X-Ray Tomographic Reconstruction — Source link

Bonnie Schmittberger

Published on: 25 Aug 2010

Topics: Tomographic reconstruction, Tomography and Industrial computed tomography

SLAC-TN-10-015

X-Ray Tomographic Reconstruction

Bonnie Schmittberger

Science Undergraduate Laboratory Internship Program

Bryn Mawr College

SLAC National Accelerator Laboratory

Menlo Park, California

August 14, 2009

Prepared in partial fulfillment of the requirement of the Department of Energy's Science Undergraduate Laboratory Internship program under the direction of Samuel Webb at the Stanford Synchrotron Radiation Lightsource at SLAC National Accelerator Laboratory.

Participant: ______________________________
Signature

Project Advisor: ______________________________
Signature

Work supported in part by US Department of Energy contract DE-AC02-76SF00515.

Table of Contents

ABSTRACT

X-Ray Tomographic Reconstruction. BONNIE SCHMITTBERGER (Bryn Mawr College, Bryn Mawr, PA, 19010) DR. SAMUEL WEBB (Stanford Synchrotron Radiation Laboratory at SLAC National Accelerator Laboratory, Menlo Park, CA 94025)

Tomographic scans have revolutionized imaging techniques used in medical and biological research by resolving individual sample slices instead of several superimposed images that are obtained from regular x-ray scans. X-Ray fluorescence computed tomography, a more specific tomography technique, bombards the sample with synchrotron x-rays and detects the fluorescent photons emitted from the sample. However, since x-rays are attenuated as they pass through the sample, tomographic scans often produce images with erroneous low densities in areas where the x-rays have already passed through most of the sample. To correct for this and correctly reconstruct the data in order to obtain the most accurate images, a program employing iterative methods based on the inverse Radon transform was written. Applying this reconstruction method to a tomographic image recovered some of the lost densities, providing a more accurate image from which element concentrations and internal structure can be determined.

1. INTRODUCTION

X-Ray fluorescence computed tomography (XFCT) is a synchrotron-based imaging technique used for mapping the distribution of elements within a sample. In XFCT, a sample is bombarded with x-rays that excite k-shell electrons. When these atoms return to their stable state, they emit fluorescent x-rays at energies characteristic of the element. These photons are collected by a solid state silicon detector that records multiple energies simultaneously. The total number of photons recorded is a function of the sum of the various element concentrations along the line of the incident beam. By rotating the object and compiling horizontal scans, it is possible to obtain a complete tomographic reconstruction of the distribution of the elements within a sample.

Since the incident beam is attenuated through the sample and part of the emission is absorbed by the sample, attenuation correction is necessary in order to obtain accurate results. If reconstruction techniques are not employed, the image of the center of the sample is blurred, and its density, as recorded by the scan, is much lower than its true density. Previous reconstruction techniques required a known attenuation at each fluorescence energy, which necessitated the time-consuming process of rescanning the sample at all the relevant energies [1]. Tomographic reconstruction is also possible by a series of mathematical corrections based on the inverse Radon transform, which is a faster and simpler method.

These reconstruction techniques have attracted numerous scientific disciplines to XFCT. In particular, the high sensitivity and sub-micrometer resolution of this method is useful in medicine [2]. The presence of metals and other trace elements drastically affect intracellular processes in any organism [3]. XFCT is the only sub-micrometer technique that can map these elements within cells and search for abnormal quantities and distributions accompanying the development of certain diseases [3].

2. MATERIALS AND METHODS

i. Data Collection

Once the code was completed, data was collected at the Stanford Synchrotron Radiation Lightsource. X-Rays obtained from the synchrotron are sent through an ion chamber to measure the energy of the incident beam. The x-rays are then directed into a helium-purged chamber where they are focused down to a 2 μm diameter by two elliptical mirrors. This focused beam is then sent out to the sample, which is scanned by moving completely across the incident beam, then rotating by a certain small angle, typically 1 to 3 degrees, and repeating until 180 degrees are covered. This is called a full translation, half rotation tomographic scan. If the scan were to cover a full rotation, the amount of attenuation correction would be minimized because the image would only contain artifacts towards the center of the sample, but that process doubles the scanning time, generally requiring three to four extra hours.

The detector is placed behind the sample to collect the transmitted x-rays, and another is placed at 90 degrees to the incident beam to collect the fluorescent photons, as depicted in Figure 1. A uniform fluorescence around the sample is assumed, so that one fluorescent photon detector is sufficient. Because elements have signature fluorescence energies, a fluorescent photon detector that can distinguish different photon energy levels is used. This detector counts the number of photons that it receives at each energy level, so the concentrations of different elements in the sample can be determined.

ii. Basic Reconstruction Technique

The data from a tomographic scan is formatted into a two-dimensional matrix composed of pixel intensities. Assuming uniform attenuation throughout the sample, a simple reconstruction method employs the inverse Radon transform.

The inverse Radon transform is the fundamental method for image reconstruction because the Radon Transform acts as a continuous mathematical model for exponentially decreasing density based on a coefficient. A line integral represents the total attenuation of an x-ray as it travels in a straight line through the sample. A tomographic image can be defined by $f(x,y)$, and the attenuation is therefore represented by

$$P_\theta(t) = \int_{(\theta,t)} f(x,y)ds, \text{ (1)}$$

where (θ,t) are the line integral parameters. This can be rewritten as

$$P_\theta(t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f(x,y)\delta(x\cos\theta + y\sin\theta - t)dxdy, \text{ (2)}$$

which is the definition of the Radon transform [4].

To understand the effects of attenuation in XFCT, a Radon transform can be applied to simulated data to create an approximate attenuated image. Figure 2(a) shows a simulated sample of iron without correction for attenuation. Applying an inverse Radon transform with the proper attenuation coefficient produces Figure 2(b), the reconstructed image. Figure 2(c) and 2(d) are analogous images of a calcium sample. The calcium sample is originally more attenuated than the iron sample because the fluorescent photon energies of calcium are lower than those of iron, which makes it harder for them to escape the sample.

A simple reconstruction method takes the inverse Radon transform of the data, and creates a second matrix that defines the shape of the sample, having a "1" wherever the object is, and a "0" everywhere else. The absorption effect can be estimated by taking the Radon Transform of the shape, which is analogous to the survival probability of a photon reaching a point in the sample. Defining this transform as $f(\theta, s, u)$, it can then be estimated that

$$f(\theta, s, u) = \exp\left(-\int_{-\infty}^{u} \mu(s, u')du' \right), \ (3)$$

where μ is the attenuation coefficient, and (s, u) is a point in the matrix as shown in Figure 3, where the s and u axes are parallel to the translation direction and the beam, respectively [5]. The exponential in Equation 1 is therefore the absorption effect. Dividing the initial data by the absorption effect will give the corrected data, and taking the inverse Radon transform of that will give the final corrected image.

This method is useful for ideal samples with uniform attenuation. Otherwise, this technique only works for very small attenuation coefficients on the order of μ = 0.01. For realistic situations, a much more comprehensive method is needed. It must be able to integrate over varying densities with different attenuation coefficients. A successful technique is described in Part II; it employs iterative methods for image reconstruction.

iii. Algorithm Theory

The number of emitted photons detected from a given point in the sample can be defined as

$d(t,\theta) =$ [illegible] $f(t,\theta) = \int_{x \bullet \xi = t} W(x,\theta) f(x) dx$, (4)

where t and θ are parameters defining the line, $\xi = (\cos\theta, \sin\theta)$ [6]. The first integrated function,

$$W(x,\theta) = \exp[-\mathcal{D}\lambda(x,\theta+\pi)]\int_{\gamma_1}^{\gamma_2} \exp[-\mathcal{D}\mu(x,\theta+\gamma)d\gamma\ ],\ (5)$$

accounts for the different fluorescence energies of elements and gives more weight to elements whose fluorescence energies are lower [6]. The limits of integration refer to the angle range $[\gamma_1, \gamma_2]$ at each emission point [6]. The operator $\mathcal{D}$ is the divergent beam transform defined by

$$\mathcal{D}\mu(x,\theta) = \int_0^\infty \mu(x + q\xi^\perp)dq\ ,\ (6)$$

where μ is the estimated fluorescence attenuation coefficient and λ is the known transmission coefficient [6]. When $\mu = \lambda = 0$, $\mathcal{R}_W$ is the general Radon transform [6]. Therefore, similar to the method used in Part I, the attenuation problem can be solved by inverting the operator $\mathcal{R}_W$ [6].

When μ = 0, the density of a sample can be approximated as

$$f_c(x) = \frac{1}{2\pi}\frac{\mathcal{R}^{-1}\{d\}(x)}{a(x)},\ (7)$$

where

$$a(x) = \frac{1}{2\pi}\int_0^{2\pi} W(x,\theta)d\theta\ \ (8)$$

is the correction factor [6]. Miqueles and De Pierro [4] discovered that this general formula is actually the first iteration of an iterative method, where each iteration consists of an inverse Radon Transform $\mathcal{R}^{-1}$. They define the sequence

$$f^{(k+1)} = f^{(k)} + \alpha_k e^{(k)},\ (9)$$

where α_k is the positive relaxation factor and $e^{(k)}$ is defined as

$$e^k = \frac{\mathcal{R}^{-1}(d - \mathcal{R}_W f^{(k)})}{a}\ [6].\ (10)$$

Therefore the complete reconstruction algorithm by iteration is

$$f^{(k+1)}(x) = f^{(k)}(x)\frac{\mathsf{B}_W d^{(k)}(x)}{\mathsf{B}_W e(x)} \quad (11)$$

where $e(t,\theta) = 1$, and

$$d^{(k)}(t,\theta) = \frac{d(t,\theta)}{\mathsf{R}_W f^{(k)}(t,\theta)} \text{ [4].} \quad (12)$$

B_W is the attenuated backprojection operator, defined as

$$\mathsf{B}_W d(x) = \int_0^{2\pi} W(x,\theta)d(x\cdot\xi,\theta)d\theta\,, \quad (13)$$

and it can be shown that it is the adjoint operator of R_W [6].

This method was translated into a code that processes images from tomographic scans. The matrix data was corrected so that each pixel was given a size of 1. This method was then applied, treating each element of the matrix as a point in space.

In order to translate this into a working code, the algorithm from Equation 9 was applied to every point in the tomographic image for both a beam attenuation correction, which accounts for any attenuation from the source to a point in the sample, and for a fluorescence attenuation correction, which accounts for any attenuation from that point in the sample to the detector. From both sets of attenuation, a matrix of absorption corrections was constructed. A filter was applied to the initial data, which distinguished the areas where the most attenuation occurred and where the least attenuation occurred. In order to get the final image, the filtered image was divided by the absorption corrections at each pixel.

3. RESULTS

This method was applied to a tomographic scan of a stem of the plant Arabidopsis thaliana with an estimated attenuation coefficient of 0.08. The initial scan is shown in Figure 4a, and its inverse Radon transform is shown in Figure 4b. The tomographic image with the applied reconstruction is shown in Figure 4c, and while the elimination of attenuation artifacts is not immediately noticed, data analysis showed an improvement.

In order to quantitatively measure the effect of the reconstruction program, the original data was subtracted from the corrected data, providing a matrix of absorption corrections that had been made. The maximum attenuation occurred in the back of the sample, opposite from the incident x-rays, which is expected from a half rotation, full translation scan. The pixels in the back of the sample had lost at least 16% of their original density, which was added back during the reconstruction.

Because there are still attenuation artifacts present in the corrected image, the inputted attenuation coefficient was not high enough. Inputting coefficients that are too low barely affect the attenuated data, and inputting coefficients that are too high add too much density back to the image and results in a loss of real data. A solution to this problem is discussed further in Section 4*i*.

4. DISCUSSION

i. Method Discussion

The proposed method yields accurate images that provide more detailed information about the sample. Although running the program requires approximately an hour, changing with the size of the sample, it is much faster than the time that would need to be allotted to take a full translation, full rotation scan.

A potential disadvantage of the method is the required estimation for the minimum and maximum attenuation coefficients. An ideal attenuation coefficient input is a matrix of varying attenuation coefficients that can be constructed from the density map of the sample obtained from the transmission photon detector. Once an estimated matrix is entered, the iterative method from Equation 11 can be applied to adjust the coefficients until an ideal image is obtained. Miqueles and De Pierro [4] found that using several iterations will limit the attenuation coefficients and further improve the data.

The reconstruction code is given as an appendix. The filtering functions and the mean correlated matrix function perform the reconstruction and create a two-dimensional array of self-absorption corrections. These functions can be added to any module that processes tomographic scans. Dividing the inverse Radon transform of the original scan by this matrix will produce the corrected tomographic image; a simple version of this is given in the fluorescent matrix function, but most modules already include an inverse Radon transform code.

ii. Sample Discussion

The sample used was brought to SSRL by Tracy Punshon from Dartmouth College. The plant Arabidopsis thaliana is of interest because it is the first plant to have its entire genome sequenced, thus being a model organism for understanding plant biology and plant traits. This sample was the wild type Arabidopsis, and other measurements consisted of plants that had a gene removed for iron transport; the intention was to study how the distribution of iron changes without the gene for iron transport.

This is significant for the current interest in trying to figure out ways to encourage iron transport in other plants, such as rice. If the process can be encouraged, rice could be infused with iron, which could potentially solve the growing iron deficiency problems in developing countries.

5. CONCLUSION

XFCT has the potential to provide incredibly informative images once a reconstruction has been applied to the data. An effective reconstruction technique employs the inverse Radon transform and accounts for changing attenuation coefficients and the variance of fluorescence energies. The attenuation coefficient map is the key input for a successful reconstruction, and its creation can be perfected by using iterative methods. Using a reconstruction method can provide accurate results despite the large attenuation effects accumulated from a half rotation scan. By correcting for the attenuation effects, the inaccuracies associated with XFCT become nearly irrelevant, which makes it very attractive, especially to the biological and medical communities. This method has become vital for the analysis and mapping of elemental content in cells and tissues.

6. ACKNOWLEDGEMENTS

This data was obtained at the Stanford Synchrotron Radiation Lightsource at SLAC National Accelerator Laboratory. I would like to thank my mentor, Samuel Webb, for his help and guidance, as well as his persistent positive encouragement throughout this program. I would like to thank Tracy Punshon for providing interesting samples, and Apurva Mehta for all his feedback. I would also like to thank Stephen Rock, SueVon Gee, Elizabeth Smith, and Vivian Lee for their support of the SULI Program. Finally, I would like to thank the Department of Energy and Stanford University for the opportunity to participate in this internship.

7. REFERENCES

[1] P. La Rivière, D. Billmire, P. Vargas, M. Rivers, S. Sutton, *Penalized-likelihood image reconstruction x-ray fluorescence computed tomography*, Optical Engineering, Vol. 45(7). (2006)

[2] C. Schroer, *Reconstructing x-ray fluorescence microtomograms*, Applied Physics Letters, Vol. 79, Number 12. (2001)

[3] T. Paunesku, S. Vogt, J. Maser, B. Lai, G. Woloschak, *X-Ray Fluorescence Microprobe Imaging in Biology and Medicine*, Journal of Cellular Biochemistry 99, 1489-1502. (2006)

[4] A. Kak, M. Slaney, *Principles of Computerized Tomographic Imaging*, Society for Industrial and Applied Mathematics, IEEE Press, New York. (2001)

[5] A. Brunetti, B. Golosio, *Software for X-Ray fluorescence and scattering tomographic reconstruction*, Computer Physics Communications 141, 412-425. (2001)

[6] E. Miqueles, A. De Pierro, *Fluorescence Tomography: Reconstruction By Iterative Methods*, Applied Mathematics Department, State University of Campinas, Brazil. (2008)

FIGURES

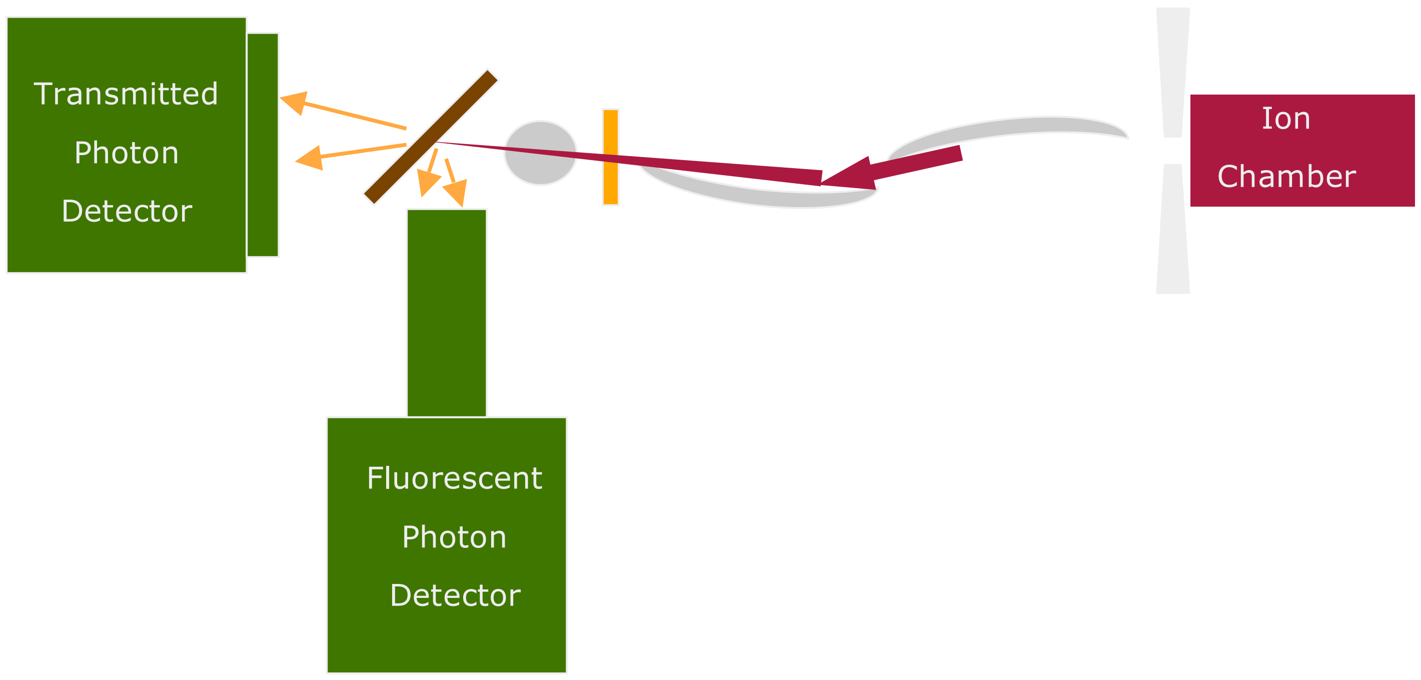


Figure 1: Schematic of experimental setup.

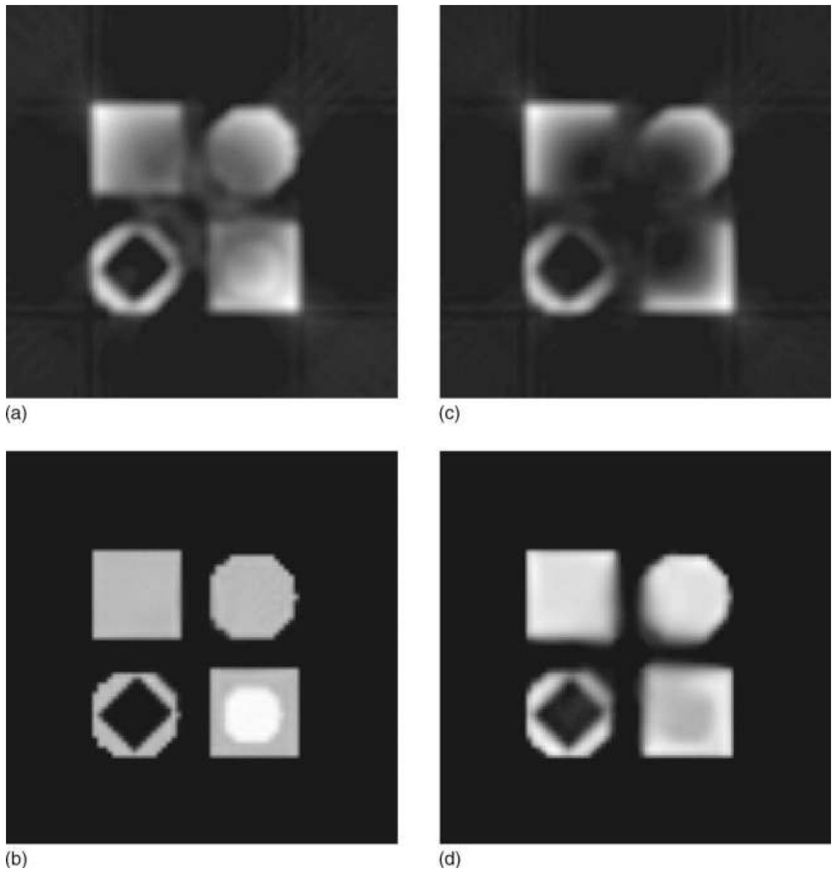


Figure 2: Simulated iron and calcium distributions. (a) Iron without correction for attenuation. (b) Iron with correction for attenuation. (c) Calcium without correction for attenuation. (d) Calcium with correction for attenuation. [1]

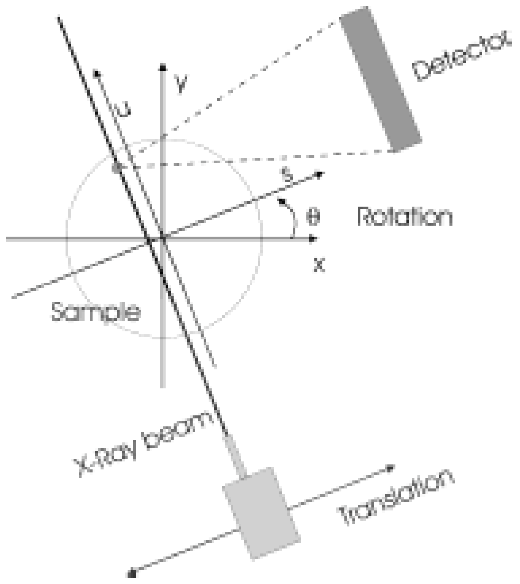
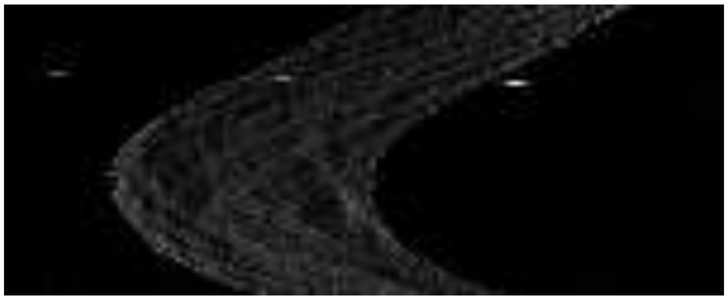
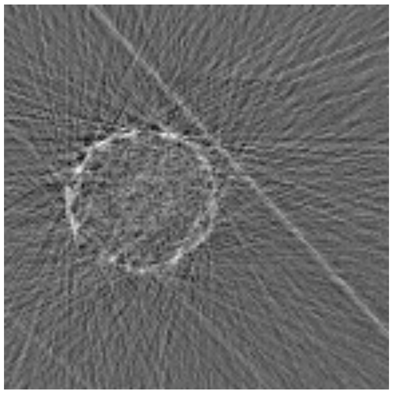


Figure 3: Diagram of fluorescence tomography [5].


(a)


(b)

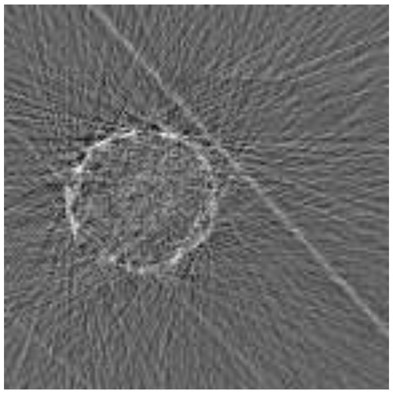

(c)

Figure 4: Tomographic scan and reconstruction of Arabidopsis thaliana. (a) The initial scan. (b) The inverse Radon transform of the scan, producing the tomographic image. (c) The fluorescence corrected image.

APPENDIX

Copy of Reconstruction Code:

```
from math import *
from numpy import *

class Mkcorralgo:
   def __init__(self, x, y, Dim, Dim2, PhiMin, PhiMax, PhiStep, DetPos, DetPixelNum,
DetArea, data_file, AttLowVal, AttHiVal, AttLowSize, AttHiSize, CorrMatrixSize,
Alpha, SLeft, SRight, Scan):
        self.x = x # used as spatial x-coordinate of the pixel
        self.y = y # used as spatial y-coordinate of the pixel
        self.xIdx = 0
        self.yIdx = 0
        self.Dim = Dim # Number of pixels across one sample axis, assuming square sample
        self.Dim2 = Dim2 # Number of angles covered by sample rotation
        self.PhiMin = PhiMin*pi/180.
        self.PhiMax = PhiMax*pi/180.
        self.PhiStep = PhiStep # Angle Incrementation
        self.DetPos = DetPos # Distance from center of sample to detector
        self.DetPixelNum = DetPixelNum # Number of pixels across one sample axis
        self.DetArea = DetArea # Area of detector
        self.DetLen = 2*math.sqrt(DetArea/pi)
        self.DetPixelSize = self.DetLen/self.DetPixelNum
        self.data = data_file
        self.FluorAbsorptionArray = zeros((self.Dim), float)
            # Integrals of the absorption coefficient at fluorescence energy
        self.AttLow = zeros((self.Dim,self.Dim),float)
        self.AttLow = AttLowVal # Matrix of absorption coefficients at fluorescent energy
        self.AttLowSize = AttLowSize # Physical size of sample along one axis
        self.AttLowDim = self.Dim
        self.AttHi = zeros((self.Dim, self.Dim),float)
        self.AttHi = AttHiVal # Matrix of absorption coefficients at beam energy
        self.AttHiSize = AttHiSize # Physical size of sample along one axis
        self.AttHiDim = self.Dim
        self.maxDim = self.Dim
        self.SLeft = SLeft # Parameter of self.Proj
        self.SRight = SRight # Paramter of self.Proj
        self.Scan = Scan # Parameter of self.Proj
        self.ProjNum = int((self.PhiMax - self.PhiMin) / self.PhiStep) + 1
        self.ProjSize = self.SRight - self.SLeft # for self.Proj
        self.ScanNum = int(self.ProjSize/Scan) # for self.Proj
        self.CorrMatrixDim = Dim
        self.CorrMatrixSize = CorrMatrixSize
        self.FluorMatrixDim = Dim
        self.FluorMatrixSize = self.CorrMatrixSixe*self.Dim
        self.Rho2 = 0 # Distance from the fluorescence point to detector points
        self.FilteredProj = zeros((self.ScanNum, self.ProjNum), float)
        self.MeanCorrMatrix = zeros((self.Dim, self.Dim), float)
        self.FluorMatrix = zeros((self.Dim, self.Dim), float) # Fluor Reconstruction Image
        self.Filter = zeros(self.ScanNum, float) # Filter for backprojection
        self.alpha = Alpha
        self.FluorAbs = 0
            # Orthogonal distance from the fluorescence point to the detector

    def idxToCoord(self, Idx, Dim, Size):
        # Converts an array index to a spatial coordinate.
        return (float(Idx)/float(Dim))*Size

    def coordToAngle(self, x, y):
        # Finds the polar angle of a point (x, y).
        r = sqrt(x*x + y*y)
        if r <= x:
            return 0
        Phi = acos(x/r)
        if y < 0:
            Phi = 2*pi - Phi
            return Phi
        else:
            return Phi

    def fluorAttenuation(self, x, y, Phi):
        # Attenuation of the fluorescent photons from point (x, y) to the detector.
        xDet0 = self.DetPos*cos(Phi) # Coordinates of detector center.
        yDet0 = self.DetPos*sin(Phi)
        self.FluorToDetDist = self.DetPos - self.x*cos(Phi) - self.y*sin(Phi)

        #The detector surface is sampled.
        xDet = xDet0
        yDet = yDet0
        x1 = xDet - x
        y1 = yDet - y
        self.Rho2 = x1*x1 + y1*y1 # Square distance from (x, y) to the pixel.
        Gamma = self.coordToAngle(x1, y1) # Polar angle from (x, y) to the pixel.
        integ = ArrayIntegral(x, y, Gamma, self.AttLow, 0, 0,
self.AttLowSize/self.AttLowDim, self.AttLowDim, self.AttLowDim, self.maxDim)
            # Integral of the absorption coefficient at fluorescence energy \\
            # from (x, y) to the pixel.
        self.FluorAbs = integ.integral()
        return self.planeAttenuation()
    def planeAttenuation(self):
        self.Attenuation = 0
        # Integrates the attenuation from (x, y) to the detector surface in the \\
        # angular range covered by the detector.
        for i in range(self.DetPixelNum):
            DetIdx = i
            DGamma = self.DetPixelSize*self.FluorToDetDist/self.Rho2
            self.Attenuation += exp(-self.FluorAbs)*DGamma

    def beamAttenuation(self, x, y, Phi):
        # Finds the attenuation of the x-ray beam from the source to (x, y).

        PhiBeam = Phi - pi/2
        integ = ArrayIntegral(x, y, PhiBeam, self.AttHi, 0, 0, self.AttHiSize/self.AttHiDim,
self.AttHiDim, self.AttHiDim, self.maxDim)
        temp = integ.integral()
        self.beam = exp(-temp)

    def makeCorrMatrix(self):
        # Evaluates and stores 2-D array of self absorption corrections.
        for i in range(int(self.ProjNum)):
            for j in range(int(self.CorrMatrixDim)):
                for k in range(int(self.CorrMatrixDim)):
                    print i, j, k
                    AngleIdx = i
                    xIdx = j
                    yIdx = k
                    if AngleIdx == 0:
                        self.MeanCorrMatrix[xIdx, yIdx] = 0
                    x = self.idxToCoord(xIdx, self.CorrMatrixDim, self.CorrMatrixSize)
                    y = self.idxToCoord(yIdx, self.CorrMatrixDim, self.CorrMatrixSize)
                    Phi = self.PhiMin + AngleIdx*self.PhiStep
                    self.beamAttenuation(x, y, Phi)
                    self.fluorAttenuation(x, y, Phi)
                    self.CorrElem = self.beam*self.Attenuation
                    self.MeanCorrMatrix[xIdx, yIdx] = self.CorrElem/self.ProjNum

    def makeFluorMatrix(self):
        self.makeCorrMatrix()
        self.tomoFilter(self.data, 1)

        # Evaluates fluorescence image through backprojection algorithm using
        # self absorption corrections data.
        Delta = pi/self.ProjNum
        for i in range(self.FluorMatrixDim):
            for j in range(self.FluorMatrixDim):
```

```
            xIdx = i # xIdx, yIdx are indices of a pixel of the Fluorescence Image
            yIdx = j
            # xIdx, yIdx are indices of a pixel of the Fluorescence Image
            x = self.idxToCoord(xIdx, self.FluorMatrixDim, self.FluorMatrixSize)
            y = self.idxToCoord(yIdx, self.FluorMatrixDim, self.FluorMatrixSize)
               # spatial coordinates of the pixel
            BeamSum = 0
            for k in range(self.ProjNum):
               AngleIdx = k
               # finds all the rays that pass through (x, y)
               Phi = self.PhiMin + AngleIdx*self.PhiStep
               s = x*cos(Phi) + y*sin(Phi)
               sIdx = int((s - self.SLeft) / self.Scan) - 1
               print AngleIdx, sIdx
               BeamSum += Delta*self.FilteredProj[AngleIdx, sIdx]
                  # All such rays are summed
            BeamSum = max(BeamSum, 0)
            self.FluorMatrix[xIdx, yIdx] = BeamSum/self.MeanCorrMatrix[xIdx, yIdx]
            # the pixel content is divded by the absorption correction factor

   def SLFilter(self, x, d):
      # if self.alpha == 0, the Shepp-Logan filter is used
      Filter=(pi**2)*d*(1-4*(x**2))
      for i in range(len(Filter) - 1):
         if abs(Filter[i]) < 1e-6:
            Filter[i] = 0.001
      return 2/Filter

   def RLFilter(self, x, d):
      # if self.alpha > 0, the Ram-Lak Filter calculated and then the GH Filter is used
      q = x
      y = -(sin(x*pi/2))**2/((pi**2)*(q**2)*d)
      for i in range(len(q)):
         if (x[i] - int(d/2)) == 0:
            y[i] = 1/(4.*d)
      return y

   def GHFilter(self, x, d):
      # the General Hamming Filter
      return self.alpha*self.RLFilter(x, d) + 0.5*(1 - self.alpha)*(self.RLFilter(x-1, d) +
self.RLFilter(x+1, d))

   def tomoFilter(self, image, d, filter_size=0):
      dims = shape(image)
      if not filter_size:
         filter_size=int(dims[0]/4.)
```

```
        nfilter = 2*filter_size + 1
        x = arange(nfilter, typecode = Float) - filter_size
        if self.alpha == 0:
            filter = self.SLFilter(x, d)
        else:
            filter = self.GHFilter(x, d)
        print filter
        (ncols,nrows) = shape(image)
        self.FilteredProj = image
        temp = zeros(ncols+2*nfilter, Float)
        for i in range(nrows):
            #pad with data from first and last columns
            temp[0 : nfilter-1] = image[0, i]
            temp[nfilter+ncols-1 : ncols+2*nfilter-1] = image[ncols-1, i]
            temp[nfilter : nfilter+ncols] = image[:, i]
            temp = convolve(temp, filter, mode=1)
            self.FilteredProj[:,i] = temp[nfilter : nfilter+ncols]


class ArrayIntegral:
    def __init__(self, X, Y, Phi, Array, XRect, YRect, PixelSize, Nx, Ny, MAXNY):
        self.XRect = XRect # x-component of corner of the rectangle
        self.YRect = YRect # y-component of corner of the rectangle
        self.Nx = Nx # width of the rectangle*PixelSize
        self.Ny = Ny # height of the rectangle*PixelSize
        self.array = Array # the two-dimensional distribution
        self.PixelSize = PixelSize # ...pretty self-explanatory
        self.maxNy = MAXNY # maximum height of the rectangle
        self.Dx = 1 # incrementation in x-direction
        self.Dy = 1 # incrementation in y-direction
        self.Phi = Phi # angle from x-axis to point (x, y)
        self.x = (X - XRect)/PixelSize # reduced x-coordinate
        self.y = (Y - YRect)/PixelSize # reduced y-coordinate
        self.IncX = {}
        self.IncY = {}
        self.IncX[0] = self.Dx
        self.IncX[1] = self.Dx
        self.IncY[0] = self.Dy
        self.IncY[1] = self.Dy
        self.length = 0
        self.Integral = 0
        self.finalIntegral = 0

    def xMatrixCheck(self):
        # Checks if the starting point (x, y) is outside the rectangle.
        if self.x < 0:
```

```
        # If x < 0, finds the intersection of the ray with the x = 0 axis.
        if self.x + cos(self.Phi) != self.x:
            xAlpha = self.y - self.x*sin(self.Phi)/cos(self.Phi)
            if xAlpha >= 0 and xAlpha <= self.Ny:
                # If the intersection lies on the rectangle edge,it is taken
                # as a new starting point for the integral.
                self.x = 0
                self.y = xAlpha
    elif self.x > self.Nx:
        # If x > Nx, finds the intersection of the ray with the x = Nx axis.
        xDelta = self.Nx - self.x
        if xDelta + cos(self.Phi) != xDelta:
            xAlpha = self.y + xDelta*sin(self.Phi)/cos(self.Phi)
            if xAlpha >= 0 and xAlpha <= self.Ny:
                self.x = self.Nx
                self.y = xAlpha
    else:
        # The x-coordinate is in the rectangle
        self.x = self.x

def yMatrixCheck(self):
    if self.y < 0:
        # If y < 0, finds the intersection of the ray with the y = 0 axis.
        if self.y + sin(self.Phi) != self.y:
            yAlpha = self.x - self.y*cos(self.Phi)/sin(self.Phi)
            if yAlpha >= 0 and yAlpha <= self.Nx:
                self.y = 0
                self.x = yAlpha
    elif self.y > self.Ny:
        # If y > Ny, finds the intersection of the ray with the y = Ny axis.
        yDelta = self.Ny - self.y
        if yDelta + sin(self.Phi) != yDelta:
            yAlpha = yDelta*cos(self.Phi)/sin(self.Phi)
            if yAlpha >= 0 and yAlpha <= self.Nx:
                self.y = self.Ny
                self.x = yAlpha
    else:
        # The y-coord is in the rectangle
        self.y = self.y

def integral(self):

    self.xMatrixCheck
    self.yMatrixCheck

    self.ix = int(self.x) # integral part of x
```

```
        self.iy = int(self.y) # integral part of y
        self.tx = self.x - self.ix # fractional part of x
        self.ty = self.y - self.iy # fractional part of y
        if self.ix == self.Nx:
            self.ix = self.ix - 1
            self.tx = 1
        if self.iy == self.Ny:
            self.iy = self.iy - 1
            self.ty = 1

        # Angle Restriction
        # Used to reduce the problem to the case of 0 < Phi < pi/4.
        if self.Phi < 0: #makes Phi element of [0, 2pi]
            while self.Phi < 0:
                self.Phi = self.Phi + 2*pi
        if self.Phi > pi: #makes Phi element of [0, pi]
            self.Phi = 2*pi - self.Phi
            self.Dy = -1
            self.ty = 1 - self.ty
        if self.Phi > pi/2: #makes Phi element of [0, pi/2)
            self.Phi = pi - self.Phi
            self.Dx = -1
            self.tx = 1- self.tx

# When the line integral crosses from one pixel to another, only one of the
        #   two indices, ix or iy, must be changed, depending on which axis,
        #   y or x, is intersected. The indices 0, 1 of the variables IncX, IncY
        #   refer to the x, y axes, respectively.

        if self.Phi > pi/4:
            self.Phi = pi/2 - self.Phi
            self.IncX[1] = 0
            self.IncY[0] = 0
            self.swap(self.tx, self.ty)
        else:
            self.IncX[0] = 0
            self.IncY[1] = 0

# Calculates the integral over the first pixel.

        self.ux = 1 - self.tx
        self.uy = 1 - self.ty
        self.t = self.ux*tan(self.Phi)

        if self.t <= self.uy:
            self.t += self.ty
            self.axis = 1
            self.length = self.ux*(1/cos(self.Phi))
        else:
            self.t = self.tx + self.uy/tan(self.Phi)
            self.axis = 0
            self.length = self.uy/sin(self.Phi)

        self.array = ravel(self.array)
        self.Integral = self.length*self.array[self.ix*self.maxNy + self.iy]
        self.L = self.length

        self.ix += self.IncX[self.axis]
        self.iy += self.IncY[self.axis]

        while self.ix >= 0 and self.ix < self.Nx and self.iy >= 0 and self.iy < self.Ny:
            # Finds all pixels of the rectangle that are intersected by the ray.
            self.integralStep(tan(self.Phi), 1/cos(self.Phi))
            self.length += self.L
            self.Integral += self.L*self.array[self.ix*self.maxNy + self.iy]
            self.ix += self.IncX[self.axis]
            self.iy += self.IncY[self.axis]
        self.length = self.length*self.PixelSize
        self.Integral = self.Integral*self.PixelSize
        return self.Integral

    def integralStep(self, TgPhi, SecPhi):
        # Finds the length of the intersection of the ray with a single pixel.
        self.u = 1 - self.t
        if self.axis == 0:
            self.axis = 1
            self.t = self.u*TgPhi
            self.L = self.u*SecPhi
        else:
            if self.t < (1 - TgPhi):
                self.t += TgPhi
                self.L = SecPhi
            else:
                self.axis = 0
                self.t = self.u/TgPhi
                self.L = self.t*SecPhi

    def swap(self, a, b):
        temp = b
        b = a
        a = temp
        return a, b
```