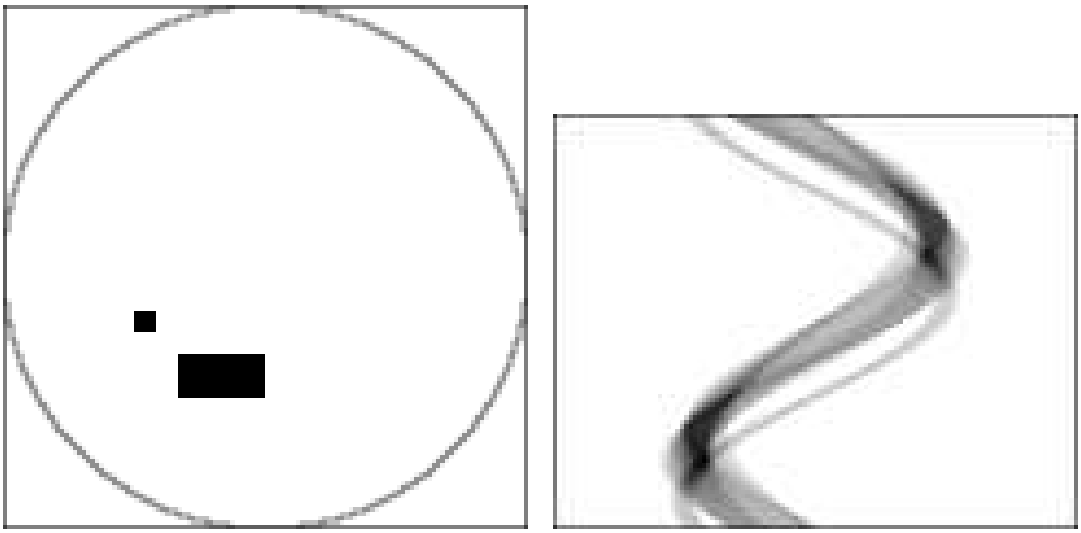
was applied.



(a) Original Phantom (b) Generated Sinogram

Figure 5.1: A mask applied to the theoretical phantom, and the generated sinogram (in reversed colours)

Otherwise, if the boolean is **false** and a mask is passed, will be generated a partial sinogram: for every sampled point in the ray, every contribution from the voxels is tested through the mask, in order to verify that the related voxels were selected.
An example of sinogram generated using a mask can be seen in figure 5.2, where two different regions of the phantom combine to generate a partial sinogram.
The sum of the signals from the selected voxels is then multiplied by the computed value of the attenuation function 3.2 for the considered point in the ray.
Thus the generated sinograms are corrected by the same physical corrections used in the reconstruction code. This makes the generation of sinograms a formidable tool for some interesting purposes.
One possible use of this feature is in verifying that the reconstruction code works well. Starting from a theoretical image, called *phantom*, it is possible to generate theoretical sinogram, and then reconstruct it. The result can be compared with the original image, to prove that the reconstruction works, and using some tools provided in the library, to even measure which is the real performance in the reconstruction quality.



(a) Mask on Phantom (b) Generated Sinogram

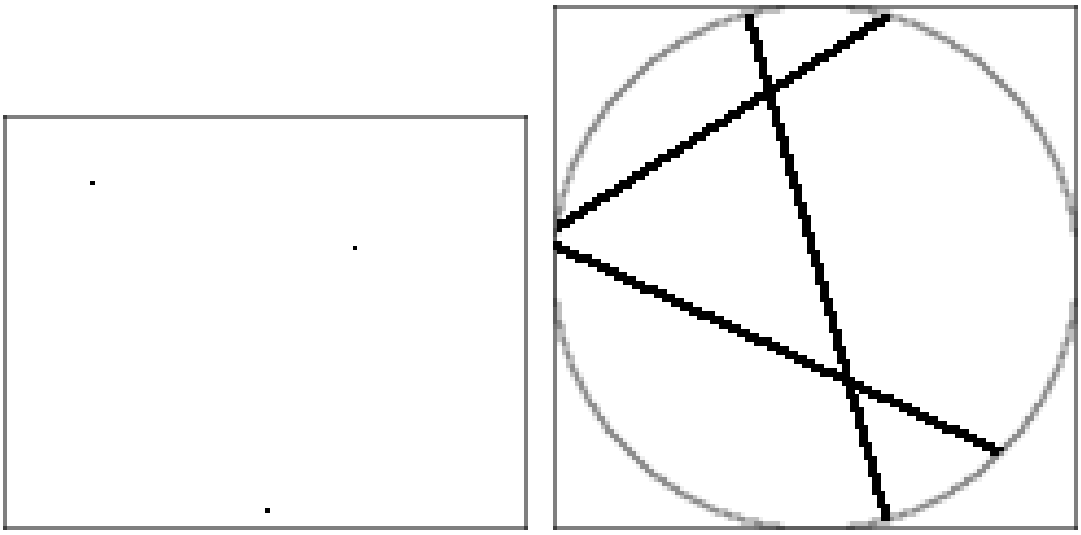
Figure 5.2: A mask applied to the theoretical phantom, and the generated sinogram

Another interesting use of this feature is in predicting quantitative results for the tomographic reconstruction. Usually the reconstruction can be taken just as a qualitative tool, that establishes a correlation between the intensities of the voxels, but the absolute values cannot be taken as the real quantities in the voxels. Since a reconstruction was performed on the real dataset, if the generated sinogram agrees with the real dataset, the reconstructed image is converged to a physical result, and more over the single values in the image can be scaled in order to fit the real data. After the fit, the quantities in the reconstructed image can be said to be a quantitative estimation of the points in the sample.

5.2 Regions of Interest

The partial generation of a sinogram, based on a mask to be applied to the phantom is what can be called generation from a *region of interest*. In my library can be found more functions for such kind of operations. In particular there are functions that generate masks for both the sinogram and the phantom.
The principle is simple, and is based on the geometric correlations between the phantom and the sinogram.

A point in the sinogram is directly related to a line that passes through the image phantom, and samples it. So the sampled voxels along the line will be related to such point in the sinogram.
A point in the image phantom is related to other points in the sinogram, in exactly the same way as before, but along a sinusoidal curve, instead of a straight line.

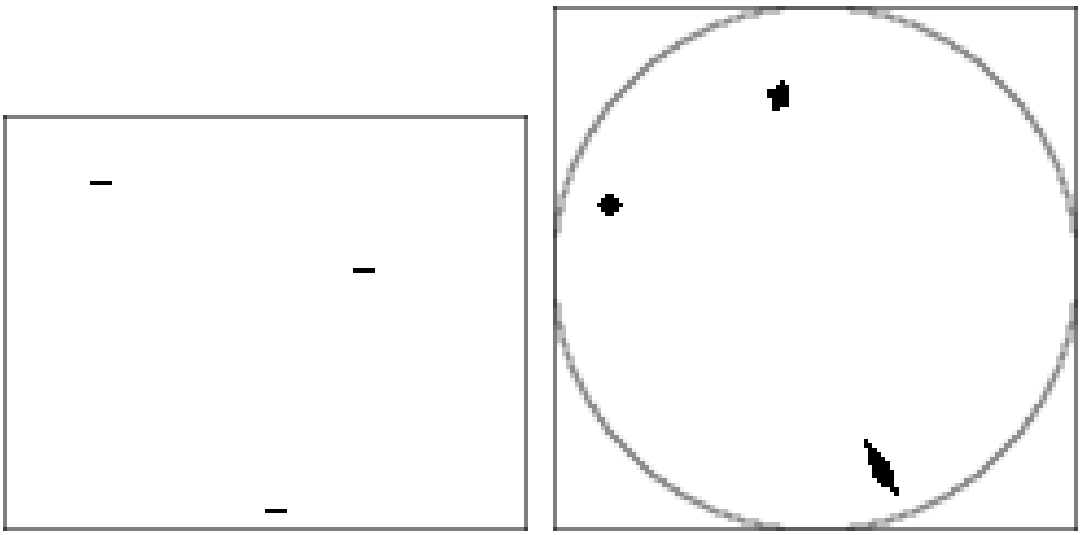


(a) Mask on Sinogram (b) Region of Interest on Phantom

Figure 5.3: The the point mask applied to the sinogram gives rise to lines on the phantom

These relationship can be striking at first sight, but can be better understood looking at some visual examples. In figure 5.3 can be seen the effect of selecting three points in the sinogram to the left. It generates three lines on the phantom to the right. Another example of this kind of correlation could be seen in image 5.2 in the previous section.
What relates the two subsets of points are the weights assigned to the voxels during the sampling phase. So for example, in the generation of a region of interest for the phantom, a matrix of the same shape of the phantom is filled with the sum of the weights associated to the voxels touched by the lines related to the points selected in the sinogram. But finally, in the generated region of interest, pixels are chosen on the basis of two factors: the sum of the weights and the application of a threshold to those sums. So pixels in the ROI are selected only if the sum of the weights that

link them to the rays associated to the points in the sinogram is enough.



(a) Mask on Sinogram (b) Region of Interest on Phantom

Figure 5.4: Broader selections on the sinogram, with an applied threshold, correspond to spots of ROI on the phantom

A visual example of this procedure is in figure 5.4, where more points in the sinogram were selected, but a higher threshold was applied to the sum of weights in the phantom.

Let's now report and comment the code that does this translation. In listing 5.3 is reported the function that generates a raw mask without any threshold applied. The code is pretty simple: it uses a function that was already explained before, and reported in code block 4.14.

The key points in listing 5.3 are the check of the mask to be of the same shape of the sinogram, and the loop over all the rays of every projection, that if selected are re-projected on the new ROI.

Unfortunately the names for the mask and the ROI are misleading since the first is called *selec*, and the second is called *mask*.

Listing 5.3: Function to generate a ROI for the phantom, from a selection on the sinogram

```
void
GeometryClient::getMaskFromSinoSelection( const BaseGeometryTable & gt,
    const Sinogram & sino, const BinVec2D_B & selec, BinVec2D_D & mask)
{
  /* Initial checks */
  CHECK_THROW(!gt.empty(),
      InitializationException("The geometry table is empty"));

  checkMask(selec, sino);

  /* Give the shape to the mask */
  mask.reset(gt.getPhantomWidth(), gt.getPhantomHeight());

  /* Let's fill the mask */
  const uint32_t rotSize = gt.size();
  for(uint32_t rot = 0; rot < rotSize; rot++) {
    const Rotation & rotRec = gt.getRotation(rot);
    for(uint32_t ray = 0; ray < gt.getTotRaysPerRot(); ray++) {
      if ( selec.get(rot, ray) ) {
        applyCorrections(mask, rotRec[ray]);
      }
    }
  }
}
```

5.3 Python Wrapper

The last topic in this chapter is about one of the main and largest features of this library. Of extreme importance has been developing an interface for *python*, because this interpreted computing language is becoming more and more used in the scientific community, thanks to the mathematical library called *NumPy*.

The strategy behind this interface implementation has been to create a multi layered interface, with access to both the lowest level functions, accessible in C++, and to some high level calls that can make life a lot easier.

To give an idea about the size of this python interface, let's give rough sizes of the code, based on code lines for the different sections. The whole line count for the C++ code is of $\sim$ 8800 lines, and the python interface written in C++ is of $\sim$ 2200 lines. Then the pure python part is of $\sim$ 1800 lines of code and examples. This explains why the python wrapper is one of the most important parts of the library. From the user side it materializes as a collection of loadable python modules. The main ones are:

1. ARTHelper (from file arthelper.py)

2. BatchReconstr (from file batch_reconstr.py)

3. ImageSignals (from file image_signals.py)

The first is the true interface to the C++ library, since it inherits the python class *art* (from file art.so) which is the compiled raw interface from the C++ part of the wrapper. If the user wants to gain access to all the low level functions of the C++ library from the python side, aside from some high level utility functions, this is the module to import.
The second module is a layer more over the previous module and can really compact the number of lines to perform a job. It will be discussed in a lengthier way in 5.3.3.
The third module is a collection of utilities for signal processing and the generation of theoretical phantoms with different shapes and features.

5.3.1 Basic API

The API is organized in an Object Oriented way, with a much higher effort on consistency than in the C++ side. While in the C++ library the programmer needs to keep track of changes between interdependent objects, usually in the python module this is not needed. A big effort on making auto-configuration as effective as possible was made.
Auto-configuration is exploited very well in geometry determination from sinograms or phantoms. Usually a sinogram is enough to fully determine the geometry of the problem and suddenly perform the reconstruction.
From the first module described before, can be instantiated an object of the class *ARTHelper*, which exposes a broad API. The object can manage two sinograms and two phantom, with associated geometry and preferences. One sinogram and one phantom are dedicated to the transmission tomography, while the other sinogram and phantom are dedicated to the other quantities. This makes it easy for the user to perform attenuation corrections, since a specific API is exposed.

Listing 5.4: High Level API of ARTHelper class

```
1 class ARTHelper(art.ART):
      def __init__(self, args = None):
```

```
    def hasTransmission(self, geometryOps):
    def hasFluorescence(self, geometryOps):
    def hasDiffraction(self, geometryOps):
    def hasCompton(self, geometryOps):
    def applyLowerThreshold(self, matr, lower_threshold = 0, offset = True):
    def applyPadding(self, matr, padding = 0):
    def makeSinogramFromFileFixedAngles(self, filenameOrMatrix):
    def makeSinogramFromFileAtGivenAngles(self, filenameOrMatrix, angles):
    def prepareAbsorption(self, sinogram, angles = None):
    def reconstruct(self, sinogram, angles = None, hasGeometry = False):
    def iterateOnce(self):
```

On how to develop using these classes and utilities, the reader is suggested to take a look at the examples and at the code itself, for example in the batching class where it makes use of the lowest level functions.

5.3.2 Saving Operations

Another key feature of this interface is the transparent and automated recycle of the geometry for similar reconstructions.

Listing 5.5: Method for the generation of the geometry

```
static PyObject *
makeGeometryFromSinogram(PyObject * self, PyObject * args)
{
  PYFT_BEGIN_FUNCTION_BODY( art )
  PYFT_CHECK_THROW(art.sinogram, PyExc_RuntimeError,
      "Sinogram object not instantiated!");

  bool hasGeometry = false, forceAbsorpSino = false;
  PyObject * _hasGeom = NULL, * _forceAbsorpSino = NULL;
  if (args && PyArg_ParseTuple(args, "|OO", &_hasGeom, &_forceAbsorpSino)) {
    if(_hasGeom == Py_True) {
      hasGeometry = true;
    }
    if(_forceAbsorpSino == Py_True) {
      forceAbsorpSino = true;
    }
  }

  const bool useAbsorpSino = !art.sinogram->size() || forceAbsorpSino;
  if (useAbsorpSino) {
    PYFT_CHECK_THROW(art.sinogramAbsorption && art.sinogramAbsorption->size(),
        PyExc_RuntimeError,"Absorption Sinogram selected but not initialized!");
```

```
    }

    InfoPrintf(("Initializing Geometry Table\n"));
    try {
27    Sinogram & sino = *(useAbsorpSino ? art.sinogramAbsorption : art.sinogram);

      if ( !(art.geomTable && hasGeometry
              && art.geomTable->isCompatibleWith(sino)))
      {
32      GeometryFactory factory(*art.prefs);
        ASSIGN_NEW(art.geomTable, factory.getGeometryFromSinogram(sino));
      }
    } catch (const InitializationException & e) {
      PYFT_REPORT_EXC(PyExc_ArithmeticError, e.what());
37  } catch (const BadSolidAngleException & e) {
      PYFT_REPORT_EXC(PyExc_ArithmeticError, e.what());
    }
    Py_RETURN_NONE;
    PYFT_END_FUNCTION_BODY
42 }
```

In listing 5.5 is clearly visible at lines 29 and 30 that before attempting to build a new geometry, it is evaluated the possibility to reuse it.

Rebuilding the geometry can be an extremely expensive task, so saving such a rebuild can save time in both batching and interactive mode. On a slow computer where the total time of geometry definition and tomographic reconstruction can be of ~ 60 seconds, on a second reconstruction from a sinogram compatible with the geometry, the total time can drop to $\sim$ 40-45 seconds.

5.3.3 Batching

One of the most interesting features is Batching. The module *BatchRecontr* gives access to some interesting classes for processing multiple files with just few commands. It makes it possible, with just few lines of code, to process all the sinograms from a directory, perform physical corrections, and put the reconstructed images in another directory.

Listing 5.6: Example script for processing a load of diffraction sinograms

```
geomType = batch_reconstr.arthelper.ARTHelper.TRANSMISSION | \
           batch_reconstr.arthelper.ARTHelper.DIFFRACTION
```

```
Indir       = "path/to/Sinograms"
Outdir      = "path/to/Reconstructions"
AbsorbSino = "path/to/Transmission_sinogram.edf"
MinAndMaxAngles = (5., 30.)

options = { "MinAngle" : 0, "MaxAngle" : 180, "DampingFactor": 0.1,
            "MaxIterations" : 20, "GeometryOps" : geomType,
            "SelfAbsorptionCorrection" : True }

rec = batch_reconstr.SingleThreadBatchReconstr(
        Indir, Outdir, options, AbsorbSino, MinAndMaxAngles, 0.000, 0.0 )

print( "Start batch reconstruction from data without metadata:\n"
       " Indir:      %s\n Outdir:     %s\n AbsorbSino: %s\n"
       " Min angle:  %f, Mmax angle:  %f" %
       (Indir, Outdir, AbsorbSino, MinAndMaxAngles[0], MinAndMaxAngles[1]) )

rec.run()
```

In code block 5.6, there is a clear example of how to perform a batch reconstruction. To be noted is the variable *MinAndMaxAngles*, which gives an interval of aperture for the angles measured on the CCD detector. This is based on the supposition that in the input directory, all the sinograms are ordered and associated to an increasing angle of aperture on the CCD. As an alternative it is possible to pass an array that explicitly specifies the aperture angles for different sinogram. Of course, this variable is only needed if performing self-absorption corrections, otherwise can be **None**.

Another interesting batching class is the one that, exploiting python threads, can use multiple threads and process more than one reconstruction per time. On machines with more than one core and enough ram, this feature, along with the geometry recycle, can save a big amount of time. Every thread will have a private geometry definition, so the memory needed will be multiplied by the number of spawned threads, but the time required will be almost divided by the number of CPU cores available.

5.3.4 Other utilities

Finally to the set of tools in the library, have been added some nice utilities. As already said, one of the classes in the python module *ImageSignals* is the generator

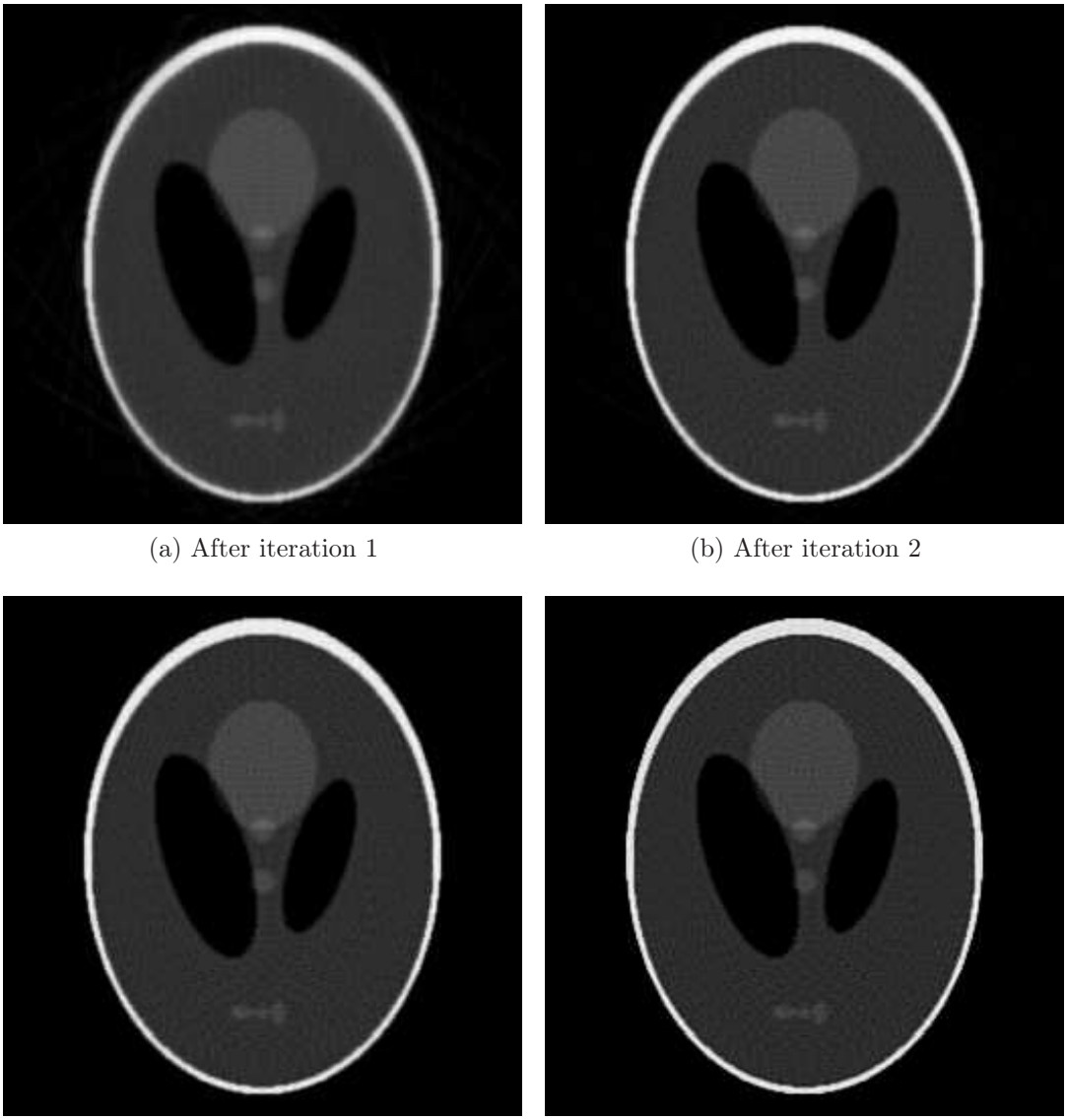
of new theoretical phantoms.
Theoretical phantoms are extremely useful during development, in order to verify the result and correctness of the algorithmic implementations. They can also serve to verify extreme conditions and compare them with the reconstructed images in order to verify how good is the reconstruction algorithm at work.
In the *PhantomGenerator* class, two classic phantoms are already included. One of them is defined on three dimensions, so different sections at different resolutions of them same pattern can be generated.

Along with the theoretical phantom generator, there is a class called *ImageSignals* that perform Signal-to-Noise Ratio analysis of two images. This can be very useful in verifying which is the residual noise in a reconstruction from a theoretical sinogram, and how different algorithms and implementations do perform on the SNR side.
Associated to the *ImageSignals* class, is the *NoiseMaker* class that can generated white noise, and could be very useful in verifying how good scales the quality performance of the reconstruction algorithm with the increase of the noise in the theoretical phantom or sinogram.

In this last chapter will be showed some results from the usage of the FreeART tomographic library, comparing also them to the ones of other pre-existent and well performing reconstruction tools.
The discussion will begin with the reconstruction of synthetic data, and then real data for some different kinds of experiment will be introduced.
This chapter will not add any technical detail on the code itself, but instead will try to demonstrate that the library gives good results. To prove it, it will be shown that they are in agreement with the ones from the other tools.
Some limitations of the current implementation may also be discussed when dealing with helical scans.

6.1 Theoretical Reconstruction

As already stated before, the theoretical phantom generation can be a good way to test if the reconstruction works as expected, and evaluate its performance.
As a test, the sinogram in image 5.1b can be taken as input and the reconstructions are the ones in the group of pictures 6.1. Taking a look at those reconstructions, it can be deduced that while the first reconstruction is a bit blurred and with some streak artefacts, the subsequent ones are more and more sharp.
One would be tempted to do an overwhelming number of iterations, in order to gain as much as possible from the algorithm. Sadly this is not possible because of approximations in the model, and because of numerical errors arising from the computation.
The kind of noise that is thrown in the reconstruction by the algorithm is the *Salt*



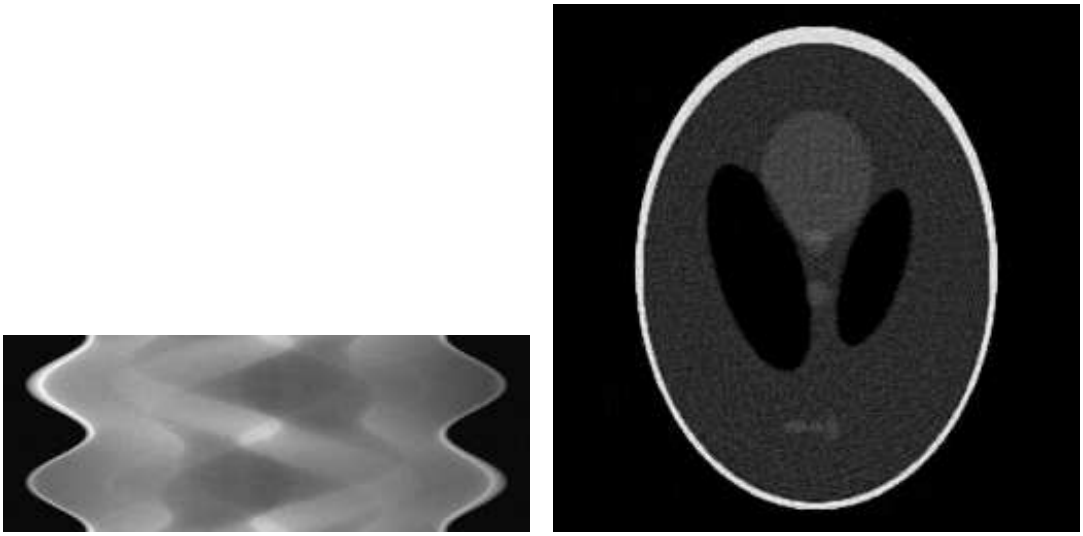
(a) After iteration 1 (b) After iteration 2

(c) After iteration 3 (d) After iteration 8

Figure 6.1: Reconstructions of the most known theoretical phantom

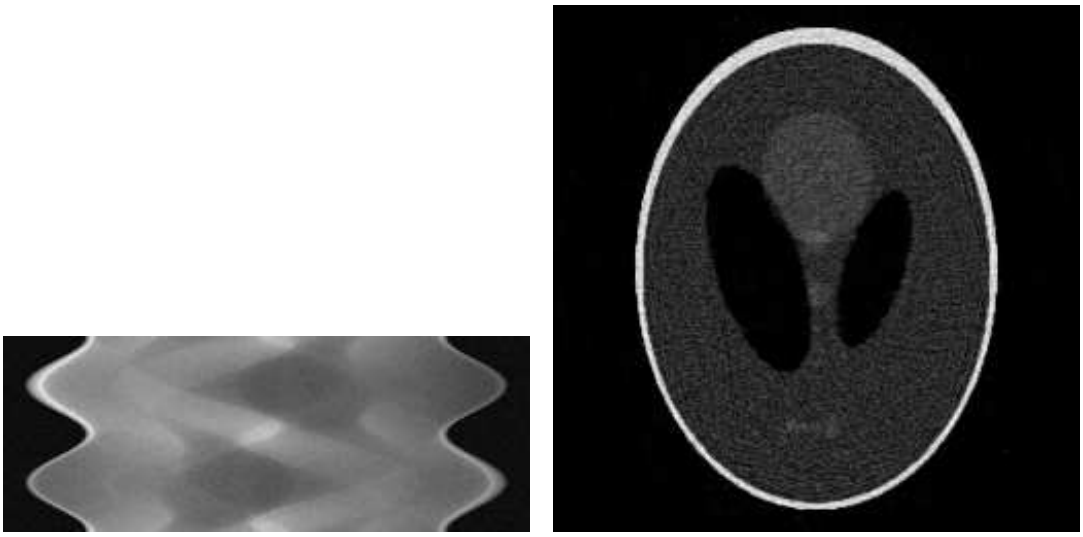
and Pepper noise, which is located in the higher frequencies. One of the filters usually used to limit the SAP is the so called *Median Filter*. It was not implement a median filter in the library, because a lot of good implementations can be found in other packages.

Using the classes from the module *ImageSignals*, let's now have a look to some tests on how the algorithm behaves under the addition of white noise to the theoretically generated sinogram.



(a) Sinogram (noise 5%) (b) Reconstructed Phantom

Figure 6.2: Reconstruction after 10 iterations of theoretical phantoms, with the addition of noise on 90% of the pixels of the sinogram, with a flat distribution of noise between +5% and −5% of the highest peak in the sinogram



(a) Sinogram (noise 10%) (b) Reconstructed Phantom

Figure 6.3: Reconstruction after 10 iterations of theoretical phantoms, with the addition of noise on 90% of the pixels of the sinogram, with a flat distribution of noise between +10% and −10% of the highest peak in the sinogram

Two different tests were performed: they have two different noise intensity levels but the probability distribution function is the same. The 90% of the sinogram pixels

was covered with a flat error distribution, that, in the most noisy case, had range between +10% and −10% of the highest peak in the sinogram, while in the other the range was between +5% and −5% of the highest peak.
In the clean reconstruction the tools for the analysis of the SNR level gave as result:

```
Clean reconstruction
S/N: 39.755570, Peak S/N: 655.603562, noise^2: 99.962849
```

In the less noisy reconstruction, the highest peak in the sinogram was 0.000516, while the most intense noise peak was 0.000023, so the percentage of the noise peak against the highest feature was around 4.4%.
Applying again the SNR tools:

```
Noised reconstruction
S/N: 34.517160, Peak S/N: 569.217666, noise^2: 115.133461
```

Finally in the most noisy case, the highest peak in the sinogram was again 0.000516, while the most intense noise peak was 0.000053. In this case the percentage of the noise peak against the highest feature was around 10.25%. The result from the tools for SNR analysis was:

```
Noised reconstruction
S/N: 24.371858, Peak S/N: 401.912917, noise^2: 163.060199
```

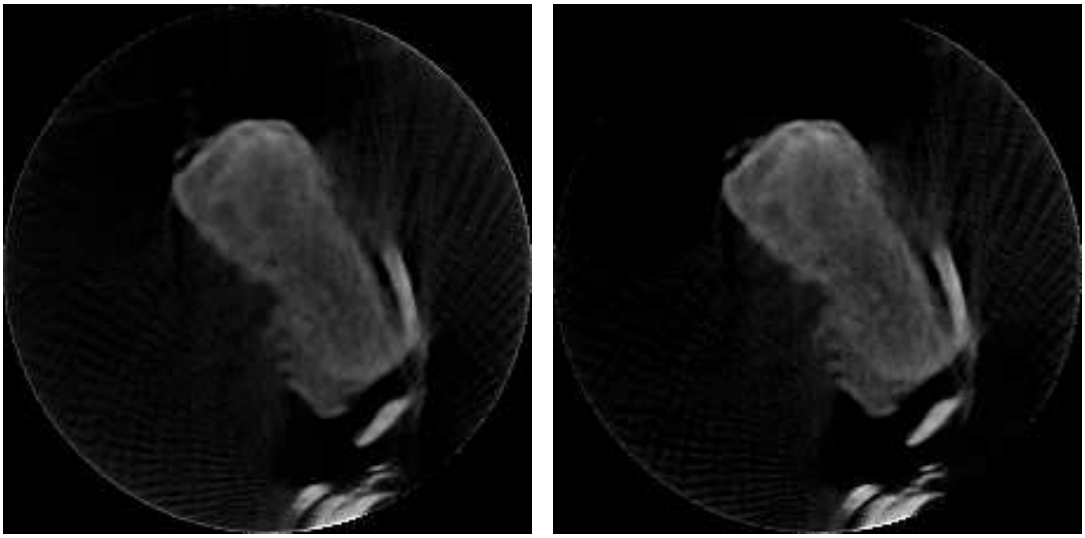
Before commenting the numbers, let's take a look to the pictures, because the most important thing is that this implementation demonstrates both to be able to handle synthetic data in the right way, and to be robust against the introduction of a good amount of noise. Even on the most noisy sinogram the reconstruction was good and the smallest features were still visible.
Then, when it comes to numbers, it is quite evident the correlation between the introduction of noise and the decrease in the SNR level. In the less noisy case the SNR was still good, and the image 6.2b shows a really good agreement with the theoretical phantom.
Unfortunately, the drop in SNR was higher in the second case, and the "reconstructed noise" was also much higher than the other.

6.2 Diffraction Data Reconstruction

It will now be shown how the library performs with real data, and then a comparison with the other reconstruction tools will be made.

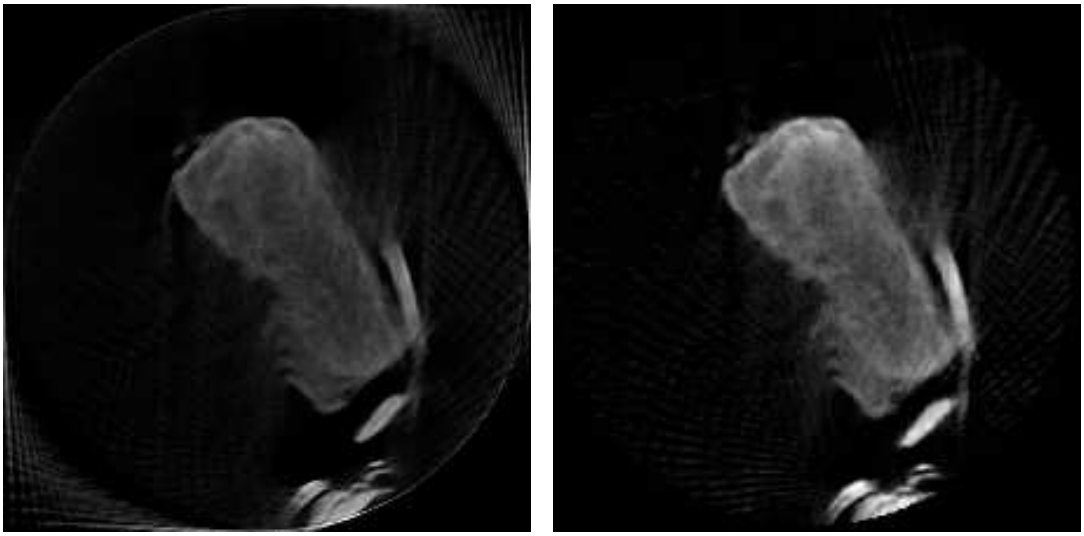
Figure 6.4: Sinogram of micro-sample from ID22 (ESRF)

As a courtesy of ID22 at the ESRF, a set of sinograms coming from a diffraction experiment were made available. Since a tool developed by Pierre Bleuet was already in use at ID22, it was easy to make a comparison with it.
One of the sinograms from the set of the experiment is reported in picture 6.4. It is part of a much larger set, with around 2000 sinograms from the different channels of the acquisition.
It was a diffraction experiment, so all the channels correspond to a different circle on the CCD detector, with an associated aperture angle.
The choice to show this sinogram is due to the fact that the dataset is cleaner than most of the others, without major artefacts. It suffers from a part of the sample going out of the scanned region, and giving rise to the ring artefact, as can be clearly seen in picture 6.5a.
In picture 6.5b, it is instead possible to see the effect of applying a simple filter to enhance the contrast. Passing a negative value to the function *applyLowerThreshold* it detects automatically the minimum non-zero value in the sinogram, and filters it using such value as a threshold. This is not supposed to give quantitative results, but instead a much better contrast.
The result can be appreciated comparing the two reconstructions in image 6.5.
It seems to reconstruct something physically consistent, but to prove it is the right reconstruction, let's now compare it with the results from the other tools.



(a) Without minimum filter (b) With minimum filter

Figure 6.5: Reconstruction of sinogram 6.4, both with and without applying a lower threshold equal to the minimum value in the sinogram



(a) P. Bleuet (b) PyHST

Figure 6.6: Reconstruction of sinogram 6.4, with other tomographic softwares in use at the ESRF

These reconstructions are reported in image 6.6: they show clearly that my code gave a similar reconstruction, and do validate my results. Things could be tweaked a bit more, because in my reconstruction in some inner features of the sample, there seems to be an aliasing problem, due to a sub-pixel misalignment of the image centre.

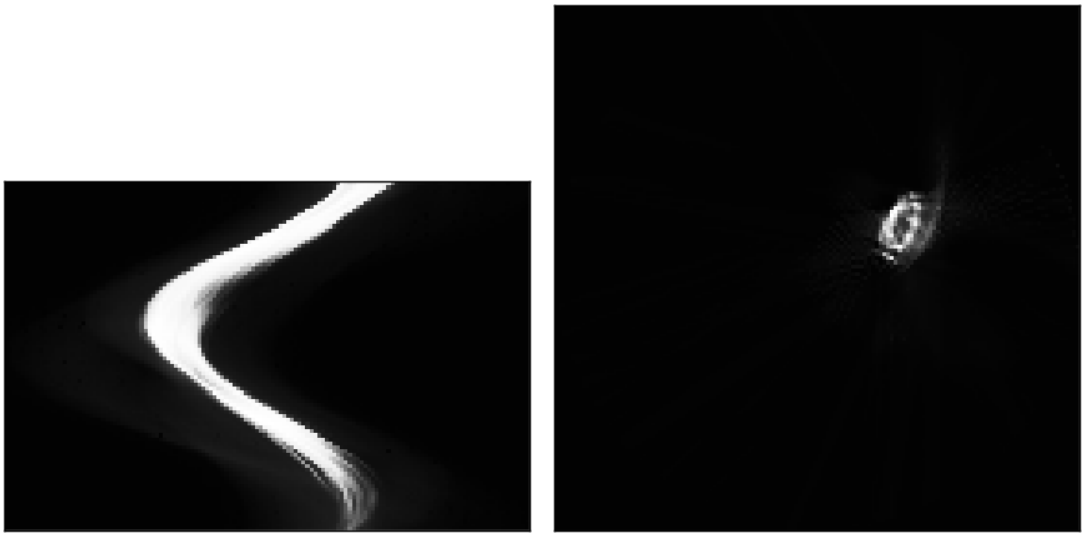
It is currently not possible in FreeART to do a sub-pixel alignment, but could be a future improvement.
Interestingly enough, the reconstruction made with PyHST does not seem to suffer from the ring artefact, but this should be related to a built-in filter that cuts out the ring in a post-process phase of the reconstruction.
However, removing the ring artefact seems to give a gain in contrast. So this could be another interesting feature for future developments.

6.3 Fluorescence Data Reconstruction

In this last section about reconstruction results, two different cases will be discussed, and some strength points and downsides of the library will be analysed.

6.3.1 Helical scan

This first case is about the reconstruction of the K line of Ca fluorescence signal from a sample whose code name is “globo”, and which is shown in figure 6.7.



(a) Sinogram (b) Reconstruction

Figure 6.7: Reconstruction of K Ca line with helical scan

The *spec* command to scan it, was:

```
helisetup samr samv heliphi heliz 0.012
zapimage samh -1.345 -0.745 136 heliphi 0 10 900 500 0
```

So can be deduced it was an helical scan, and the sinogram used is the first turn of the complete scan. Since the sinogram is done over a 2-dimensional plane, this is in contrast with one of the prerequisites of the 2-dimensional tomography. The result is clearly the coming out of artefacts related to the of different space regions of the sample at different projections.
Since the vertical movement is not so dramatic, the reconstruction will still be close to the reconstruction of a real in-plane scan. A solution to this problem could interpolating the first and last projections, in order to give a better estimate of that is in the middle. Unfortunately, this is not always feasible because in the first and last turn of the helical scan, one of the two needed factor for the interpolation is missing.
Moreover, in FreeART no tool is provided to easily interpolate head and tail of a sinogram. Anyway, even in a not ideal case, the algorithm works pretty well and some sharp features in the sample are clearly visible.

6.3.2 Solid angles

The second case is about a biological sample, whose sinogram is reported in picture 6.8. Also in this case, the λ of the emitted radiation is not important since what will now be discussed is the *solid angle correction*.
This is an intrinsic problem in Fluorescence and Compton signals, so the corrections can be of great interest.
The important aspect of this sample is that it is big enough to give appreciable differences in the detector solid angle coverage seen by the different voxels in the sample. If not taken into account, this could give rise to some vertical streak artefacts on the borders of the sample's reconstructed image.
The artefact is clearly visible in image 6.9, where both reconstructions with PyHST and FreeART, are shown. PyHST, which assumes transmission tomography, does not perform any solid angle correction. FreeART instead, which is instructed that it

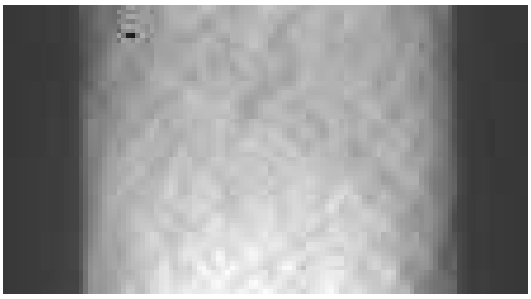
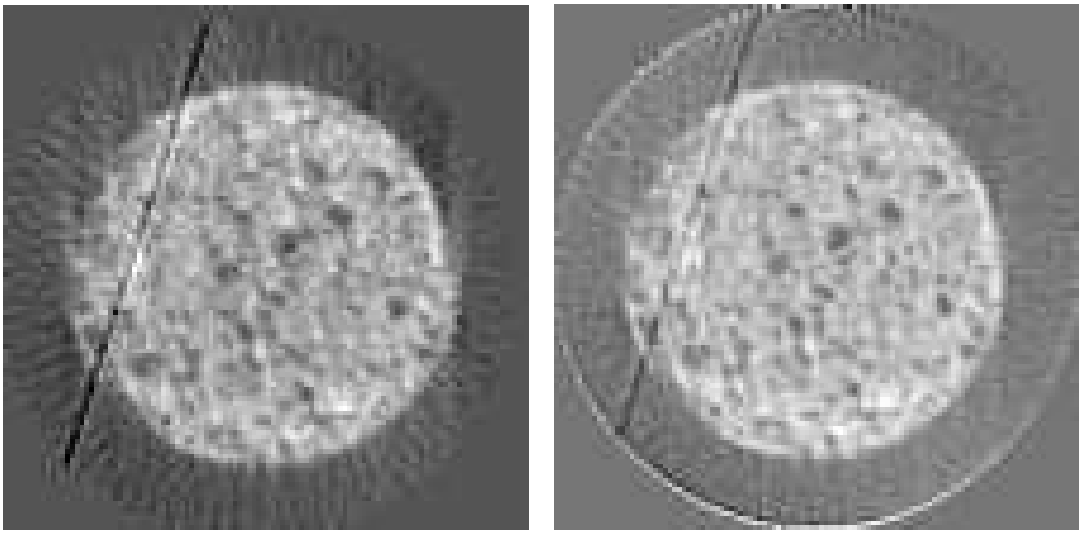


Figure 6.8: Sinogram of a biological sample



(a) PyHST (b) FreeART

Figure 6.9: Reconstructions of sinogram 6.8 without(a) and with(b) solid angle correction

is a fluorescence reconstruction, introduces a term in the attenuation function 3.2, which takes into account the different solid angles seen by every sampled point. Even in the reconstruction 6.9b, a little artefact is still visible, but this is due to the fact that the distance between the sample and the detector, and detector's shape and dimensions were not known with enough precision.

6.4 Conclusions and future outlooks

The most important conclusion that can be taken is that FreeART works well as a tomographic tool, even in respect to other pre-existent tomographic solutions.

However, it was started from scratch and it is just a six months of one man work, so it misses many features and utilities that other software solutions do have.
Let's list some of them:

Parallel processing The implementation is a sequential CPU implementation, so it is possible to gain from multi CPU machines just using python batching. No GPU implementation is available, so with big scans, it can be painfully long to reconstruct on slow machines.
Anyway, it should be mentioned the fact that the library is intended to be used for small scans, for example in fluorescence, where the time of exposure is a big limiting factor on the number of projections and offsets per rotation that can be taken during the measurement.

Interpolation of helical sinograms As already introduced in 6.3.1, helical scans are not explicitly supported in FreeART, but they should, because they are often used in Fluorescence experiments: such scans can greatly reduce the acquisition time, which is usually longer during this kind of measurements.

Deal with non parallel beams and oblique angle beams Because of the intrinsic 2-dimensional kind of the reconstruction, oblique angle beams cannot be implemented in the current form of FreeART.
For what concerns with non parallel beams, it is possible, and, if needed, it could also be relatively easy to introduce such support in the library.

3-dimensional native reconstruction The FreeART implementation is purely 2-dimensional, but in some applications it could be interesting to have a native 3-dimensional reconstruction algorithm. It could allow to support proper self-absorption in Diffraction experiments, helical scans, and oblique angle beams in a native and straight forward way.
The big constrain to this possible development is the intrinsic sequential CPU implementation, which is slow, and to be faster it stores a lot of precomputed geometric

information in memory. In 3D this would make the memory usage explode, and make the reconstruction times too long to be of any use or interest.

Other important features, which are missing from the library, may not be listed, but these are the ones that people have been interested in, during the meeting on tomographic software at the ESRF at the end of March 2011.

There is an ongoing development of an easy to use *Graphical User Interface* for this library, written in python and optionally usable from PyMca. The person in charge for this development is the author of PyMca himself.
This library will be still maintained and developed in the future, and the author may also find myself working again on it, because the physical corrections could lead to interesting applications in other fields of tomography.

Finally let's summarize what were the goals met in these six months of development:

2-dimensional tomographic reconstruction The core part that performs the 2-dimensional reconstruction of an image, from a sinogram, is complete. The reconstruction is correct and even if it is a sequential CPU implementation, it is quite fast.

Physical corrections The reconstruction can be done performing some corrections that try add into the model of the simple ART scheme, the physical nature of the sample and of the instrumentation.
There are code paths to compute and use corrections for *detector solid angle coverage*, *incoming beam attenuation*, and *self-absorption attenuation*.

Sinogram and ROIs generation As extensively talked in 5.1 and 5.2, the core code is able to perform theoretical projections of the sinogram and generate masks to investigate regions of interest.

Python Wrapper The python interface is one of the largest parts in the FreeART library and provides both access to the lowest level functions and to a simple and

quick interface.
The extensive work done on this interface is supposed to make the experience with such an interface as smooth as possible, granting consistency on parameters change, and reuse of precomputed geometric information, in order to skip unneeded operations.

Batching utilities The batching classes in the python interface give a useful tool to reconstruct a big bunch of sinograms, being also able to take advantage of multi CPU machines.

Performance evaluation utilities As already discussed in 6.1, a little set of classes, useful for evaluating the quality of the reconstructions in different noise cases, was also added to the python interface.
This feature is not so useful for an end user that uses the library as a black-box, but grants a set of tools for the developer to verify how the new features perform. This grants an higher quality of the library, which is indeed an advantage for the end user, too.

Bibliography

[1] http://www.ll.mit.edu/mission/electronics/AIT/hisensitivityimage.html

[2] http://en.wikipedia.org/wiki/X-ray_tube

[3] http://en.wikipedia.org/wiki/Synchrotron

[4] Yoni De Witte, *Improved and practically feasible reconstruction methods for high resolution X-ray tomography*, (Universiteit Gent, Faculteit Wetenschappen, 2010)

[5] J. Als-Nielsen and D. McMorrow, *Elements of Modern X-Ray Physics* (Wiley, New York, 2001)

[6] A. C. Kak and M. Slaney, *Principles of Computerized Tomographic Imaging*, IEEE Press, New York, (1988)

[7] B. Golosio, A. Simionovici, A. Somogyi, L. Lemelle, M. Chukalina, A. Brunetti, *Internal elemental microanalysis combining X-ray fluorescence, Compton and transmission tomography*, Journal of Applied Physics, vol. 94, n. 1, p. 145-156, (2003)

[8] R. Eisberg, R. Resnick, *Quantum Physics of Atoms, Molecules, Solids, Nuclei, and Particles*, (Wiley, New York, 1985)

[9] N. W. Ashcroft, N. D. Mermin, *Solid State Physics*, (Sanders College, Orlanda, Florida, USA, 1976)

[10] M. Frontini, *Fondamenti di calcolo numerico*, (Libreria Clup, Cittá Studi, Milano, 2003)

[11] S. Kaczmarz, *Angenaherte auflosung von systemen linearer gleichungen*, Bulletin International de lâĂŹAcademie Polonaise des Sciences Lettres A, 6-8A:355âĂŞ357, (1937)

[12] P. M. Joseph, *An improved algorithm for reprojecting rays through pixel images*, IEEE Transactions on Medical Imaging, 1(3):192âĂŞ196, (1982)

[13] M. Jiang and G. Wang, *Convergence of the simultaneous algebraic reconstruction technique (SART)*, IEEE Transactions on Image Processing, 12(8):957âĂŞ961, (2003)

Thanks

In queste pagine vorrei poter ringraziare molte persone, perché tante persone hanno caratterizzato questi anni di specialistica, che mi hanno portato al termine di un percorso della mia vita.

Primi fra tutti, è giusto che ringrazi chi mi ha dato l'opportunità, i mezzi, i consigli ed una guida in questo lavoro di tesi.

Il mio supervisore Armando, che molto modestamente al termine del mio stage ha detto "Io non ti ho detto quello che dovevi fare o come lo dovevi fare: io ti ho solo detto cosa non dovevi fare!". Armando che in realtà mi ha aiutato più di quanto lui creda, dandomi consigli preziosi, spunti, materiale e trasmettendomi tanto, ma veramente tanto, senso pratico.

Sono dispiaciuto solo di non aver provato la sua paella, ma non mi do ancora per vinto.

Poi viene Claudio, che mi ha insegnato a non dare mai del "lei" sul posto di lavoro, neanche ai propri capi. In realtà con Claudio ho condiviso anche molti momenti, dal puro pettegolezzo a quelli di profondo impegno scientifico, nonché la risoluzione di problemi di elettromagnetismo che rendevano necessario l'utilizzo delle trasformate di Laplace.

A Claudio va anche un ringraziamento particolare per avermi dato l'opportunità di conoscere Wolfgang ed il suo gruppo, in cui proseguirò gli studi, facendovi il dottorato.

In quanto ad opportunità devo poi ringraziare il prof Giacomo Ghiringhelli, a cui purtroppo, per via di un esame sostenuto con lui, non sono riuscito a dare del "tu" neanche ora che sono giunto alla fine di questo percorso. Lo ringrazio però per avermi fatto conoscere Claudio ed avermi così spalancato una grande finestra sul mondo.

Della gente che ho conosciuto a Grenoble, vorrei ringraziare veramente tutti, e sono sincero nel dirlo. Ho conosciuto tante persone, tanto speciali, con cui ho condiviso proprio dei bei momenti. Citarne solo i nomi non rende loro giustizia, perché sopratutto senza i più a me vicini, durante i sei mesi non sarei riuscito a tirar avanti. I colleghi di lavoro ad esempio: Eleonora[1], Niccolò[2], Dimitris[3], Matteo, Alessandro, Jerome, Tom, Igisso, Lydia, Vaggelis, Matthew. Poi ci sono le amicizie che ho fatto al di fuori del posto di lavoro, e che hanno riempito il mio tempo libero: vanno citati assolutamente Fabrizio, Francesca, Giuseppe, Mattia, Mauro, Nikos, le "francesi" (per dire) Elise[4] e Lisa, ed il proprietario del Bayard[5].

Non mi devo però dimenticare della sciura, Mme Comtat, che mi ha ospitato in casa sua per i mesi dello stage, e che mi ha praticamente insegnato il francese. La signora che ha praticamente badato a me, come se fossi un suo nipote, e mi fatto tanti favori, e a cui fare i favori da parte mia era un piacere, perché così potevo restituirgliene qualcuno.

Vengono poi i miei compagni di università in questi anni di specialistica, con cui ho passato altrettanti bei momenti, e che vorrei nominare tutti, ma rischierei di dimenticarne qualcuno. Molto probabilmente non li rivedrò più, ma ogni volta che torno giù a Milano, non perdo mai l'occasione di passar dove so che vanno a mangiare al mezzogiorno per poterli rivedere e passar qualche momento di serenità.

Ci sono anche alcuni professori del poli che vorrei ringraziare sentitamente. Uno fra tutti è il professor Matteo Tommasini, con cui oltre ad aver scoperto una materia favolosa, ho sviluppato un rapporto di amicizia, ed a cui mando tutti i miei auguri per quando diventerà padre a Settembre!

Fondamentali poi per questi anni di fisica sono stati il prof. Dupasquier, che ai tempi della triennale mi ha indicato il percorso di studi migliore per poter passare poi ad

[1]Passata a miglior vita, ma nel vero senso della parola, perché ha iniziato il suo PhD a Valencia; e con cui tutti i giorni era costantemente una Tragedia, ma con cui le giornate passavano anche veramente bene

[2]Passerà presto a miglior vita, anche lui nel vero senso della parola, perché passerà due anni del PhD a New York.

[3]Passerà presto e tristemente a miglior vita se non si decide a darsi una regolata con lo stress da superlavoro.

[4]Brescianaaa!!

[5]Un lemonade s'il vous plait!

Ing. Fisica; il prof. Duò che mi ha introdotto ed a modo suo incitato nello studio della fisica; il prof. Puppin che mi ha aiutato a capire la fisica dello stato solido, quando non avevo le basi per riuscirci; ed infine il prof. Nisoli che oltre a delle lezioni favolose mi ha offerto anche la possibilità di poter eventualmente fare il PhD in uno dei gruppi di chimica quantistica più prestigiosi.

Voglio ora ringraziare i miei genitori e la mia famiglia, con un pensiero particolare a Simone e la Laura che presto si sposeranno. In questi anni, la mia famiglia è stata per me una delle cose più importanti che abbia avuto per motivi tutti personali e che quindi tengo per me. Voglio ringraziare tutti, partendo dai miei genitori, che nonostante siano dei casinisti d.o.p. e disorganizzati d.o.c. sono stati proprio dei bravi genitori; passando per i miei zii che tanto hanno fatto e tanto fanno per noi; e arrivando infine ai miei cugini che sento come dei fratelli.

Devo poi ringraziare i miei amici, che non mi fanno sentire solo e che sono per me molto di più di un gruppo di persone con cui passo il mio tempo libero nel week end. Servirebbero pagine per elencare i motivi che mi spingono a ringraziarli. Non li vedo molto, e non li vedrò quasi più ora che partirò, ma il mio pensiero va ad Alessandro, che mi conosce da sempre e molto meglio di tante altre persone, e con cui mi consiglio quando voglio un parere da "cattivo"; Mattia, con cui converso tutte le volte amabilmente, ma sopratutto mi trovo sempre a progettare tante invenzioni, aggeggi e trappole (neanche fossimo "Mignolo col Prof"), che però poi puntualmente non realizziamo mai; Boris, che è affascinato dalla fisica dei quanti, e con cui cerco sempre il punto di contatto fra la fisica e biochimica, quando non siamo impegnati a ridercela come matti; Giulio, con cui, anche se non parlo da 10 anni, riesco sempre ad esser in sintonia, ed a riconoscermi nello stesso modo di pensare; Sara che tanto guarda al mondo con gli occhi ingenui di una sognatrice, da non essersi ancora data per vinta nel desiderio di poterne raddrizzare le storture; la Betti, con cui ingaggio discorsi eruditi sugli argomenti più disparati, riguardanti la realtà sensibile o l'epistème stesso, e che, non so se considerarlo un insulto, mi ha dato del "Formalista Russo" per via del mio utilizzo della parola *Pertinenza*; Kuki, con cui gli argomenti che spaziano dall'informatica, ai fenomeni sociali legati alla musica, passando per una pseudo-misoginia, sono tutti buoni per farsi una risata e potersi lamentare di

quanto fa schifo il mondo; la Lallina, che riesce a dare tutti i giorni il sorriso a mio cugino Stefano (in tutti i sensi); Ugo, che sempre più malinconico ed a dir poco disilluso nei riguardi del mondo, mi ricorda di tener i piedi per terra; Bea, che sa esser di compagnia, ma anche tagliente quanto serve per sistemare una parola di troppo; il Loffo, dalle idee politiche più ambigue e contraddittorie che conosca; Serena, Luisa, il Flash e Fusi, che vedo purtroppo solo di rado ormai.
Per non parlare poi dei miei amici del liceo, come Scacca[6], Preso, David e Lore, che continuo a vedere con molto piacere. Con loro infatti passo da sempre dei bei momenti, e condivido tanto anche in termini di interessi o punti di vista sul mondo.

Infine voglio ringraziare la mia Lucia, che ormai da quattro anni mi è vicina e mi sostiene in ogni mia nuova sfida. Da sempre, e forse sopratutto ora che mi conosce bene, non capisco come riesca a sopportarmi tutti i giorni.
Se anche solo fingesse, nello spronarmi, nel dimostrarmi affetto e nel farmi capire quanto creda in me, la dovrei ringraziare comunque tantissimo. Lei forse non sa quanto siano importanti per me il suo appoggio e la sua dedizione, ma io me ne rendo conto tutti i giorni, perché mi accorgo che li riempie con il suo affetto.
In quattro anni si conoscono molti dei difetti di una persona, e sicuramente quasi tutti i suoi pregi. Certe volte non è facile riuscir ad incastrare tutte le esigenze di entrambi, ed il più piccolo accenno di egoismo può far accumulare malumori anche colossali. Tutto sommato, a me invece sembra che noi reggiamo bene! In questi anni non ci sono mai stati grossi problemi, e ce la siamo cavata bene anche quando io mi trovavo in un'altra nazione, per svolgere questo lavoro di tesi.
Non posso prevedere il futuro, ma posso ringraziarla guardando al presente ed al passato. La stabilità che da lei ho avuto, mi ha aiutato in questo mio duro percorso della specialistica in Ing. Fisica, che fin da subito si era dimostrato oltremodo in salita.
Ora mi rendo anche conto che non ha mai contestato le mie scelte nello studio o nel lavoro; ed a stento posso credere che fosse per arrendevolezza, proprio perché lei è una che non si arrende facilmente. È dunque evidente che anche quando non ha

[6] A cui devo riconoscere anche un grande aiuto dato nel mio approccio iniziale alla fisica.

espresso un giudizio, mi ha supportato, persino nelle scelte che comportavano dei sacrifici per il nostro rapporto.
Voglio quindi ringraziarla per tante cose, ma in particolare per il bene che mi vuole.

Nicola